User's Manual

CubeSuite+ V2.00.00

統合開発環境

ユーザーズマニュアル　RXコーディング編

対象デバイス

RXファミリ

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2013.03

ご注意書き

1. 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・システムの設計において、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て、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。
2. 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。
3. 本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害に関し、当社は、何らの責任を負うものではありません。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。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
   各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   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、産業用ロボット等

   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、各種安全装置等

   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）　、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。 たとえ、意図し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損害が生じても、当社は一切その責任を負いません。 なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の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。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、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、社会的損害等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、お客様の機器・システムとしての出荷保証を行ってください。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、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の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。
9. 本資料に記載されている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・システムに使用することはできません。また、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に使用しないでください。当社製品または技術を輸出する場合は、「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。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
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禁じます。

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。

このマニュアルの使い方

このマニュアルは，RX ファミリ用アプリケーション・システムを開発する際の統合開発環境である CubeSuite+ について説明します。

CubeSuite+ は，RX ファミリの統合開発環境（ソフトウエア開発における，設計，実装，デバッグなどの各開発フェーズに必要なツールをプラットフォームである IDE に統合）です。統合することで，さまざまなツールを使い分ける必要がなく，本製品のみを使用して開発のすべてを行うことができます。

対 象 者　このマニュアルは，CubeSuite+ を使用してアプリケーション・システムを開発するユーザを対象としています。

目　　的　このマニュアルは，CubeSuite+ の持つソフトウエア機能をユーザに理解していただき，これらのデバイスを使用するシステムのハードウエア，ソフトウエア開発の参照用資料として役立つことを目的としています。

構　　成　このマニュアルは，大きく分けて次の内容で構成しています。

読 み 方　このマニュアルを読むにあたっては，電気，論理回路，マイクロコンピュータに関する一般知識が必要となります。

凡　　例

データ表記の重み	：左が上位桁，右が下位桁
アクティブ・ロウの表記	：$\overline{\text{XXX}}$（端子，信号名称に上線）
注	：本文中につけた注の説明
注意	：気をつけて読んでいただきたい内容
備考	：本文中の補足説明
数の表記	：10 進数　...　XXXX
	16 進数　...　0xXXXX

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

資料名		資料番号	
		和文	英文
CubeSuite+ 統合開発環境 ユーザーズ・マニュアル	起動編	R20UT2444J	R20UT2444E
	V850 設計編	R20UT2134J	R20UT2134E
	RL78 設計編	R20UT2136J	R20UT2136E
	78K0R 設計編	R20UT2137J	R20UT2137E
	78K0 設計編	R20UT2138J	R20UT2138E
	RX コーディング編	このマニュアル	R20UT2470E
	V850 コーディング編	R20UT0553J	R20UT0553E
	コーディング編（CX コンパイラ）	R20UT2139J	R20UT2139E
	RL78, 78K0R コーディング編	R20UT2140J	R20UT2140E
	78K0 コーディング編	R20UT2141J	R20UT2141E
	RX ビルド編	R20UT2472J	R20UT2472E
	V850 ビルド編	R20UT0557J	R20UT0557E
	ビルド編（CX コンパイラ）	R20UT2142J	R20UT2142E
	RL78, 78K0R ビルド編	R20UT2143J	R20UT2143E
	78K0 ビルド編	R20UT0783J	R20UT0783E
	RX デバッグ編	R20UT2350J	R20UT2350E
	V850 デバッグ編	R20UT2446J	R20UT2446E
	RL78 デバッグ編	R20UT2445J	R20UT2445E
	78K0R デバッグ編	R20UT0732J	R20UT0732E
	78K0 デバッグ編	R20UT0731J	R20UT0731E
	解析編	R20UT2447J	R20UT2447E
	メッセージ編	R20UT2448J	R20UT2448E

注意　上記関連資料は，予告なしに内容を変更することがあります。設計などには，必ず最新の資料を使用してください。

目　　次

第1章 概　　説

この章では、RX ファミリ向け C/C++ コンパイラが行うコンパイル処理の概要と、プログラム開発の事例を説明します。

1.1 概　　要

ビルド・ツール（CC-RX）は，本製品が提供しているコンポーネントの一種であり，GUI ベースで各種情報を設定することにより，ソースファイルからロードモジュールファイル，ライブラリファイルを，目的に応じて生成することができます。

CC-RX は、以下に示す実行ファイルで構成されています。

(1) ccrx: コンパイルドライバ

(2) rxc: コンパイラ

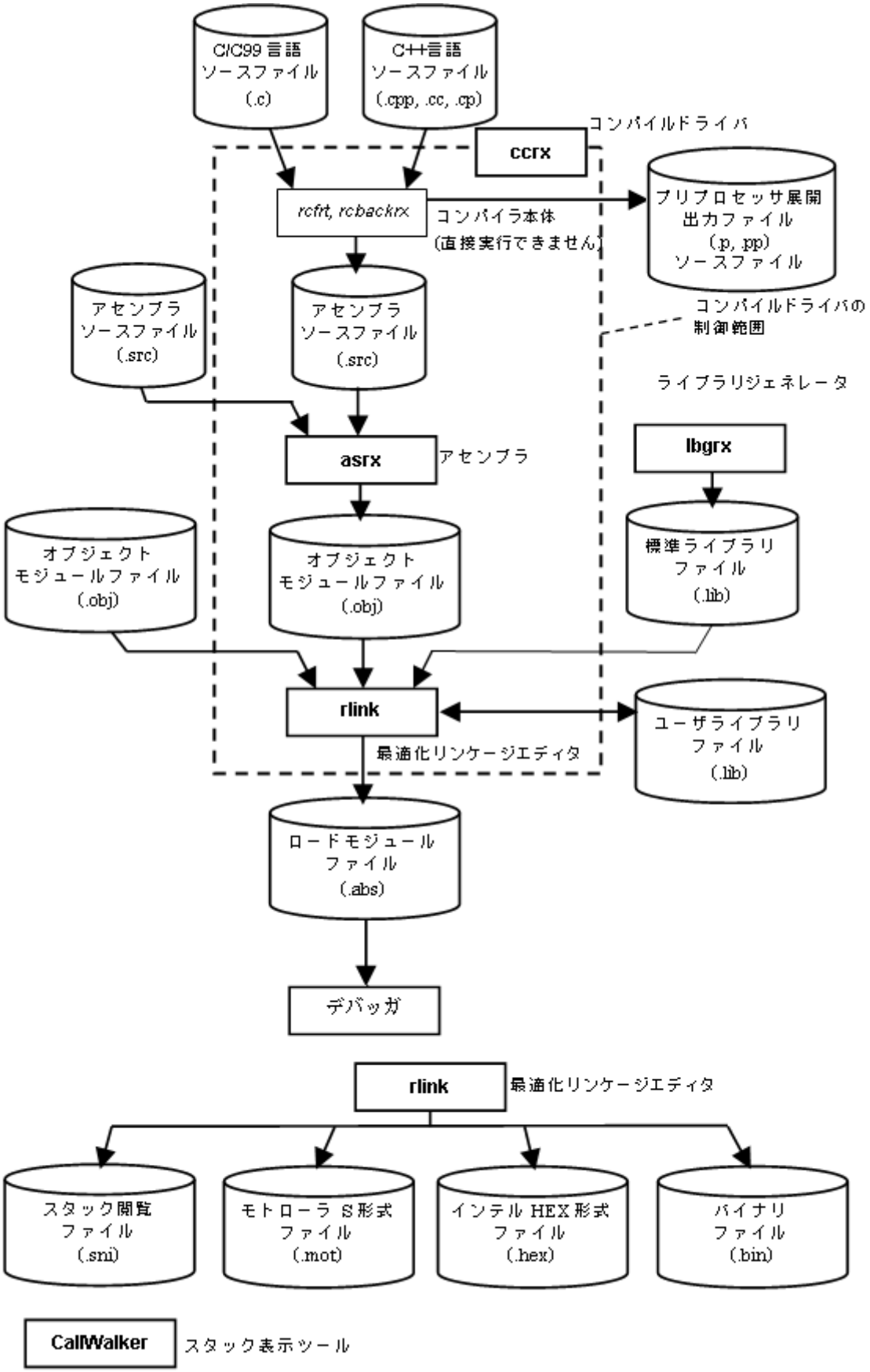
(3) asrx: アセンブラ

(4) rlink: 最適化リンケージエディタ

(5) lbgrx: ライブラリジェネレータ

以下に、ビルド・ツールの処理の流れを示します。

図 1―1　ビルド・ツールの処理の流れ


C/C99 言語
ソースファイル
(.c)
C++言語
ソースファイル
(.cpp, .cc, .cp)
コンパイルドライバ
ccrx
rcfrt, rcbackrx
コンパイラ本体
(直接実行できません)
プリプロセッサ展開
出力ファイル
(.p, .pp)
ソースファイル
アセンブラ
ソースファイル
(.src)
アセンブラ
ソースファイル
(.src)
コンパイルドライバの
制御範囲
ライブラリジェネレータ
asrx
アセンブラ
lbgrx
オブジェクト
モジュールファイル
(.obj)
オブジェクト
モジュールファイル
(.obj)
標準ライブラリ
ファイル
(.lib)
rlink
最適化リンケージエディタ
ユーザライブラリ
ファイル
(.lib)
ロードモジュール
ファイル
(.abs)
デバッガ
rlink
最適化リンケージエディタ
スタック閲覧
ファイル
(.sni)
モトローラ S形式
ファイル
(.mot)
インテル HEX形式
ファイル
(.hex)
バイナリ
ファイル
(.bin)
CallWalker
スタック表示ツール

1.2 特 長

RX ファミリ用 C/C++ コンパイラパッケージ (CCRX) は、次の特長を備えています。

(1) ANSI 規格に準拠した言語仕様

C,C99,C++ 言語仕様は，ANSI 規格に準拠しています。また，従来の C 言語仕様（K&R 仕様）との両立性も備えています。

(2) 高度な最適化

コンパイラによるコード・サイズ，および速度優先の最適化を提供しています。

(3) 記述性の向上

拡張言語仕様によりプログラミングの記述性を向上させています。

(4) 高い移植性

CCRX では単一のコンパイラですべてのマイクロコントローラをサポートしています。これにより言語仕様の統一を図り，マイクロコントローラ間の移行を容易にしています。

また，デバッグ情報には業界標準フォーマットである DWARF2/3 を採用しています。

(5) 多機能性

CubeSuite+ との連携による静的解析機能などを提供します。

1.3 最大値

1.3.1 コンパイラの最大値

ソースプログラムを作成する際は、この翻訳限界の範囲で作成してください。

表 1―1 コンパイラの翻訳限界

No.	分類	項目	翻訳限界
1	起動	define オプションで指定可能なマクロ名総数	制限なし (メモリ容量に依存)
2		ファイル名の文字数	制限なし (OS に依存)
3	ソース プログラム	1 行の文字数	32768 文字
4		1 ファイルあたりのソースプログラムの行数	制限なし (メモリ容量に依存)
5		コンパイル可能なソースプログラムの総行数	制限なし (メモリ容量に依存)

No.	分類	項目	翻訳限界
6	プリプロセッサ	#include 文のネストの深さ	制限なし (メモリ容量に依存)
7		#define 文のマクロ名総数	制限なし (メモリ容量に依存)
8		マクロ定義、マクロ呼び出しのパラメータの個数	制限なし (メモリ容量に依存)
9		マクロ名の再置き換えの数	制限なし (メモリ容量に依存)
10		条件コンパイルのネストのレベル数	制限なし (メモリ容量に依存)
11		#if, #elif 文で指定可能な演算子、非演算子の合計数	制限なし (メモリ容量に依存)
12	宣言	関数定義の個数	制限なし (メモリ容量に依存)
13		外部結合となる識別子 (外部名) の数	制限なし (メモリ容量に依存)
14		1 関数内で有効な識別子 (内部名) の数	制限なし (メモリ容量に依存)
15		基本型を修飾するポインタ、配列、および関数宣子の数	16 個
16		配列の次元数	6 次元
17		配列・構造体のサイズ	2147483647 バイト
18	文	複文のネストの深さ	制限なし (メモリ容量に依存)
19		繰り返し文 (while 文、do 文、for 文)、選択文 (if 文、switch 文) の組み合わせによるネストの深さ	4096 レベル
20		1 関数内で記述可能な複文の数	2048 個
21		1 関数内で指定可能な goto ラベルの数	2147483646 個
22		switch 文の数	2048 個
23		switch 文のネストの深さ	2048 レベル
24		１つの switch 文内で指定可能な case ラベルの数	2147483646 個
25		for 文のネストの深さ	2048 レベル
26	式	文字列の文字数	32766 文字
27		関数定義、関数呼び出しでパラメータの個数	2147483646 個
28		1 つの式で指定可能な演算子と非演算子の合計数	約 500 個
29	標準ライブラリ	open 関数で一度にオープンできるファイルの数	可変 *1
30	セクション	セクション名長 *2	8146 文字
31		1 ファイルあたりの #pragma section で指定できるセクション数	2045 個
32		セクションの最大サイズ	4294967295 バイト
33	出力ファイル	アセンブリソースとして出力できる 1 行の最大文字数	8190 文字

注 1. 詳細は「7. 3. 2 初期設定」を参照してください。

2. オブジェクト生成時に用いるアセンブラの 1 行文字数の制限を受けるため、#pragma section や section オプションで指定できる長さはこれより小さくなります。

1. 3. 2 アセンブラの最大値

ソースプログラムを作成する際は、この翻訳限界の範囲で作成してください。

表 1―2 アセンブラの翻訳限界

	項目	翻訳限界
1	1 行文字数	32760 文字
2	シンボル長	1 行文字数 *1
3	シンボル数	制限なし (メモリ容量に依存)
4	外部参照シンボル数	制限なし (メモリ容量に依存)
5	外部定義シンボル数	制限なし (メモリ容量に依存)
6	セクションの最大サイズ	0FFFFFFFFH バイト
7	セクション数	65265 個 (デバッグ情報あり)、65274 個 (デバッグ情報なし)
8	ファイルインクルード	ネストは 30 レベル
9	文字列長	1 行文字数 *1
10	ファイル名の文字数	1 行文字数 *1
11	環境変数設定文字数	2048 バイト
12	マクロ定義数	65535 個

注 1. 同じ行に指定した文字列の長さにより、これよりも小さい値となります。

第2章　機　　能

この章では、RX ファミリをより効果的に用いるためのプログラミング技法、および拡張機能の利用方法について説明します。

2.1　変数 (C 言語)

この節では，変数（C 言語）について説明します。

2.1.1　配置領域を変更する

変数のデフォルトの配置セクションは，次のとおりになります。

- 初期値なし変数：B, B_2, B_1 セクション
- 初期値あり変数：D, D_2, D_1 セクション (ROM)，R, R_2, R_1 セクション (RAM)
- const 変数：C, C_2, C_1 セクション

配置する領域（セクション）を変更するには，#pragma section でセクション種別、セクション名を指定します。

#pragma section セクション種別　セクション名
　　変数宣言 / 定義
#pragma section

セクション種別を指定すると指定した種別のみがセクション名変更の対象になります。

尚、RX ファミリ C/C++ コンパイラでは，変数のアライメント数に応じて配置するセクションを分けています。
例)

B　：アライメント数が 4 バイトの初期値無し変数を配置
B_2：アライメント数が 2 バイトの初期値無し変数を配置
B_1：アライメント数が 1 バイトの初期値無し変数を配置

初期値あり変数の場合は，初期値が ROM に配置されて，変数自体は RAM に配置されます（ROM/RAM 両方の領域が必要になります）。スタートアップ・ルーチンの resetprg.c ファイルを使用した場合は、INITSCT 関数で ROM の初期値を RAM の変数にコピーします。

セクション種別と生成されるセクションの関係は次のとおりです。

名称	セクション名称	属性	形式種別	初期値 / 書き込み	アライメント
定数領域	C[*1*2]	romdata	相対	有 / 不可	4byte
	C_2[*1*2]	romdata	相対	有 / 不可	2byte
	C_1[*1*2]	romdata	相対	有 / 不可	1byte
初期化データ	D[*1*2]	romdata	相対	有 / 可	4byte
	D_2[*1*2]	romdata	相対	有 / 可	2byte
	D_1[*1*2]	romdata	相対	有 / 可	1byte
未初期化データ	B[*1*2]	data	相対	無 / 可	4byte
	B_2[*1*2]	data	相対	無 / 可	2byte
	B_1[*1*2]	data	相対	無 / 可	1byte
switch 文分岐	W[*1]	romdata	相対	有 / 不可	4byte
テーブル領域	W_2[*1]	romdata	相対	有 / 不可	2byte
	W_1[*1]	romdata	相対	有 / 不可	1byte
C++ 初期処理後処理データ	C$INT	romdata	相対	有 / 不可	4byte
C++ 仮想関数表	C$VTBL	romdata	相対	有 / 不可	4byte
絶対アドレス変数	$ADDR_<section>_<address>[*3]	data	絶対	有無 / 可不可 [*4]	-
可変ベクタ領域	C$VECT	romdata	相対	無 / 可	-

注 1. section オプションまたは拡張子 #pragma section でセクション名を切り替えることができます。
ただし、文字列リテラルなどデータの一部に #pragma section の影響を受けないものがあります。
詳しくは、3. 2 拡張言語仕様 #pragma section の詳細説明を参照ください。

2. セクション名切り替えの際に、アライメント数が 4 のセクションを指定することで、アライメントが 1 または 2 のセクション名も変更されます。#pragma endian で endian オプションと異なる指定のエンディアンを指定した場合、#pragma endian big であれば _B を、#pragma endian little であれば _L を、セクション名の後ろに付加した専用のセクションを生成し、該当データを格納します。
ただし、文字列リテラルなどデータの一部に #pragma endian の影響を受けないものがあります。
詳しくは、3. 2 拡張言語仕様 #pragma endian の詳細説明を参照ください。

3. <section> は C, D, B のセクション名称、<address> は絶対アドレス値 (16 進数) になります。

4. 初期値、書き込み操作は <section> の属性に従います。

2.1.2 通常時と割り込み時に使用する変数を定義する

通常時の処理と割り込み処理の両方で使用する変数は，volatile 指定してください。

volatile 修飾子をつけて変数宣言すると，その変数は最適化の対象からはずされ，レジスタに割り付ける最適化などを行わなくなります。volatile 指定された変数に対する操作を行うときは，必ずメモリから値を読み込み，操作後にメモリへ値を書き込むコードになります。volatile 指定されていない変数は，最適化によってレジスタに割り付けられ，その変数をメモリからロードするコードが削除されることがあります。また volatile 指定されていない変数に同じ値を代入する場合，冗長な処理と解釈されて最適化によりコードが削除されることもあります。

2.1.3 変数を宣言したサイズでアクセスするコードを生成する

変数を宣言したサイズでアクセスする場合は、__evenaccess の拡張機能を使用します。

__evenaccess の宣言により変数の型のサイズでアクセスすることを保証します。保証対象サイズは、4 バイト以下の整数スカラ型 (signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, unsigned long) です。

構造体または共用体に指定した場合、全てのメンバに __evenaccess を指定したのと同じ効果になります。その場合、4 バイト以下の整数スカラ型メンバのアクセスサイズは保証しますが、構造体または共用体単位でのアクセスサイズは保証しません。

【記述例】

C ソースコード

```
#pragma address A=0xff0178
unsigned long __evenaccess A;
void test(void)
{
      A &= ~0x20;
}
```

出力コード（__evenaccess 非指定時）

```
_test:
        MOV.L #16712056,R1
        BCLR #5,[R1] ; 1 バイトメモリアクセス
        RTS
```

出力コード（__evenaccess 指定時）

```
_test:
        MOV.L #16712056,R1
        MOV.L [R1],R5 ; 4 バイトメモリアクセス
        BCLR #5,R5
        MOV.L R5,[ R1] ; 4 バイトメモリアクセス
        RTS
```

2.1.4 値を変更しない初期化変数は const 宣言をする

初期値のある変数は、通常、起動時に ROM エリアから RAM エリアに転送して、RAM エリアを使って処理を行います。このため、プログラム内で値が不変な初期化データの場合、確保した RAM エリアが無駄になります。初期化データに const 演算子をつけておくと、起動時の RAM エリアへの転送が抑止され、使用メモリ量の節約になります。

また初期値は変更しない、というルールでプログラムを作成すると、ROM 化が容易になります。

【改善前記述例】

char a[] = { 1, 2, 3, 4, 5 };

初期値を ROM から RAM へ転送して処理を行います。

【改善後記述例】

const char a[] = { 1, 2, 3, 4, 5 };

ROM 上の初期値を使用して処理を行います。

2. 1. 5 const 定数ポインタを定義する

ポインタについては，“ const ” の指定場所により，異なる解釈がされます。

【記述例 1】

```
const char *p；
```

ポインタが示すオブジェクト（*p）を書き換えできないことを示します。

ポインタ自体（p）は書き換え可能です。

したがって，以下のようになり，ポインタ自体は RAM（B セクション）に配置されます。

```
*p = 0；/* エラー */
p = 0；/* 正しい */
```

【記述例 2】

```
char *const p；
```

ポインタ自体（p）を書き換えできないことを示します。

ポインタが示すオブジェクト（*p）は書き換え可能です。

したがって，以下のようになり，ポインタ自体は ROM（C セクション）に配置されます。

```
*p = 0；/* 正しい */
p = 0；/* エラー */
```

【記述例 3】

```
const char *const p；
```

ポインタ自体（p），ポインタが示すオブジェクト（*p）を書き換えできないことを示します。

したがって，以下のようになり，ポインタ自体は ROM（C セクション）に配置されます。

```
*p = 0；/* エラー */
p = 0；/* エラー */
```

2. 1. 6 セクションのアドレスを参照する

セクションアドレス演算子を使用することでセクションの先頭アドレス、末尾 +1 アドレス、サイズを参照することができます。

__sectop ("< セクション名 >") < セクション名 > の先頭アドレスを参照します。

__secend ("< セクション名 >") < セクション名 > の末尾 +1 アドレスを参照します。

__secsize ("< セクション名 >") < セクション名 > のサイズを生成します。

【記述例】

```
#pragma section $DSEC
static const struct {
        void *rom_s; /* 初期化データセクションの ROM 上の先頭アドレス値を取得 */
        void *rom_e; /* 初期化データセクションの ROM 上の最終アドレス値を取得 */
        void *ram_s; /* 初期化データセクションの RAM 上の先頭アドレス値を取得 */
} DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};
```

スタートアップ・ルーチンの resetprg.c ファイルの INITSCT 関数内では、ROM から RAM への転送、及び未初期化領域の初期化を実行します。この実行の際、dbsct.c ファイル内に記述されている __sectop、__secend で取得したアドレスを参照しています。

2.2 関数

この節では，関数について説明します。

2.2.1 アセンブラ命令の埋め込み

RX C/C++ コンパイラでは，#pragma inline_asm において，C 言語ソース・プログラム中にアセンブラ命令が記述できます。

【記述例】

```
#pragma inline_asm func
static int func(int a, int b){
        ADD R2,R1 ; アセンブリ記述
}
main(int *p){
        *p = func(10,20);
}
```

#pragma inline_asm で宣言したアセンブリ記述関数をインライン展開します。

アセンブラ埋め込みインライン関数の呼び出し規則は通常関数の呼び出し規則と同様です。

2.2.2 関数のインライン展開を行う

#pragma inline は、インライン展開する関数を宣言します。

インライン展開の有無はコンパイラオプションの inline/noinline でも制御しますが、noinline オプションが指定された場合でも、#pragma inline 指定された関数はインライン展開の対象となります。

関数名には、グローバル関数および静的関数メンバを指定できます。#pragma inline で指定した関数名の関数と関数指定子 inline(C++ 言語および C(C99) 言語) を指定した関数は、その関数を呼び出したところにインライン展開されます。

【記述例】

C ソースコード

```
#pragma inline(func)
static int func (int a, int b)
{
          return (a+b)/2;
}
int x;
main()
{
          x=func(10,20);
}
```

展開イメージ

```
int x;
main()
{
          int func_result;
          {
                    int a_1=10, b_1=20;
                    func_result=(a_1+b_1)/2;
          }
            x=func_result;
}
```

2. 2. 3 関数のインライン展開を行う (ファイル間)

通常、インライン展開はファイル内の関数のみがインライン展開の対象になりますが、コンパイラの -file_inline オプションを使用することでファイル間の関数呼び出しもインライン展開の対象になります。

【記述例】

```
<a.c>
  func(){
          g();
   }
<b.c>
  g(){
          h();
}
ccrx -inline -file_inline=b.c a.c
```

と指定してコンパイルすることにより a.c の中の関数 g の呼び出しが展開され以下のようになります。

```
func(){
          h();
}
```

2. 3 マイコン機能の使用

この節では，マイコン機能の使用について説明します。

2. 3. 1 C 言語で割り込み処理を行う

割り込み関数は、#pragma interrupt によって宣言します。

【記述例】

C ソース

```
#pragma interrupt func
void func(){ .... }
```

生成コード

```
_func:
          PUSHM R1-R3 ; 関数内で使用しているレジスタを退避
                    ・・・
          (R1,R2,R3 を関数内で使用 )
                    ・・・
          POPM R1-R3 ; 入口で退避したレジスタを回復
          RTE
```

2. 3. 2 C 言語で CPU 命令を使用する

制御レジスタへアクセスする場合や C 言語で表現できない特殊命令に関しては、組込み関数を提供しています。

- 最大値、最小値
- データ内バイト入れ替え
- データ交換
- 積和演算
- 回転
- 特殊命令 (BRK, WAIT, INT, NOP)
- BRK、WAIT などの RX ファミリ用特殊命令
- 制御レジスタ設定、参照

組込み関数の一覧

機能	仕様
signed long max(signed long data1, signed long data2)	最大値の選択
signed long min(signed long data1, signed long data2)	最小値の選択
unsigned long revl(unsigned long data)	ロングワードデータをバイトリバース
unsigned long revw(unsigned long data)	ロングワードデータをワード毎にバイトリバース
void xchg(signed long *data1, signed long *data2)	データ交換
long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)	積和演算 (バイト)
long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)	積和演算 (ワード)
long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)	積和演算 (ロングワード)
unsigned long rolc(unsigned long data)	キャリーを含めて 1 ビット左回転
unsigned long rorc(unsigned long data)	キャリーを含めて 1 ビット右回転
unsigned long rotl(unsigned long data, unsigned long num)	左回転
unsigned long rotr (unsigned long data, unsigned long num)	右回転
void brk(void)	BRK 命令例外
void int_exception(signed long num)	INT 命令例外
void wait(void)	プログラム実行停止
void nop(void)	NOP 命令に展開
void set_ipl(signed long level)	割り込み優先レベルの設定
unsigned char get_ipl(void)	割り込み優先レベルの参照
void set_psw(unsigned long data)	PSW の設定
unsigned long get_psw(void)	PSW の参照
void set_fpsw(unsigned long data)	FPSW の設定
unsigned long get_fpsw(void)	FPSW の参照
void set_usp(void *data)	USP の設定
void *get_usp(void)	USP の参照
void set_isp(void *data)	ISP の設定

機能	仕様
void *get_isp(void)	ISP の参照
void set_intb(void *data)	INTB の設定
void *get_intb(void)	INTB の参照
void set_bpsw(unsigned long data)	BPSW の設定
unsigned long get_bpsw(void)	BPSW の参照
void set_bpc(void *data)	BPC の設定
void *get_bpc(void)	BPC の参照
void set_fintv(void *data)	FINTV の設定
void *get_fintv(void)	FINTV の参照
unsigned long long emulu(unsigned long, unsigned long)	有効桁 64bit の符号なし乗算
signed long long emul(signed long, signed long)	有効桁 64bit の符号付き乗算

2.4 変数（アセンブラ）

この節では，変数（アセンブラ）について説明します。

2.4.1 初期値なし変数を定義する

DATA セクション内にメモリ領域を確保します。

DATA セクションを定義するには、.SECTION 指示命令を使用し、メモリ領域には、1 バイト単位の場合には .BLKB 指示命令を、2 バイト単位の場合には .BLKW 指示命令を、4 バイト単位の場合には .BLKL 指示命令を、8 バイト単位の場合には .BLKD 指示命令を使用します。

【記述例】

```
        .SECTION area,DATA
work1:  .BLKB 1; 1バイトの単位で RAM 領域を確保
work2:  .BLKW 1; 2バイトの単位で RAM 領域を確保
work3:  .BLKL 1; 4バイトの単位で RAM 領域を確保
work4:  .BLKD 1; 8バイトの単位で RAM 領域を確保
```

2.4.2 初期値あり const 定数を定義する

ROMDATA セクション内のメモリ領域を初期化します。

ROMDATA セクションを定義するには .SECTION 指示命令を使用し、メモリ初期化には、1 バイトの場合には .BYTE 指示命令を、2 バイトの場合には .WORD 指示命令を、4 バイトの場合には .LWORD 指示命令を、浮動小数点の 4 バイトの場合には .FLOAT 指示命令を、浮動小数点の 8 バイトの場合には .DOUBLE 指示命令を使用します。

【記述例】

```
       .SECTION value,ROMDATA
work1: .BYTE "data"   ;  1バイト長の固定データを ROM に格納
work2: .WORD "data"   ;  2バイト長の固定データを ROM に格納
work3: .LWORD "data"  ;  4バイト長の固定データを ROM に格納
work4: .FLOAT 5E2     ;  4バイト長の浮動小数点データを ROM に格納
work5: .DOUBLE 5E2    ;  8バイト長の浮動小数点データを ROM に格納
```

2.4.3 セクションのアドレスを参照する

SIZEOF、TOPOF 演算子によりオペランドに指定したセクションのサイズ、開始アドレスを値として扱います。

【記述例】

```
・・・
MVTC        #(TOPOF SU + SIZEOF SU),USP
; SU の開始アドレス +SU のサイズによりユーザスタック領域のアドレスを USP へ設定
MVTC        #(TOPOF SI + SIZEOF SI),ISP
; SI の開始アドレス +SI のサイズにより割り込みスタック領域のアドレスを ISP へ設定
・・・
```

2.5 スタートアップ・ルーチン

この節では、スタートアップ・ルーチンについて説明します。

2.5.1 スタック領域を確保する

スタートアップ・ルーチンの resetprg.c ファイルの PowerON_Reset 関数を "#pragma entry" 宣言している事により、下記の設定によりコンパイラとリンカが自動的にユーザスタック USP/ 割り込みスタック ISP 初期化コードを関数先頭に生成します。

(1) ユーザスタックの設定

stacksct.h ファイルの #pragma stacksize su=0xXXX でスタック領域のサイズ、最適化リンカの -start オプションで SU セクションの配置を指定します。

(2) 割り込みスタックの設定

stacksct.h ファイルの #pragma stacksize si=0xXXX でスタック領域のサイズ、最適化リンカの -start オプションで SI セクションの配置を指定します。

【記述例】

```
<resetprg.c>
 ・・・
 #pragma section ResetPRG
 #pragma entry PowerON_Reset_PC
 void PowerON_Reset_PC(void)
 {
 ・・・
<stacksct.h>
 #pragma stacksize su=0x300
 #pragma stacksize si=0x100
```

【生成コード例】

```
 // -start=SU,SI/01000 と指定した場合
_PowerON_Reset_PC          MVTC        #00001300H,USP
                           MVTC        #00001400H,ISP
                           ・・・
```

2.5.2 RAM を初期化する

スタートアップ・ルーチンの esetprg.c ファイルの _INITSCT 関数内で未初期化領域の初期化を行います。初期化の対象となるセクションの追加は、dbsct.c ファイル内の下記に記述を追記することで可能です。

【記述例】

```
<dbsct.c>
  ・・・
  #pragma section C C$BSEC
  extern const struct {
     _UBYTE *b_s;   /* Start address of non-initialized data section */
     _UBYTE *b_e;   /* End address of non-initialized data section */
  } _BTBL[] = {
       { __sectop("B"), __secend("B") },
       { __sectop("B_2"), __secend("B_2") },
       { __sectop("B_1"), __secend("B_1") }
  };
  ・・・
```

上記では、B, B_2, B_1 **セクションを初期化するために** INITSCT **で使用するアドレスをテーブルに格納します。**

2. 5. 3 初期値あり変数を ROM から RAM へ転送する

スタートアップ・ルーチンの resetprg.c **ファイルの** _INITSCT **関数内で初期値あり変数を** ROM **から** RAM **への転送を行います。転送の対象となるセクションの追加は、**dbsct.**c ファイル内の下記に記述を追加することで可能です。**

【記述例】

```
<dbsct.c>
  ・・・
  #pragma section C C$DSEC
  extern const struct {
     _UBYTE *rom_s;      /* Start address of the initialized data section in ROM */
     _UBYTE *rom_e;      /* End address of the initialized data section in ROM */
     _UBYTE *ram_s;      /* Start address of the initialized data section in RAM */
  }   _DTBL[] = {
         { __sectop("D"), __secend("D"), __sectop("R") },
         { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
         { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
  };
  ・・・
```

上記では、D, D_2, D_1 **セクションを** R, R_2, R_1 **セクションへの転送するために** INITSCT **関数で使用するアドレスをテーブルに格納します。尚、**D, D_2, D_1 **と** R, R_2, R_1 **の配置アドレスは、最適化リンカの** -start **オプションで指定し、**ROM **から** RAM **への転送によるリロケーション解決は、最適化リンカの** -rom **オプションで指定します。**

2. 6 コードサイズの削減

この節では、コードサイズの削減について説明します。

2. 6. 1 データの構造

関連するデータを同一関数の中で何度も参照している場合、構造体を用いると相対アクセスを利用したコードが生成され易くなり、効率向上が期待できます。また、引数として渡す場合も効率が向上します。相対アクセスにはアクセス範囲に制限があるため、頻繁にアクセスするデータは構造体の先頭に集めると効果的です。

データを構造化すると、データの表現を変更するようなチューニングが容易になります。

【使用例】

変数 a, b, c **に数値を代入します。**

改善前ソースコード

```
int a, b, c;
void func()
{
        a = 1;
        b = 2;
        c = 3;
}
```

改善前アセンブリ展開コード

```
_ func:
        MOV.L #_a,R4
        MOV.L #00000001H,[R4]
        MOV.L #_b,R4
        MOV.L #00000002H,[R4]
        MOV.L #_c,R4
        MOV.L #00000003H,[R4]
        RTS
```

改善後ソースコード

```
struct s{
        int a;
        int b;
        int c;
} s1;
void func()
{
        register struct s *p=&s1;
        p->a = 1;
        p->b = 2;
        p->c = 3;
}
```

改善後アセンブリ展開コード

```
_func:
       MOV.L #_s1,R5
       MOV.L #00000001H,[R5]
       MOV.L #00000002H,04H[R5]
       MOV.L #00000003H,08H[R5]
       RTS
```

2. 6. 2 局所変数と大域変数

局所変数として使用できるものは、大域変数として宣言しないで必ず局所変数として宣言してください。大域変数は、関数呼び出しやポインタ操作によって値が変化してしまう可能性があるため、最適化の効率が悪くなります。

局所変数を使用すると次の利点があります。

a. アクセスコストが安い。

b. レジスタに割り付けられる可能性がある。

c. 最適化の効率が良い

【使用例】

一時変数に大域変数を使った場合 (改善前) と局所変数を使った場合 (改善後)

改善前ソースコード

```
int tmp;
void func(int* a, int* b)
{
        tmp = *a;
        *a = *b;
        *b = tmp;
}
```

改善前アセンブリ展開コード

```
__func:
        MOV.L #_tmp,R4
        MOV.L [R1],[R4]
        MOV.L [R2],[R1]
        MOV.L [R4],[R2]
        RTS
```

改善後ソースコード

```
void func(int* a, int* b)
{
       int tmp;
       tmp = *a;
       *a = *b;
       *b = tmp;
}
```

改善後アセンブリ展開コード

```
__func:
        MOV.L [R1],R5
        MOV.L [R2],[R1]
        MOV.L R5,[R2]
        RTS
```

2. 6. 3 構造体宣言のメンバオフセット

構造体メンバは、構造体アドレスにオフセットを加算してアクセスします。オフセットを小さくするとサイズが有利になるので、よく使用するメンバを先頭に宣言するようにしてください。

もっとも効果的なのは、signed char, unsigned char **型で先頭から** 32byte **未満** , short, unsigned short **型で先頭から** 64byte **未満** , int, unsigned, long, unsigned long **型で先頭から** 128byte **未満です。**

【使用例】

以下の例は、構造体のオフセットによってコードが変わる例を示します。

改善前ソースコード

```
struct str {
         long L1[8];
         char C1;
};
struct str STR1;
char x;
void func()
{
         x = STR1.C1;
}
```

改善前アセンブリ展開コード

```
_func:
       MOV.L #_STR1,R4
       MOVU.B 20H[R4],R5
       MOV.L #_x,R4
       MOV.B R5,[R4]
       RTS
```

改善後ソースコード

```
struct str {
         char C1;
         long L1[8];
};
struct str STR1;
char x;
void func()
{
         x = STR1.C1;
}
```

改善後アセンブリ展開コード

```
__func:
        MOV.L #_STR1,R4
        MOVU.B [R4],R5
        MOV.L #_x,R4
        MOV.B R5,[R4]
        RTS
```

■注意事項

構造体を定義する際には、境界調整数を意識してメンバの宣言をおこなってください。

構造体の境界調整数は構造体内の最も大きな境界調整値に合わせられ、構造体のサイズは境界調整数の倍数となります。その為、構造体の末尾が構造体自身の境界調整数と合わない場合に、次の境界調整を保証するために生成される、未使用領域もサイズに含めてしまいます。

改善前ソースコード

```
/* 最大メンバが int 型の為、境界調整数は 4 */
struct str {
         char C1; /* 1byte + 境界調整分 3byte */
         long L1; /* 4byte */
         char C2; /* 1byte */
         char C3; /* 1byte */
         char C4; /* 1byte + 境界調整分 1byte */
}STR1;
```

改善前 str サイズ

```
       .SECTION B,DATA,ALIGN=4
       .glb _STR1
_STR1:                          ; static: STR1
       .blkl 3
```

改善後ソースコード

```
/* 最大メンバが int 型の為、境界調整数は 4 */
struct str {
         char C1; /* 1byte */
         char C2; /* 1byte */
         char C3; /* 1byte */
         char C4; /* 1byte */
         long L1; /* 4byte */
}STR1;
```

改善後 str サイズ

```
       .SECTION B,DATA,ALIGN=4
       .glb _STR1
_STR1:                          ; static: STR1
       .blkl 2
```

2.6.4 ビットフィールドの割り付け

異なるビットフィールドのメンバを設定するためには、そのたびにビットフィールドを含むデータにアクセスしなければなりません。関連するビットフィールドを同じ構造体内にまとめて割り付けることによって、このアクセスを一度で済ますことができます。

【使用例】

同じ構造体に関連するビットフィールドを割り付けることによってサイズが改善する例を示します。

改善前ソースコード

```
struct str
{
        Int flag1:1;
}b1,b2,b3;
void func()
{
        b1.flag1 = 1;
        b2.flag1 = 1;
        b3.flag1 = 1;
}
```

改善前アセンブリ展開コード

```
_func:
       MOV.L #_b1,R5
       BSET #00H,[R5]
       MOV.L #_b2,R5
       BSET #00H,[R5]
       MOV.L #_b3,R5
       BSET #00H,[R5]
       RTS
```

改善後ソースコード

```
struct str
{
          int flag1:1;
          int flag2:1;
          int flag3:1;
}a1;
void func()
{
          a1.flag1 = 1;
          a1.flag2 = 1;
          a1.flag3 = 1;
}
```

改善後アセンブリ展開コード

```
__func:
        MOV.L #_a1,R4
        MOVU.B [R4],R5
        OR #07H,R5
        MOV.B R5,[R4]
        RTS
```

2.6.5 ベースレジスタ指定時の外部変数アクセス最適化

RAM セクションのベースレジスタに R13 を指定した場合、RAM セクションへのアクセスが、R13 レジスタ相対となります。更にモジュール間の外部変数アクセス最適化を有効にした場合、R13 レジスタ相対値が最適化されて、8bit 範囲以下の値であれば、命令サイズが小さくなる場合があります。

【使用例】

改善前ソースコード

```
int a;
int b;
int c;
int d;
void fu{
        a=0;
        b=1;
        c=2;
        d=3;
}
```

改善前アセンブリ展開コード

```
_func:
        MOV.L #_a,R4
        MOV.L #00000000H,[R4]
        MOV.L #_b,R4
        MOV.L #00000001H,{R4}
        MOV.L #_c,R4
        MOV.L #00000002H,[R4]
        MOV.L #_d,[R4]
        MOV.L #00000003H,[R4]
        RTS
```

改善後ソースコード

```
int a;
int b;
int c;
int d;
void fu{
        a=0;
        b=1;
        c=2;
        d=3;
}
```

改善後アセンブリ展開コード

```
__func:
         MOV.L #0000000H,_a-__RAM_TOP:16[R13]
         MOV.L #0000001H,_b-__RAM_TOP:16[R13]
         MOV.L #0000002H,_c-__RAM_TOP:16[R13]
         MOV.L #0000003H,_d-__RAM_TOP:16[R13]
         RTS
```

2. 6. 6 外部変数アクセス最適化時のリンカのセクションアドレス指定順

レジスタ相対形式でメモリにアクセスする命令では、ディスプレースメント値が小さいほうが、命令サイズが小さくなります。

以下の指標を参考にリンカでのセクション割り付け順を変更するとコードサイズを改善できる場合があります。

- 関数内でのアクセス回数の多い外部変数のセクションを前にする。
- 型サイズの小さい外部変数のセクションを前にする。

但し、外部変数アクセス最適化は、コンパイラが 2 度実行されるのでビルド時間は長くなります。

【使用例】

改善前ソースコード

```
/* D_1 セクション */
char d11=0, d12=0, d13=0, d14=0;
/* D_2 セクション */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* D セクション */
int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0}

void func(int a){
        d11 = a;
        d12 = a;
        d13 = a;
        d14 = a;
        d21 = a;
        d22 = a;
        d23 = a;
        d24 = a;
        d41 = a;
        d42 = a;
        d43 = a;
        d44 = a;
}
```

改善前アセンブリ展開コード

```
< セクションの割り付け順を D, D_2, D_1  または D* とした場合 >
_func:
       MOV.L #d41,R4
       MOV.B R1,0120H[R4]
       MOV.B R1,0121H[R4]
       MOV.B R1,0122H[R4]
       MOV.B R1,0123H[R4]
       MOV.W R1,0100H[R4]
       MOV.W R1,0102H[R4]
       MOV.W R1,0104H[R4]
       MOV.W R1,0106H[R4]
       MOV.L R1,[R4]
       MOV.L R1,04H[R4]
       MOV.L R1,08H[R4]
       MOV.L R1,0CH[R4]
       RTS
```

改善後ソースコード

```
/* D_1 セクション */
char d11=0, d12=0, d13=0, d14=0;
/* D_2 セクション */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* D セクション */
int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0}

void func(int a){
        d11 = a;
        d12 = a;
        d13 = a;
        d14 = a;
        d21 = a;
        d22 = a;
        d23 = a;
        d24 = a;
        d41 = a;
        d42 = a;
        d43 = a;
        d44 = a;
}
```

改善後アセンブリ展開コード

```
<セクションの割り付け順を D_1, D_2, D  または D* とした場合>
_func:
        MOV.L #d11,R4
        MOV.B R1,[R4]
        MOV.B R1,01H[R4]
        MOV.B R1,02H[R4]
        MOV.B R1,03H[R4]
        MOV.W R1,04H[R4]
        MOV.W R1,06H[R4]
        MOV.W R1,08H[R4]
        MOV.W R1,0AH[R4]
        MOV.L R1,24H[R4]
        MOV.L R1,28H[R4]
        MOV.L R1,2CH[R4]
        MOV.L R1,30H[R4]
        RTS
```

2.6.7 関数のモジュール化

異なるファイルにある関数を呼び出す場合、4 バイトの BSR 命令に展開されますが、同一ファイル内の関数呼び出しでは、呼び出し範囲が近いと 3 バイトの BSR 命令に展開され、コンパクトなオブジェクトが生成されます。

また、モジュール化によって、チューンアップ時の修正が容易になります。

【使用方法】

関数 f から関数 g を呼び出します。

改善前ソースコード

```
extern void sub(void);
int func()
{
        sub();
        return(0);
}
```

改善前アセンブリ展開コード

```
_func:
        BSR   _sub ;length A
        MOV.L #00000000H,R1
        RTS
```

改善後ソースコード

```
void sub(void);
int func()
{
         sub();
         return(0);
}
```

改善後アセンブリ展開コード

```
_func:
        BSR   _sub ;length W
        MOV.L #00000000H,R1
        RTS
```

2.6.8 割り込み

割り込み処理の前後では多くのレジスタ退避・回復が発生し、期待する割り込み応答時間を得られない場合があります。その場合は、高速割り込み指定（fint）と fint_register オプションを利用することで、レジスタの退避・回復を抑えることができ、割り込み応答時間の短縮を図ることができます。

ただし、fint_register オプションを利用すると他の関数での使用可能なレジスタが減るため、プログラム全体での効率が低下する場合があるのでご注意ください。

【使用例】

改善前ソースコード

```
#pragma interrupt int_func
volatile int count;

void  int_func()
{
        count++;
}
```

改善前アセンブリ展開コード

```
_int_func:
          PUSHM R4-R5
          MOV.L #_count,R4
          MOV.L [R4],R5
          ADD #01H,R5
          MOV.L R5,[R4]
          POPM R4-R5
          RTE
```

改善後ソースコード

```
#pragma interrupt int_func(fint)
volatile int count;

void  int_func()
{
        count++;
}
```

改善後アセンブリ展開コード

```
<fint_register=2 オプション指定時 >
_int_func:
           MOV.L #_count,R12
           MOV.L [R12],[R13]
           ADD #01H,R13
           MOV.L R13,[R12]
           RTFI
```

2. 7 処理の高速化

この節では，処理の高速化について説明します。

2. 7. 1 ループ制御変数

ループ終了条件の判定で、サイズの違いによりループ制御変数がその比較対象のデータを表現できない可能性のある場合は、ループ展開最適化がかかりません。たとえばループ制御変数が signed char で比較対象のデータが signed long の場合はループ展開最適化がかかりません。

そのため、signed char, signed short に比べ、signed long の方がループ展開最適化を適用し易くなります。ループ展開最適化を活用したい場合はループ制御変数を 4 バイト整数型としてください。

【使用例】

改善前ソースコード

```
signed long array_size=16;
signed char array[16];

void func()
{
        signed char I;
        for(i=0;i<array_size;i++)
        {
                    array[i]=0;
        }
}
```

改善前アセンブリ展開コード

```
<loop=2 指定時 >
_func:
       MOV.L #_array_size,R4
       MOV.L [R4],R2
       MOV.L #00000000H,R5
       BRA L11
L12:
       MOV.L #_array,R14
       MOV.L #00000000H,R3
       MOV.B R3,[R5,R4]
       ADD #01H,R5
L11:
       MOV.B R5,R5
       CMP R2,R5
       BLT L12
L13:
       RTS
```

改善後ソースコード

```
signed long array_size=16;
signed char array[16];

void func()
{
        signed long I;
        for(i=0;i<array_size;i++)
        {
                    array[i]=0;
        }
}
```

改善後アセンブリ展開コード

```
<loop=2 指定時 >
_func:
        MOV.L #_array_size,R 5
        MOV.L [R5],R2
        MOV.L #00000000H,R 4
        ADD #0FFFFFFFFH,R2,R3
        CMP R3,R2
        BLE L12
L11:
        MOV.L #_array,R1
        MOV.L R1,R5
        BRA L13
L14:
        MOV.W #0000H,[R5]
        ADD #02H,R5
        ADD #02H,R4
L13:
        CMP R3,R4
        BLT L14
L15:
        CMP R2,R4
        BGE L17 L16:
        MOV.L #00000000H,R5
        MOV.B R5,[R4,R1]
        RTS
L12:
        MOV.L #_array,R5
        MOV.L #00000000H,R3
L19:
        CMPR2,R4
        BGE L17
L20:
        MOV.B R3,[R5+]
        ADD #01H,R4
        BRA L19
L17:
        RTS
```

2. 7. 2 関数のインタフェース

引数がすべてレジスタに乗るように (4 個まで) 引数の数を厳選してください。引数が多い場合は、構造体にしてポインタで渡してください。もし、構造体のポインタではなく、構造体そのものを受け渡すとレジスタに乗らない場合があります。引数がレジスタに乗れば、呼び出し、関数の出入り口の処理が簡単になります。またスタック領域も節約できます。なお、レジスタは R1 ～ R4 が引数用です。

【使用方法】

関数 f の引数が引数用レジスタ個数よりも 4 個多くあります。

改善前ソースコード

```
void call_func()
{
        func(1,2,3,4,5,6,7,8);
}
```

改善前アセンブリ展開コード

```
_call_func:
            SUB #04H,R0
            MOV.L #08070605H,[R0]
            MOV.L #00000004H,R4
            MOV.L #00000003H,R3
            MOV.L #00000002H,R2
            MOV.L #00000001H,R1
            BSR _func
            ADD #04H,R0
            RTS
```

改善後ソースコード

```
struct str{
         char a;
         char b;
         char c;
         char d;
         char e;
         char f;
         char g;
         char h;
};
struct str arg = {1,2,3,4,5,6,7,8};

void call_func()
{
          func(&arg);
}
```

改善後アセンブリ展開コード

```
_call_func:
         MOV.L #arg,R1
         BRA _func
```

2. 7. 3 ループ回数の削減

ループの展開は特に内側のループが有効です。ループの展開によりプログラムサイズは増大するので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

【使用例】

配列 a[] **を初期化します。**

改善前ソースコード

```
extern int a[100];
void func()
{
        int I;
        for( i = 0 ; i < 100 ; i++ ){
                    a[i] = 0;
        }
}
```

改善前アセンブリ展開コード

```
_func:
       MOV.L #00000064H,R4
       MOV.L #_a,R5
       MOV.L #00000000H,R3
L11:
       MOV.L R3,[R5+]
       SUB #01H,R4
       BNE L11
L12:
       RTS
```

改善後ソースコード

```
extern int a[100];
void func()
{
        int I;
        for( i = 0 ; i < 100 ; i+=2 )
        {
             a[i] = 0;
             a[i+1] = 0;
        }
}
```

改善後アセンブリ展開コード

```
_func:
        MOV.L #00000032H,R4
        MOV.L #_a,R5
L11:
        MOV.L #00000000H,[R5]
        MOV.L #00000000H,04H[R5]
        ADD #08H,R5
        SUB #01H,R4
        BNE L11
L12:
        RTS
```

2. 7. 4 テーブルの活用

switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。

【使用方法】

変数 i の値により変数 ch に代入する文字定数を変えます。

改善前ソースコード

```
char func(int i)
{
        char ch;
        switch (i) {
                case 0:
                        ch = ‘a’; break;
                case 1:
                        ch = ‘x’; break;
                case 2:
                        ch = ‘b’; break;
        }
        return(ch);
}
```

改善前アセンブリ展開コード

```
_func:
         CMP #00H,R1
         BEQ L17
L16:
         CMP #01H,R1
         BEQ L19
         CMP #02H,R1
         BEQ L20
         BRA L21
L12:
L17:
         MOV.L #00000061H,R1
         BRA L21
L13:
L19:
         MOV.L #00000078H,R1
         BRA L21
L14:
L20:
         MOV.L #00000062H,R1
L11:
L21:
         MOVU.B R1,R1
         RTS
```

改善後ソースコード

```
char chbuf[] = { ‘a’, ‘x’, ‘b’};

char func(int i)
{
        return (chbuf[i]);
}
```

改善後アセンブリ展開コード

```
_f
        MOV.L #_chbuf,R4
        MOVU.B [R1,R4],R1
        RTS
```

2.7.5 分岐

else if 文のように上から順に比較をする場合、場合分けが増えると末端のケースの実行速度は低下します。頻繁に分岐するケースは先頭近くに配置してください。

【使用例】

引数の値によりリターン値を変えます。

改善前ソースコード

```
int func(int a)
{
        if (a==1)
                a = 2;
        else if (a==2)
                a = 4;
        else if (a==3)
                a = 0;
        else
                a = 0;
        return(a);
}
```

改善前アセンブリ展開コード

```
_func:
       CMP #01H,R1
       BEQ L11
L12:
       CMP #02H,R1
       BNE L14
L13:
       MOV.L #00000004H,R1
       RTS
L14:
       CMP #03,R1
       BNE L17
L16:
       MOV.L #00000008H,R1
       RTS
L17:
       MOV.L #00000000H,R1
       RTS
L11:
       MOV.L #00000002H,R1
       RTS
```

改善後ソースコード

```
int func(int a)
{
       if (a==3)
               a = 8;
       else if (a==2)
               a = 4;
       else if (a==1)
               a = 2;
       else
               a = 0;
       return (a);
}
```

改善後アセンブリ展開コード

```
_func:
        CMP #03H,R1
        BEQ L11
L12:
        CMP #02H,R1
        BNE L14
L13:
        MOV.L #00000004H,R1
        RTS
L14:
        CMP #01H,R1
        BNE L17
L16:
        MOV.L #00000002H,R1
        RTS
L17:
        MOV.L #00000000H,R1
        RTS
L11:
        MOV.L #00000008H,R1
        RTS
```

2.7.6 インライン展開

頻繁によびだされる関数をインライン展開することにより、実行速度の向上が図れます。特にループ内で呼ばれる関数などを展開すると大きな効果を得られる場合もあります。しかし、インライン展開をした場合、プログラムサイズが増大する傾向にありますので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

【使用例】

配列 a と配列 b の要素を交換します。

改善前ソースコード

```
int x[10], y[10];
static void sub(int *a, int *b, int I)
{
        int temp;
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
}

void func()
{
        int I;
        for(i=0;i<10;i++)
        {
                sub(x,y,i);
        }
}
```

改善前アセンブリ展開コード

```
__$sub:
        SHLL #02H,R3
        ADD R3,R1
        MOV.L [R1],R5
        ADD R3,R2
        MOV.L [R2],[R1]
        MOV.L R5,[R2]
        RTS
_func:
        PUSHM R6-R8
        MOV.L #00000000H,R6
        MOV.L #_x,R7
        MOV.L #_y,R8
L12:
        MOV.L R6,R3
        MOV.L R7,R1
        MOV.L R8,R2
        ADD #01H,R6
        BSR __$sub
        CMP #0AH,R6
        BLT L12
L13:
        RTSD #0CH,R6-R8
```

改善後ソースコード

```
int x[10], y[10];
#pragma inline(sub)
static void sub(int *a, int *b, int I)
{
         int temp;
         temp = a[i];
         a[i] = b[i];
         b[i] = temp;
}

void func()
{
         int I;
         for(i=0;i<10;i++)
         {
                   sub(x,y,i);
         }
}
```

改善後アセンブリ展開コード

```
;  インライン展開により
; _sub のコードが削減されている
_func:
        MOV.L #0000000AH,R1
        MOV.L #_y,R2
        MOV.L #_x,R3
L11:
        MOV.L [R3],R4
        MOV.L [R2],R5
        MOV.L R4,[R2+]
        MOV.L R5,[R3+]
        SUB #01H,R1
        BNE L11
L12:
        RTS
```

2.8 コンパイラとアセンブラの相互参照

この節では、コンパイラとアセンブラの相互参照について説明します。

C/C++ プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。コンパイラは、次のものを外部名として扱います。

- 大域変数であって、かつ static 記憶クラスでないもの (C/C++ プログラム)
- extern 記憶クラスで宣言されている変数名 (C/C++ プログラム)
- static 記憶クラスを指定されていない関数名 (C プログラム)

- static 記憶クラスを指定されてない非メンバ非インライン関数名 (C++ プログラム)
- 非インラインメンバ関数名 (C++ プログラム)
- 静的データメンバ名 (C++ プログラム)

2. 8. 1 アセンブリプログラムの外部名を C/C++ プログラムで参照

アセンブリプログラムでは、.EXPORT を用いてシンボル名 (先頭に下線 "_" を付与) を外部定義宣言します。

C/C++ プログラムでは、シンボル名 (先頭に下線 "_" がない) を「extern」宣言します。

例 アセンブリソース

```
            .glb _a, _b
            .SECTION D,ROMDATA,ALIGN=4
_a:         .LWORD 1
_b:         .LWORD 1
            .END
```

例 C ソース

```
extern int a,b;
void f()
{
        a+=b;
}
```

2. 8. 2 C/C++ プログラムの外部 (変数および C 関数) 名をアセンブリプログラムで参照

C/C++ プログラムでは、変数名 (先頭に下線 "_" がない) を外部定義します。

アセンブリプログラムでは、.IMPORT を用いて外部名 (先頭に下線 "_" を付与) を外部参照宣言します。

例 C ソース

```
int a;
```

例 アセンブリソース

```
        .GLB _a
        .SECTION P,CODE
        MOV.L #A_a,R1
        MOV.L [R1],R2
        ADD #1,R2
        MOV.L R2,[R1]
        RTS
        .SECTION D,ROMDATA,ALIGN=4
A_a:    .LWORD _a
        .END
```

2. 8. 3 C++ プログラムの外部 (関数) 名をアセンブリプログラムで参照

アセンブリプログラムで参照する関数を「extern "C"」を用いて宣言することにより、(2) と同じ規則で参照できます。ただし、「extern "C"」を用いて宣言した関数は多重定義できません。

例 C ソース

```
extern "C"
void sub()
{
        :
}
```

例 アセンブリソース

```
.GLB _sub
.SECTION P,CODE
  :
PUSH.L R13
MOV.L 4[R0],R1
MOV.L R3,R12
MOV.L #_sub,R14
JSR R14
POP R13
RTS
  :
.END
```

第 3 章　コンパイラ言語仕様

3. 1　基本言語仕様

RXC は，ANSI 規格で規定された言語仕様をサポートしていますが、その中には処理系定義として規定されている項目があります。ここでは、RX マイクロコントローラの処理系に依存した項目の言語仕様について説明します。

なお，RXC で独自に追加されている拡張言語仕様については，「3. 2　拡張言語仕様」を参照してください。

3. 1. 1　未規定の動作

この項では、ANSI 規格における未規定の動作項目について説明します。

(1) 実行環境ー 静的記憶域の初期化

静的データは，コンパイル時にデータ・セクションとして出力されます。

(2) 文字表示の意味ー後退（\b），水平タブ（\t），垂直タブ（\v）

表示装置設計依存となります。

(3) 型ー浮動小数点

IEEE754 注準拠です。

注　IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。

また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

(4) 式ー評価順序

基本的には式は前方より評価します。ただし，最適化を行った場合は未規定とします。オプションなどにより，順序が変更になることがあるので，副作用のある式の記述は行わないでください。

(5) 関数呼び出しー引数の評価順序

基本的には第一引数（先頭の引数）より順に評価します。ただし，最適化を行った場合は未規定とします。オプションなどにより，順序が変更になることがあるので，副作用のある式の記述は行わないでください。

(6) 構造体指定子，および共用体指定子

ビットフィールドの型の整列境界をまたがないように調整します。オプションや #pragma 指令でパッキングを行った場合は，整列境界調整は行わず，ビットフィールドは詰めて配置されます。

(7) 関数定義ー 仮引数の記憶域

スタック，およびレジスタに割り付けます。詳細は，「3. 4　関数呼び出しインタフェース」を参照してください。

(8) # 演算子

前方より評価します。

3. 1. 2　未定義の動作

この項では、ANSI 規格における未定義の動作項目について説明します。

(1) 文字集合

ソースファイル中に文字集合に定められた文字以外がある場合，メッセージを出力します。

(2) 字句要素

文字 “ ' ”，または文字 “ " ” がその最後の分類（区切り子，および字句的に他の前処理字句の種類に一致しない単一の非空白類文字）に入る場合，メッセージを出力します。

(3) 識別子

識別子全文字を意味がある文字とするため，意味のない文字は存在しません。

(4) 識別子の結合

翻訳単位の中で同じ識別子が内部結合と外部結合の両方で現れた場合，メッセージを出力します。

(5) 適合型と合成型

同じオブジェクト，または関数を参照するすべての宣言は，適合しなければなりません。それ以外の場合，メッセージを出力します。

(6) 文字定数

特定の非図形文字は，\ に続く英小文字から構成する拡張表記 \a，\b，\f，\n，\r，\t，および \v によって表現できます。その他の拡張表記はもたず，\ に続く文字は，その文字自身とします。

(7) 文字列リテラルー結合

単純文字列リテラルとワイド文字列リテラル字句が隣り合うとき，単純に文字列結合を行います。

(8) 文字列リテラルー変更

文字列リテラルの変更はユーザ責任となります。RAM に配置した場合は変更されますが，ROM に配置された場合は変更されません。

(9) ヘッダ名

文字，'，"，//，または /* が，区切り記号＜と＞の間の文字列中，または二つの区切り記号 " の文字列中に現れた場合は，そのままファイル名として扱います。\ 文字はフォルダ区切りとして扱います。

(10) 浮動小数点型と汎整数型

浮動小数点型の値を汎整数型に型変換する場合，整数部の値が汎整数型で表現できなければ，汎整数型で表現できる値に切りつめを行います。

(11) 左辺値及び関数指示子

不完全型が左辺値となった場合は，メッセージを出力します。

(12) 関数呼び出しー引数の個数

実引数の数が少ない場合，仮引数は不定値となります。実引数が多い場合，余剰な実引数はないものとして関数が実行され，実引数は意味を持ちません。

関数呼び出しに先立ち，関数宣言がある場合は，メッセージを出力します。

(13) 関数呼び出しー拡張後の引数の型

関数原型を含まない形で関数を定義し，かつ拡張後の実引数の型が，拡張後の仮引数の型と一致しない場合，仮引数は不定値となります。

(14) 関数呼び出しー適合しない型

呼び出される関数を表す式によって指される型と適合しない型で関数が定義されている場合，関数の戻り値は不正な値となります。

(15) 関数宣言ー適合しない型

関数原型を含む型で関数を定義し，かつ拡張後の実引数の型が仮引数の型と適合しない場合，または関数原型が省略記号で終わっている場合，仮引数の型として解釈されます。

(16) アドレス，および間接演算子

正しくない値がポインタに代入されている場合の，単項 * 演算子の動作は，ハードウエア設計，および正しくない値の内容により，不定な値を取るか，不正なアクセスとなります。

(17) キャスト演算子ー関数ポインタのキャスト

型変換されたポインタが元の型以外の関数を呼び出すために使われた場合，関数を呼び出すことは可能です。引数，戻り値が不適合な場合は，不正となります。

(18) キャスト演算子ー汎整数型のキャスト

ポインタを汎整数型にキャストした場合で，領域の大きさが不十分な場合は，キャストした型の領域の大きさに切り詰められます。

(19) 乗除演算子

コンパイル中に 0 による除算剰余算を検出した場合，メッセージを出力します。

実行時は，0 除算例外が発生します。エラー処理ルーチンを記述した場合は，それに従います。

(20) 加減演算子ー配列以外のポインタ

配列オブジェクトの要素を指すかのように動作するもの以外のポインタに対して，加算，または減算を行っている場合，あたかも配列の要素を指しているように振舞います。

(21) 加減演算子ー別な配列へのポインタ減算

同じ配列オブジェクトの中を指すかのように動作するもの以外の二つのポインタに対し，減算を行なっている場合，あたかも配列の要素を指しているように振舞います。

(22) ビット単位のシフト演算子

右オペランドの値が負であるか，または拡張した左オペランドのビット幅以上の場合，左オペランドのビット幅で右オペランドをマスクした値でシフトした値となります。

(23) 関係演算子ーポインタ

比較対象のポインタで指されているオブジェクトが同一の集成体オブジェクト，または共用体オブジェクトのメンバでない場合，あたかも同一のオブジェクトを指しているポインタ同士の関係演算として動作します。

(24) 単純代入

オブジェクトに格納されている値が，何らかの形でそのオブジェクトの記憶域に重なる他のオブジェクトを通してアクセスされる場合，重なりは完全に一致していなければなりません。さらに，二つのオブジェクトの型は，適合する型の修飾版，または非修飾版でなければなりません。一致しない重なりの代入は，代入によって代入元の値が破壊されます。

(25) 構造体指定子及び共用体指定子

メンバ宣言並びが名前付のメンバを含まない場合，意味を持たない旨の警告メッセージを出力します。ただし，-Xansi オプションを指定した場合は，同一メッセージでエラーとします。

(26) 型修飾子ー const

メンバ宣言並びが名前付のメンバを含まない場合，意味を持たない旨の警告メッセージを出力します。

(27) 型修飾子ー volatile

volatile 修飾型で定義されたオブジェクトを，非 volatile 修飾型の左辺値を使って変更しようとした場合，メッセージを出力します。

(28) return 文

式を持たない return 文を実行し，呼び出し元で関数呼び出しの値を使用している場合で，宣言がある場合はメッセージを出力します。宣言がない場合は，関数の戻り値が不定値となります。

(29) 関数定義

可変個引数の実引数を受け付ける関数が，省略記号表記で終わる仮引数型並びをもたずに定義された場合，仮引数の値が不定となります。

(30) 条件付取り込み

置き換え処理によって字句 defined が生成される場合，または defined 単項演算子のマクロ置き換え前の使用法が制約の中で規定した二つの形式のどちらにも一致しない場合，通常の defined として扱います。

(31) マクロ置き換え－前処理句を含まない実引数

実引数が（実引数の置換前に）前処理句を含まない場合，メッセージを出力します。

(32) マクロ置き換え－前処理指令を持つ実引数

実引数の並びの中に，ほかの場合であれば前処理指令として働く前処理字句列がある場合，メッセージを出力します。

(33) # 演算子

置き換えの結果が，正しい単純文字列リテラルでない場合，メッセージを出力します。

(34) ## 演算子

置き換えの結果が，正しい単純文字列リテラルでない場合，メッセージを出力します。

3.1.3 処理系依存

言語仕様で規定していない処理系定義項目について、コンパイラの仕様を示します

(1) 環境

表 3―1 環境の仕様

	項目	コンパイラの仕様
1	main 関数への実引数の意味	規定しません。
2	対話的入出力装置の構成	規定しません。

(2) 識別子

表 3―2 識別子の仕様

	項目	コンパイラの仕様
1	外部結合とならない識別子（内部名）の有効文字数	8189 文字まで有効です。
2	外部結合となる識別子（外部名）の有効文字数	8191 文字まで有効です。
3	外部結合となる識別子（外部名）の大文字と小文字の区別	大文字と小文字を区別します。

(3) 文字

表 3―3 文字の仕様

	項目	コンパイラの仕様
1	ソース文字集合および実行環境文字集合の要素	どちらも ASCII 文字集合です。ただし、文字列、文字定数にはシフト JIS、EUC 漢字コード、Latin1 コードまたは UTF-8 コードを記述できます。
2	多バイト文字のコード化で使用されるシフト状態	シフト状態はサポートしていません。
3	プログラム実行時の文字集合の文字のビット数	ビット数は 8 ビットです。
4	文字定数内、文字列内のソース文字集合の文字と実行環境文字集合の文字との対応付け	同じ ASCII 文字に対応します。
5	言語で規定していない文字や拡張表記を含む整数文字定数の値	言語で規定する以外の文字、拡張表記はサポートしていません。
6	2 文字異常の文字を含む文字定数または 2 文字異常の多バイト文字を含む広角文字定数の値	文字定数は上位 2 バイトを有効とします。広角文字定数はサポートしていません。また、1 文字より多く指定した場合はウォーニングエラーを出力します。
7	多バイト文字を広角文字に変換するために使用される locale の仕様	locale はサポートしていません。
8	char 型の値	unsigned char 型と同じ値の範囲を持ちます。*1

注 1. signed_char オプションを指定した場合、signed char 型と同じ値の範囲を持ちます。

(4) 整数

表 3―4 整数の仕様

	項目	コンパイラの仕様
1	整数型の表現方法とその値	表 9.5 に示します。
2	整数の値がより短いサイズの符号付き整数型、または符号なし整数型を同一のサイズの符号付き整数型に変換したときの値（結果の値が変換先の型で表現できない場合）	整数の値の下位 4 バイト（変換後の型のサイズが 4 バイトの場合）、下位 2 バイト（変換後の型のサイズが 2 バイトの場合）あるいは下位 1 バイト（変換後の型のサイズが 1 バイトの場合）が変換後の値となります。
3	符号付き整数に対するビットごとの演算の結果	符号付きの値になります。
4	整数除算における剰余の符号	被除数の符号と同符号になります。
5	負の値を持つ符号付きスカラ型の右シフトの結果	符号ビットを保持します。

表 3―5 整数型とその値の範囲

	型	値の範囲	データサイズ
1	char*1	0 ～ 255	1 バイト
2	signed char	-128 ～ 127	1 バイト
3	unsigned char	0 ～ 255	1 バイト
4	short signed short	-32768 ～ 32767	2 バイト
5	unsigned short	0 ～ 65535	2 バイト
6	int*2 signed int*2	-2147483648 ～ 2147483647	4 バイト
7	unsigned int*2	0 ～ 4294967295	4 バイト
8	long signed long	-2147483648 ～ 2147483647	4 バイト
9	unsigned long	0 ～ 4294967295	4 バイト
10	long long signed long long	-9223372036854775808 ～ 9223372036854775807	8 バイト
11	unsigned long long	0 ～ 18446744073709551615	8 バイト

注 1. signed_char オプションを指定した場合、signed char 型として扱います。

2. int_to_short オプションを指定した場合、int 型は short 型、signed int 型は signed short 型、unsigned int 型は unsigned short 型としてそれぞれ扱います。

(5) 浮動小数点

表 3―6 浮動小数点の仕様

	項目	コンパイラの仕様
1	浮動小数点の表現方法とその値	浮動小数点型には、float 型、double 型と long double 型があります。浮動小数点型の内部表現や変換仕様、演算仕様等の性質は「3.1.4 (5) 浮動小数点型の仕様」で説明します。表 3.7 に、浮動小数点型の表現可能な値の限界値を示します。
2	整数を本来の値に正確に表現することができない浮動小数点型に変換したときの切り捨て方向	
3	浮動小数点型をより狭い浮動小数点型に変換したときの切り捨てまたは丸め方法	

表 3―7 浮動小数点型の限界値

	項目			限界値	
				10 進数表現 *1	内部表現（16 進数）
1	float 型の最大値			3.4028235677973364e+38f (3.4028234663852886e+38f)	7f7fffff
2	float 型の正の最小値			7.0064923216240862e-46f (1.4012984643248171e-45f)	00000001
3	double*2 long double*2	}	型の最大値	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffffff
4	double*2 long double*2	}	型の正の最小値	4.9406564584124655e-324 (4.9406564584124654e-324)	0000000000000001

注 1. 10 進数表現の限界値は 0 または無限大にならない限界値です。また、( ) 内は理論値を示します。

2. dbl_size=8 を指定した場合の解釈です。dbl_size=4 を指定した場合、double 型および long double 型は float 型と同じ値となります。

(6) 配列とポインタ

表 3―8 配列とポインタの仕様

	項目	コンパイラの仕様
1	配列の大きさの最大値を保持するために必要な整数の型 (size_t)	unsigned long 型
2	ポインタ型から整数型への変換 （ポインタ型のサイズ ≥ 整数型のサイズ）	ポインタ型の下位バイトの値になります。
3	ポインタ型から整数型への変換 （ポインタ型のサイズ < 整数型のサイズ）	ゼロ拡張します。
4	整数型からポインタ型への変換 （整数型のサイズ ≥ ポインタ型のサイズ）	整数型の下位バイトの値となります。
5	整数型からポインタ型への変換 （整数型のサイズ < ポインタ型のサイズ）	符号拡張します。
6	同じ配列内のメンバのポインタ間の差を保持するために必要な整数の方 (ptrdiff_t)	int 型

(7) レジスタ

表 3―9 レジスタの仕様

	項目	コンパイラの仕様
1	レジスタに割り付けることができる変数の型	char, signed char, unsigned char, bool, _Bool, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, float, ポインタ

(8) クラス、構造体、共用体、列挙型、ビットフィールド

表 3―10 クラス、構造体、共用体、列挙型、ビットフィールドの仕様

	項目	コンパイラの仕様
1	異なる型のメンバでアクセスされる共用体型のメンバ参照	参照はできますが、値は保証しません。
2	クラス・構造体メンバのアライメント	クラス・構造体メンバ中のアライメント数の最大値がそのクラス・構造体のアライメント数になります。「9.1.2(2) 構造体 / 共用体、クラス型」を参照してください。
3	単なる int 型のビットフィールドの符号	unsigned int 型 *3
4	int 型のサイズ内のビットフィールドの割り付け順序	下位ビットから割り付けます。*1*2
5	int 型のサイズ内にビットフィールドが割り付けられているとき、次に割り付けるビットフィールドのサイズが int 型内の残っているサイズを超えたときの割り付け方	次の int 型の領域に割り付けます。*1
6	ビットフィールドで許される型指定子	char、unsigned char、bool、_Bool、short、unsigned short、int、unsigned int、long、unsigned long、enum、long long、unsigned long long
7	列挙型の値を表現する整数型	int 型です。*4

注 1. ビットフィールドの割り付け方の詳細については、「3. 1. 4 データの内部表現と領域」を参照してください。

2. bit_order=left オプションを指定した場合、上位ビットから割り付けられます。
3. signed_bitfield オプションを指定した場合、signed int 型となります。
4. auto_enum オプションを指定した場合、列挙値が収まる最小の型となります。詳細は、ビルド編の auto_enum オプションの説明を参照ください。

(9) 型修飾子

表 3―11 型修飾子の仕様

	項目	コンパイラの仕様
1	volatile 修飾したデータへのアクセスの種類	規定しません。

(10) 宣言

表 3―12 宣言の仕様

	項目	コンパイラの仕様
1	基本型（算術型、構造体型、共用体型）を修飾する宣言子の数	16 個まで指定できます。

基本型を修飾する型の数の数え方を、以下に例を用いて示します。

例

(i) int a; a は int 型（基本型）であり、基本型を修飾する型の数は 0 個です。

(ii) char *f(); f は char 型（基本型）へのポインタ型を返す関数型であり、基本型を修飾する型の数は 2 個です。

(11) 文

表 3―13 文の仕様

	項目	コンパイラの仕様
1	一つの switch 文中で指定できる case ラベルの数	2147483646 個まで指定できます。

(12) プリプロセッサ

表 3―14 プリプロセッサの仕様

	項目	コンパイラの仕様
1	条件コンパイルの定数式内の単一文字の文字定数と実行環境文字集合の対応	プリプロセッサ文の文字定数と実行環境文字集合は一致します。
2	インクルードファイルの読み込み方法	「<」、「>」で囲まれたファイルは include オプションで指定されたパスから読み込みます。 ファイルが見つからない場合、環境変数 INC_RX 指定フォルダ、環境変数 BIN_RX 指定フォルダの順序で各フォルダを検索します。
3	二重引用符で囲まれたインクルードファイルのサポートの有無	サポートします。インクルードファイルをカレントフォルダから読み込みます。カレントフォルダになければ、本表 2 項の読み込み方法に従います。
4	ソースファイルの文字の並びの対応（マクロ展開後の文字列の空白文字）	空白文字列は、空白文字 1 文字として展開します。
5	#pragma の動作	「3. 2. 3 #pragma 指令」を参照してください。
6	__DATE__、__TIME__ の値	コンパイル開始時のホストマシンのタイマに基づく値が設定されます。

3. 1. 4 データの内部表現と領域

本節では、型名と、データの内部表現の対応について述べます。データの内部表現は、以下の項目から成り立っています。

- データのサイズ

  データの占有する領域のサイズです。

- データのアライメント数

  データを割り付けるアドレスに関する制約です。任意のアドレスに割り付ける 1 バイトアライメント、偶数バイトに割り付ける 2 バイトアライメント、4 の倍数バイトに割り付ける 4 バイトアライメントがあります。

- 値の範囲

  スカラ型 (C 言語)、基本型 (C++ 言語) の値のとり得る範囲を示します。

- データの割り付け例

  構造体 / 共用体 (C 言語)、クラス型 (C++ 言語) の要素となるデータの割り付け方を示します。

(1) スカラ型 (C 言語)、基本型 (C++ 言語)

C 言語におけるスカラ型および、C++ 言語における基本型の内部表現を表 3.15 に示します。

表 3―15 スカラ型・基本型の内部表現

	型名	サイズ (byte)	アライメント数 (byte)	符号の有無	値の範囲	
					最小値	最大値
1	char*1	1	1	無	0	2^8-1 (255)
2	signed char	1	1	有	-2^7 (-128)	2^7-1 (127)
3	unsigned char	1	1	無	0	2^8-1 (255)
4	short	2	2	有	-2^{15} (-32768)	2^{15}-1 (32767)
5	signed short	2	2	有	-2^{15} (-32768)	2^{15}-1 (32767)
6	unsigned short	2	2	無	0	2^{16}-1 (65535)
7	int*2	4	4	有	-2^{31}(-2147483648)	2^{31}-1 (2147483647)
8	signed int*2	4	4	有	-2^{31}(-2147483648)	2^{31}-1 (2147483647)
9	unsigned int*2	4	4	無	0	2^{32}-1 (4294967295)
10	long	4	4	有	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
11	signed long	4	4	有	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
12	unsigned long	4	4	無	0	2^{32}-1 (4294967295)
13	long long	8	4	有	-2^{63} (-9223372036854775808)	2^{63}-1 (9223372036854775807)
14	signed long long	8	4	有	-2^{63} (-9223372036854775808)	2^{63}-1 (9223372036854775807)
15	unsigned long long	8	4	無	0	2^{64}-1 (18446744073709551615)
16	float	4	4	有	-∞	+∞

	型名	サイズ (byte)	アライメント数 (byte)	符号の有無	値の範囲	
					最小値	最大値
17	double long double	4*4	4	有	-∞	+∞
18	size_t	4	4	無	0	2^{32}-1 (4294967295)
19	ptrdiff_t	4	4	有	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
20	enum*3	4	4	有	-2^{31} (-2147483648)	2^{31}-1 (2147483647)
21	ポインタ	4	4	無	0	2^{32}-1 (4294967295)
22	bool*5 _Bool*8	1	1	–*9	–	–
23	リファレンス *6	4	4	無	0	2^{32}-1 (4294967295)
24	データメンバへのポインタ *6	4	4	有	0	2^{32}-1 (4294967295)
25	関数メンバへのポインタ *6 *7	12	4	–*9	–	–

注 1. signed_char オプションを指定した場合、signed char 型と同じになります。

2. int_to_short オプションを指定した場合、int 型は short 型と、signed int 型は signed short 型と、unsigned int 型は unsigned short 型とそれぞれ同じになります。

3. auto_enum オプションを指定した場合、列挙値が収まる最小の型となります。

4. dbl_size=8 を指定した場合、double 型および long double 型のサイズは 8 バイトになります。

5. C++ プログラム、または stdbool.h をインクルードした C99 プログラムのコンパイル時のみ有効です。

6. C++ プログラムのコンパイル時のみ有効です。

7. 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:
   long d;          // オブジェクトのオフセット値
   long 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のアドレス
     long offset;   // 対象メンバ関数が仮想関数のときの仮想関数表のオブジェクト
                    // 中のオフセット
   };
};
```

8. C99 コンパイルのみ有効です。_Bool 型は bool 型と同じ型としてコンパイルされます。

9. 符号の設定はありません。

(2) 構造体 / 共用体 (C 言語)、クラス型 (C++ 言語)

本項では、C 言語における配列型、構造体型、共用体型および、C++ 言語におけるクラス型の内部表現について説明します。

表 3.16 に構造体 / 共用体、クラス型の内部表現を示します。

表 3―16 構造体 / 共用体、クラス型の内部表現

	型名	アライメント数 (byte)	サイズ (byte)	データの割り付け例
1	配列型	配列要素のアライメント数	配列要素の数 × 要素サイズ	char a[10]; アライメント数 1byte サイズ 10byte
2	構造体型	構造体メンバのアライメント数のうち最大値	メンバのサイズの和 「(a) 構造体データの割り付け方」参照	struct { アライメント数 1byte char a,b; サイズ 2byte };
3	共用体型	共用体メンバのアライメント数のうち最大値	最大メンバのサイズ 「(b) 共用体データの割り付け方」参照	union { アライメント数 1byte char a,b; サイズ 1byte };

4	クラス型	仮想関数がある場合 :
		常に 4
		
		2) 上記以外 :
		データメンバのアライメント数のうち最大値
			データメンバ、仮想関数表へのポインタ、仮想基底クラスへのポインタの和
			「(c) クラスデータの割り付け方」参照
				class B: public A{
				virtual void f(); アライメント数 4byte
				}; サイズ 8byte
				class A{
				char a; アライメント数 1byte
				}; サイズ 1byte

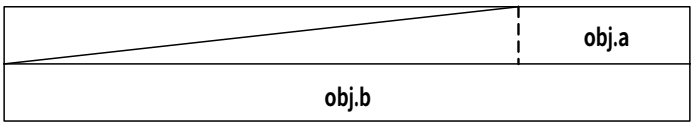
以下の例でサイズを明記していない □ は、4 バイトを表します。／ はパディングを表します。
また、アドレスの増える向きは、右から左とします（左側が上位アドレス）。

(a) 構造体データの割り付け方

構造体型の各メンバを割り付ける場合、そのメンバの型名の境界調整数に合わせるために直前のメンバとの間にパディングが生じる場合があります。

例

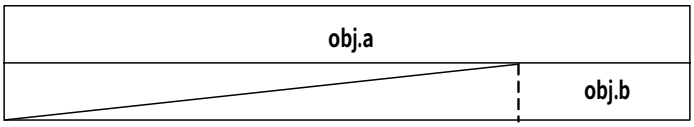
```
struct {
  char a;
  int b;
} obj;
```



構造体が 4 バイトのアライメント数を持ち、最後のメンバが 1,2,3 バイト目で終わっている場合、その次のバイトも含めて構造体型の領域として扱います。

例

```
struct {
  int a;
  char b;
} obj;
```

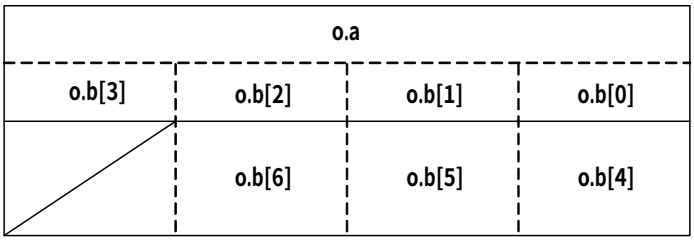


(b) 共用体データの割り付け方

共用体が 4 バイトのアライメント数を持ち、最大メンバのサイズが 4 の倍数バイトでない場合、4 の倍数になるまで残りのバイトも含めて共用体型の領域として扱います。

例

```
union {
  int a;
  char b[7];
} o;
```

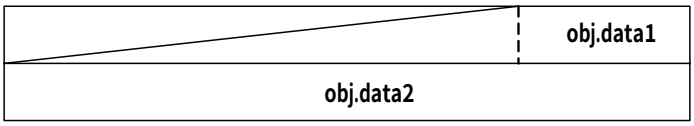


(c) クラスデータの割り付け方

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

例

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



アライメント数が 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、パディングを作らないようにデータメンバを割り付けます。

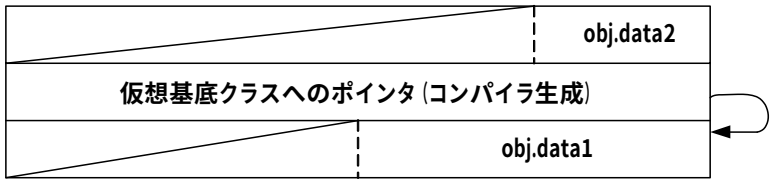
例

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

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

例

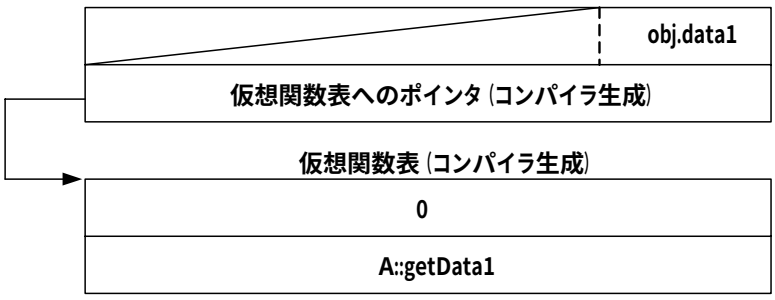
```
class A{
   short data1;
};
class B: virtual protected A{
   char data2;
}obj;
```



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

例

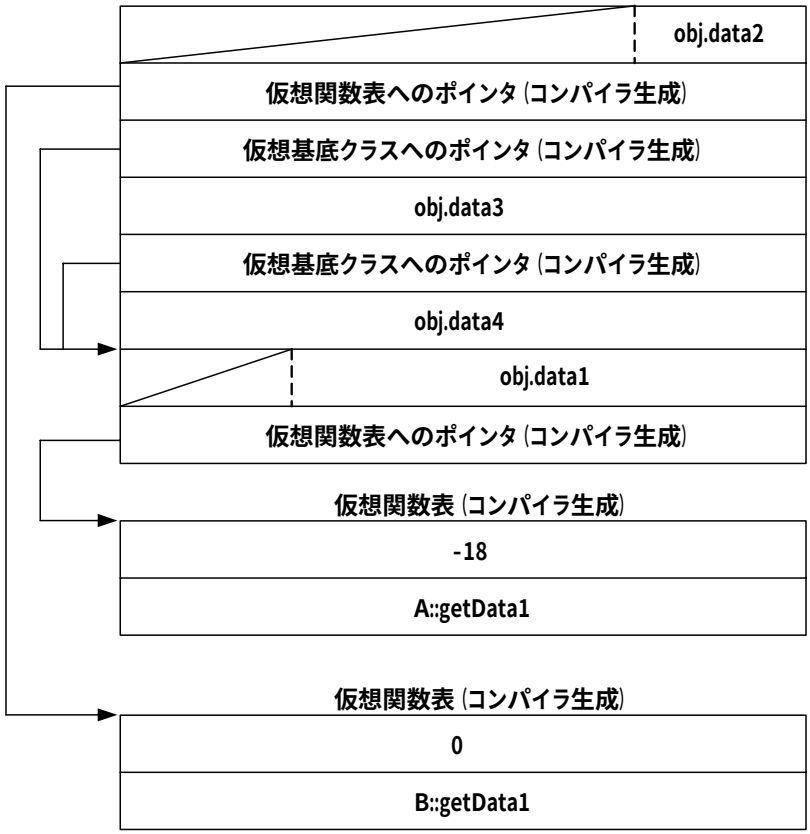
```
class A{
   char data1;
 public:
   virtual char getData1();
}obj;
```



仮想基底クラス、基底クラス、仮想関数があるクラスの例を示します。

例

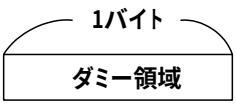
```
class A{
   char data1 ;
   virtual char getData1();
};
class B:virtual public A{
   char data2;
   char getData2();
   char getData1();
};
class C:virtual protected A{
   int data3;
};
class D:virtual public A,public B,public C{
 public:
   int data4;
   char getData1();
}obj;
```



空クラスの場合、1 バイトのダミー領域を割り付けます。

例

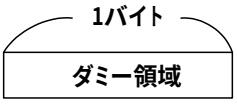
```
class A{
   void fun();
}obj;
```



空クラスを基底クラスに持つ空クラスの場合でも、ダミー領域は 1 バイトになります。

例

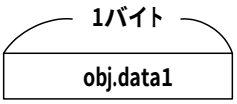
```
class A{
   void fun();
};
class B: A{
   void sub();
}obj;
```



空クラスのダミー領域は、クラスサイズが 0 の場合に割り付けます。基底クラスや派生クラスにデータメンバがある場合や、仮想関数があるクラスの場合には、ダミー領域は割り付けません。

例

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



(3) ビットフィールド

ビットフィールドは、構造体、共用体、クラスの中にビット幅を指定して割り付けるメンバです。

本項では、ビットフィールド特有の割り付け規則について説明します。

(a) ビットフィールドのメンバ

表 3.17 にビットフィールドメンバの仕様を示します。

表 3―17 ビットフィールドメンバの仕様

	項目	仕様
1	ビットフィールドで許される型指定子	(unsigned) char、signed char、bool*1、_Bool*5、 (unsigned) short、signed short、enum、 (unsigned) int、signed int、 (unsigned) long、signed long、 (unsigned) long long、signed long long

	項目	仕様
2	宣言された型に拡張するときの符号の扱い *2	符号なし (unsigned) は、ゼロ拡張 *3 符号付き (signed) は、符号拡張 *4
3	符号指定なしの型の符号型	符号なし (unsigned) 但し、signed_bitfield オプションが指定された場合は、符号付き (signed)
4	enum 型の符号型	符号付き (signed) 但し、auto_enum オプションが指定された場合は、その結果の型に従う

注 1. C++ プログラム、または stdbool.h をインクルードした C99 プログラムのコンパイル時のみ bool を指定できます。

2. ビットフィールドのメンバを使用する場合、ビットフィールドに格納したデータを宣言した型に拡張して使用します。符号付き (signed) で宣言されたサイズが 1 ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は 0 と -1 だけになります。

3. ゼロ拡張：拡張するときに上位のビットにゼロを補います。

4. 符号拡張：拡張するときにビットフィールドデータの最上位ビットを符号として解釈し、データより上位のビット全てに符号ビットを補います。

5. C99 プログラムのみ有効です。_Bool 型は bool 型と同じ型としてコンパイルされます。

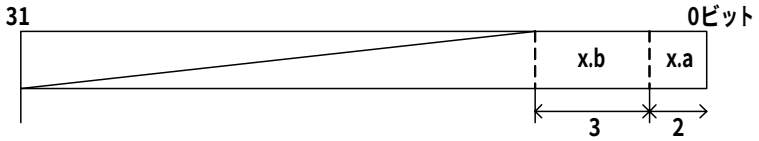
(b) ビットフィールドの割り付け方

ビットフィールドは、以下の 5 つの規則に従って割り付けます。

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

例

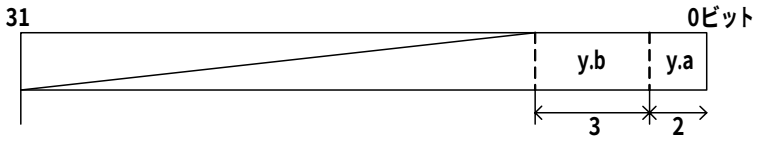
```
struct b1 {
  int a:2;
  int b:3;
} x;
```



- 同じサイズの型指定子が連続している場合は、可能な限り同じ領域に詰め込みます。

例

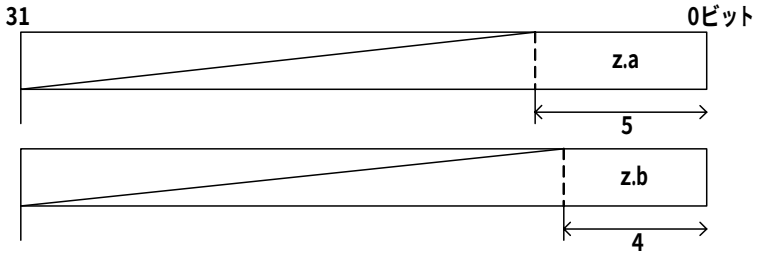
```
struct b1 {
  long          a:2;
  unsigned int  b:3;
} y;
```



- 異なるサイズの型指定子で宣言されたメンバは、次の領域に割り付けます。

例

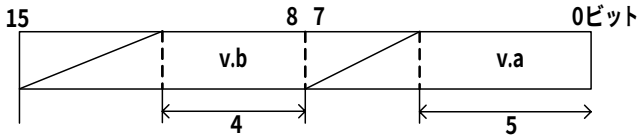
```
struct b1 {
  int   a:5;
  char  b:4;
} z;
```



- 同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。

例

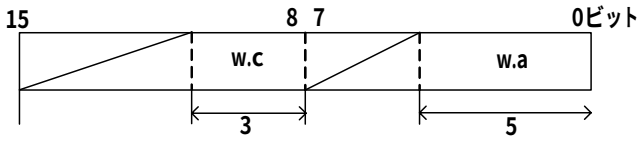
```
struct b2 {
  char a:5;
  char b:4;
} v;
```



- ビット幅 0 のビットフィールドのメンバを指定すると、次のメンバからは、強制的に次の領域に割り付けます。

例

```
struct b2 {
  char a:5;
  char  :0;
  char c:3;
} w;
```



備考 ビットフィールドメンバを上位ビット側から詰め込むことも可能です。
詳細は、「3. 2 拡張言語仕様」の #pragma bit_order およびビルド編の bit_order オプションを参照してください。

(4) Big Endian のメモリ割り付け

Big Endian でのメモリ上のデータ配列は以下のとおりです。

(a) 1 バイトデータ ((signed)char、unsigned char、bool、_Bool 型)

1 バイトデータの中のビット並び順は、Little Endian の場合も、Big Endian の場合も同じです。

(b) 2 バイトデータ ((signed)short、unsigned short 型)

2 バイトデータの中のバイト並び順は、Little Endian と Big Endian で上位、下位のバイトが逆になります。

例

0x100 番地に 2 バイトデータ 0x1234 がある場合

Little Endian:	0x100 番地：0x34	Big Endian:	0x100 番地：0x12
	0x101 番地：0x12		0x101 番地：0x34

(c) 4 バイトデータ ((signed)int、unsigned int、(signed)long、unsigned long、float 型)

4 バイトデータの中のバイト並び順は、Little Endian と Big Endian で 4 バイトのデータの順序が逆になります。

例

0x100 番地に 4 バイトデータ 0x12345678 がある場合

Little Endian:	0x100 番地：0x78	Big Endian:	0x100 番地：0x12
	0x101 番地：0x56		0x101 番地：0x34
	0x102 番地：0x34		0x102 番地：0x56
	0x103 番地：0x12		0x103 番地：0x78

(d) 8 バイトデータ ((signed)long long、unsigned long long、double 型)

8 バイトデータの中のバイト並び順は、Little Endian と Big Endian で 8 バイトのデータの順序が逆になります。

例

0x100 番地に 8 バイトデータ 0x0123456789abcdef がある場合

Little Endian:	0x100 番地：0xef	Big Endian:	0x100 番地：0x01
	0x101 番地：0xcd		0x101 番地：0x23
	0x102 番地：0xab		0x102 番地：0x45
	0x103 番地：0x89		0x103 番地：0x67
	0x104 番地：0x67		0x104 番地：0x89
	0x105 番地：0x45		0x105 番地：0xab
	0x106 番地：0x23		0x106 番地：0xcd
	0x107 番地：0x01		0x107 番地：0xef

(e) 構造体 / 共用体、クラス型データ

構造体 / 共用体、クラス型データの各メンバの割り付けは Little Endian のときと同様です。ただし、各メンバのバイト並び順はそのデータサイズの規則に従って反転します。

例

0x100 番地に、

```
struct {
    short a;
    int b;
}z = {0x1234, 0x56789abc};
```

ある場合

Little Endian:	0x100 番地：0x34	Big Endian:	0x100 番地：0x12
	0x101 番地：0x12		0x101 番地：0x34
	0x102 番地：パディング		0x102 番地：パディング
	0x103 番地：パディング		0x103 番地：パディング
	0x104 番地：0xbc		0x104 番地：0x56
	0x105 番地：0x9a		0x105 番地：0x78
	0x106 番地：0x78		0x106 番地：0x9a
	0x107 番地：0x56		0x107 番地：0xbc

(f) ビットフィールド

ビットフィールドの各領域の割り付けも Little Endian のときと同様です。ただし、各領域のバイト並び順はそのデータサイズの規則に従って反転します。

例

0x100 番地に、

```
struct {
    long a:16;
    unsigned int b:15;
    short c:5;
}y={1,1,1};
```

ある場合

Little Endian:	0x100 番地：0x01	Big Endian:	0x100 番地：0x00
	0x101 番地：0x00		0x101 番地：0x01
	0x102 番地：0x01		0x102 番地：0x00
	0x103 番地：0x00		0x103 番地：0x01
	0x104 番地：0x01		0x104 番地：0x00
	0x105 番地：0x00		0x105 番地：0x01
	0x106 番地：パディング		0x106 番地：パディング
	0x107 番地：パディング		0x107 番地：パディング

(5) 浮動小数点型の仕様

(a) 浮動小数点型の内部表現

コンパイラで扱う浮動小数点型の内部表現は、IEEE の形式に準拠しています。ここでは、IEEE 形式の浮動小数点型の内部表現の概要について述べます。

なお、本節では dbl_size=8 オプションが指定されたものとして説明しています。dbl_size=4 オプションが指定された場合は、double 型および long double 型の内部表現は、float 型と同じになります。

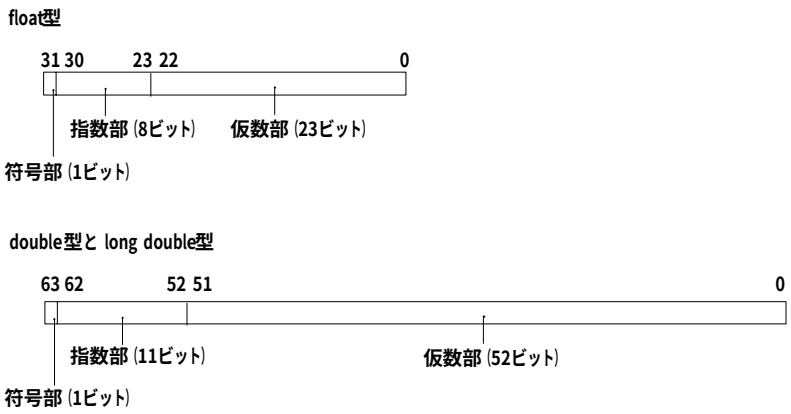
(b) 内部表現の形式

float 型は IEEE の単精度形式 (32 ビット)、double 型と long double 型は IEEE の倍精度形式 (64 ビット) で表現します。

(c) 浮動小数点データフォーマット

float 型および double 型と long double 型の浮動小数点データフォーマットを図 3.1 に示します。

図 3―1 浮動小数点データフォーマット



内部表現の各構成要素の意味を以下に示します。

(i) 符号部

浮動小数点型の符号を示します。0 のとき正、1 のとき負を示します。

(ii) 指数部

浮動小数点型の指数を 2 のべき乗で示します。

(iii) 仮数部

浮動小数点型の有効数字に対応するデータです。

(d) 表現する値の種類

浮動小数点型は、通常の実数値のほかに、無限大等の値も表現することができます。浮動小数点型が表現する値の種類を以下に示します。

(i) 正規化数

指数部が 0 または全ビット 1 ではない場合です。通常の実数値を表現します。

(ii) 非正規化数

指数部が 0 で、仮数部が 0 でない場合です。絶対値の小さな実数値を表現します。

(iii) ゼロ

指数部および仮数部が 0 の場合です。値 0.0 を表現します。

(iv) 無限大

指数部が全ビット 1 で仮数部が 0 の場合です。無限大を表現します。

(v) 非数

指数部が全ビット 1 で仮数部が 0 でない場合です。「0.0/0.0」、「∞ / ∞」、「∞ - ∞」等、結果が数値に対応しない演算の結果として得られます。

浮動小数点型の表現する値を決定する条件を表 3.18 に示します。

表 3―18 浮動小数点型の表現する値の種類

仮数部	指数部		
	0	0 でも全ビット 1 でもない	全ビット 1
0	0	正規化数	無限大
0 以外	非正規化数		非数

注 非正規化数は、正規化数で表現できない範囲の絶対値の小さな浮動小数点型を表現しますが、正規化数と比較して有効桁数が少なくなっています。したがって、演算の結果あるいは途中結果が非正規化数となる場合、結果の有効桁数は保証しません。

denormalize=off を指定した場合、非正規化数は 0 として扱います。

denormalize=on を指定した場合、非正規化数は非正規化数のまま扱います。

(e) float 型

float 型の内部表現は、1 ビットの符号部、8 ビットの指数部、23 ビットの仮数部からなります。

(i) 正規化数

符号部は、0(正) または 1(負) で、値の符号を示します。

指数部は、1 ～ 254(2^8-2) の値をとります。実際の指数は、この値から 127 を引いた値で、その範囲は -126 ～ 127 です。

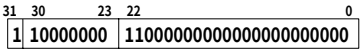
仮数部は、0 ～ 2^{23}-1 の値をとります。実際の仮数は、2^{23} のビットを 1 と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$(-1)^{<符号部>} \times 2^{<指数部>-127} \times (1 + <仮数部> \times 2^{-23})$

となります。

例



符号： －

指数： $10000000_{(2)} - 127 = 1$

仮数： $1.11_{(2)} = 1.75$

値： $-1.75 \times 2^1 = -3.5$

(2) は2進数を意味します。

(ii) 非正規化数

符号部は 0(正) または 1(負) で、値の符号を示します。

指数部は 0 で、実際の指数は -126 になります。

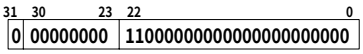
仮数部は、1 ～ 2^{23}-1 で、実際の仮数は、2^{23} のビットを 0 と仮定し、その直後に小数点があるものとして解釈します。

非正規化数の表現する値は、

$(-1)^{<符号部>} \times 2^{-126} \times (<仮数部> \times 2^{-23})$

となります。

例



符号： ＋

指数： －126

仮数： $0.11_{(2)} = 0.75$

値： 0.75×2^{-126}

(2) は2進数を意味します。

(iii) ゼロ

符号部は 0(正) または 1(負) で、それぞれ +0.0、-0.0 を示します。

指数部、仮数部はともに 0 です。

+0.0、-0.0 は、ともに値としては 0.0 を示します。

(iv) 無限大

符号部は 0(正) または 1(負) で、それぞれ + ∞、- ∞を示します。

指数部は 255(2^8-1) です。

仮数部は 0 です。

(v) 非数

指数部は 255(2^8-1) です。

仮数部は 0 以外の値です。

注 仮数部の最上位ビットが 1 の非数を qNaN、仮数部の最上位ビットが 0 の非数を sNaN と呼びます。

その他の仮数フィールドの値、および符号部については規定していません。

(f) double 型と long double 型

double 型と long double 型の内部表現は、1 ビットの符号部、11 ビットの指数部、52 ビットの仮数部からなります。

(i) 正規化数

符号部は 0(正) または 1(負) で、値の符号を示します。

指数部は、1 ～ 2046(2^{11}-2) の値をとります。実際の指数は、この値から 1023 を引いた値で、その範囲は -1022 ～ 1023 です。

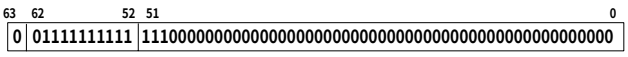
仮数部は、0 ～ 2^{52}-1 の値となります。実際の仮数は、2^{52} のビットを 1 と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$(-1)^{<\text{符号部}>} \times 2^{<\text{指数部}>-1023} \times (1 + <\text{仮数部}> \times 2^{-52})$

となります。

例



符号： ＋

指数： $1111111111_{(2)} - 1023 = 0$

仮数： $1.111_{(2)} = 1.875$

値： $1.875 \times 2^0 = 1.875$

(2) は2進数を意味します。

(ii) 非正規化数

符号部は 0(正) または 1(負) で、値の符号を示します。

指数部は 0 で、実際の指数は -1022 になります。

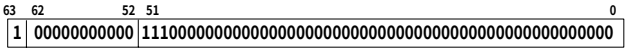
仮数部は、1 ～ 2^{52}-1 で、実際の仮数は、2^{52} のビットを 0 と仮定し、その直後に小数点があるものとして解釈します。

非正規化数が表現する値は、

$(-1)^{<\text{符号部}>} \times 2^{-1022} \times (<\text{仮数部}> \times 2^{-52})$

となります。

例



符号： －

指数： － 1022

仮数： 0.111 (2) ＝ 0.875　　(2) は2進数を意味します。

値： 0.875 × 2 -1022

(iii) ゼロ

符号部は 0(正) または 1(負) で、それぞれ +0.0、-0.0 を示します。

指数部、仮数部はともに 0 です。

+0.0、-0.0 は、ともに値としては 0.0 を示します。

(iv) 無限大

符号部は 0(正) または 1(負) で、それぞれ + ∞、- ∞を示します。

指数部は 2047(2^{11}-1) です。

仮数部は 0 です。

(v) 非数

指数部は 2047(2^{11}-1) です。

仮数部は 0 以外の値です。

注 仮数部の最上位ビットが 1 の非数を qNaN、仮数部の最上位ビットが 0 の非数を sNaN と呼びます。
その他の仮数フィールドの値、および符号部については規定していません。

3. 1. 5 演算子の評価順序

式の中に複数の演算子がある場合、それらの演算子の評価順序は、優先順位と「右」または「左」で表わされる結合性によって決まります。

各演算子の優先順位と結合性を表 3.19 に示します。

表 3―19 演算子の優先順位と結合性

優先順位	演算子	結合性	適用される式
1	++ -- (後置) ( ) [ ] -> .	左	後置式
2	++ -- (前置) ! ~ + - * & sizeof	右	単項式
3	(型名)	右	キャスト式
4	* / %	左	乗除式
5	+ -	左	加減式
6	<< >>	左	ビット単位のシフト式
7	< <= > >=	左	関係式
8	== ! =	左	等価式
9	&	左	ビット単位の AND 式

10	^	左	ビット単位の排他 OR 式
11	|	左	ビット単位の OR 式
12	&&	左	論理 AND 式
13	||	左	論理 OR 式
14	? :	右	条件式
15	= += -= *= /= %= <<= >>= &= |= ^=	右	代入式
16	,	左	カンマ式

3. 1. 6 準拠する言語仕様

(1) C 言語仕様 (lang=c オプション選択時)

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

(2) C 言語仕様 (lang=c99 オプション選択時)

ISO/IEC 9899:1999 INTERNATIONAL STANDARD Programming Languages - C

(3) C++ 仕様 (lang=cpp オプション選択時)

Microsoft(R) Visual C/C++ 6.0 と互換性のある言語仕様に基づいています。

3. 2 拡張言語仕様

この節では、CCRX で拡張されている言語仕様について説明します。

拡張仕様には、#pragma、キーワード、組み込み関数、およびセクションアドレス演算子があります。

3. 2. 1 マクロ名

次にサポートしているマクロ名を示します。

表 3―20 サポートしているマクロ

	オプション	プリデファインドマクロ	
1	–	__DATE__	ソース・ファイルの翻訳日付(“ Mmm dd yyyy ” の形式をもつ文字列定数。ここで，月の名前は ANSI 規格で規定されている asctime 関数で生成されるもの（英字 3 文字の並びで最初の 1 文字のみ大文字）と同じもの。dd の最初の文字は値が 10 より小さい場合空白とします)。
	–	__FILE__	仮定されたソース・ファイルの名前（文字列定数)。

	オプション	プリデファインドマクロ	
	–	__LINE__	その時点でのソース行の行番号（10 進数）。
	–	__STDC__	1
		__STDC_HOSTED__	1
	–	__STDC_VERSION__	199409L(lang=c 時) 199901L(lang=c99 時)
		__cplusplus	1(lang=cpp 時)
	–	__TIME__	ソース・ファイルの翻訳時間（" hh:mm:ss " の形式をもつ文字列定数。
1	cpu=rx600 cpu=rx200	#define __RX600 #define __RX200	1 1
2	endian=big endian=little	#define __BIG #define __LIT	1 1
3	dbl_size=4 dbl_size=8	#define __DBL4 #define __DBL8	1 1
4	int_to_short	#define __ INT_SHORT	1
5	signed_char unsigned_char	#define __SCHAR #define __UCHAR	1 1
6	signed_bitfield unsigned_bitfield	#define __SBIT #define __UBIT	1 1
7	round=zero round=nearest	#define __ROZ #define __RON	1 1
8	denormalize=off denormalize=on	#define __DOFF #define __DON	1 1
9	bit_order=left bit_order=right	#define __BITLEFT #define __BITRIGHT	1 1
10	auto_enum	#define __AUTO_ENUM	1
11	library=function library=intrinsic	#define __FUNCTION_LIB #define __INTRINSIC_LIB	1 1
12	fpu	#define __FPU	1
13	–	#define __RENESAS__ *1	1
14	–	#define __RENESAS_VERSION__ *1	0xAABBCC00 *2
15	–	#define __RX*1	1
16	pic	#define __PIC	1
17	pid	#define __PID	1

注 1. オプションに関わらず常に定義されます。

2. バージョンが V.AA.BB.CC **の場合、**__RENESAS_VERSION__ **の値は** 0xAABBCC00 **となります。**

例) V.1.01.00 **の場合、**#define __RENESAS_VERSION__ 0x01010000

3. 2. 2　キーワード

CCRX では，拡張機能を実現するために次の字句をキーワードとして追加しています。これらの字句も ANSI-C のキーワードと同様，ラベルや変数名として使用することはできません。

次に，CCRX で追加されているキーワード一覧を示します。

__evenaccess, far, _far, near, _near

表 3―21 キーワード一覧

	キーワード名	機能
1	#pragma STDC CX_LIMITED_RANGE #pragma STDC FENV_ACCESS #pragma STDC FP_CONTRACT	予約キーワード。C99 言語選択時のみ有効。 (C99 言語の文法確認のみ行い、内容は無視します。)
2	#pragma キーワード	言語拡張機能を提供します。 「3. 2. 3 #pragma 指令 」を参照ください。
3	__evenaccess	変数の型のサイズのアクセスを保証
4	far _far near _near	予約キーワード (型名として認識しますが、無視します。)

3. 2. 3 #pragma 指令

(1) セクション切り替え指定

コンパイラの出力するセクション名を切り替えます。

なお、記述方法の詳細については、「(1) セクション切り替え記述」を参照してください。

```
#pragma section [<セクション種別>] [△<変更セクション名>]
<セクション種別>: { P | C | D | B }
```

(2) スタックセクション作成

スタックセクションを作成します。

なお、記述方法の詳細については、「(2) スタックセクション作成記述」を参照してください。

```
#pragma stacksize {si=<定数> | su=<定数>}
```

(3) 割り込み関数作成

割り込み関数を作成します。

なお、記述方法の詳細については、「(3) 割り込み関数作成記述」を参照してください。

```
#pragma interrupt [(]<関数名>[(＜割り込み仕様＞[,…])][,…][)]
```

(4) 関数インライン展開

関数をインライン展開します。

なお、記述方法の詳細については、「(4) 関数のインライン展開記述」を参照してください。

```
#pragma inline [(]<関数名>[,…][)]
```

(5) 関数インライン展開抑止

関数をインライン展開抑止します。

なお、記述方法の詳細については、「(4) 関数のインライン展開記述」を参照してください。

```
#pragma noinline [(]<関数名>[,…][)]
```

(6) アセンブリ記述関数インライン展開

アセンブラ埋め込みインライン関数を作成します。

なお、記述方法の詳細については、「(5) アセンブラ記述関数インライン展開」を参照してください。

```
#pragma inline_asm[(]<関数名>[,…][)]
```

(7) エントリ関数指定

エントリ関数を指定します。

なお、記述方法の詳細については、「(6) エントリ関数指定」を参照してください。

```
#pragma entry[(]<関数名>[)]
```

(8) ビットフィールド並び順指定

ビットフィールド並び順を指定します。

なお、記述方法の詳細については、「(7) ビットフィールド並び順指定」を参照してください。

```
#pragma bit_order [{left | right}]
```

(9) 構造体 / クラスメンバの 1 バイトアライメント指定

構造体 / クラスメンバのアライメントを 1 バイトにします。

なお、記述方法の詳細については、「(8) 構造体 / クラスメンバのアライメント指定」を参照してください。

```
#pragma pack
```

(10) 構造体 / クラスメンバのデフォルトアライメント指定

構造体 / クラスメンバのアライメントをメンバのアライメント数にします。

なお、記述方法の詳細については、「(8) 構造体 / クラスメンバのアライメント指定」を参照してください。

```
#pragma unpack
```

(11) 構造体 / クラスメンバのオプションアライメント指定

構造体 / クラスメンバのアライメントをオプション指定にします。

なお、記述方法の詳細については、「(8) 構造体 / クラスメンバのアライメント指定」を参照してください。

```
#pragma packoption
```

(12) 変数の絶対アドレス割り付け指定

変数を絶対アドレスに割り付けます。

なお、記述方法の詳細については、「(9) 変数の絶対アドレス割り付け指定」を参照してください。

```
#pragma address [(]< 変数名 >=< 絶対アドレス >[,…][)]
```

(13) 初期値のエンディアン指定

初期値のエンディアン指定をします。

なお、記述方法の詳細については、「(10) 初期値のエンディアン指定」を参照してください。

```
#pragma endian [{big | little}]
```

(14) 分岐先 4 バイト整合指定

分岐先を 4 バイト整合する関数を指定します。

なお、記述方法の詳細については、「(11) 分岐先の命令実行向け整合を行う関数の指定」を参照してください。

```
#pragma instalign4 [(]< 関数名 >[( ＜分岐先の種類＞ )][,…][)]
```

(15) 分岐先 8 バイト整合指定

分岐先を 8 バイト整合する関数を指定します。

なお、記述方法の詳細については、「(11) 分岐先の命令実行向け整合を行う関数の指定」を参照してください。

```
#pragma instalign8 [(]< 関数名 >[( ＜分岐先の種類＞ )][,…][)]
```

(16) 分岐先整合なし指定

分岐先を整合しない関数を指定します。

なお、記述方法の詳細については、「(11) 分岐先の命令実行向け整合を行う関数の指定」を参照してください。

```
#pragma noinstalign [(]< 関数名 >[,…][)]
```

3. 2. 4　拡張仕様の使用方法

この項では、下記の拡張機能の使用方法について説明します。

- セクション切り替え記述
- スタックセクション作成記述
- 割り込み関数作成記述
- 関数のインライン展開記述
- アセンブラ記述関数インライン展開
- エントリ関数指定
- ビットフィールド並び順指定
- 構造体 / クラスメンバのアライメント指定
- 変数の絶対アドレス割り付け指定

- 初期値のエンディアン指定
- 分岐先の命令実行向け整合を行う関数の指定

(1) セクション切り替え記述

コンパイラの出力するセクション名を切り替えます。

セクション種別と変更セクションを指定した場合、セクション種別が P であればその #pragma 宣言以降に記述された関数のセクション名を変更します。セクション種別が C、D、または B の場合は、その #pragma 宣言以降に実体を定義した全てのセクション名を変更します。

変更セクション名のみを指定した場合、その #pragma 宣言以降にあるプログラム領域、定数領域、初期化データ領域、および未初期化データ領域のすべてのセクション名を変更します。この場合、各セクションの変更後のセクション名は、各デフォルトセクション名の後に < 変更セクション名 > の文字列を追加したセクション名となります。

セクション種別も変更セクション名も記述しなかった場合は、その #pragma 宣言以降にあるプログラム領域、定数領域、初期化データ領域、および未初期化データ領域の全てのセクション名をデフォルトセクション名に戻します。

各セクション種別のデフォルトセクション名は、section オプションの指定があればそれに従います。ない場合はセクション種別名をそのまま用います。

例 1. セクション名とセクション種別を指定した場合

```
#pragma section B Ba
int i; // Ba セクションに配置
void func(void)
{
( 省略 )
}

#pragma section B Bb
int j; // Bb セクションに配置
void sub(void)
{
( 省略 )
}
```

2. セクション種別を省略した場合

```
#pragma section abc
int a; // Babc セクションに配置
const int c=1; // Cabc セクションに配置
void f(void)// Pabc セクションに配置
{
    a=c;
}

#pragma section
int b;// B セクションに配置
void g(void)// P セクションに配置
{
    b=c;
}
```

```
#pragma section [<セクション種別>] [△<変更セクション名>]
<セクション種別>: { P | C | D | B }
```

#pragma section **は関数定義の外で宣言しなければなりません。**

次の項目のセクション名は変更できません。section **オプションを使用してください。**

(1) **文字列リテラルおよび集成体の動的初期化で用いる初期化子**

(2) switch **文の分岐テーブル**

1 **ファイルあたりの** #pragma section **で指定できるセクション数は最大** 2045 **個です。**

静的クラスメンバ変数のセクションを指定する場合は、クラスのメンバ宣言と実体の定義の両方に #pragma section **の指定が必要になります。**

例

```
/*
 ** クラスメンバ宣言
 */

class A
{
private:

// 初期値なし
#pragma section DATA
static int data_;
#pragma section

// 初期値あり
#pragma section TABLE
static int table_[2];
#pragma section
};

/*
** 実体定義
*/

// 初期値なし
#pragma section DATA
int A::data_;
#pragma section

// 初期値あり
#pragma section TABLE
int A::table_[2]={0, 1};
#pragma section
```

(2) スタックセクション作成記述

si=< 定数 > を指定した場合、セクション名 SI、サイズ < 定数 > のスタックとして使用するデータセクションを作成します。

su=< 定数 > を指定した場合、セクション名 SU、サイズ < 定数 > のスタックとして使用するデータセクションを作成します。

例　C ソース：

```
#pragma stacksize si=100
#pragma stacksize su=200
```

コード展開例：

```
.SECTION    SI,DATA,ALIGN=4
.BLKB       100
.SECTION    SU,DATA,ALIGN=4
.BLKB       200
```

```
#pragma stacksize {si=<定数> | su=<定数>}
```

si, su 指定はファイル内でそれぞれ 1 回しか指定できません。

< 定数 > は必ず 4 の倍数を指定してください。

< 定数 > に記述できる値の範囲は、4 から 2147483644(0x7ffffffc) までです。

(3) 割り込み関数作成記述

#pragma interrupt を用いて割り込み関数となる関数を宣言します。

関数名には、グローバル関数および静的関数メンバを指定できます。

割り込み仕様の一覧を表 3.22 に示します。

表 3―22 割り込み仕様の一覧

	項目	形式	オプション	指定内容
1	ベクタテーブル指定	vect=	< ベクタ番号 >	割り込み関数のアドレスを配置するベクタ番号
2	高速割り込み指定	fint	なし	高速割り込みに使用する関数の指定 RTFI 命令でリターン
3	割り込み関数レジスタ制限指定	save	なし	割り込み関数内で使用するレジスタの本数を制限し、退避・回復の数を減らす
4	多重割り込み許可指定	enable	なし	関数の先頭で PSW の I フラグを 1 にし、多重割り込みを許可する
5	ACC 保存指定	acc	なし	割り込み関数内で ACC を退避・回復する
6	ACC 非保存指定	no_acc	なし	割り込み関数内で ACC を退避・回復しない

#pragma interrupt を用いて宣言した関数は、関数の処理の前後で全レジスタを保証 (関数入口 / 出口において関数内で使用する全レジスタを退避・回復) し、通常 RTE 命令でリターンします。

割り込み仕様を指定しない場合は単純な割り込み関数として処理します。

ベクタテーブル指定 (vect=) をした場合は C$VECT セクション内の指定したベクタテーブル番号位置にその関数アドレスを設定します。

高速割り込み指定 (fint) をした場合は、RTFI 命令でリターンします。また、fint_register オプションを指定した場合は、オプションで指定したレジスタを退避、回復せずに割り込み関数で使用します。

割り込み関数レジスタ制限指定 (save) をした場合は、割り込み関数内で使用するレジスタの本数を R1 ～ R5、および R14 ～ R15 に制限します。R6 ～ R13 は割り込みで使用しないため、退避・回復命令は生成しません。

多重割り込み許可指定 (enable) をした場合は、割り込み関数の先頭で PSW の I フラグを 1 にし、多重割り込みを許可します。

ACC 保存指定 (acc) をした場合で、指定された関数から別の関数呼び出しがあったり、関数内で ACC を書き換える命令を使う場合は、ACC を退避および回復する命令を生成します。

ACC 非保存指定 (no_acc) をした場合は、ACC を退避および回復する命令を生成しません。

acc および no_acc のどちらの指定もない場合は、コンパイルオプションに従います。

割り込み関数の定義に対して指定できる関数は、グローバル関数 (C/C++ 言語) と静的関数メンバ (C++ 言語) です。

関数の返却値の型は void のみです。return 文の返却値を指定することはできません。指定があった場合はエラーを出力します。

例 1. 正しい宣言と誤った宣言の例

```
#pragma interrupt (f1, f2)
void f1(){…}// 正しい宣言です。
int f2(){…}// 返却値の型が void ではないのでエラーになります。
```

2. 通常の割り込み関数の例

C ソース：

```
#pragma interrupt func
void func(){ .... }
```

出力コード：

```
_func:
PUSHM R1-R3; 関数内で使用しているレジスタを退避
....
(R1,R2,R3 を関数内で使用 )
....
POPM R1-R3; 入口で退避したレジスタを回復
RTE
```

3. 関数呼び出しがある割り込み関数の例

関数内で使用しているレジスタに加えて、関数呼び出し前後で保証しないレジスタについても、割り込み関数の入口で退避し、出口で回復します。

C ソース：

```
#pragma interrupt func
void func(){
  ...
  sub();
  ...
}
```

出力コード：

```
_func:
    PUSHM R14-R15
    PUSHM R1-R5
    ...
    BSR _sub
    ...
    POPM R1-R5
    POPM R14-R15
    RTE
```

4.　割り込み仕様 fint **を使用した場合の例**

C **ソース：**fint_register=2 **オプションを指定してコンパイル**

```
#pragma interrupt func1(fint)
void func1(){ a=1; }    //  割り込み関数
void func2(){ a=2; }    //  通常関数
```

出力コード：

```
_func1:
    PUSHM R1-R3 ;  関数内で使用しているレジスタを退避
    ...         ; ( 但し、R12,R13 は退避しない )
    ...
    (R1,R2,R3,R12,R13 を関数内で使用 )
    ...
    POPM R1-R3  ;  入口で退避したレジスタを回復
    RTFI

_func2:
    ...         ; #pragma interrupt fint 指定した関数以外では、
    ...         ; R12,R13 を使用しないコードを生成する
    RTE
```

5.　割り込み仕様 acc **を使用した場合の例**

C **ソース：**

```
void func5(void);
#pragma interrupt accsaved_ih(acc)      /* "acc" を指定 */
void accsaved_ih(void)
{
     func5();
}
```

出力コード：

```
_accsaved_ih:
     PUSHM R14-R15
     PUSHM R1-R5
     MVFACMI R4
     SHLL #10H, R4
     MVFACHI R5
     PUSHM R4-R5
     BSR _func5
     POPM R4-R5
     MVTACLO R4
     MVTACHI R5
     POPM R1-R5
     POPM R14-R15
     RTE
```

[備考]

最適化により削除される場合がありますので、static 関数は指定しないでください。

RX 命令セットの仕様のため、acc フラグで退避・回復できるのは、ACC の上位 48 ビットに限られます。ACC の下位 16 ビットは退避・回復されません。

各割り込み仕様は、小文字のみ指定できます。大文字を指定してもエラーになります。

割り込み仕様として vect を使った場合、指定の無い空きベクタのアドレスは 0 になります。このアドレスは、最適化リンケージエディタで任意のアドレスやシンボルに変更することができます。詳しくは、VECT および VECTN オプションの項目を参照してください。

#pragma interrupt の関数には、引数 (仮引数) を定義することはできません。定義してもエラーにはなりませんが、引数から値を読み出しても、不定値が返されます。

＜ acc、no_acc の用途＞

acc、no_acc は次のような用途を考慮したものです。

・ACC の補償を save_acc で行う場合の割り込み応答速度低下の対策 (no_acc)

既存の割り込み関数で ACC の補償には save_acc オプションが有効だが、割り込み応答速度が悪化することがあるため、不要な ACC の退避・回復を関数単位で抑止する手段をとして no_acc を用意する。

・ACC の退避・回復をソースコードで制御 (acc と no_acc)

ACC の退避・回復の考慮が完了している割り込み関数には、明示的に acc、no_acc を選択しておくことで、ソースプログラムで ACC の退避・回復を save_acc に依存せずに定義できる。

(4) 関数のインライン展開記述

#pragma inline は、インライン展開する関数を宣言します。

noinline オプションが指定された場合でも、#pragma inline 指定された関数はインライン展開の対象となります。

#pragma noinline は、inline オプションの指定を抑止する関数を宣言します。

関数名には、グローバル関数および静的関数メンバを指定できます。

#pragma inline で指定した関数名の関数と関数指定子 inline(C++ 言語および C(C99) 言語) を指定した関数は、その関数を呼び出したところにインライン展開されます。

例 ソースファイル

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
main()
{
    x=func(10,20);
}
```

展開イメージ

```
int x;
main()
{
    int func_result;
    {
        int a_1=10, b_1=20;
        func_result=(a_1+b_1)/2;
    }
    x=func_result;
}
```

```
#pragma inline [(]<関数名>[,…][)]
#pragma noinline [(]<関数名>[,…][)]
```

#pragma inline が指定された場合でも、以下のいずれかに該当する場合はインライン展開しません。

- 可変引数を持つ関数である。
- 関数内で仮引数のアドレスを参照している。
- 展開対象関数のアドレスを介して呼び出しを行っている。

#pragma inline は、インライン展開されることを保証するものではありません。コンパイル時間やメモリ使用量の増大を考慮した制限により、インライン展開を抑止することがあります。なお、インライン展開が抑止される際に、noscope オプションを指定すると、インライン展開されるようになる場合があります。

#pragma inline は、関数本体の定義の前に指定してください。

#pragma inline で指定した関数に対しても外部定義を生成します。static 関数に #pragma inline を指定した場合、関数定義はインライン展開後に削除されます。

C++ 言語のコンパイルでは、inline が指定された関数には、外部定義を生成しません。

C(C99) 言語のコンパイルでは、inline 指定された関数には、その関数に extern 宣言がなければ、外部定義を生成しません。

(5) アセンブラ記述関数インライン展開

#pragma inline_asm で宣言したアセンブリ記述関数をインライン展開します。

アセンブラ埋め込みインライン関数の呼び出し規則は通常関数の呼び出し規則と同様です。

例　C ソース

```
#pragma inline_asm func
static int func(int a, int b){
   ADD R2,R1; アセンブリ記述
}
main(int *p){
*p = func(10,20);
}
```

生成コード

```
_main:
   PUSH.L R6
   MOV.L R1, R6
   MOV.L #20, R2
   MOV.L #10, R1
   ADD  R2,R1; アセンブリ記述
   MOV.L R1, [R6]
   MOV.L #0, R1
   RTSD #04H, R6-R6
```

#pragma inline_asm[(]< 関数名 >[,…][)]

#pragma inline_asm は、関数本体の定義の前に指定してください。

#pragma inline_asm で指定した関数に対しても外部定義を生成します。

アセンブラ埋め込みインライン関数内で関数の出入口で保証するレジスタ（表 8―1　レジスタ使用規則を参照）を使用する場合は、アセンブラ埋め込みインライン関数の先頭と最後でこれらのレジスタの退避・回復が必要です。

アセンブラ埋め込みインライン関数には、RX ファミリの命令およびテンポラリラベルだけを記述してください。テンポラリラベル以外のラベルを定義したり、アセンブラ制御命令を記述することはできません。

アセンブラ埋め込みインライン関数の最後に RTS を記述しないでください。

関数名に関数メンバを指定することはできません。

static 関数に #pragma inline_asm を指定した場合、関数定義はインライン展開後に削除されます。

アセンブリ記述は、プリプロセッサの処理対象となります。このため、アセンブリ言語で使用される命令やレジスタと同じ名前のマクロ (例 : "MOV" や "R5" など) を #define でマクロ定義する場合はご注意ください。

(6) エントリ関数指定

< 関数名 > で指定した関数をエントリ関数として扱います。

エントリ関数では、レジスタの退避・回復コードを一切作成しません。

#pragma stacksize 宣言があると、関数先頭でスタックポインタの初期設定コードを出力します。

base オプションを指定した場合、オプションで指定したベースレジスタへの設定を行います。

例　C ソース：-base=rom=R13 を指定

```
#pragma stacksize su=100
#pragma entry INIT
void INIT() {
:
}
```

出力コード

```
.SECTION    SU,DATA,ALIGN=4
.BLKB       100
.SECTION    P,CODE
_INIT:
MVTC        (TOPOF SU + SIZEOF SU),USP
MOV.L       #__ROM_TOP,R13
```

```
#pragma entry[(]< 関数名 >[)]
```

#pragma entry **指定は、関数の宣言前に行ってください。**

ロードモジュール全体でエントリ関数を複数指定することはできません。

(7) ビットフィールド並び順指定

ビットフィールドの並び順の切り替えを指定します。

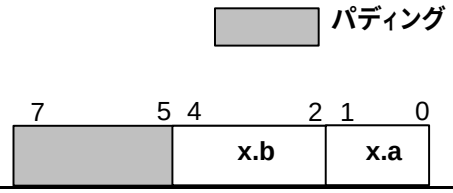
left を指定した場合は上位ビット側から、right を指定した場合は下位ビット側から、それぞれメンバが割り付けられます。

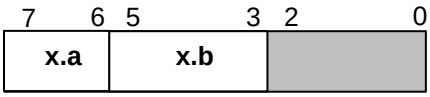
デフォルトの設定は right です。

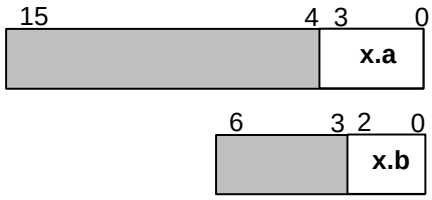
left|right を省略すると、以降はオプションに従います。

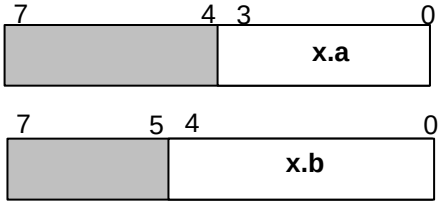
例

C ソース	ビット配置

#pragma bit_order right
struct tbl_r {
  unsigned char a:2;
  unsigned char b:3;
} x;
	

#pragma bit_order left
struct tbl_l {
  unsigned char a:2;
  unsigned char b:3;}
y;
	

// 異なるサイズのメンバの場合
#pragma bit_order right
struct tbl_r {
  unsigned short a:4;
  unsigned char b:3;
} x;
	

// 型のサイズを超える場合
#pragma bit_order right
struct tbl_r {
  unsigned char a:4;
  unsigned char b:5;
} x;
	

```
#pragma bit_order [{left | right}]
```

(8) 構造体 / クラスメンバのアライメント指定

ソースプログラム中の指定位置以降の構造体メンバおよびクラスメンバのアライメント数を指定します。本拡張子が指定されていない場合または #pragma packoption 指定位置以降で宣言された構造体メンバおよびクラスメンバのアライメント数は pack オプションの指定に従います。#pragma pack 拡張子とアライメント数の関係を表 3.24 に示します。

表 3―23 #pragma pack とメンバのアライメント数

メンバの型	#pragma pack	#pragma unpack	#pragma packoption または指定なし
(signed) char	1	1	1
(unsigned) short	1	2	pack オプションに従う
(unsigned) int *, (unsigned) long, (unsigned) long long, 浮動小数点型 , ポインタ型	1	4	pack オプションに従う

例

```
#pragma pack
struct S1 {
   char a;/* バイトオフセット =0*/
   int b;/* バイトオフセット =1*/
   char c;/* バイトオフセット =5*/
} ST1;/* 合計サイズ 6 バイト */

#pragma unpack
struct S2 {
   char a;/* バイトオフセット =0*/
/* 3 バイト空き領域 */
   int b;/* バイトオフセット =4*/
   char c;/* バイトオフセット =8*/
/* 3 バイト空き領域 */
} ST2;/* 合計サイズ 12 バイト */

#pragma pack
#pragma unpack
#pragma packoption
```

構造体、共用体、クラスメンバのアライメント数は pack オプションでも指定できます。オプションと #pragma の両方が指定された場合は、#pragma の指定を優先します。

(9) 変数の絶対アドレス割り付け指定

指定した変数を指定したアドレスに割り付けます。その際、コンパイラが指定した変数ごとにセクションを設定し、リンク時に指定した絶対アドレスに割り付けます。連続したアドレスに変数を指定した場合、それらの変数は同一セクションにします。

例 C ソース

```
#pragma address X=0x7f00
int X;
main(){
X=0;
}
```

出力コード

```
_main:
MOV.L       #0,R5
MOV.L       #7F00H,R14;
MOV.L       R5,[R14]
RTS
.SECTION    $ADDR_B_7F00,DATA
.ORG        7F00H
.glb        _X
_X:                             ; static: X
.blkl       1
```

#pragma address [(]< 変数名 >=< 絶対アドレス >[,…][)]

#pragma address 指定は、変数の宣言前に行ってください。

構造体 / 共用体のメンバ、もしくは変数以外を指定した場合はエラーとなります。

#pragma address を同一の変数に対して複数回指定した場合はエラーとなります。

(10) 初期値のエンディアン指定

静的オブジェクトを格納する領域のエンディアンを指定します。

#pragma endian を記述した行から、ファイルの末尾か、または次の #pragma endian を記述した行の手前までに定義したものが対象となります。

big を指定した場合は big endian になります。オプション指定が endian=little である場合は、セクション名の後に _B をつけたセクションに配置されます。

little を指定した場合は little endian になります。オプション指定が endian=big である場合は、セクション名の後に _L をつけたセクションに配置されます。

big | little を省略すると、以降はオプションに従います。

例 endian=little オプション指定時 (デフォルト)

C ソース :

```
#pragma endian big
int A=100;/* D_B セクション */
#pragma endian
int B=200;/* D セクション */
```

出力コード :

```
    .glb   _A
    .glb   _B
    .SECTION   D,ROMDATA,ALIGN=4
_B:
    .lword   200
    .SECTION  D_B,ROMDATA,ALIGN=4
    .ENDIAN  BIG
_A:
    .lword   100

#pragma endian [{big | little}]
```

endian オプションと異なる #pragma endian 対象のオブジェクトに、long long 型、double 型 (dbl_size=8 オプション指定時) および long double 型 (同) の領域を含む場合は、これらの領域に対するアドレスやポインタを用いた間接的なアクセスはしないでください。この場合の動作は保証されません。

もし、このような領域のアドレスを取得するコードを記述した場合は、警告を表示します。

endian オプションと異なる #pragma endian 対象のオブジェクトに、long long 型のビットフィールドを含む場合は、この領域に書き込みを行わないでください。この場合の動作は保証されません。

もし、このような領域への書き込みを行うコードを記述した場合は、警告を表示します。

次の項目のエンディアンは変更できません。endian オプションを使用してください。

(1) 文字列リテラルおよび集成体の動的初期化で用いる初期化子

(2) switch 文の分岐テーブル

(3) 外部参照宣言されたオブジェクト（初期化式なく extern 宣言されたオブジェクト）

(4) #pragma address により指定されたオブジェクト

（11）分岐先の命令実行向け整合を行う関数の指定

分岐先の命令実行向け整合を行う関数を指定します。

指定された関数に対し、#pragma instalign4 の場合は 4 バイト、#pragma instalign8 の場合は 8 バイトでそれぞれ配置アドレスを命令実行向け整合します。

#pragma noinstalign が指定された関数は、整合は行いません。

分岐先の種類は、以下から選択します (*1)。

指定なし : 関数先頭、switch 文の case および default ラベル

inmostloop: 各最内周ループの先頭、関数先頭、switch 文の case および default ラベル

loop: 各ループの先頭、関数先頭、switch 文の case および default ラベル

注 1. ここに挙げたもの以外の分岐先は、命令実行向け整合の対象ではありません。たとえば、loop を選択した場合はループの先頭は対象ですが、ループ内にあるループを構成しない if 文などの分岐先は対象ではありません。

それぞれ、指定した関数ごとに有効になることを除き、instalign4, instalign8, noinstalign オプションと機能は同じです。これらのオプションと同時に指定した場合は、#pragma の指定が優先されます。

instalign4 または instalign8 を選択した関数が属するコードセクションは、そのアライメント数は 4(instalign4 の場合) または 8(instalign8 の場合) に変わります。同じコードセクション内に、

instalign4 と instalign8 が指定された関数が混在する場合、そのコードセクションのアライメント数は 8 になります。

その他の内容については、instalign4, instalign8, noinstalign オプションと同様ですので、これらのオプションの項目を参照ください。

```
#pragma instalign4 [(]< 関数名 >[( ＜分岐先の種類＞ )][,…][)]
#pragma instalign8 [(]< 関数名 >[( ＜分岐先の種類＞ )][,…][)]
#pragma noinstalign [(]< 関数名 >[,…][)]
```

3. 2. 5 キーワードの使用方法

この項では、下記のキーワードの使用方法について説明します。

- 指定サイズのアクセス記述

(1) 指定サイズのアクセス記述

宣言または定義したサイズで変数をアクセスします。

変数の型のサイズでアクセスすることを保証します。

保証対象サイズは、4 バイト以下の整数スカラ型 (signed char, unsigned char, signed short, unsigned short, signed int, unsigned int, signed long, unsigned long) です。

例 C ソース

```
#pragma address A=0xff0178
unsigned long __evenaccess A;
void test(void)
{
    A &= ~0x20;
}
```

出力コード (__evenaccess 非指定時)

```
_test:
 MOV.L #16712056,R1
 BCLR #5,[R1]          ; 1 バイトメモリアクセス
 RTS
```

出力コード (__evenaccess 指定時)

```
_test:
 MOV.L #16712056,R1
 MOV.L [R1],R5         ; 4 バイトメモリアクセス
 BCLR  #5,R5
 MOV.L R5,[ R1]         ; 4 バイトメモリアクセス
 RTS
```

構造体または共用体に指定した場合、全てのメンバに __evenaccess を指定したのと同じ効果になります。その場合、4 バイト以下の整数スカラ型メンバのアクセスサイズは保証しますが、構造体または共用体単位でのアクセスサイズは保証しません。

__evenaccess < 型指定子 > < 変数名 >
< 型指定子 > __evenaccess < 変数名 >

3. 2. 6　組み込み関数

CCRX では、アセンブラ命令の一部を “ 組み込み関数 ” として C ソースに記述することができます。ただし，“ アセンブラ命令そのもの ” を記述するのではなく，RX で用意した関数の形式で記述します。これらの関数を使用した場合，出力コードは通常の関数呼び出しを行わず，対応するアセンブラを出力します。

表 3―24 組み込み関数の一覧

	項目	仕様	機能	ユーザモードの制限 *1
1	最大値・最小値	signed long max(signed long data1, signed long data2)	最大値の選択	○
2		signed long min(signed long data1, signed long data2)	最小値の選択	○
3	バイト並べ替え	unsigned long revl(unsigned long data)	ロングワードデータをバイトリバース	○
4		unsigned long revw(unsigned long data)	ロングワードデータをワード毎にバイトリバース	○
5	データ交換	void xchg(signed long *data1, signed long *data2)	データ交換	○
6	積和演算	long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)	積和演算 (バイト)	○
7		long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)	積和演算 (ワード)	○
8		long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)	積和演算 (ロングワード)	○
9	回転	unsigned long rolc(unsigned long data)	キャリーを含めて 1 ビット左回転	○
10		unsigned long rorc(unsigned long data)	キャリーを含めて 1 ビット右回転	○
11		unsigned long rotl(unsigned long data, unsigned long num)	左回転	○
12		unsigned long rotr (unsigned long data, unsigned long num)	右回転	○
13	特殊命令	void brk(void)	BRK 命令例外	○
14		void int_exception(signed long num)	INT 命令例外	○
15		void wait(void)	プログラム実行停止	×
16		void nop(void)	NOP 命令に展開	○
17	プロセッサ割り込み優先レベル (IPL)	void set_ipl(signed long level)	割り込み優先レベルの設定	×
18		unsigned char get_ipl(void)	割り込み優先レベルの参照	○
19	プロセッサステータスワード (PSW)	void set_psw(unsigned long data)	PSW の設定	△
20		unsigned long get_psw(void)	PSW の参照	○
21	浮動小数点ステータスワード (FPSW)	void set_fpsw(unsigned long data)	FPSW の設定	○
22		unsigned long get_fpsw(void)	FPSW の参照	○
23	ユーザスタックポインタ (USP)	void set_usp(void *data)	USP の設定	○
24		void *get_usp(void)	USP の参照	○

	項目	仕様	機能	ユーザモードの制限 *1
25	割り込みスタックポインタ (ISP)	void set_isp(void *data)	ISP の設定	△
26		void *get_isp(void)	ISP の参照	○
27	割り込みテーブルレジスタ (INTB)	void set_intb (void *data)	INTB の設定	△
28		void *get_intb(void)	INTB の参照	○
29	バックアップ PSW(BPSW)	void set_bpsw(unsigned long data)	BPSW の設定	△
30		unsigned long get_bpsw(void)	BPSW の参照	○
31	バックアップ PC(BPC)	void set_bpc(void *data)	BPC の設定	△
32		void *get_bpc(void)	BPC の参照	○
33	高速割り込みベクタレジスタ (FINTV)	void set_fintv(void *data)	FINTV の設定	△
34		void *get_fintv(void)	FINTV の参照	○
35	有効桁 64bit の乗算	signed long long emul(signed long data1, signed long data2)	有効桁 64bit の符号付き乗算	○
36		unsigned long long emulu(unsigned long data1, unsigned long data2)	有効桁 64bit の符号なし乗算	○
37	プロセッサモード (PM)	void chg_pmusr(void)	ユーザモードへの切り換え	△
38	アキュムレータ (ACC)	void set_acc(signed long long data)	ACC の設定	○
39		signed long long get_acc(void)	ACC の参照	○
40	割り込み許可ビットの制御	void setpsw_i(void)	割り込み許可ビットを 1 に設定	△
41		void clrpsw_i(void)	割り込み許可ビットを 0 に設定	△
42	積和演算	long macl(short *data1, short *data2, unsigned long count)	2byte データの積和演算	○
43		short macw1(short *data1, short *data2, unsigned long count) short macw2(short *data1, short *data2, unsigned long count)	固定小数点データ向けの積和演算	○

注 1. RX のプロセッサモードがユーザモードの場合に制限があるものを示します。
×印の関数は特権命令例外が発生するため、ユーザモードでは使用しないでください。
△印の関数はユーザモードで実行しても効果はありません。

```
signed long max(signed long data1, signed long data2)
```

[機能]

2 つの入力値のうち大きい方を選択します（MAX 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

data1 入力値1

data2 入力値2

[リターン値]

data1 と data2 のうち大きい方の値

[例]

```
#include < machine.h>
extern signed long ret,in1,in2;
void main(void);
{
ret = max(in1,in2);// in1 と in2 のうち大きい方の値を ret に設定します。
}
```

```
signed long min(signed long data1, signed long data2)
```

[機能]

2つの入力値のうち小さい方を選択します（MIN 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

data1 入力値1

data2 入力値2

[リターン値]

data1 と data2 のうち小さい方の値

[例]

```
#include < machine.h>
extern signed long ret,in1,in2;
void main(void)
{
ret = min(in1,in2); // in1 と in2 のうち小さい方の値を ret に設定します。
}
```

```
unsigned long revl(unsigned long data)
```

[機能]

4バイトデータのバイト並び順をリバースします（REVL 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

data バイト並びをリバースするデータ

[リターン値]

data のバイト並びをリバースした値

[例]

```
#include <machine.h>
extern unsigned long ret,indata=0x12345678;
void main(void)
{
ret = revl(indata);// ret=0x78563412 となる
}
```

```
unsigned long revw(unsigned long data)
```

[機能]

4 バイトデータの上位２バイトと下位２バイトでそれぞれのバイト並びをリバースします（REVW 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

data　バイト並びをリバースするデータ

[リターン値]

data の上位２バイトと下位２バイトでそれぞれのバイト並びをリバースした

[例]

```
#include <machine.h>
extern unsigned long ret;indata=0x12345678;
void main(void)
{
ret = revw(indata);// ret=0x34127856 となる
}
```

```
void xchg(signed long *data1, signed long *data2)
```

[機能]

引数が指す領域の内容を入れ替えます（XCHG 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

*data1　入力値１

*data2　入力値２

[リターン値]

－

[例]

```
#include <machine.h>
extern signed long *in1,*in2;
void main(void)
{
xchg (in1,in2);// アドレス in1、アドレス in2 のデータを交換します。
}
```

```
long long rmpab(long long init, unsigned long count, signed char *addr1, signed char *addr2)
```

[機能]

初期値を init、回数を count、乗数の格納されている先頭アドレスを addr1、addr2 として積和演算を行います（RMPA.B 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

init 初期値

count 積和演算の回数

*addr1 乗数 1 の先頭アドレス

*addr2 乗数 2 の先頭アドレス

[リターン値]

init + S(data1[n] * data2[n]) の下位 64 ビットの結果 (n=0, 1, …, const-1)

[例]

```
#include <machine.h>
extern signed char data1[8],data2[8];
long long sum;
void main(void)
{
sum=rmpab(0, 8, data1, data2);// 0 を初期値とし、配列 data1,data2 の
                              // 乗算結果を加算して sum に設定する
}
```

[備考]

RMPA 命令は 80bit の範囲で結果を計算しますが、本組み込み関数では 64bit 範囲のみ扱います。

```
long long rmpaw(long long init, unsigned long count, short *addr1, short *addr2)
```

[機能]

初期値を init、回数を count、乗数の格納されている先頭アドレスを addr1、addr2 として積和演算を行います（RMPA.W 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

init 初期値

count 積和演算の回数

*addr1 乗数 1 の先頭アドレス

*addr2 乗数 2 の先頭アドレス

[リターン値]

init + S(data1[n] * data2[n]) の下位 64 ビットの結果 (n=0, 1, …, const-1)

[例]

```
#include <machine.h>
extern signed short data1[8],data2[8];
long long sum;
void main(void)
{
sum=rmpaw(0, 8, data1, data2); // 0 を初期値とし、配列 data1,data2 の
                               // 乗算結果を加算して sum に設定する
}
```

[備考]

RMPA 命令は 80bit の範囲で結果を計算しますが、本組み込み関数では 64bit 範囲のみ扱います。

```
long long rmpal(long long init, unsigned long count, long *addr1, long *addr2)
```

[機能]

初期値を init、回数を count、乗数の格納されている先頭アドレスを addr1、addr2 として積和演算を行います（RMPA.L 命令に展開します）。

[ヘッダ]

<machine.h>

[引数]

init 初期値

count 積和演算の回数

*addr1 乗数 1 の先頭アドレス

*addr2 乗数 2 の先頭アドレス

[リターン値]

init + S(data1[n] * data2[n]) の下位 64 ビットの結果 (n=0, 1, …, const-1)

[例]

```
#include <machine.h>
extern signed long data1[8],data2[8];
long long sum;
void main(void)
{
sum=rmpaw(0, 8, data1, data2); // 0 を初期値とし、配列 data1,data2 の
                               // 乗算結果を加算して sum に設定する
}
```

```
unsigned long rolc(unsigned long data)
```

[機能]

C フラグを含めて 1 ビット左回転した結果を返します（ROLC 命令に展開します）。オペランドの外へ出たビットを C フラグに反映します。

[ヘッダ]

<machine.h>

[引数]

data 左回転するデータ

[リターン値]

data を C フラグを含めて左に１ビット回転した結果

[例]

```
#include <machine.h>
extern unsigned long ret;indata;
void main(void)
{
ret = rolc(indata);// indata を C フラグを含めて 1 ビット左回転し
                   // ret に設定します。
}
```

```
unsigned long rorc(unsigned long data)
```

[機能]

C フラグを含めて 1 ビット右回転した結果を返します（RORC 命令に展開します）。オペランドの外へ出たビットを C フラグに反映します。

[ヘッダ]

<machine.h>

[引数]

data 右回転するデータ

[リターン値]

data を C フラグを含めて右に１ビット回転した結果

[例]

```
#include <machine.h>
extern unsigned long ret;indata;
void main(void)
{
ret = rorc(indata);// indata を C フラグを含めて 1 ビット右回転し
                   // ret に設定します。
}
```

```
unsigned long rotl(unsigned long data, unsigned long num)
```

[機能]

任意ビット左回転した結果を返します（ROTL 命令に展開します）。オペランドの外へ出たビットを C フラグに反映します。

[ヘッダ]

<machine.h>

[引数]

data 左回転するデータ

num 回転するビット数

[リターン値]

data を左に num ビット回転した結果

[例]

```
#include <machine.h>
extern unsigned long ret;indata;
void main(void)
{
ret = rotl(indata, 31); // indata を 31 ビット左回転し
                        // ret に設定します。
}
```

```
unsigned long rotr (unsigned long data, unsigned long num)
```

[機能]

任意ビット右回転した結果を返します（ROTR 命令に展開します）。オペランドの外へ出たビットを C フラグに反映します。

[ヘッダ]

<machine.h>

[引数]

data 右回転するデータ

num 回転するビット数

[リターン値]

data を右に num ビット回転した結果

[例]

```
#include <machine.h>
extern unsigned long ret;indata;
void main(void)
{
ret = rotr(indata, 31); // indata を 31 ビット右回転し
                        // ret に設定します。
}
```

```
void brk(void)
```

[機能]

BRK 命令に展開します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
brk();// BRK 命令
}
```

```
void int_exception(signed long num)
```

[機能]

INT num 命令に展開します。

[ヘッダ]

<machine.h>

[引数]

num　INT 命令番号

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
int_exception(10);// INT #10 命令
}
```

[備考]

num に設定できる数は、0 ～ 255 の整数のみです。

```
void wait(void)
```

[機能]

WAIT 命令に展開します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
wait();// WAIT 命令
}
```

[備考]

本関数は RX のプロセッサモードがユーザモードの場合は実行しないでください。実行すると、WAIT 命令の仕様により、RX の特権命令例外が発生します。

```
void nop(void)
```

[機能]

NOP 命令に展開します。

[ヘッダ]

<machine.h>

[引数]

ー

[リターン値]

ー

[例]

```
#include <machine.h>
void main(void)
{
nop();// NOP 命令
}
```

```
void set_ipl(signed long level)
```

[機能]

割り込みマスクレベルを変更します。

[ヘッダ]

<machine.h>

[引数]

level　設定する割り込みマスクレベル

[リターン値]

ー

[例]

```
#include <machine.h>
void main(void)
{
set_ipl(7);// PSW.IPL に 7 を設定
}
```

[備考]

デフォルトでは level には 0 ～ 15 の値が、-patch=rx610 を指定した場合は 0 ～ 7 の値がそれぞれ指定できます。

level が定数のとき、範囲外の値を指定した場合はエラーとなります。

本関数は RX のプロセッサモードがユーザモードの場合は使用しないでください。実行すると、MVTIPL 命令の仕様により、RX の特権命令例外が発生します。

```
unsigned char get_ipl(void)
```

[機能]

割り込みマスクレベルを参照します。

[ヘッダ]

<machine.h>

[引数]

ー

[リターン値]

割り込みマスクレベル

[例]

```
#include <machine.h>
extern unsigned char level;
void main(void)
{
level=get_ipl();// PSW.IPL の値を取得し level に設定
}
```

```
void set_psw(unsigned long data)
```

[機能]

PSW を設定します。

[ヘッダ]

<machine.h>

[引数]

data 設定値

[リターン値]

ー

[例]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
set_psw(data);// PSW に data の値を設定
}
```

[備考]

RX の命令セット仕様のため、PSW の PM ビットの書き込みは無視されます。また、RX のプロセッサモードがユーザモードの場合は、PSW への書き込みは無視されます。

```
unsigned long get_psw(void)
```

[機能]

PSW を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

PSW の値

[例]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
ret=get_psw();// PSW の値を取得し、ret に設定
}
```

[備考]

最適化の作用により、get_psw の呼び出し箇所とは違うタイミングで PSW レジスタの値が取得される場合があります。このため、何らかの演算後に、本関数の戻り値に含まれる C,Z,S および O フラグのいずれかを利用するコードを記述した場合、その動作は保証されません。

```
void set_fpsw(unsigned long data)
```

[機能]

FPSW を設定します。

[ヘッダ]

<machine.h>

[引数]

data 設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
set_fpsw(data);// FPSW に data の値を設定
}
```

```
unsigned long get_fpsw(void)
```

[機能]

FPSW を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

FPSW の値

[例]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
ret=get_fpsw();// FPSW の値を取得し、ret に設定
}
```

[備考]

最適化の作用により、get_fpsw の呼び出し箇所とは違うタイミングで FPSW レジスタの値が取得される場合があります。このため、何らかの演算後に、本関数の戻り値に含まれる CV,CO,CZ,CU,CX,CE,FV,FO,FZ,FU,FX および FS フラグのいずれかを利用するコードを記述した場合、その動作は保証されません。

```
void set_usp(void *data)
```

[機能]

USP を設定します。

[ヘッダ]

<machine.h>

[引数]

data 設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern void * data;
void main(void)
{
set_usp(data);// USP に data の値を設定
}
```

```
void *get_usp(void)
```

[機能]

USP を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

USP の値

[例]

```
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_usp();// USP の値を取得し、ret に設定
}
```

```
void set_isp(void *data)
```

[機能]

ISP を設定します。

[ヘッダ]

<machine.h>

[引数]

data　設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern void * data;
void main(void)
{
set_isp(data);// ISP に data の値を設定
}
```

[備考]

本関数で使用する MVTC 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、ISP への書き込みは無視されます。

```
void *get_isp(void)
```

[機能]

ISP を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

ISP の値

[例]

```
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_isp();// ISP の値を取得し、ret に設定
}
```

```
void set_intb (void *data)
```

[機能]

INTB を設定します。

[ヘッダ]

<machine.h>

[引数]

data　設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern void * data;
void main(void)
{
set_intb (data);// INTB に data の値を設定
}
```

[備考]

本関数で使用する MVTC 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、INTB への書き込みは無視されます。

```
void *get_intb(void)
```

[機能]

INTB を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

INTB の値

[例]

```
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_intb();// INTB の値を取得し、ret に設定
}
```

```
void set_bpsw(unsigned long data)
```

[機能]

BPSW を設定します。

[ヘッダ]

<machine.h>

[引数]

data　設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern unsigned long data;
void main(void)
{
set_bpsw (data);// BPSW に data の値を設定
}
```

[備考]

本関数で使用する MVTC 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、BPSW への書き込みは無視されます。

```
unsigned long get_bpsw(void)
```

[機能]

BPSW を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

BPSW の値

[例]

```
#include <machine.h>
extern unsigned long ret;
void main(void)
{
ret=get_bpsw ();// BPSW の値を取得し、ret に設定
}
```

```
void set_bpc(void *data)
```

[機能]

BPC を設定します。

[ヘッダ]

<machine.h>

[引数]

data 設定値

[リターン値]

ー

[例]

```
#include <machine.h>
extern void * data;
void main(void)
{
set_bpc(data);// BPC に data の値を設定
}
```

[備考]

本関数で使用する MVTC 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、BPC への書き込みは無視されます。

```
void *get_bpc(void)
```

[機能]

BPC を参照します。

[ヘッダ]

<machine.h>

[引数]

ー

[リターン値]

BPC の値

[例]

```
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_bpc();// BPC の値を取得し、ret に設定
}
```

```
void set_fintv(void *data)
```

[機能]

FINTV を設定します。

[ヘッダ]

<machine.h>

[引数]

data 設定値

[リターン値]

―

[例]

```
#include <machine.h>
extern void * data;
void main(void)
{
set_fintv(data);// FINTV に data の値を設定
}
```

[備考]

本関数で使用する MVTC 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、FINTV への書き込みは無視されます。

```
void *get_fintv(void)
```

[機能]

FINTV を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

FINTV の値

[例]

```
#include <machine.h>
extern void * ret;
void main(void)
{
ret=get_fintv();// FINTV の値を取得し、ret に設定
}
```

```
signed long long emul(signed long data1, signed long data2)
```

[機能]

有効桁 64bit の符号付き乗算を行います。

[ヘッダ]

<machine.h>

[引数]

data1 入力値 1

data2 入力値 2

[リターン値]

符号付き乗算の結果（64bit 符号付き）

[例]

```
#include <machine.h>
extern signed long long ret;
extern signed long data1, data2;
void main(void)
{
ret=emul(data1, data2);// data1 * data2 の値を計算し、ret に設定
}
```

```
unsigned long long emulu(unsigned long data1, unsigned long data2)
```

[機能]

有効桁 64bit の符号なし乗算を行います。

[ヘッダ]

<machine.h>

[引数]

data1 入力値 1

data2 入力値 2

[リターン値]

符号なし乗算の結果（64bit 符号なし）

[例]

```
#include <machine.h>
extern unsigned long long ret;
extern unsigned long data1, data2;
void main(void)
{
ret=emulu(data1, data2);// data1 * data2 の値を計算し、ret に設定
}
```

```
void chg_pmusr(void)
```

[機能]

RX のプロセッサモードをユーザに切り換えます。

[ヘッダ]

<machine.h>

[引数]

ー

[リターン値]

ー

[例]

```
#include <machine.h>
void main(void);
void Do_Main_on_UserMode(void)
{
chg_pmusr();  // プロセッサモードをユーザモードに切り換え
main();       // main を実行
}
```

[備考]

本関数はリセット処理関数あるいは割り込み関数のために用意されています。それ以外の関数での使用は推奨しません。

RX のプロセッサモードがユーザモードのときは、プロセッサモードは切り替わりません。

chg_pmusr 関数の実行により、スタックは割り込みスタックからユーザスタックに切り換えが起こりますので、本関数を呼び出す関数では、必ず次の条件を守ってください。守られない場合、本関数の実行前後のスタックの相違が起こるため、コードは正常に動作しません。

- 呼び出し元に return することはできません。
- auto 変数を宣言することはできません。
- 引数を宣言することはできません。

```
void set_acc(signed long long data)
```

[機能]

ACC を設定します。

[ヘッダ]

<machine.h>

[引数]

data ACC への設定値

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
signed long long data = 0x123456789ab0000LL;
set_acc(data);// ACC に data の値を設定
}
```

```
signed long long get_acc(void)
```

[機能]

ACC を参照します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

ACC の値

[例]

```
/* get_acc と set_acc による ACC の退避 / 回復を利用したプログラムの例 */
#include <machine.h>
signed long func(signed long a, signed long b)
{
signed long long bak_acc = get_acc(); // ACC の値を取得し、bak_acc に退避
a *= b;               // 乗算 (ACC を破壊する )
set_acc(bak_acc);     // bak_acc で退避していた値で ACC を回復
return a;
}
```

[備考]

RX 命令セットの仕様のため、ACC の下位 16 ビットは取得できません。本関数は、これらのビットには 0 を返します。

```
void setpsw_i(void)
```

[機能]

PSW の割り込み許可ビット (I ビット) を 1 に設定します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
setpsw_i();// 割り込み許可ビットを 1 にする
}
```

[備考]

本関数で使用する SETPSW 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、割り込み許可ビットへの書き込みは無視されます。

```
void clrpsw_i(void)
```

[機能]

PSW の割り込み許可ビット (I ビット) を 0 に設定します。

[ヘッダ]

<machine.h>

[引数]

―

[リターン値]

―

[例]

```
#include <machine.h>
void main(void)
{
clrpsw_i();// 割り込み許可ビットを 0 にする
}
```

[備考]

本関数で使用する CLRPSW 命令の仕様により、RX のプロセッサモードがユーザモードの場合は、割り込み許可ビットへの書き込みは無視されます。

```
long macl(short *data1, short *data2, unsigned long count)
```

[機能]

積和演算を行います。

2byte のデータ同士の積和演算を行い、その結果を 4byte で返します。

積和演算は、DSP 機能命令 (MULLO,MACLO,MACHI) を使用して演算します。

積和演算の途中のデータは、ACC に 48bit 長データとして保持されます。

全ての積和演算が終わったら、ACC の内容を MVFACHI 命令で取り出し、組み込み関数の戻り値とします。

本組み込み関数を使用することにより、組み込み関数を使用せずに積和演算を記述する場合よりも、高速な積和演算処理が期待できます。

2byte の整数データの積和演算をする場合に利用できます。積和演算の演算結果には、飽和処理や丸め処理はされません。

[ヘッダ]

<machine.h>

[引数]

data1 乗数 1 の先頭アドレス

data2 乗数 2 の先頭アドレス

count 積和演算の乗算回数

[リターン値]

Σ(data1[n] × data2[n]) の演算結果

[例]

```
#include <machine.h>
short data1[3] = {a1, b1, c1};
short data2[3] = {a2, b2, c2};
void mac_calc()
{
result = macl(data1, data2, 3);    /* a1*a2+b1*b2+c1*c2 の結果を求めます */
}
```

[備考]

積和演算に使用される各種 DSP 機能命令の詳細な内容を確認するには、プログラミングマニュアルを参照してください。

乗算回数が 0 の場合、組み込み関数の戻り値は 0 です。

本組み込み関数を使用する場合、ACC の内容が書き換わる割り込み処理では、ACC を退避回復してください。

ACC を退避回復する機能については、コンパイルオプションの save_acc、もしくは、拡張言語仕様の #pragma interrupt を参照してください。

```
short macw1(short *data1, short *data2, unsigned long count)
short macw2(short *data1, short *data2, unsigned long count)
```

[機能]

2byte のデータ同士の積和演算を行い、その結果を 2byte で返します。

積和演算は、DSP 機能命令 (MULLO,MACLO,MACHI) を使用して演算します。

積和演算の途中のデータは、ACC に 48bit 長データとして保持されます。

全ての積和演算が終わった後に、ACC の積和演算結果に対して丸め処理を行います。

macw1 関数は "RACW #1" 命令、macw2 関数は "RACW #2" 命令で丸め処理を行います。

丸め処理の処理内容は、以下の手順になります。

- ACC の内容を、macw1 関数は 1 ビット、macw2 関数は 2 ビット、左シフトします
- ACC の下位 32 ビットの最上位ビットを 0 捨 1 入します
- ACC の上位 32 ビットを、上限値 0x00007FFF、下限値を 0xFFFF8000 として飽和処理します

最後に、MVFACHI 命令で ACC から上位 32 ビットを取得し、組み込み関数の戻り値とします。

通常、固定小数点データ同士の乗算をする場合、乗算結果の小数点位置を調整する必要があります。たとえば、Q15 形式の固定小数点データ同士の乗算の場合、乗算結果を同じ Q15 形式にするためには、乗算結果を 1 ビット左シフトする必要があります。
この小数点位置を調整するための左シフトは、RACW 命令の左シフト動作によって実現できます。そのため、2byte の固定小数点データの積和演算をする場合に、本組み込み関数を利用することにより、容易に積和演算処理を実現できます。なお、macw1 と macw2 は演算結果の丸め方法が異なりますので、演算結果に求められる精度に応じて、使用する組み込み関数を選択してください。

[ヘッダ]

<machine.h>

[引数]

data1 乗数 1 の先頭アドレス

data2 乗数 2 の先頭アドレス

count 積和演算の乗算回数

[リターン値]

積和演算の演算結果を、RACW 命令で丸めた値

[例]

```
#include <machine.h>
short data1[3] = {a1, b1, c1};
short data2[3] = {a2, b2, c2};
void mac_calc()
{
result = macw1(data1, data2, 3);
/* a1*a2+b1*b2+c1*c2 の結果を、"RACW #1" 命令で丸めた値を求めます。 */
}
```

[備考]

積和演算に使用される各種 DSP 機能命令の詳細な内容を確認するには、プログラミングマニュアルを参照してください。
乗算回数が 0 の場合、組み込み関数の戻り値は 0 です。
本組み込み関数を使用する場合、ACC の内容が書き換わる割り込み関数では、ACC を退避回復してください。
ACC を退避回復する機能については、コンパイルオプションの save_acc、もしくは、拡張言語仕様の #pragma interrupt を参照してください。

3. 2. 7 セクションアドレス演算子

セクションアドレス演算子の一覧を表 3.26 に示します。

表 3―25 セクション演算子一覧

	セクション演算子名	説明
1	__sectop("< セクション名 >")	__sectop で指定した < セクション名 > の先頭アドレスを参照
2	__secend("< セクション名 >")	__secend で指定した < セクション名 > の末尾 +1 アドレスを参照

3	__secsize("< セクション名 >")	__secsize で指定した < セクション名 > のサイズを生成

__sectop("< セクション名 >")
__secend("< セクション名 >")
__secsize("< セクション名 >")

[機能]

__sectop で指定した < セクション名 > の先頭アドレスを参照します。

__secend で指定した < セクション名 > の末尾 +1 アドレスを参照します。

__secsize で指定した < セクション名 > のサイズを生成します。

[型]

__sectop の型は、void * です。

__secend の型は、void * です。

__secsize の型は、unsigned long です。

[例]

例 1: __sectop, __secend

```
#include <machine.h>
#pragma section $DSEC
static const struct {
    void *rom_s; /* 初期化データセクションの ROM 上の先頭アドレス */
    void *rom_e; /* 初期化データセクションの ROM 上の最終アドレス */
    void *ram_s; /* 初期化データセクションの RAM 上の先頭アドレス */
} DTBL[]={__sectop("D"), __secend("D"), __sectop("R")};

#pragma section $BSEC
static const struct {
    void *b_s; /* 未初期化データセクションの先頭アドレス */
    void *b_e; /* 未初期化データセクションの最終アドレス */
} BTBL[]={__sectop("B"), __secend("B")};
#pragma section
#pragma stacksize si=0x100
#pragma entry INIT
void main(void);
void INIT(void)
{
    _INITSCT();
    main();
    sleep();
}
```

例 2: __secsize

```
/* size of section B */
unsigned int size_of_B = __secsize("B");
```

[備考]

PIC/PID 機能が有効なアプリケーションの場合、__sectop および __secend はリンク時のアドレスで処理します。

PIC/PID 機能の詳細は、pic および pid オプションと、「8.4 PIC/PID 機能の利用」の項目をそれぞれ参照してください。

3.3　C ソースの修正

拡張機能を使用することにより，効率の良いオブジェクトを生成することができます。

ここでは，他のコンパイラから CCRX への移植と，CCRX から他の C コンパイラへの移植の 2 つの場合について，その方法を説明します。

＜他のコンパイラから CCRX ＞

- #pragma

  他のコンパイラが #pragma をサポートしている場合は，ソースを修正する必要があります。修正方法は，そのコンパイラの仕様によって検討します。

- 拡張仕様

  他のコンパイラがキーワードを追加するなどの仕様の拡張を行っている場合は，修正する必要があります。修正方法はそのコンパイラの仕様によって検討します。

注　ANSI でサポートされている前処理指令の 1 つで，#pragma に続く文字列をコンパイラへの指令として認識させるものです。その指令がコンパイラによってサポートされていなければ，#pragma 指令は無視され，コンパイルが続けられて正常に終了します。

＜ CCRX から他の C コンパイラ＞

- CCRX は，拡張機能としてキーワードの追加を行っているため，他の C コンパイラへ移植するためには，キーワードを削除するか，#ifdef で切り分けなければなりません。

例 1.　キーワードを無効にする

```
#ifndef __RX
#define interrupt       /*interrupt 関数を通常の関数にします */
#endif
```

2.　他の型に変更する

```
#ifdef __RX
#define bit char        /*bit 型変数を char 型変数にします */
#endif
```

3.4　関数呼び出しインタフェース

この節では，CCRX におけるプログラム呼び出し時の引数などの扱い方について説明します。

- スタックに関する規則
- レジスタに関する規則
- 引数の設定、参照に関する規則
- リターン値の設定、参照に関する規則

- 外部名の相互参照方法

3. 4. 1 スタックに関する規則

(1) スタックポインタ

スタックポインタの指すアドレスよりも下位 (0 番地の方向) のスタック領域に有効なデータを格納してはいけません。スタックポインタより下位アドレスに格納されたデータは、割り込み処理で破壊される可能性があります。

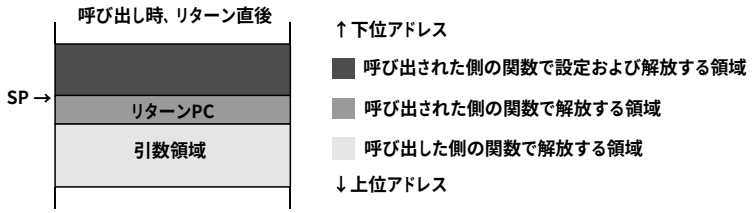
(2) スタックフレームの割り付け、解放

関数呼び出しが行われた時点 (JSR または BSR 命令の実行直後) では、スタックポインタは呼び出した関数側で使用したスタックの最下位アドレスを指しています。このスタックポインタの指している領域より上位アドレスのデータの割り付け、設定は呼び出す側の関数の役目です。

関数のリターン時は、呼び出された関数で確保した領域を解放してから、通常 RTS 命令を用いて呼び出した関数へ返ります。これより上位アドレスの領域 (リターン値アドレスおよび引数の領域) は、呼び出した側の関数で解放します。

図 3.2 は、関数呼び出し直後のスタックフレームの状態を説明したものです。

図 3―2 スタックフレームの割り付け、解放に関する規則



3. 4. 2 レジスタに関する規則

関数呼び出し前後において、レジスタの値が同一であることを保証するかどうかは、レジスタにより異なります。また、オプションにより特定の用途向けに使用するレジスタがあります。レジスタの使用規則を表 3.27 に示します。

表 3―26 レジスタ使用規則

レジスタ	関数呼び出し前後で値を保証	関数入口	関数出口	高速割り込み用レジスタ *1	ベースレジスタ *2	PID レジスタ *3
R0	保証する	スタックポインタ	スタックポインタ	-	-	-
R1	保証しない	引数 1	戻り値 1	-	-	-
R2	保証しない	引数 2	戻り値 2	-	-	-
R3	保証しない	引数 3	戻り値 3	-	-	-
R4	保証しない	引数 4	戻り値 4	-	-	-

レジスタ	関数呼び出し前後で値を保証	関数入口	関数出口	高速割り込み用レジスタ *1	ベースレジスタ *2	PIDレジスタ *3
R5	保証しない	-	(不定)	-	-	-
R6	保証する	-	(入口の値を保持)	-	-	-
R7	保証する	-	(入口の値を保持)	-	-	-
R8	保証する	-	(入口の値を保持)	-	○	-
R9	保証する	-	(入口の値を保持)	-	○	○
R10	保証する	-	(入口の値を保持)	○	○	○
R11	保証する	-	(入口の値を保持)	○	○	○
R12	保証する	-	(入口の値を保持)	○	○	○
R13	保証する	-	(入口の値を保持)	○	○	○
R14	保証しない	-	(不定)	-	-	-
R15	保証しない	構造体戻り値へのポインタ	(不定)	-	-	-
ISP USP	スタックポインタの場合は R0 と同じ。 そうでない場合は変化しません。*4			-	-	-
PC	-	プログラムカウンタ *5		-	-	-
PSW	保証しない	-	(不定)	-	-	-
FPSW	保証しない	-	(不定)	-	-	-
ACC	保証しない *6	-	(不定) *6	-	-	-
INTB BPC BPSW FINTV CPEN	-	変化しません *4	-	-	-	-

注 1. R10 ～ R13 の 4 本は、fint_register オプションにより、一部または全部が「高速割り込み機能」に使われることがあります。「高速割り込み機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。

2. R8 ～ R13 の 6 本は、base オプションにより、一部または全部が「ベースレジスタ機能」に使われることがあります。「ベースレジスタ機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。
3. R9 ～ R13 のうちの 1 本は、pid オプションにより「PID 機能」に使われることがあります。「PID 機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。
4. 組み込み関数または #pragma inline_asm で、これらのレジスタを設定したり更新したりする場合を除きます。
5. 関数の呼び出しに使用する命令の仕様に従います。関数の呼び出しには、BSR, JSR, BRA および JMP のいずれかの命令を用います。
6. ACC(アキュムレータ) を更新する命令は、RX のソフトウェアマニュアルを参照してください。

3. 4. 3　引数の設定、参照に関する規則

引数に対する一般的な規則と、引数の割り付け方について述べます。

引数が実際どのように割り付けられるかは、「8.2.5　引数割り付けの具体例」を参照ください。

(1) 引数の渡し方

引数の値を、必ずレジスタまたはスタック上の引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出した側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出した側の処理は直接には影響を受けません。

(2) 型変換の規則

(a) 関数原型によって型が宣言されている引数は、宣言された型に変換します。

(b) 関数原型によって型が宣言されていない引数の型変換は、以下の規則に従って変換します。

- 2 バイト以下の整数型は、4 バイト整数型に変換されます。
- float 型の引数は、double 型に変換します。
- 上記以外の引数は、変換しません。

例

```
void p(int,... );
void f( )
{
  char c;
  p(1.0, c);
}
```

cは、対応する引数の型宣言がないので、4バイト整数型に変換されます。

1.0は、対応する引数の型がint型なので、4バイト整数型に変換されます。

(3) 引数の割り付け領域

引数は、レジスタに割り付ける場合とスタック上の引数領域に割り付ける場合があります。引数の割り付け領域を図 3―3　に示します。

通常、ソースプログラムにおける引数の宣言順に、番号の小さいレジスタから順に割り付けを行い、レジスタが全て割り付いたらスタックに割り付けます。但し、可変個の引数を持つ関数など、レジスタが余っていてもスタックに割り付ける場合もあります。また、C++ プログラムの非静的関数メンバの this ポインタは、常に R1 に割り付けられます。

引数割り付け領域の一般規則を表 3.28 にそれぞれ示します。

図 3―3 スタックフレームの割り付け、解放に関する規則

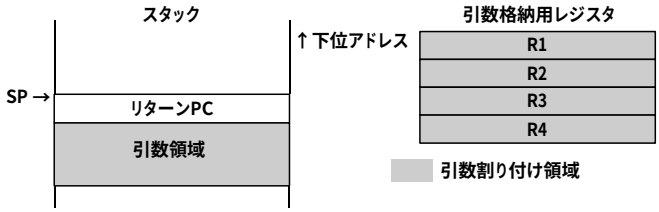


表 3―27 引数割り付け領域の一般規則

レジスタで渡される引数			スタック渡しになる引数
対象の型	引数格納用レジスタ	割り付け方	
signed char, (unsigned)char, bool, _Bool, (signed)short, unsigned short, (signed)int, unsigned int, (signed)long, unsigned long, float、double*1, long double*1, ポインタ , データメンバへのポインタ , リファレンス	R1 ～ R4 のうち 1 つ	signed char, (signed)short は符号拡張、(unsigned)char, unsigned short はゼロ拡張を行った結果を割り付ける その他の型はそのままレジスタに割り付ける	(1) 引数の型がレジスタ渡しの対象の型以外のもの (2) 関数原型により可変個の引数を持つ関数として宣言しているもの *3 (3) R1 ～ R4 のうち、まだ他の引数に割り当てられていないものの本数が、割り当てに必要な本数より少ない場合
(signed)long long, unsigned long long, double*2, long double*2	R1 ～ R4 のうち 2 つ	下位 4 バイトを番号の少ない方に、上位 4 バイトを番号の大きい方に割り付ける	
16 バイト以内でサイズが 4 の倍数の構造体型、共用体型、クラス型	R1 ～ R4 のうち、サイズを 4 で割った数	メモリイメージの先頭から 4 バイトずつ、レジスタ番号が増える方向に割り付ける	

注 1. dbl_size=8 を指定しなかった場合です。

2. dbl_size=8 を指定した場合です。

3. 関数原型により可変個の引数をもつ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタック渡しになります。型のない引数は、2 バイト以下の整数は long 型に、float 型は double 型にそれぞれ変換して、全て境界調整数が 4 の引数として取り扱います。

例

```
int f2(int,int,int,int,...);
:
f2(a,b,c,x,y,z); → x、y、z はスタック渡しになります。
```

(4) スタック渡しとなる引数の割り付け方

表 3.28 で、スタック渡しとなる引数の、配置アドレス、およびスタックへの配置の仕方は以下となります。

- 各引数は、その境界調整数に応じたアドレスに配置します。

- 引数並びの左から右の順に、スタックが深くなる方向に配置されるように、スタックの引数用領域に格納します。すなわち、引数 A とその右隣の引数 B がともにスタック渡しとなる場合、引数 B のアドレスは、引数 A の配置アドレスに引数 A の占有サイズを加えたアドレスを、引数 B の境界調整数に整合させたアドレス、となります。

3. 4. 4 リターン値の設定、参照に関する規則

リターン値に対する一般的な規則と、リターン値の設定場所について述べます。

(1) リターン値の型変換

リターン値は、その関数の返す型に変換します。

例

```
long f();
long f()
{
    float x;
    return x; // 関数原型にしたがってリターン値は long 型に変換されます。
}
```

(2) リターン値の設定場所

関数のリターン値の型によっては、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は表 3.29 を参照してください。

表 3―28 リターン値の型と設定場所

No.	リターン値の型	リターン値の設定場所
1	singed char, (unsigned)char, (singed)short, unsigned short, (singed)int, unsigned int, (signed)long, unsigned long, float, double*2, long double*2, ポインタ , bool, _Bool, リファレンス , データメンバへのポインタ	R1 但し、signed char, (signed)short は符号拡張、(unsigned)char, unsigned short はゼロ拡張を行った結果を設定
2	double*3, long double*3, (signed)long long, unsigned long long	R1, R2 下位 4 バイトを R1 に、上位 4 バイトを R2 に設定
3	16バイト以内かつ4の倍数であるサイズの構造体、共用体、クラス型	メモリイメージの先頭から 4 バイトずつ R1,R2,R3,R4 の順に設定
4	3. 以外の構造体、共用体、クラス型	リターン値設定領域 (メモリ)*1

注 1. 関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスを R15 に設定してから関数を呼び出します。

2. dbl_size=8 を指定しなかった場合です。

3. dbl_size=8 を指定した場合です。

3. 4. 5 外部名の相互参照方法

C/C++ プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。コンパイラは、次のものを外部名として扱います。

- 大域変数であって、かつ static 記憶クラスでないもの (C/C++ プログラム)
- extern 記憶クラスで宣言されている変数名 (C/C++ プログラム)
- static 記憶クラスを指定されていない関数名 (C プログラム)
- static 記憶クラスを指定されてない非メンバ非インライン関数名 (C++ プログラム)
- 非インラインメンバ関数名 (C++ プログラム)
- 静的データメンバ名 (C++ プログラム)

(1) アセンブリプログラムの外部名を C/C++ プログラムで参照する方法

アセンブリプログラムでは、.glb を用いてシンボル名 (先頭に下線 "_" を付与) を外部定義宣言します。

C/C++ プログラムでは、シンボル名 (先頭に下線 "_" がない) を「extern」宣言します。

アセンブリプログラム (定義する側)	C/C++ プログラム (参照する側)

.glb _a, _b
.SECTION D,ROMDATA,ALIGN=4
a: .LWORD 1
b: .LWORD 1
.END
	extern int a,b;
	void f()
	{
	a+=b;
	}

(2) C/C++ プログラムの外部 (変数および C 関数) 名をアセンブリプログラムから参照する方法

C/C++ プログラムでは、変数名 (先頭に下線 "_" がない) を外部定義します。

アセンブリプログラムでは、.IMPORT を用いて外部名 (先頭に下線 "_" を付与) を外部参照宣言します。

C/C++ プログラム (定義する側)	アセンブリプログラム (参照する側)
int a;	.GLB _a .SECTION P,CODE MOV.L #A_a,R1 MOV.L [R1],R2 ADD #1,R2 MOV.L R2,[R1] RTS .SECTION D,ROMDATA,ALIGN=4 A_a: .LWORD _a .END

(3) C++ プログラムの外部 (関数) 名をアセンブリプログラムから参照する方法

アセンブリプログラムで参照する関数を「extern "C"」を用いて宣言することにより、(2) と同じ規則で参照できます。ただし、「extern "C"」を用いて宣言した関数は多重定義できません。

C++ プログラム (呼び出される側)

```
extern "C"
void sub ( )
{
    :
}
```

アセンブリプログラム (呼び出す側)

```
.GLB_sub
.SECTION P,CODE
          :




PUSH.L R13
MOV.L 4[R0],R1
MOV.L R3,R12
MOV.L #_sub,R14
JSR     R14
POP     R13
RTS
          :
.END
```

3.5 セクション名一覧

ccrx で扱うセクションについて説明します。

アセンブラが出力するリロケータブルファイルの実行命令、データの各領域は、セクションを構成します。セクションは、メモリ上の配置を行う最小単位です。セクションの性質には、以下の項目があります。

- セクション属性

code	実行命令を格納します。
data	変更可能なデータを格納します。
romdata	固定データを格納します。

- 形式種別

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

- 初期値

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

- 書き込み操作

プログラム実行時における書き込み操作の可 / 不可を示します。

- アライメント数

セクションの配置アドレスを補正するための値です。最適化リンケージエディタでは、各セクションの配置アドレスを、それぞれのアライメント数の倍数になるように補正します。

3. 5. 1 C/C++ プログラムのセクション

C/C++ プログラム、標準ライブラリの使用メモリ領域の種類とセクションとの対応を表 30.30 に示します。

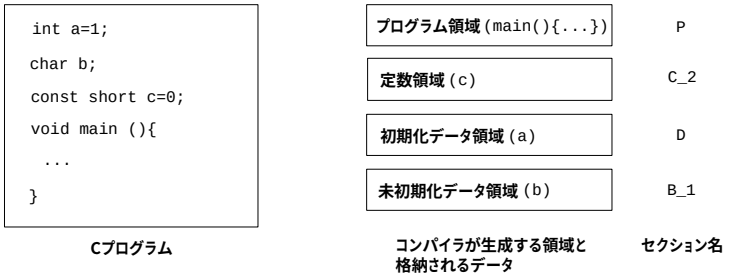
表 3―29 メモリ領域の種類とその性質の概要

No.	名称	セクション		形式種別	初期値	アライメント数	内容
		名称	属性		書き込み操作		
1	プログラム領域	P*1*6	code	相対	有 不可	1byte*7	機械語を格納
2	定数領域	C*1*2*6*8	romdata	相対	有 不可	4byte	const 型のデータを格納
		C_2*1*2*6*8	romdata	相対	有 不可	2byte	
		C_1*1*2*6*8	romdata	相対	有 不可	1byte	
3	初期化データ領域	D*1*2*6*8	romdata	相対	有 可	4byte	初期値のあるデータを格納
		D_2*1*2*6*8	romdata	相対	有 可	2byte	
		D_1*1*2*6*8	romdata	相対	有 可	1byte	
4	未初期化データ領域	B*1*2*6*8	data	相対	無 可	4byte	初期値のないデータを格納
		B_2*1*2*6*8	data	相対	無 可	2byte	
		B_1*1*2*6*8	data	相対	無 可	1byte	
5	switch 文分岐テーブル領域	W*1*2	romdata	相対	有 不可	4byte	switch 文の分岐テーブルを格納
		W_2*1*2	romdata	相対	有 不可	2byte	
		W_1*1*2	romdata	相対	有 不可	1byte	
6	C++ 初期処理／後処理データ領域	C$INIT	romdata	相対	有 不可	4byte	グローバルクラスオブジェクトに対して呼び出されるコンストラクタおよびデストラクタのアドレスを格納

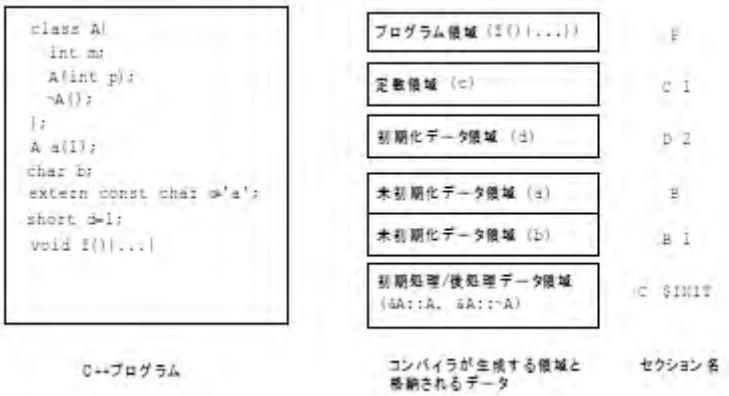
No.	名称	セクション		形式種別	初期値	アライメント数	内容
		名称	属性		書き込み操作		
7	C++ 仮想関数表領域	C$VTBL	romdata	相対	有 不可	4byte	クラス宣言中に仮想関数があるときに仮想関数をコールするためのデータを格納
8	ユーザスタック領域	SU	data	相対	無 可	4byte	プログラム実行に必要な領域
9	割り込みスタック領域	SI	data	相対	無 可	4byte	プログラム実行に必要な領域
10	ヒープ領域	—	—	相対	無 可	—	ライブラリ関数 malloc、realloc、calloc、new で使用する領域
11	絶対アドレス変数領域	$ADDR_<section>_<address> *3	data	絶対	有 / 無 可 / 不可 *4	—	#pragma address 指定した変数を格納
12	可変ベクタ領域	C$VECT	romdata	相対	無 可	4byte	可変ベクタテーブル
13	リテラル領域	L*5	romdata	相対	有 可 / 不可	4byte	文字列リテラルおよび集成体の動的初期化で用いる初期化子を格納

注 1. section オプションでセクション名を切り替えることができます。
2. セクション名切り替えの際に、アライメント数が 4 のセクションを指定することで、アライメントが 1 または 2 のセクション名も変更されます。
3. <section> は C,D,B のセクション名称、<address> は絶対アドレス値 (16 進数) になります。
4. 初期値、書き込み操作は <section> の属性に従います。
5. section オプションでセクション名を変更することができます。このとき、変更後の名前に C セクションを選択することも可能です。
6. #pragma section でセクション名を変更することができます。
7. instalign4 オプション、instalign8 オプション、#pragma instalign4 または #pragma instalign8 のいずれかを使用すると、アライメント数は 4 または 8 になります。
8. #pragma endian で endian オプションと異なる指定のエンディアンを指定した場合、#pragma endian big であれば _B を、#pragma endian little であれば _L を、セクション名の後ろに付加した専用のセクションを生成し、該当データを格納します。

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



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



3. 5. 2 アセンブリプログラムのセクション

アセンブリプログラムでは、.SECTION 制御命令を用いてセクションの開始や属性を、.ORG 制御命令を用いてセクションの形式種別を、それぞれ宣言します。

各制御命令の詳細については「10.3 アセンブラ制御命令の記述方法」を参照してください。

例 アセンブリプログラムのセクション宣言例を示します。

```
                .SECTION        A,CODE,ALIGN=4      ;(1)

START:
                MOV.L           #CONST,R4
                MOV.L           [R4],R5
                ADD             #10,R5,R3
                MOV.L           #100,R4
                MOV.L           #ARRAY,R5
LOOP:
                MOV.L           R3,[R5+]
                SUB             #1,R4
                CMP             #0,R4
                BNE             LOOP
EXIT:
                RTS

;
                .SECTION        B,ROMDATA           ;(2)
                .ORG            02000H
                .glb            CONST
CONST:
                .LWORD          05H
;
                .SECTION        C,DATA,ALIGN=4      ;(3)
                .glb            BASE
BASE:
                .blkl           100
                .END
```

(1) セクション名 A、アライメント数 4、相対アドレス形式の code セクションを宣言しています。

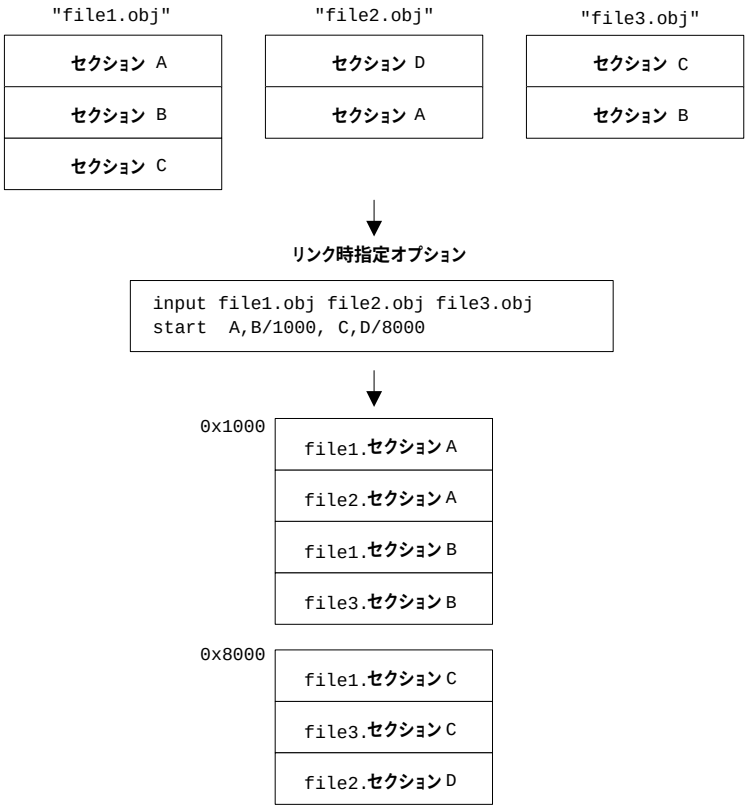
(2) セクション名 B、割り付けアドレス 2000H、絶対アドレス形式の romdata セクションを宣言しています。

(3) セクション名 C、アライメント数 4、相対アドレス形式の stack セクションを宣言しています。

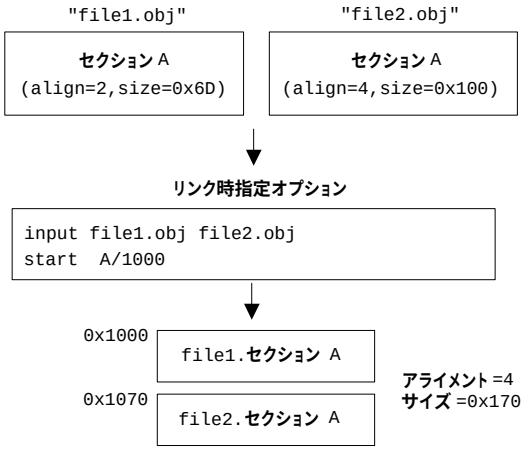
3. 5. 3　セクションの結合

最適化リンケージエディタでは、入力リロケータブルファイル内の同一セクションを結合し、start オプションによって指定されたアドレスに割り付けます。

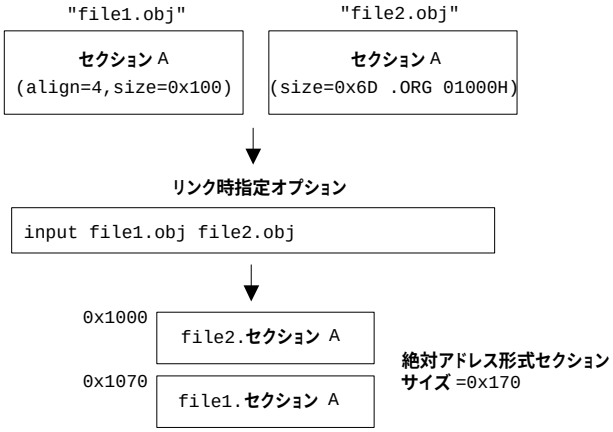
[1] 異なるファイルの同名セクションは、ファイルの入力順に連続して割り付けます。



[2] **アライメント数の異なる同名セクションは、アライメント調整後に結合します。セクションのアライメント数は大きい方に合わせます。**

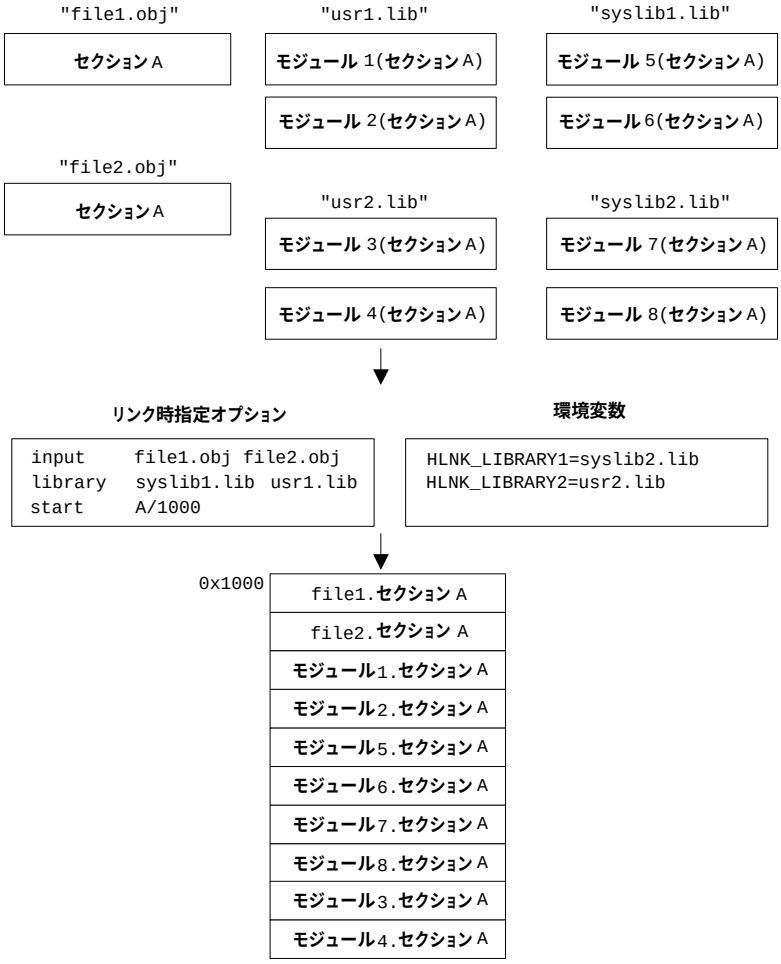


[3] **同名セクションに絶対アドレス形式と相対アドレス形式が含まれている場合、絶対アドレス形式セクションの後に相対アドレス形式セクションを結合します。**



[4] **同名セクションの結合順序に関する規則は、優先度の高い順に以下の通りです。**

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



第 4 章　アセンブラ言語仕様

この章では，RX がサポートするアセンブリ言語仕様について説明します。

4. 1　ソースの記述方法

この節では，ソースの記述方法，式と演算子などについて説明します。

4. 1. 1　記述方法

ニーモニック記述行の構成を以下に記します。

[ラベル][オペレーション [△オペランド]][コメント]

(コーディング例)

```
LABEL1:         MOV.L     [R1], R2            ; ニーモニック記述行の例です。
```

ラベル　オペレーション　オペランド　コメント

(1) ラベル

ニーモニック記述行のアドレスに対して名前を定義します。

(2) オペレーション

ニーモニック、制御命令を書きます。

(3) オペランド

オペレーションの実行対象となるものを書きます。オペランドの個数と種類はオペレーションによって決まります。オペランドを必要としないオペレーションもあります。

(4) コメント

プログラムを分かりやすくするための注釈を書きます。

4. 1. 2　名前

名前はアセンブリ言語ファイルの中で任意に定義し使用できます。

名前は次の種類に分けられます。

表 4―1 名前の種類

名前の種類	内容
ラベル名	アドレスを値として持つ名前
シンボル名	定数を値として持つ名前 (シンボル名には、ラベル名も含みます)
セクション名	.SECTION 制御命令で定義されるセクションの名前
ロケーションシンボル名	ロケーションシンボル '$' が記述されている行のオペレーション部の先頭アドレスを示します
マクロ名	マクロの定義名

名前の記述規則

- 名前の文字数に制限はありません。
- 名前は大文字と小文字を区別して扱います。"LAB" と "Lab" は異なる名前として扱われます。
- 名前には英数字とアンダーライン (_)、ドル ($) が使用できます。
- 名前の先頭文字に数字は使用できません。
- 名前に予約語を使用することはできません。

* ただし、セクション名にのみ予約語のフラグ名 (U,I,O,S,Z,C) を使用することができます。

4. 1. 3 ラベルの記述方法

ラベルを記述する場合、名前の最後に必ずコロン (:) を付けます。

- 記述例

```
LABEL1:
```

定義されているセクション名と同じ名前のシンボルを定義することはできません。セクションとシンボルを同じ名前で定義した場合、後で定義したものは A2118 エラーとなります。

4. 1. 4 オペレーション部の記述方法

- 記述方法

```
ニーモニック [ サイズ指定子 ( 分岐距離指定子 ) ]
```

- 内容

命令は、以下２つの要素から構成されます。

(1) ニーモニック ・・・ 命令の動作を示します。

(2) サイズ指定子 ・・・ 処理対象のデータサイズを指定します。

(1) ニーモニック

ニーモニックは命令の動作を示します。

例

```
MOV ・・・ 転送命令
ADD ・・・ 算術演算命令 ( 加算命令 )
```

(2) サイズ指定子

サイズ指定子は、命令コードのオペランドサイズを指定するものです。

- 記述方法

```
.size
```

- 内容

オペランドの演算サイズを指定します。正確にはその命令が処理を行うために読み出すデータのサイズを指定します。size は下記表の通りです。

表 4―2 サイズ指定子

size	内容
B	バイト (8 ビット)
W	ワード (16 ビット)
L	ロングワード (32 ビット)

size は大文字小文字いずれも可。

(例) MOV.B #0, R3 ・・・バイト指定

本指定子は、「RX ファミリソフトウェアマニュアル」の命令フォーマットで (.size) が明記されているものについてのみ指定が可能でかつ必須となります。

(3) 分岐距離指定子

分岐距離指定子は、分岐命令および相対サブルーチン分岐命令で指定します。

- 記述方法

```
.length
```

- 内容

length は下記表の通りです。

表 4―3 分岐距離指定子

length	内容	
S	3 ビット PC 前方相対	(+3 ～ +10)
B	8 ビット PC 相対	(-128 ～ +127)
W	16 ビット PC 相対	(-32768 ～ +32767)
A	24 ビット PC 相対	(-8388608 ～ +8388607)
L	レジスタ相対	(-2147483648 ～ +2147183647)

length は大文字小文字いずれも可。

(例)

BRA.W label ・・・16 ビット相対指定

BRA.L R1 ・・・レジスタ相対指定

本指定子は省略可能です。省略した場合は次の条件を全て満たした場合のみ、もっともオペコードが小さくなるように (S/B/W/A の中から) アセンブラがコードを選択します。

(1) オペランドが、レジスタ以外で記述されている場合

(2) オペランドが、アセンブル時に分岐距離が確定する分岐先である場合

(例) ラベル＋アセンブル時確定値

ラベル－アセンブル時確定値

アセンブル時確定値＋ラベル

(3) オペランドのラベルが同一セクション内で定義されている場合

また、オペランドがレジスタの場合、分岐距離指定子 L が選択されます。

条件分岐命令の場合、分岐距離が規定の範囲を超えているときは条件を反転してコードを生成します。

各命令で指定可能な分岐距離指定子は、下記表の通りです。

表 4―4 分岐命令ごとの分岐距離指定子

命令		.S	.B	.W	.A	.L
BCnd	(Cnd=EQ/Z)	○	○	○	×	×
	(Cnd=NE/NZ)	○	○	○	×	×
	(Cnd= 上記以外)	×	○	×	×	×
BRA		○	○	○	○	○
BSR		×	×	○	○	○

4. 1. 5 オペランド部の記述方法

(1) 数値

プログラムに記述できる数値の種類は以下 5 つがあります。

記述した値は 32 ビット符号付きで処理されます。(浮動小数点数を除く)

(a) 2 進数

0 ～ 1 のいずれかの数字で記述し、接尾辞として B または b を添付します。

- 記述例

```
1011000B
1011000b
```

(b) 8 進数

0 ～ 7 までの数字で記述し、接尾辞として O または o を添付します。

- 記述例

```
60702O
60702o
```

(c) 10 進数

0 ～ 9 までの数字で記述します。

- 記述例

```
9243
```

(d) 16 進数

0 ～ 9, A ～ F, a ～ f で記述し、接尾辞として H または h を添付します。

アルファベットで始まる数値の場合は接頭辞として 0 を添付します。

- 記述例

```
0A5FH
5FH
0a5fh
5fh
```

(e) 浮動小数点数

浮動小数点数は制御命令 ".FLOAT" と ".DOUBLE" のオペランドにのみ記述できます。

浮動小数点数は式に記述できません。

浮動小数点数で表される次の範囲の値を記述できます。

FLOAT (32bits)1.17549435×10^{-38} ～ 3.40282347×10^{38}

DOUBLE (64bits)$2.2250738585072014\times10^{-308}$ ～ $1.7976931348623157\times10^{308}$

- 記述方法

```
( 仮数部 )E( 指数部 )
( 仮数部 )e( 指数部 )
```

- 記述例

```
3.4E35    ;3.4x10**35
3.4e-35   ;3.4x10**-35
-.5E20    ;-0.5x10**20
5e-20     ;5.0x10**-20
```

4.1.6 式

数値、シンボルおよび演算子を組み合わせた式を記述できます。

- 演算子と数値の間には空白文字またはタブを記述できます。
- 演算子は複数組み合わせて記述できます。
- シンボル値として式を記述する場合は、式の値がアセンブル時に確定するように式を記述する必要があります。
- 式の項に文字定数は使用できません。
- 演算結果の範囲は、-2147483648 ～ 2147483647 となります。演算結果がこの範囲を超えた場合のオーバフローの判断は行いません。

(a) 演算子

プログラムに記述できる演算子の一覧を示します。

- 単項演算子

表 4―5 単項演算子

演算子	機能
+	続く値を正の値として扱います。
-	続く値を負の値として扱います。
~	続く値の論理否定値を扱います。
SIZEOF	オペランドに指定したセクションのサイズ（バイト数）を値として扱います。
TOPOF	オペランドに指定したセクションの開始アドレス値として扱います。

SIZEOF, TOPOF は、オペランドとの間に空白文字またはタブを記述します。

（例） SIZEOF program

- 二項演算子

表 4―6 二項演算子

演算子	機能
+	左辺値と右辺値を加算します。
-	左辺値から右辺値を減算します。
*	左辺値と右辺値を乗算します。
/	左辺値を右辺値で除算します。
%	左辺値を右辺値で割った余りを扱います。
>>	左辺値を右辺値回右へビットシフトします。
<<	左辺値を右辺値回左へビットシフトします。
&	左辺値と右辺値のビット毎の論理積値を扱います。
|	左辺値と右辺値のビット毎の論理和値を扱います。
^	左辺値と右辺値のビット毎の排他的論理和値を扱います。

- 条件演算子

条件演算子は制御命令 ".IF", ".ELIF" のオペランドにだけ記述できます。

表 4―7 条件演算子

演算子	機能
>	左辺値が右辺値より大きいことを評価します。
<	左辺値が右辺値より小さいことを評価します。
>=	左辺値が右辺値以上であることを評価します。
<=	左辺値が右辺値以下であることを評価します。
==	左辺値が右辺値と等しいことを評価します。
!=	左辺値が右辺値と等しくないことを評価します。

- 演算優先順位変更演算子

表 4―8 演算優先順位変更演算子

演算子	機能
()	() で囲った演算を最優先で行います。一つの式に複数の () が記述されている場合は、左側が優先されます。 () はネストした記述ができます。

(b) 式の演算優先順位

オペランドに記述されている式について、次に示す優先順位に従い演算を行った結果の数値を値として扱います。

- 演算子がもつ優先順位の高いものから演算します。演算子の優先順位を以下表に示します。
- 同一の優先順位を持つ演算子は、左から順に演算を行います。
- () で囲ったものが、優先順位が一番高くなります。

表 4―9 式の演算優先順位

優先順位	演算子の種類	演算子
1	演算優先順位変更演算子	()
2	単項演算子	+, -, ~, SIZEOF, TOPOF
3	二項演算子 1	*, /, %
4	二項演算子 2	+, -
5	二項演算子 3	>>, <<
6	二項演算子 4	&
7	二項演算子 5	|, ^
8	条件演算子	>, <, >=, <=, ==, !=

(1) アドレッシングモード

命令のオペランド部に記述できるアドレッシングモードは以下 3 つがあります。

(a) 一般命令アドレッシング

- レジスタ直接

指定したレジスタが演算の対象となります。R0 ～ R15、SP を記述することができます。SP は R0 と解釈します。(R0=SP)

```
Rn (Rn=R0 ～ R15, SP)
```

- 記述例

```
ADD  R1, R2
```

- 即値

#imm で示した即値は整数を表します。

#uimm で示した即値は符号なし整数を表します。

#simm で示した即値は符号付き整数を表します。

#imm:n、#uimm:n、および #simm:n は、n ビット長の即値を表します。

```
#imm:8, #uimm:8, #simm:8, #imm:16, #simm:16, #simm:24, #imm:32
```

注 RTSD 命令の #uimm:8 は、確定値でなければなりません。

- 記述例

```
MOV.L  #-100, R2          ; #simm:8
```

- レジスタ間接

レジスタの値が演算対象の実効アドレスになります。実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。

```
[Rn] (Rn=R0 ～ R15, SP)
```

- 記述例

```
ADD  [R1], R2
```

- レジスタ相対

ディスプレースメント (dsp) を 32 ビットにゼロ拡張した値と、レジスタ値を加算した結果が演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。dsp:n は、n ビット長のディスプレースメントを表します。

dsp の値は以下規則によりスケーリングされた値で指定します。アセンブラではスケーリング前の値に戻し、命令のビットパターンに埋め込みます。

表 4―10 dsp 値スケーリング規則

命令	規則
サイズ指定子をとる転送命令	サイズ指定子 .B/.W/.L に応じてそれぞれ、1/2/4 倍
サイズ拡張指定子をとる演算命令	サイズ拡張指定子 .B/.UB/.W/.UW/.L に応じてそれぞれ、1/1/2/2/4 倍
ビット操作命令	1 倍
上記以外	4 倍

```
dsp:8[Rn], dsp:16[Rn] (Rn=R0 ～ R15, SP)
```

- 記述例

```
ADD  400[R1], R2              ; dsp:8[Rn] (400/4 = 100)
```

サイズ指定子 W/L で 2/4 の倍数でない場合、アセンブル時確定値はアセンブラエラー
アセンブル時未確定値はリンク時エラー

(b) 拡張命令アドレッシング

- 短縮即値

#imm で示した即値が演算の対象となります。即値がアセンブル時確定値でない場合はエラー処理されます。

```
#imm:1
```

このアドレッシングは、DSP 機能命令（RACW）の src にのみ使用されます。1 または 2 を記述できます。

- 記述例

```
RACW  #1                      ; RACW  #imm:1
```

#imm:2

#imm で示した 2 ビット即値が演算の対象となります。このアドレッシングは、コプロセッサ命令 (MVFCP,MVTCP,OPECP) のコプロセッサ番号指定にのみ使用されます。

- 記述例

```
MVTCP  #3, R1, #4:16          ; MVTCP  #imm:2, Rn, #imm:16
```

#imm:3

#imm で示した 3 ビット即値が演算の対象となります。このアドレッシングは、ビット操作命令 (BCLR,BMCnd,BNOT,BSET,BTST) のビット番号指定に使用されます。

- 記述例

```
BSET   #7, R10                ; BSET   #imm:3, Rn
```

#imm:4

ADD,AND,CMP,MOV,MUL,OR,SUB 命令のソースに使用される場合は、#imm で示した 4 ビット即値を 32 ビットにゼロ拡張した結果が演算の対象となります。

MVTIPL 命令の割り込み優先レベル指定に使用される場合は、#imm で示した 4 ビット即値が演算の対象となります。

- 記述例

```
ADD   #15, R8                 ; ADD   #imm:4, Rn
```

#imm:5

#imm で示した 5 ビット即値が演算の対象となります。このアドレッシングは、ビット操作命令 (BCLR, BMCnd, BNOT, BSET, BTST) のビット番号指定、シフト命令 (SHAR,SHLL,SHLR) のシフト幅指定、および、ローテート命令 (ROTL,ROTR) のローテート幅指定にのみ使用されます。

- 記述例

```
BSET  #31, R10                ; BSET   #imm:5, Rn
```

- 短縮レジスタ相対

5 ビットディスプレースメント (dsp) を 32 ビットにゼロ拡張した値と、レジスタ値を加算した結果が演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。dsp の値はサイズ指定子 .B/.W/.L に応じて、それぞれ 1/2/4 倍した値で指定します。dsp の値がアセンブル時確定値でない場合、エラー処理をします。このアドレッシングは、MOV,MOVU 命令にのみ使用されます。

```
dsp:5[Rn] (Rn=R0～R7, SP)
```

- 記述例

```
MOV.L  R3,124[R1]              ; dsp:5[Rn] (124/4 = 31)
```

※ src/dest のレジスタも R0～R7 でなくてはなりません。

- ポストインクリメントレジスタ間接

レジスタの値に、サイズ指定子 .B/.W/.L に応じて、それぞれ 1/2/4 を加算します。更新前のレジスタの値が演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h～FFFFFFFFh です。このアドレッシングは、MOV,MOVU 命令にのみ使用されます。

```
[Rn+] (Rn=R0～R15, SP)
```

- 記述例

```
MOV.L [R3+],R1
```

- プリデクリメントレジスタ間接

レジスタの値に、サイズ指定子 .B/.W/.L に応じて、それぞれ 1/2/4 を減算します。更新後のレジスタの値が演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h～FFFFFFFFh です。このアドレッシングは、MOV,MOVU 命令にのみ使用されます。

```
[-Rn] (Rn=R0～R15, SP)
```

- 記述例

```
MOV.L [-R3],R1
```

- インデックス付きレジスタ間接

インデックスレジスタ (Ri) の値をサイズ指定子 .B/.W/.L に応じて、それぞれ 1/2/4 倍した値とベースレジスタ (Rb) の値を加算した結果の下位 32 ビットが演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h～FFFFFFFFh です。このアドレッシングは、MOV,MOVU 命令にのみ使用されます。

```
[Ri,Rb] (Ri=R0～R15, SP)  (Rb=R0～R15, SP)
```

- 記述例

```
MOV.L [R3,R1],R2
MOV.L R3, [R1,R2]
```

(c) 特定命令アドレッシング

- 制御レジスタ直接

指定した制御レジスタが演算の対象となります。

このアドレッシングは、MVTC,POPC,PUSHC,MVFC 命令にのみ使用されます。

```
PSW, FPSW, USP, ISP, INTB, BPSW, BPC, FINTV, PC, CPEN
```

- 記述例

```
STC  PSW,R2
```

- PSW 直接

指定したフラグ、または、ビットが演算の対象となります。このアドレッシングは、CLRPSW,SETPSW 命令にのみ使用されます。

```
U, I, O, S, Z, C
```

- 記述例

```
CLRPSW  U
```

- プログラムカウンタ相対

分岐命令の分岐先を指定するためのアドレッシング。

```
Rn(Rn=R0～R15, SP)
```

プログラムカウンタの値と、Rn の値を符号付きで加算した結果が実効アドレスとなります。Rn の値の範囲は、-2147483648 ～ 2147483647 です。実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。このアドレッシングモードは、BRA(.L)、BSR(.L) 命令に使用されます。

```
label(PC + pcdsp:3)
```

分岐命令の分岐先アドレスを表します。指定したシンボル、数値が実効アドレスとなります。
指定した分岐先アドレスからプログラムカウンタの値を引いたものをディスプレースメント (pcdsp) として命令のビットパターンに埋め込みます。
分岐距離指定子が “ .S ” の場合、プログラムカウンタの値とディスプレースメントの値を符号なしで加算した結果の下位 32 ビットが実効アドレスとなります。
pcdsp の範囲は、3 £ pcdsp:3£10 です。
実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。このアドレッシングは、BRA,BCnd(Cnd==EQ,NE,Z,NZ のみ) のみに使用できます。

```
label(PC + pcdsp:8/pcdsp:16/pcdsp:24)
```

分岐命令の分岐先アドレスを表します。指定したシンボル、数値が実効アドレスとなります。
指定した分岐先アドレスからプログラムカウンタの値を引いたものをディスプレースメント (pcdsp) として命令のビットパターンに埋め込みます。
分岐距離指定子が “ .B ” または “ .W ” または “ .A ” の場合、プログラムカウンタの値とディスプレースメントの値を符号付きで加算した結果の下位 32 ビットが実効アドレスとなります。
pcdsp の範囲は以下の通りです。
“ .B ” の場合 128£pcdsp:8£ +127
“ .W ” の場合 32768 £ pcdsp:16£ +32767
“ .A ” の場合 8388608 £ pcdsp:24 £ +8388607
実効アドレスの範囲は、00000000h ～ FFFFFFFFh です。

(2) ビット長指定子

ビット長指定子はオペランドの即値、またはディスプレースメントのサイズを指定します。

- 記述方法

```
:width
```

- 内容

本指定子はオペランドに記述された即値、またはディスプレースメントの直後に指定します。
アセンブラは指定されたビット長のアドレッシングモードを選択します。
本指定子を省略した場合はアセンブラがもっとも効率のよいビット長を選択します。
本指定子が記述されている場合には最適選択は行わず、指定されたビット長とします。
本指定子はアセンブラ制御命令のオペランドには記述できません。
即値、ディスプレースメントの式と本指定子の間には、１つ以上の空白文字を入れることができます。
命令フォーマットに存在しないビット長が指定された場合は、エラー処理されます。
width に指定できるものは次の通りです。

2: 有効ビット長が 2 ビットであることを表します。

```
#imm:2
```

3: 有効ビット長が 3 ビットであることを表します。

```
#imm:3
```

4: 有効ビット長が 4 ビットであることを表します。

```
#imm:4
```

5: 有効ビット長が 5 ビットであることを表します。

```
#imm:5, dsp:5
```

8: 有効ビット長が 8 ビットであることを表します。

```
#uimm:8, #simm:8, dsp:8
```

16: 有効ビット長が 16 ビットであることを表します。

```
#uimm:16, #simm:16, dsp:16
```

24: 有効ビット長が 24 ビットであることを表します。

```
#simm:24
```

32: 有効ビット長が 32 ビットであることを表します。

```
#imm:32
```

(3) サイズ拡張指定子

サイズ拡張指定子は、演算命令でソースがメモリオペランドの場合、メモリオペランドのサイズと拡張方法を指定するために付加されます。

- 記述方法

```
.memex
```

- 内容

本指定子はメモリオペランドの直後に記述し、間に空白文字を入れることはできません。

サイズ拡張指定子は特定の命令と、メモリオペランドの組み合わせに対してのみ有効で、有効でない命令とオペランドの組み合わせで指定した場合は、エラー処理をします。

指定可能な命令とオペランドの組み合わせは、RX ファミリソフトウェアマニュアルの命令フォーマットのオペランドに .memex が付いているパターンのみです。

省略時はビット操作命令では 'B' として扱い、それ以外の命令では 'L' として扱います。

指定可能なサイズ拡張指定子と効果を、以下表に記します。

表 4―11 サイズ拡張指定子

サイズ拡張指定子	効果
B	8 ビットデータを 32 ビット符号拡張
UB	8 ビットデータを 32 ビットゼロ拡張
W	16 ビットデータを 32 ビット符号拡張
UW	16 ビットデータを 32 ビットゼロ拡張
L	32 ビットデータをロード

(記述例)

```
ADD  [R1].B, R2
AND  125[R1].UB, R2
```

4. 1. 7 コメントの記述方法

セミコロン (;) の後に続けて記述します。セミコロンから行末までをコメントと見なします。

- 記述例

```
ADD     R1, R2          ; R2 に R1 を加えます。
```

4. 1. 8 命令フォーマットの最適選択

RX ファミリの命令には、同一処理に対して複数の命令フォーマットを持つものがあります。

アセンブラでは、命令およびアドレッシングモードの指定に応じて、最短コードの命令フォーマットを選択する最適選択を行います。

(1) 即値について

アセンブラではオペランドに即値を持つ命令である場合、オペランドに指定された即値の範囲に従い選択可能なアドレッシングから最適選択を行います。以下に即値の範囲について優先順位の高い順に示します。

表 4―12 即値の範囲

#imm	10 進記法	16 進記法
#imm:1	1 ～ 2	1H ～ 2H
#imm:2	0 ～ 3	0H ～ 3H
#imm:3	0 ～ 7	0H ～ 7H
#imm:4	0 ～ 15	0H ～ 0FH
#imm:5	0 ～ 31	0H ～ 1FH
#imm:8	-128 ～ 255	-80H ～ 0FFH
#uimm:8	0 ～ 255	0H ～ 0FFH
#simm:8	-128 ～ 127	-80H ～ 7FH
#imm:16	-32768 ～ 65535	-8000H ～ 0FFFFH
#simm:16	-32768 ～ 32767	-8000H ～ 7FFFH
#simm:24	-8388608 ～ 8388607	-800000H ～ 7FFFFFH
#imm:32	-2147483648 ～ 4294967295	-80000000H ～ 0FFFFFFFFH

注 1. 16 進表記は 32 ビット表記も可能です。
例：10 進表記 ”-127”、16 進表記 ”-7FH” は ”0FFFFFF81H” と表記できます。

2. INT 命令の src の #imm の範囲は 0 ～ 255 となります。

3. RTSD 命令の src の #imm の範囲は #uimm:8 を 4 倍した値となります。

(2) ADC, SBB 命令

ADC, SBB 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

注 最適選択対象とならない命令フォーマットとオペランドは記述していません。表内の処理サイズは、特に明記がない場合は "L" となります。

表 4―13 ADC, SBB 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
ADC src,dest	#simm:8	-	Rd	4
	#simm:16	-	Rd	5
	#simm:24	-	Rd	6
	#imm:32	-	Rd	7
ADC/SBB src,dest	dsp:8[Rs].L	-	Rd	4
	dsp:16[Rs].L	-	Rd	5

SBB 命令では、src に即値を指定することはできません。

(3) ADD 命令

ADD 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―14 ADD 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
(1) ADD src,dest	#uimm:4	-	Rd	2
	#simm:8	-	Rd	3
	#simm:16	-	Rd	4
	#simm:24	-	Rd	5
	#imm:32	-	Rd	6
	dsp:8[Rs].memex	-	Rd	3(memex =UB), 4(memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	4(memex =UB), 5(memex ≠ UB)
(2) ADD src,src2,dest	#simm:8	Rs	Rd	3
	#simm:16	Rs	Rd	4
	#simm:24	Rs	Rd	5
	#imm:32	Rs	Rd	6

(4) AND, OR, SUB, MUL 命令

AND, OR, SUB, MUL 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―15 AND, OR, SUB および MUL 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
AND/OR/SUB/MUL src,dest	#uimm:4	-	Rd	2
	#simm:8	-	Rd	3
	#simm:16	-	Rd	4
	#simm:24	-	Rd	5
	#imm:32	-	Rd	6
	dsp:8[Rs].memex	-	Rd	3(memex = UB), 4(memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	4(memex = UB), 5(memex ≠ UB)

SUB 命令では、src に #simm:8/16/24, #imm32 を指定することはできません。

(5) BMCnd 命令

BMCnd 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―16 BMCnd 命令の命令フォーマット

命令フォーマット	処理サイズ	対象			コードサイズ [バイト]
		src	src2	dest	
BMCnd src,dest	B	#imm:3	-	dsp:8[Rs].B	4
	B	#imm:3	-	dsp:16[Rs].B	5

(6) CMP 命令

CMP 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―17 CMP 命令の命令フォーマット

命令フォーマット	処理サイズ	対象			コードサイズ [バイト]
		src	src2	dest	
CMP src,src2	L	#uimm:4	Rd	-	2
	L	#uimm:8	Rd	-	3
	L	#simm:8	Rd	-	3
	L	#simm:16	Rd	-	4
	L	#simm:24	Rd	-	5
	L	#imm:32	Rd	-	6
	L	dsp:8[Rs].memex	Rd	-	3(memex = UB), 4(memex ≠ UB)
	L	dsp:16[Rs].memex	Rd	-	4(memex = UB), 5(memex ≠UB)

(7) DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, TST, XOR 命令

DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, MUL, TST, XOR 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―18 DIV, DIVU, EMUL, EMULU, ITOF, MAX, MIN, TST および XOR 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
DIV/DIVU/	#simm:8	-	Rd	4
EMUL/EMULU/ITOF/	#simm:16	-	Rd	5
MAX/MIN/TST/XOR	#simm:24	-	Rd	6
	#imm:32	-	Rd	7
src,dest	dsp:8[Rs].memex	-	Rd	4(memex=UB), 5(memex ≠ UB)
	dsp:16[Rs].memex	-	Rd	5(memex=UB), 6(memex ≠ UB)

ITOF 命令では、src に #simm:8/16/24, #imm32 を指定することはできません。

(8) FADD, FCMP, FDIV, FMUL, FTOI 命令

FADD, FCMP, FDIV, FMUL, FTOI 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―19 FADD, FCMP, FDIV, FMUL および FTOI 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
FADD/FCMP/FDIV/	#imm:32	-	Rd	7
FMUL/FTOI	dsp:8[Rs].L	-	Rd	4
src,dest	dsp:16[Rs].L	-	Rd	5

FTOI 命令では、src に #imm32 を指定することはできません。

(9) MVTC, STNZ, STZ 命令

MVTC, STNZ, STZ 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―20 MVTC, STNZ および STZ 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
MVTC/STNZ/STZ	#simm:8	-	Rd	4
src,dest	#simm:16	-	Rd	5
	#simm:24	-	Rd	6
	#imm:32	-	Rd	7

(10) MOV 命令

MOV 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―21 MOV 命令の命令フォーマット

命令フォーマット	size	処理サイズ	対象			コードサイズ [バイト]
			src	src2	dest	
MOV(.size) src,dest	B/W/L	size	Rs(Rs=R0-R7)	-	dsp:5[Rd](Rd=R0-R7)	2
	B/W/L	L	dsp:5[Rs](Rs=R0-R7)	-	Rd(Rd=R0-R7)	2
	B/W/L	L	#uimm:8	-	dsp:5[Rd](Rd=R0-R7)	3
	L	L	#uimm:4	-	Rd	2
	L	L	#uimm:8	-	Rd	3
	L	L	#simm:8	-	Rd	3
	L	L	#simm:16	-	Rd	4
	L	L	#simm:24	-	Rd	5
	L	L	#imm:32	-	Rd	6
	B	B	#imm:8	-	[Rd]	3
	W/L	W/L	#simm:8	-	[Rd]	3
	W	W	#imm:16	-	[Rd]	4
	L	L	#simm:16	-	[Rd]	4
	L	L	#simm:24	-	[Rd]	5
	L	L	#imm:32	-	[Rd]	6
	B	B	#imm:8	-	dsp:8[Rd]	4
	W/L	W/L	#simm:8	-	dsp:8[Rd]	4
	W	W	#imm:16	-	dsp:8[Rd]	5
	L	L	#simm:16	-	dsp:8[Rd]	5
	L	L	#simm:24	-	dsp:8[Rd]	6
	L	L	#imm:32	-	dsp:8[Rd]	7
	B	B	#imm:8	-	dsp:16[Rd]	5
	W/L	W/L	#simm:8	-	dsp:16[Rd]	5
	W	W	#imm:16	-	dsp:16[Rd]	6
	L	L	#simm:16	-	dsp:16[Rd]	6
	L	L	#simm:24	-	dsp:16[Rd]	7
	L	L	#imm:32	-	dsp:16[Rd]	8
	B/W/L	L	dsp:8[Rs]	-	Rd	3
	B/W/L	L	dsp:16[Rs]	-	Rd	4
	B/W/L	size	Rs	-	dsp:8[Rd]	3
	B/W/L	size	Rs	-	dsp:16[Rd]	4
	B/W/L	size	[Rs]	-	dsp:8[Rd]	3
	B/W/L	size	[Rs]	-	dsp:16[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	[Rd]	3
	B/W/L	size	dsp:16[Rs]	-	[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	dsp:8[Rd]	4
	B/W/L	size	dsp:8[Rs]	-	dsp:16[Rd]	5
	B/W/L	size	dsp:16[Rs]	-	dsp:8[Rd]	5
	B/W/L	size	dsp:16[Rs]	-	dsp:16[Rd]	6

(11) MOVU 命令

MOVU 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―22 MOVU 命令の命令フォーマット

命令フォーマット	size	処理サイズ	対象			コードサイズ [バイト]
			src	src2	dest	
MOVU(.size) src,dest	B/W	L	dsp:5[Rs](Rs=R0-R7)	-	Rd(Rd=R0-R7)	2
	B/W	L	dsp:8[Rs]	-	Rd	3
	B/W	L	dsp:16[Rs]	-	Rd	4

(12) PUSH 命令

PUSH 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―23 PUSH 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
PUSH src	dsp:8[Rs]	-	-	3
	dsp:16[Rs]	-	-	4

(13) ROUND 命令

ROUND 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―24 ROUND 命令の命令フォーマット

命令フォーマット	対象			コードサイズ [バイト]
	src	src2	dest	
ROUND src,dest	dsp:8[Rs]	-	Rd	4
	dsp:16[Rs]	-	Rd	5

(14) SCCnd 命令

SCCnd 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―25 SCCnd 命令の命令フォーマット

命令フォーマット	size	対象			コードサイズ [バイト]
		src	src2	dest	
SCCnd(.size) src,dest	B/W/L	-	-	dsp:8[Rd]	4
	B/W/L	-	-	dsp:16[Rd]	5

(15) XCHG 命令

XCHG 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―26 XCHG 命令の命令フォーマット

命令フォーマット	処理サイズ	対象			コードサイズ [バイト]
		src	src2	dest	
XCHG src,dest	L	dsp:8[Rs].memex	-	Rd	4(memex = UB), 5(memex ≠ UB)
	L	dsp:16[Rs].memex	-	Rd	5(memex = UB), 6(memex ≠ UB)

(16) BCLR, BNOT, BSET, BTST 命令

BCLR, BNOT, BSET, BTST 命令に対して、最適選択の対象となる命令フォーマットとオペランドを優先順位の高い順に示します。

表 4―27 BCLR, BNOT, BSET および BTST 命令の命令フォーマット

命令フォーマット	処理サイズ	対象			コードサイズ [バイト]
		src	src2	dest	
BCLR/BNOT/BSET/BTST src,dest	B	#imm:3	-	dsp:8[Rd].B	3
	B	#imm:3	-	dsp:16[Rd].B	4
	B	Rs	-	dsp:8[Rd].B	4
	B	Rs	-	dsp:16[Rd].B	5

4. 1. 9 分岐命令の最適選択

(1) 相対無条件分岐命令 (BRA) の最適選択

(a) 指定可能な分岐距離指定子

.S 3 ビット PC 相対 (PC+pcdsp:3,3 £ pcdsp:3 £ 10)

.B 8 ビット PC 相対 (PC+pcdsp:8,-128 £ pcdsp:8 £ 127)

.W 16 ビット PC 相対 (PC+pcdsp:16,-32768 £ pcdsp:16 £ 32767)

.A 24 ビット PC 相対 (PC+pcdsp:24,-8388608 £ pcdsp:24 £ 8388607)

.L レジスタ相対 (PC+Rs,-2147483648 £ Rs £ 2147483647)

※レジスタ相対はオペランドがレジスタの場合のみ選択され、最適選択によって選択されることはありません。

(b) 最適選択

- アセンブラでは相対無条件分岐命令のオペランドが分岐最適化対象条件を満たす場合、最短の分岐距離を選択します。条件については、「10.1.5 (3) 分岐距離指定子」を参照してください。
- 条件を満たさないものについては、24 ビット PC 相対 (.A) を選択します。

(2) 相対サブルーチン分岐命令 (BSR) の最適選択

(a) 指定可能な分岐距離指定子

.W 16 ビット PC 相対 (PC+pcdsp:16,-32768 £ pcdsp:16 £ 32767)

.A 24 ビット PC 相対 (PC+pcdsp:24,-8388608 £ pcdsp:24 £ 8388607)

.L レジスタ相対 (PC+Rs,-2147483648 £ Rs £ 2147483647)

※レジスタ相対はオペランドがレジスタの場合のみ選択され、最適選択によって選択されることはありません。

(b) 最適選択

- アセンブラでは相対サブルーチン分岐命令のオペランドが分岐最適化対象条件を満たす場合、最短の分岐距離を選択します。条件については、「10.1.5 (3) 分岐距離指定子」を参照してください。
- 条件を満たさないものについては、24 ビット PC 相対 (.A) を選択します。

(3) 条件分岐命令 (BCnd) の最適選択

(a) 指定可能な分岐距離指定子

BEQ.S 3 ビット PC 相対 (PC+pcdsp:3,3 £ pcdsp:3 £ 10)

BNE.S 3 ビット PC 相対 (PC+pcdsp:3,3 £ pcdsp:3 £ 10)

BCnd.B 8 ビット PC 相対 (PC+pcdsp:8,-128 £ pcdsp:8 £ 127)

BEQ.W 16 ビット PC 相対 (PC+pcdsp:16,-32768 £ pcdsp:16 £ 32767)

BNE.W 16 ビット PC 相対 (PC+pcdsp:16,-32768 £ pcdsp:16 £ 32767)

(b) 最適選択

- アセンブラでは条件分岐命令のオペランドが分岐最適化対象条件を満たす場合、論理を反転した条件分岐命令と最適な分岐距離の相対無条件分岐命令を組み合わせた最適な条件分岐コードを選択して生成します。
- 条件を満たさないものについては、8 ビット PC 相対 (.B) または 16 ビット PC 相対 (.W) を選択します。

(c) 変換される条件分岐命令に対する代替命令

表 4―28 条件分岐命令の代替規則

条件分岐命令	代替分岐命令	条件分岐命令	代替分岐命令
BNC/BLTU dest	BC ..xx BRA.A dest ..xx:	BC/BGEU dest	BNC ..xx BRA.A dest ..xx:
BLEU dest	BGTU ..xx BRA.A dest ..xx:	BGTU dest	BLEU ..xx BRA.A dest ..xx:
BNZ/BNE dest	BZ ..xx BRA.A dest ..xx:	BZ/BEQ dest	BNZ ..xx BRA.A dest ..xx:
BPZ dest	BN ..xx BRA.A dest ..xx:	BO dest	BNO ..xx BRA.A dest ..xx:
BGT dest	BLE ..xx BRA.A dest ..xx:	BLE dest	BGT ..xx BRA.A dest ..xx:
BGE dest	BLT ..xx BRA.A dest ..xx:	BLT dest	BGE ..xx BRA.A dest ..xx:

上記は相対無条件分岐命令の分岐距離が 24 ビット PC 相対の場合を記します。
“..xx” ラベルおよび相対無条件分岐命令は内部的に処理されるものであり、アセンブルリストファイルにはコードのみ生成されます。

4. 2 擬似命令

この節では，疑似命令について説明します。
疑似命令とは，アセンブラが一連の処理を行う際に必要な各種の指示を行うものです。

4. 2. 1 概要

インストラクションは，アセンブルの結果，オブジェクト・コード（機械語）に変換されますが，疑似命令は，原則としてオブジェクト・コードに変換されません。

疑似命令は，主に次の機能を持ちます。

- ソースの記述を容易にします。
- メモリの初期化や領域の確保を行います。
- アセンブラ，リンカがその処理を行うために必要となる情報を与えます。

次に，疑似命令の種類を示します。

種類	擬似命令
リンク制御擬似命令	.SECTION, .GLB, .RVECTOR
アセンブル制御擬似命令	.EQU, .END, .INCLUDE
アドレス制御擬似命令	.ORG, .OFFSET, .ENDIAN, .BLKB, .BLKW, .BLKL, .BLKD, .BYTE, .WORD, .LWORD, .FLOAT, .DOUBLE, .ALIGN
マクロ制御擬似命令	.MACRO, .EXITM, .LOCAL, .ENDM, .MREPEAT, .ENDR, ..MACPARA, ..MACREP, .LEN, .INSTR, .SUBSTR
コンパイラ専用制御擬似命令	._LINE_TOP, ._LINE_END, .SWSECTION, .SWMOV, .SWITCH, .INSTALIGN

以降，各疑似命令について詳細な説明を行います。

4. 2. 2 リンク制御擬似命令

プログラムを複数のファイルに分割して記述するリロケータブルアセンブルを実行するための制御命令です。

.SECTION

セクションの宣言、再開を指定します。

[指定形式]

.SECTION △ < セクション名 >
.SECTION △ < セクション名 >,< セクション属性 >
.SECTION △ < セクション名 >,< セクション属性 >,ALIGN=[2|4|8]
.SECTION △ < セクション名 >,ALIGN=[2|4|8]
< セクション属性 >:[CODE|ROMDATA|DATA]

[詳細説明]

セクションの宣言、再開を指定します。

(1) セクションの宣言

セクション名、セクション属性を指定し、セクションの始まりを定義します。

(2) セクションの再開

ソースプログラム中にすでに存在しているセクションを再開します。セクションの再開ではすでに存在するセクション名を指定します。セクション属性とアライメント補正値は最初に宣言したものを継続します。

‘ ALIGN= ’ 指定がある場合、指定されたセクションに対してアライメント補正値を変更することができます。

ALIGN 指定をした相対アドレス形式セクションまたは絶対アドレス形式セクションに、制御命令 ".ALIGN" が記述できます。

ALIGN 指定がない場合、そのセクションの境界調整数は 1 となります。

例 SECTIONprogram,CODE
NOP
.SECTIONram,DATA
.BLKB10
.SECTIONtbl1,ROMDATA

```
.BYTE"abcd"
.SECTIONtbl2,ROMDATA,ALIGN=8
.LWORD11111111H,22222222H
.END
```

[備考]

セクション名は必ず記述してください。

メモリ領域を確保したりメモリにデータを格納するアセンブラ制御命令を記述する場合は、必ず本制御命令でセクションを定義してください。

ニーモニックを記述する場合は必ず、本制御命令でセクションを定義してください。

セクション属性と ALIGN は、セクション名の後に記述してください。

セクション属性および、ALIGN 指定をする場合は、カンマで区切って記述してください。

セクション属性と ALIGN の記述順序は任意です。

セクション属性は、’ CODE ’ , ’ ROMDATA ’ , ’ DATA ’ のいずれかを記述できます。

セクション属性は省略できます。このとき、アセンブラはセクション属性を CODE として処理します。

-endian=big 指定時、絶対アドレス形式の CODE セクションの開始アドレスには 4 の倍数以外の値を指定することはできません。

-endian=big 指定時、絶対アドレス形式の CODE セクションはウォーニングを出力し、セクションサイズが 4 の倍数になるようにアセンブラがセクション末尾に NOP(0x03) を書き込みます。

定義されているセクション名と同じ名前のシンボルを定義することはできません。セクションとシンボルを同じ名前で定義した場合、後で定義したものは A2118 エラーとなります。

$iop という名前のセクション名を定義することはできません。$iop を定義した場合 A2049 エラーとなります。

.GLB

本擬似命令で指定したラベルおよびシンボルがグローバルであることを宣言します。

[指定形式]

.GLB △ < 名前 > .GLB △ < 名前 >[,< 名前 > …]

[詳細説明]

本制御命令で指定したラベルおよびシンボルがグローバルであることを宣言します。

本制御命令で指定したラベルおよびシンボルで、ファイル内で定義されていないものは、外部のファイルで定義されているものとして処理します。

本制御命令で指定したラベルおよび、シンボルで、ファイル内で定義されているものは、外部から参照できるように処理します。

例

```
.GLB    name1,name2,name3
.GLB    name4
.SECTION program
MOV.L   #name1,R1
```

[備考]

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドにグローバルラベルとするラベル名を記述します。

オペランドにグローバルシンボルとするシンボル名を記述します。
オペランドに複数のシンボル名を記述する場合は、カンマ (,) で区切って記述してください。

.RVECTOR

本制御命令で指定したラベルおよびシンボルを、可変ベクタとして登録します。
[指定形式]

.RVECTOR △ < 番号 >,< 名前 >

[詳細説明]
本制御命令で指定したラベルおよびシンボルを、可変ベクタとして登録します。
本制御命令の < 番号 > には、ベクタ番号として 0 ～ 255 の定数を記述することができます。
本制御命令の < 名前 > には、ファイル内で定義されたラベルまたはシンボルを指定することができます。
登録された可変ベクタは、最適化リンケージエディタにより、ひとつの C$VECT セクションにまとめられます。
例 .RVECTOR 50,_rvfunc

```
_rvfunc:
MOV.L  #0,R1
RTE
```

[備考]
制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

4. 2. 3 アセンブル制御擬似命令

制御命令自身はデータを生成しません。命令に対する機械語コードの生成を制御する制御命令です。アドレスの更新は行いません。

.EQU

シンボルに 32 ビット符号付き整数値 (-2147483648 ～ 2147483647) の範囲の値を定義します。
[指定形式]

名前 > △ .EQU △ < 数値 >

[詳細説明]
シンボルに 32 ビット符号付き整数値 (-2147483648 ～ 2147483647) の範囲の値を定義します。
本制御命令でシンボルを定義することにより、シンボリックデバッグ機能が使用できます。

例

```
symbol  .EQU 1
symbol1 .EQU symbol+symbol
symbol2 .EQU 2
```

[備考]
シンボルに定義できる値は、アセンブル時に確定しなければなりません。
制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。
シンボル定義のオペランドには、シンボルを記述できます。ただし、前方参照となるシンボル名は記述できません。

オペランドには式を記述できます。

シンボルはグローバル指定ができます。

本制御命令と .DEFINE 制御命令で同名のシンボルを宣言した場合、先に記述した方が優先されます。

.END

アセンブリ言語ファイルの終了を宣言します。

[指定形式]

.END

[詳細説明]

アセンブリ言語ファイルの終了を宣言します。

本制御命令を記述した行以降の記述内容はアセンブルリストファイルに出力するのみで、コード生成などの処理は行いません。

例 .END

[備考]

本制御命令は、1 つのアセンブリ言語ファイルに必ず 1 つ記述する必要があります。

.INCLUDE

アセンブリ言語ファイルの行に、インクルードファイルの内容全てを読み込みます。

[指定形式]

.INCLUDE △ < インクルードファイル名 >

[詳細説明]

アセンブリ言語ファイルの行に、インクルードファイルの内容全てを読み込みます。

本制御命令で読み込まれたインクルードファイルの内容は、読み込んだアセンブリ言語ファイル内に記述した場合と、同じ 1 つのアセンブリ言語ファイルとして処理されます。

インクルードファイルは 30 レベルまでネストできます。

インクルードファイル名に絶対パスを記述した場合は、記述したディレクトリ内のファイルを検索します。

ファイルが見つからない場合はエラーとなります。

インクルードファイル名に絶対パスを記述していない場合は、次に示す順序でファイルを検索します。

(1)アセンブラ起動時にコマンド行で指定したアセンブリ言語ファイル名にディレクトリ指定がない場合は、.INCLUDE 制御命令で指定されたインクルードファイル名を検索します。アセンブラ起動時にコマンド行で指定したアセンブリ言語ファイル名にディレクトリ指定がある場合は、.INCLUDE 制御命令で指定されたインクルードファイル名にコマンド行で指定されたディレクトリ名を付加して検索します。

(2)アセンブラオプション -include で指定されたディレクトリを検索します。

(3)環境変数 INC_RXA に設定されているディレクトリを検索します。

例 .INCLUDE initial.src

.INCLUDE ..FILE@.inc

[備考]

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドのインクルードファイル名には、必ずファイル拡張子を記述してください。

オペランドには、制御命令 "..FILE" や "@" を含む文字列が記述できます。
ファイル名は、その先頭を除き、空白文字を含むことができます。
ファイル名をダブルクォーテーション「"」で囲わないでください。
自分自身をインクルードファイルに指定することはできません。

4.2.4　アドレス制御擬似命令

アセンブラがアドレス更新をする場合の指示を行う制御命令です。
絶対アドレス形式セクション内のアドレスを除いて、アセンブラが制御を行うアドレスはリロケータブル値となります。

.ORG

本制御命令を記述したセクションを絶対アドレス形式とします。
[指定形式]

.ORG △ < 数値 >

[詳細説明]
本制御命令を記述したセクションを絶対アドレス形式とします。
本制御命令を記述したセクションのアドレスはアブソリュート値になります。
本制御命令を記述した直後の行から記述したニーモニックのコードが格納されるアドレスを決定します。
本制御命令の直後の行から記述した領域確保制御命令で、確保されるメモリのアドレスを決定します。

例 .SECTIONvalue,ROMDATA
.ORG0FF00H
.BYTE"abcdefghijklmnopqrstuvwxyz"
.ORG0FF80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
以下の場合は .SECTION の直後に .ORG が記述されていないため、エラーとなります。
.SECTIONvalue,ROMDATA
.BYTE"abcdefghijklmnopqrstuvwxyz"
.ORG0FF80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END
[備考]
本制御命令は、必ずセクション制御命令の直後に記述してください。
".SECTION" を記述した直後の行に ".ORG" の記述がない場合は、そのセクションは相対アドレス形式セクションとなります。
制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。
オペランドに記述できる値は、0 ～ 0FFFFFFFFH の範囲の数値です。
オペランドには式、シンボルを記述できます。ただし、アセンブル時に確定する値でなければなりません。
本制御命令は、相対アドレス形式指定を行ったセクション内には記述できません。

絶対アドレス形式セクション内であれば複数回記述できます。ただし本制御命令記述行のアドレスよりも小さい値を指定した場合にはエラーとなります。

.OFFSET

セクション先頭からのオフセットを指定します。

[指定形式]

.OFFSET △ < 数値 >

[詳細説明]

セクション先頭からのオフセットを指定します。

本制御命令を記述した直後の行から記述したニーモニックのコードが格納される、セクション先頭からのオフセットを決定します。

本制御命令の直後の行から記述した領域確保制御命令で確保されるメモリの、セクション先頭からのオフセットを決定します。

例 .SECTIONvalue,ROMDATA
.BYTE"abcdefghijklmnopqrstuvwxyz"
.OFFSET80H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END

以下の場合は .OFFSET 記述行のオフセットよりも小さい値が指定されているため、エラーとなります。

.SECTIONvalue,ROMDATA
.OFFSET80H
.BYTE"abcdefghijklmnopqrstuvwxyz"
.OFFSET70H
.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
.END

[備考]

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドに記述できる値は、0 ～ 0FFFFFFFFH の範囲の数値です。

オペランドには式、シンボルを記述できます。ただし、アセンブル時に確定する値でなければなりません。

本制御命令は、絶対アドレス形式指定を行ったセクション内には記述できません。

相対アドレス形式セクション内であれば複数回記述できます。ただし本制御命令記述行のオフセットよりも小さい値を指定した場合にはエラーとなります。

.ENDIAN

本制御命令を記述したセクションのエンディアンを指定します。

[指定形式]

.ENDIAN △ BIG
.ENDIAN △ LITTLE

[詳細説明]

本制御命令を記述したセクションのエンディアンを指定します。

.ENDIAN BIG を記述したセクションのデータのバイト並びは Big Endian になります。

.ENDIAN LITTLE を記述したセクションのデータのバイト並びは Little Endian になります。

本制御命令が記述されていないセクションのデータのバイト並びは、-endian オプションに依存します。

例 .SECTIONvalue,ROMDATA

.ORGFF00H

.ENDIANBIG

.BYTE"abcdefghijklmnopqrstuvwxyz"

以下の場合は .SECTION または .ORG の直後に .ENDIAN が記述されていないため、エラーとなります。

.SECTIONvalue,ROMDATA

.ORG0FF00H

.BYTE"abcdefghijklmnopqrstuvwxyz"

.ENDIANBIG

.BYTE"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

[備考]

本制御命令は必ず .SECTION 制御命令、またはそれに続く .ORG 制御命令の直後に記述してください。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

セクション属性が CODE のセクションに本制御命令を追加することはできません。

.BLKB

1 バイト単位で、指定したバイト数の RAM 領域を確保します。

[指定形式]

△ .BLKB △ < オペランド >

△ < ラベル名 :> △ .BLKB △ < オペランド >

[詳細説明]

1 バイト単位で、指定したバイト数の RAM 領域を確保します。

確保した RAM のアドレスに、ラベル名を定義することもできます。

例 symbolEQU 1

.SECTION area,DATA

work1:BLKB 1

work2:BLKB symbol

.BLKB symbol+1

[備考]

本制御命令は必ず、DATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ” ,DATA ” を記述することでセクション属性が DATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドの値は、アセンブル時に確定しなければなりません。

領域にラベル名を定義する場合は、制御命令の前にラベルを記述してください。

ラベル名には、必ずコロン (:) を記述してください。

オペランドに指定できる値の最大値は 7FFFFFFFH です。

.BLKW

2 バイト単位で、指定した個数の RAM 領域を確保します。

[指定形式]

△ .BLKW △ < オペランド > △ < ラベル名 :> △ .BLKW △ < オペランド >

[詳細機能]

2 バイト単位で、指定した個数の RAM 領域を確保します。

確保した RAM のアドレスに、ラベル名を定義することもできます。

例 symbolEQU 1
.SECTION area,DATA
work1:BLKW 1
work2:BLKW symbol
.BLKW symbol+1

[備考]

本制御命令は必ず、DATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ” ,DATA ” を記述することでセクション属性が DATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドの値は、アセンブル時に確定しなければなりません。

領域にラベル名を定義する場合は、制御命令の前にラベルを記述してください。

ラベル名には、必ずコロン (:) を記述してください。

オペランドに指定できる値の最大値は 3FFFFFFFH です。

.BLKL

4 バイト単位で、指定した個数の RAM 領域を確保します。

[指定形式]

△ .BLKL △ < オペランド > △ < ラベル名 :> △ .BLKL △ < オペランド >

[詳細説明]

4 バイト単位で、指定した個数の RAM 領域を確保します。

確保した RAM のアドレスに、ラベル名を定義することもできます。

例 symbolEQU 1
.SECTION area,DATA
work1:BLKL 1
work2:BLKL symbol
.BLKL symbol+1

[備考]

本制御命令は必ず、DATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ” ,DATA ” を記述することでセクション属性が DATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドの値は、アセンブル時に確定しなければなりません。

領域にラベル名を定義する場合は、制御命令の前にラベルを記述してください。

ラベル名には、必ずコロン (:) を記述してください。

オペランドに指定できる値の最大値は 1FFFFFFFH です。

.BLKD

8 バイト単位で、指定した個数の RAM 領域を確保します。

[指定形式]

△ .BLKD △ < オペランド >
△ < ラベル名 :> △ .BLKD △ < オペランド >

[詳細説明]

8 バイト単位で、指定した個数の RAM 領域を確保します。

確保した RAM のアドレスに、ラベル名を定義することもできます。

例 symbolEQU 1

.SECTION area,DATA

work1:BLKD 1

work2:BLKD symbol

.BLKD symbol+1

[備考]

本制御命令は必ず、DATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ” ,DATA ” を記述することでセクション属性が DATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドの値は、アセンブル時に確定しなければなりません。

領域にラベル名を定義する場合は、制御命令の前にラベルを記述してください。

ラベル名には、必ずコロン (:) を記述してください。

オペランドに指定できる値の最大値は 0FFFFFFFH です。

.BYTE

1 バイト長の固定データを ROM に格納します。

[指定形式]

△ .BYTE △ < オペランド >
△ < ラベル名 :> △ .BYTE △ < オペランド >

[詳細説明]

1 バイト長の固定データを ROM に格納します。

データを格納したアドレスに、ラベル名を定義することもできます。

例 <endian=little オプション指定時 >

```
.SECTION value,ROMDATA
.BYTE 1
.BYTE "data"
.BYTE symbol
.BYTE symbol+1
.BYTE 1,2,3,4,5
.END
```

<endian=big オプション指定時 >

```
.SECTION  program,CODE,ALIGN=4
MOV.L  R1,R2
.ALIGN 4
.BYTE 080H,00H,00H,00H
.END
```

[備考]

本制御命令は必ず、ROMDATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ",ROMDATA" を記述することでセクション属性が ROMDATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドに文字・文字列を記述するときは、クォーテーション (‘) またはダブルクォーテーション (“) で囲ってください。このとき格納されるデータは、文字の ASCII コードになります。

ラベルを定義する場合には、制御命令の前にラベル名を記述してください。

ラベル名には、必ずコロン (:) を記述してください。

endian=big オプション指定時、本制御命令は次の条件に当てはまるセクション内にのみ記述できます。条件に当てはまらないセクション内に記述した場合はエラーとなります。

(1) ROMDATA セクション

```
.SECTION data,ROMDATA
```

(2) セクション定義の際のアドレス補正に 4、もしくは 8 を指示している相対アドレス形式の CODE セクション

```
.SECTION program,CODE,ALIGN=4
```

(3) 絶対アドレス形式の CODE セクション

```
.SECTION program,CODE
.ORG 0fff00000H
```

endian=big オプション指定時、本制御命令をセクション属性が CODE のセクションに記述する場合、直前の行にアドレス補正制御命令 (.ALIGN 4) を記述し、4 バイト境界に配置されるようにしてください。記述されていない場合、アセンブラはウォーニングを出力し、自動的に 4 バイト境界に配置します。

.WORD

2 バイト長の固定データを ROM に格納します。

[指定形式]

△ .WORD △ < オペランド >
△ < ラベル名 :> △ .WORD △ < オペランド >

[詳細説明]

2 バイト長の固定データを ROM に格納します。

データを格納したアドレスに、ラベル名を定義することもできます。

例 .SECTION value,ROMDATA

.WORD1

.WORDsymbol

.WORDsymbol+1

.WORD1,2,3,4,5

.END

[備考]

本制御命令は必ず、ROMDATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ",ROMDATA" を記述することでセクション属性が ROMDATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドに文字・文字列を記述することはできません。

ラベルを定義する場合には、制御命令の前にラベル名を記述してください。

ラベル名には、必ずコロン (:) を記述してください。

.LWORD

4 バイト長の固定データを ROM に格納します。

[指定形式]

△ .LWORD △ < オペランド >
△ < ラベル名 :> △ .LWORD △ < オペランド >

[詳細説明]

4 バイト長の固定データを ROM に格納します。

データを格納したアドレスに、ラベル名を定義することもできます。

例 .SECTION value,ROMDATA

.LWORD1

.LWORDsymbol

.LWORDsymbol+1

.LWORD1,2,3,4,5

.END

[備考]

本制御命令は必ず、ROMDATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ",ROMDATA" を記述することでセクション属性が ROMDATA となります。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

オペランドには数値、シンボル、式を記述できます。

オペランドに文字・文字列を記述することはできません。

ラベルを定義する場合には、制御命令の前にラベル名を記述してください。

ラベル名には、必ずコロン (:) を記述してください。

.FLOAT

4 バイト長の固定データを ROM に格納します。

[指定形式]

△ .FLOAT △ < 数値 > △ < ラベル名 :> △ .FLOAT △ < 数値 >

[詳細説明]

4 バイト長の固定データを ROM に格納します。

データを格納したアドレスに、ラベル名を定義することもできます。

例 .FLOAT 5E2

constant:FLOAT 5e2

[備考]

本制御命令は必ず、ROMDATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ",ROMDATA" を記述することでセクション属性が ROMDATA となります。

オペランドに浮動小数点数を記述してください。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

ラベルを定義する場合には、制御命令の前にラベル名を記述してください。

ラベル名には、必ずコロン (:) を記述してください。

.DOUBLE

8 バイト長の固定データを ROM に格納します。

[指定形式]

△ .DOUBLE △ < 数値 > △ < ラベル名 :> △ .DOUBLE △ < 数値 >

[詳細説明]

8 バイト長の固定データを ROM に格納します。

データを格納したアドレスに、ラベル名を定義することもできます。

例 DOUBLE 5E2

constant:DOUBLE 5e2

[備考]

本制御命令は必ず、ROMDATA 属性のセクション内に記述してください。セクション定義の際に、セクション名に続けて ",ROMDATA" を記述することでセクション属性が ROMDATA となります。

オペランドに浮動小数点数を記述してください。

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

ラベルを定義する場合には、制御命令の前にラベル名を記述してください。

ラベル名には、必ずコロン (:) を記述してください。

```
.ALIGN
```

本制御命令を記述した直後の行のコードを格納するアドレスを 2、4 または 8 バイト
アライメントに補正します。

[指定形式]

```
△ .ALIGN △ < アライメント補正値 >
< アライメント補正値 >:[2|4|8]
```

[詳細説明]

本制御命令を記述した直後の行のコードを格納するアドレスを 2、4 または 8 バイトアライメントに補正します。

セクション属性が CODE または、ROMDATA の場合は、アドレスを補正した結果、空になったところに NOP のコード（03H）を書き込みます。

セクション属性が DATA の場合は、アドレス補正のみ行います。

例 .SECTION program,CODE,ALIGN=4
MOV.L R1, R2
.ALIGN 4; アドレスを 4 の倍数に補正
RTS
.END

[備考]

本制御命令は、次の条件に当てはまるセクション内に記述できます。

(1) セクション定義の際にアドレス補正を指示している相対アドレス形式セクション
.SECTION program,CODE,ALIGN=4

(2) 絶対アドレス形式セクション
.SECTION program,CODE
.ORG 0fff00000H

相対アドレス形式のセクションで .SECTION 制御命令行で ALIGN 指定のされていないセクションに本制御命令を記述した場合は、ウォーニングが出力されます。

指定した値がセクションの境界調整数よりも大きい場合は、ウォーニングが出力されます。

4. 2. 5 マクロ制御擬似命令

制御命令自身はデータを生成しません。命令に対する機械語コードの生成を制御する制御命令です。アドレスの更新は行いません。

マクロ機能および繰り返しマクロ機能を定義するための制御命令です。

表 4―29 マクロ制御命令

制御命令	機能内容
.MACRO	マクロ名を定義します。マクロボディの始まりを定義します。
.EXITM	マクロボディの展開を中止します。
.LOCAL	マクロ内ローカルラベルを宣言します。
.ENDM	マクロボディの終了を示します。
.MREPEAT	繰り返しマクロボディの始まりを示します。
.ENDR	繰り返しマクロボディの終了を示します。
..MACPARA	マクロ呼び出しの実引数の個数を値として持ちます。
..MACREP	繰り返しマクロボディの展開回数を値として持ちます。
.LEN	指定した文字列の文字数を値として持ちます。
.INSTR	指定した文字列の中で指定した文字列の始まる位置を値として持ちます。
.SUBSTR	指定した文字列の中で指定した位置から指定した文字数分の文字を切り出します。

.MACRO

マクロ名を定義します。

[指定形式]

[**マクロ定義**]
△ < **マクロ名** > △ .MACRO[< **仮引数** >[,...]]
△**ボディ**
△ .ENDM
[**マクロ呼び出し**]
△ < **マクロ名** > △ [< **実引数** >[,...]]

[詳細説明]

マクロ名を定義します。

マクロ定義の始まりを示します。

例・例 1

[マクロ定義例]

```
name.MACRO  string
.BYTE  'string'
.ENDM
```

[マクロ呼び出し例 1]

```
name"name,address"
```

.BYTE'name,address'

[**マクロ呼び出し例** 2]

name(name,address)

.BYTE(name,address)'

• **例** 2

```
mac.MACROp1,p2,p3
.IF.MACPARA == 3
.IFp1' == 'byte'
MOV.B  #p2,[p3]
.ELSE
MOV.W  #p2,[p3]
.ENDIF
.ELIF.MACPARA == 2
.IF'p1' == 'byte'
MOV.B  #p2,[R3]
.ELSE
MOV.W  #p2,[R3]
.ENDIF
.ELSE
MOV.W  R3,R1
.ENDIF
.ENDM

macword,10,R3; マクロ呼び出し

.IF3 == 3; マクロ展開後コード
.ELSE
MOV.W  #10,[R3]
.ENDIF
```

[**備考**]

マクロ名は必ず記述してください。

マクロ名は、「4.1.2　**名前」の「名前の記述規則」に従ってください。**

マクロ仮引数の名前は、「4.1.2　**名前」の「名前の記述規則」に従ってください。**

マクロ仮引数の名前は、ネストしているマクロ定義を含めて、異なる名前で定義してください。

仮引数を複数定義する場合は、仮引数をカンマ (,) **で区切って記述してください。**

制御命令 ".MACRO" **のオペランドに記述した仮引数は、必ずマクロボディ内に記述してください。**

マクロ名と実引数の間には、必ず空白文字またはタブを記述してください。

実引数は、マクロ呼び出しの際に仮引数に対応させて記述してください。

特殊文字を実引数に記述する場合は、ダブルクォーテーションで囲って記述してください。

実引数には、ラベル、グローバルラベルおよびシンボルが記述できます。

実引数には式が記述できます。

仮引数と実引数は、左から記述されている順に置き換えられます。

仮引数が定義されていて、マクロ呼び出しで実引数の記述がない場合は、仮引数にあたる部分のコードは出力されません。

仮引数の数が、実引数の数より多い場合は、対応する実引数がない仮引数にあたる部分のコードは出力されません。

ボディに記述した仮引数をシングルクォーテーション (') で囲った場合は、対応する実引数をシングルクォーテーションで囲って出力されます。

１つの実引数がカンマ (,) を含む場合に、括弧 (()) で囲った場合は、括弧を含めて変換されます。

実引数の数が、仮引数の数より多い場合は、対応する仮引数がない実引数については処理されません。

ダブルクォーテーションで囲った文字列は、全てその文字列そのものを示します。仮引数をダブルクォーテーションで囲わないでください。

仮引数は 80 個まで記述できます。

1 行に記述できる文字数の範囲内で最大 80 個まで記述できます。

実引数と仮引数の数が合わない場合は、アセンブラはウォーニングメッセージを出力します。

```
.EXITM
```

マクロボディの展開を中止し、最も近い ".ENDM" に制御を渡します。

[詳細説明]

マクロボディの展開を中止し、最も近い ".ENDM" に制御を渡します。

例 data1.MACRO value
.IFvalue == 0
.EXITM
.ELSE
.BLKBvalue
.ENDIF
.ENDM

data10; マクロ呼び出し

.IF0 == 0; マクロ展開後コード
.EXITM
.ENDIF

[備考]

マクロ定義のボディ内に記述してください。

```
.LOCAL
```

オペランドに記述されたラベルがマクロローカルラベルであることを宣言します。

[指定形式]

.LOCAL △ < ラベル名 >[,...]

[詳細説明]

オペランドに記述されたラベルがマクロローカルラベルであることを宣言します。

マクロローカルラベルは、異なるマクロ定義およびマクロ定義外であれば、同一の名前を複数個記述できます。

例 nameMACRO

```
  .LOCALm1; 'm1' is macro local label
m1:
  nop
  bram1
  .ENDM
```

[備考]

本制御命令は、必ずマクロボディ内に記述してください。

本制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

本制御命令によるマクロローカルラベル宣言は、ラベル名を定義するより前に記述してください。

マクロローカルラベル名は、「4.1.2 名前」の「名前の記述規則」に従ってください。

本制御命令のオペランドは、カンマで区切って複数のラベルを記述できます。このときの最大ラベル数は 100 個までです。

マクロ定義がネストしている場合は、マクロ定義内で定義を行っているマクロ内のマクロローカルラベルは、同一名を使用できません。

インクルードファイルの内容を含む、1 つのアセンブリ言語ファイルに記述できるマクロローカルラベルは 65535 個までです。

.ENDM

1 つのマクロ定義のボディが終了することを示します。

[指定形式]

< マクロ名 > △ .MACRO △ボディ △ .ENDM

[詳細説明]

1 つのマクロ定義のボディが終了することを示します。

例 lda.MACRO

```
MOV.L  #value,R3
.ENDM
```

lda0; MOV.L #0,R3 に展開される

.MREPEAT

繰り返しマクロの始まりを示します。

[指定形式]

```
[< ラベル >: ] △ .MREPEAT △ < 数値 >
△ボディ
△ .ENDR
```

[詳細説明]

繰り返しマクロの始まりを示します。

ボディを指定した数値回、繰り返して展開します。

繰り返し回数は、1 から 65535 の間の数を指定できます。

65535 レベルまでのネストができます。

本制御命令を記述した場所に、マクロボディを展開します。

例 rep.MACRO num

.MREPEATnum

.IFnum > 49

.EXITM

.ENDIF

nop

.ENDR

.ENDM

rep3; マクロ呼び出し

nop; マクロ展開後コード

nop

nop

[備考]

オペランドは必ず記述してください。

本制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

本制御命令行の先頭にラベルを記述できます。

オペランドには、シンボルを記述できます。

前方参照となるシンボルは記述できません。

オペランドには、式が記述できます。

ボディには、マクロ定義およびマクロ呼び出しが記述できます。

ボディ内に制御命令 ".EXITM" を記述できます。

```
.ENDR
```

繰り返しマクロの終了を示します。

[指定形式]

```
[< ラベル >:] △ .MREPEAT △ < 数値 >
△ボディ
△ .ENDR
```

[詳細説明]

繰り返しマクロの終了を示します。

[備考]

必ず制御命令 ".MREPEAT" に対応させて記述してください。

```
..MACPARA
```

マクロ呼び出しの実引数の個数を示します。

[指定形式]

```
..MACPARA
```

[詳細説明]

マクロ呼び出しの実引数の個数を示します。

".MACRO" によるマクロ定義のボディ内に記述できます。

例マクロ実引数の数を判断して、条件アセンブルを行います。

```
.GLBmem
name.MACRO  f1,f2
.IF.MACPARA == 2
ADD  f1,f2
.ELSE
ADD  R3,f1
.ENDIF
.ENDM

name  mem    ; マクロ呼び出し

.ELSE       ; マクロ展開後コード
ADD  R3,mem
.ENDIF
```

[備考]

本制御命令は式の項として記述できます。

".MACRO" によるマクロボディの外に記述した場合、値は 0 となります。

```
.MACREP
```

繰り返しマクロが展開されている回数を示します。

[指定形式]

```
.MACREP
```

[詳細説明]

繰り返しマクロが展開されている回数を示します。

".MREPEAT" によるマクロ定義のボディ内に記述できます。

条件アセンブルのオペランドに記述できます。

例 mac.MACRO value,reg

.MREPEATvalue

MOV.B#0,..MACREP[reg]

.ENDR

.ENDM

mac3,R3; マクロ呼び出し

.MREPEAT3; マクロ展開後コード

MOV.B#0,1[R3]

MOV.B#0,2[R3]

MOV.B#0,3[R3]

.ENDR

.ENDM

[備考]

本制御命令は式の項として記述できます。

".MACRO" によるマクロボディの外に記述した場合、値は 0 となります。

.LEN

オペランドに記述した文字列の長さを示します。

[指定形式]

.LEN △ {"< 文字列 >"}
.LEN △ {'< 文字列 >'}

[詳細説明]

オペランドに記述した文字列の長さを示します。

例 bufset.MACRO f1

buffer:BLKB .LEN{'f1'}

.ENDM

bufset Sample ; マクロ呼び出し

buffer: .BLKB 6 ; マクロ展開後コード

[備考]

オペランドは、必ず {} で囲ってください。

本制御命令とオペランドの間に空白文字またはタブが記述できます。

文字列には、空白文字およびタブを含む文字が記述できます。
文字列は、必ずクォーテーションで囲って記述してください。
本制御命令を式の項に記述できます。
マクロの実引数の文字列長を求める場合は、仮引数名をシングルクォーテーションで囲って記述してください。ダブルクォーテーションで囲った場合は、仮引数として指定した文字列の長さを示します。

.INSTR

オペランドで指定した文字列のなかで、検出文字列が始まる位置を示します。
[詳細説明]
オペランドで指定した文字列のなかで、検出文字列が始まる位置を示します。
文字列の検索を開始する位置を指定できます。

例 指定した文字列 (japanese) の先頭 (top) からの、"se" 文字列の位置 (7) を取り出します。

```
top.EQU   1
point_set.MACRO  source,dest,top
point.EQU    .INSTR{'source','dest',top}
.ENDM
  point_set  japanese,se,1   ; マクロ呼び出し

  point     .EQU   7      ; マクロ展開後コード
```

[備考]
オペランドは、必ず {} で囲ってください。
文字列、検出文字列および検索開始位置は、必ず記述してください。
文字列、検出文字列および検索開始位置は、カンマで区切って記述してください。
カンマの前後には、空白文字およびタブは記述できません。
検索開始位置は、シンボルを記述できます。
検索開始位置を 1 とした場合は、文字列の先頭を示します。
本制御命令は、式の項に記述できます。
文字列よりも、検索文字列が長い場合の値は 0 となります。文字列のなかに、検索文字列が含まれていなかった場合の値は 0 となります。文字列の長さよりも、検索開始位置の値が大きかった場合の値は 0 となります。
マクロの実引数を検出条件としてマクロを展開したい場合は、仮引数名をシングルクォーテーションで囲って記述してください。ダブルクォーテーションで囲って記述した場合は、その文字列を検出条件としてマクロを展開します。

.SUBSTR

文字列の指定した位置から、指定した文字列を取り出します。
[指定形式]

.SUBSTR △ { "< 文字列 >",< 切り出し開始位置 >,< 切り出し文字数 > }
.SUBSTR △ { '< 文字列 >',< 切り出し開始位置 >,< 切り出し文字数 > }

[詳細説明]
文字列の指定した位置から、指定した文字列を取り出します。

例 マクロの実引数として与えられた文字列の長さを、".MREPEAT" のオペランドに与えます。

"..MACREP" は、".BYTE" の行を 1 回展開するごとに、1 → 2 → 3 → 4 と増加します。

したがって、マクロの実引数として与えられた文字列の先頭から順に 1 文字ずつ、".BYTE" のオペランドに与えることになります。

```
name.MACRO  data
.MREPEAT.LEN{'data'}
.BYTE.SUBSTR{'data',..MACREP,1}
.ENDR
.ENDM

name  ABCD   ; マクロ呼び出し

.BYTE  "A"   ; マクロ展開後コード
.BYTE  "B"
.BYTE  "C"
.BYTE  "D"
```

[備考]

オペランドは、必ず {} で囲ってください。

文字列、切り出し開始位置および切り出し文字数は、必ず記述してください。

文字列、切り出し開始位置および切り出し文字数は、カンマで区切って記述してください。

切り出し開始位置および切り出し文字数には、シンボルが記述できます。切り出し開始位置を 1 とした場合は、文字列の先頭を示します。

文字列には、空白文字およびタブを含む文字が記述できます。

文字列は、必ずクォーテーションで囲って記述してください。

文字列の長さよりも切り出し開始位置の値が大きい場合の値は 0 となります。文字列の長さよりも切り出し文字数の値が大きい場合の値は 0 となります。切り出し文字数を 0 とした場合の値は 0 となります。

マクロの実引数を切り出し条件としてマクロを展開したい場合は、仮引数名をシングルクォーテーションで囲って記述してください。ダブルクォーテーションで囲って記述した場合は、その文字列を切り出し条件としてマクロを展開します。

4. 2. 6 コンパイラ専用制御擬似命令

コンパイラがアセンブリ言語ソースファイルを生成する際、C 言語の機能をアセンブラで適切に処理させるため、下記の制御命令を出力することがあります。

コンパイラが生成したアセンブリ言語ソースファイルを利用する場合、これらの制御命令の内容を変更せず、そのまま使用してください。また、ユーザアセンブリプログラム作成時には、これらの制御命令は使用しないでください。

表 4―30 コンパイラ専用制御命令

制御命令	内容
._LINE_TOP	#pragma inline_asm で指定された関数を展開した場合に出力されます。
._LINE_END	
.SWSECTION	switch 文で分岐テーブルを使用した場合に出力されます。
.SWMOV	
.SWITCH	
.INSTALIGN	instalign4, instalign8 オプション、または #pragma instalign4, #pragma instalign8 を使用した場合に出力されます。

4. 3 制御命令

この節では，制御命令について説明します。

制御命令とは，アセンブラの動作に対し細かい指示を与えるものです。

4. 3. 1 概要

制御命令は，アセンブラの動作に対し細かい指示を与えるもので，ソース中に記述します。

制御命令は，オブジェクト・コード生成の対象とはなりません。

種類	擬似命令
アセンブルリスト制御命令	.LIST
条件アセンブル制御命令	.IF, .ELIF, .ELSE, .ENDIF
拡張機能制御命令	.ASSERT, ?, @, ..FILE, .STACK, .LINE, .DEFINE

以降，各制御命令について詳細な説明を行います。

4. 3. 2 アセンブルリスト制御命令

アセンブルリストファイルに出力する情報や、アセンブルリストファイルのフォーマットの制御を行う制御命令です。なお、コード生成には影響を与えません。

表 4―31 アセンブルリスト制御命令

制御命令	機能内容
.LIST	アセンブルリストファイルを生成する際に、アセンブリ言語ファイルの行単位でアセンブルリストファイルへの出力を行うか行わないかを制御します。

```
.LIST
```

アセンブルリストファイルへの行の出力を停止 (OFF) することができます。

[指定形式]

```
.LIST △ [ON|OFF]
```

[詳細説明]

アセンブルリストファイルへの行の出力を停止 (OFF) することができます。

リストへの行の出力を停止している範囲においても、エラー発生行についてはアセンブルリストファイルに出力します。

アセンブルリストファイルへの行の出力を開始 (ON) することができます。

本制御命令を指定しない場合は、全ての行をアセンブルリストファイルに出力します。

例 .LIST ON
.LIST OFF

[備考]

制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

行の出力を停止する場合は、オペランドに 'OFF' を記述してください。

行の出力を開始する場合は、オペランドに 'ON' を記述してください。

4.3.3 条件アセンブル制御命令

アセンブル制御擬似命令を使って、指定した範囲の行のアセンブルを行うか、行わないかを指定できます。

.IF, .ELIF, .ELSE, .ENDIF

表 4―32 条件アセンブル制御命令

制御命令	機能内容
.IF	条件アセンブルブロックの始まりを示します。条件の判定を行います。
.ELIF	二つ以上の条件ブロックを記述する場合に、二つ目以降の条件を判定します。
.ELSE	全ての条件が偽である場合に、アセンブルを行うブロックの始まりを示します。
.ENDIF	条件アセンブルブロックの終了を示します。

[指定形式]

```
.IF △条件式
ボディ
.ELIF △条件式
ボディ
.ELSE
ボディ
.ENDIF
```

[詳細説明]

.IF, .ELIF に記述した条件に従いアセンブルを行うブロックを制御します。

.IF, .ELIF のオペランドに記述した条件を判定し、真であれば以降に続くボディをアセンブルします。

条件が真である場合にアセンブルされる行は、制御命令 ".ELIF",".ELSE" および ".ENDIF" 行の前までです。

条件アセンブルブロック内には、アセンブリ言語ファイルに記述可能な全ての命令を記述できます。

条件式の結果によって、条件アセンブルが行われます。

例＜条件式の記述例＞

```
sym < 1
sym+2 < data1
sym+2 < data1+2
'smp1' == name
```

＜条件アセンブル記述例＞

```
.IF TYPE==0
.byte "Proto Type Mode"
.ELIF TYPE>0
.byte "Mass Produciton Mode"
.ELSE
.byte "Debug Mode"
.ENDIF
```

備　考 .IF, .ELIF 制御命令のオペランドには、必ず条件式を記述してください。

.IF, .ELIF 制御命令とオペランドの間には、必ず空白文字またはタブを記述してください。

条件式は、制御命令のオペランドに 1 つだけ記述できます。

条件式には、必ず条件演算子を記述してください。

次に示す演算子が記述できます。

表 4.33　.IF および .ELIF 制御命令の条件演算子

条件演算子	内容
＞	左辺値が右辺値より大きい場合に真となります
＜	左辺値が右辺値より小さい場合に真となります
＞＝	左辺値が右辺値より大きいか等しい場合に真となります
＜＝	左辺値が右辺値より小さいか等しい場合に真となります
＝＝	左辺値と右辺値が等しい場合に真となります
！＝	左辺値と右辺値が等しくない場合に真となります

条件式は符号付き 32 ビットで演算します。

条件演算子の左辺および右辺には、シンボルが記述できます。

条件演算子の左辺および右辺には、式が記述できます。式は、「4.1.5　オペランド部の記述方法」の「(2) 式」に従って記述してください。

条件演算子の左辺および右辺には、文字列が記述できます。文字列は、必ずシングルクォーテーション (') またはダブルクォーテーション (") で囲って記述してください。このとき、文字列の大小は、文字コードの値で判定されます。

例)

"ABC"<"CBA" -> 414243 < 434241 で真となります。

"C" < "A" -> 43 < 41 で偽となります。

条件演算子の前後には、空白文字またはタブが記述できます。

条件式は、制御命令 ".IF" および ".ELIF" のオペランドに記述できます。

演算結果のオーバフローは判断しません。

シンボルは、前方参照（本制御命令行より後に定義されているシンボルを参照）はできません。

前方参照のシンボルや、未定義のシンボルを記述した場合は、値を 0 として式を判定します。

4. 3. 4 拡張機能制御命令

コード生成には影響を与えない制御命令です。

表 4―33 拡張機能制御命令

制御命令	機能内容
.ASSERT	オペランドに記述した文字列を標準エラー出力またはファイルに出力します。
?	テンポラリラベルの定義と参照を指定します。
@	@ の前後の文字列を連結し、1 つの文字列として扱います。
..FILE	アセンブラが処理を行っているアセンブリ言語ファイル名を示します。
.STACK	指定したシンボルに対してスタック値を定義します。
.LINE	行番号を変更します。
.DEFINE	置き換えシンボルを定義します。

.ASSERT

オペランドに記述した文字列をアセンブル時に、標準エラー出力に出力します。

[指定形式]

.ASSERT △“ < 文字列 > ”
.ASSERT △“ < 文字列 > ”> △ < ファイル名 >
.ASSERT △“ < 文字列 > ”>> △ < ファイル名 >

[詳細説明]

オペランドに記述した文字列をアセンブル時に、標準エラー出力に出力します。

ファイル名を指定した場合は、オペランドに記述した文字列をファイルに出力します。

ファイル名に絶対パスを記述した場合は、記述したディレクトリにファイルを生成します。

ファイル名に絶対パスを記述していない場合

(1)output オプションで指定したファイル名にディレクトリ指定がない場合は、本制御命令で指定されたファイルをカレントディレクトリに生成します。

(2)output オプションで指定したファイル名にディレクトリ指定がある場合は、本制御命令で指定されたファイル名に output オプションで指定されたファイルのディレクトリを付加したファイルを生成します。

(3)output オプションが指定されていない場合、アセンブラ起動時にコマンド行で指定したファイルと同じディレクトリにファイルを生成します。

ファイル名に制御命令 ” ..FILE ” を記述した場合は、アセンブラ起動時にコマンド行で指定したファイルと同じディレクトリにファイルを生成します。

例 sample.dat ファイルにメッセージを出力します。

.ASSERT "string" > sample.dat

sample.dat ファイルにメッセージを追加します。

.ASSERT "string" >> sample.dat

現在処理中のファイルと同じ名前で拡張子を除くファイル名のファイルにメッセージを出力します。

.ASSERT "string" > ..FILE

[備考]

オペランドと制御命令の間には、必ず空白文字またはタブを記述してください。

オペランドの文字列は必ずダブルクォーテーションで囲ってください。

文字列をファイルに出力するときは、">" または ">>" に続けてファイル名を指定してください。

> は、新規にファイルを生成して、そのファイルにメッセージを出力します。以前に同一名のファイルがある場合は、そのファイルに上書きされます。

>> は、ファイルの内容に追加して、メッセージを出力します。指定したファイルが存在しない場合は、新しくファイルを生成します。

">" または ">>" の前後には、空白文字またはタブを記述できます。

ファイル名に制御命令 "..FILE" を記述できます。

?

テンポラリラベルを定義します。

[指定形式]

```
?:
△ < ニーモニック > △ ?+
△ < ニーモニック > △ ?-
```

[詳細説明]

テンポラリラベルを定義します。

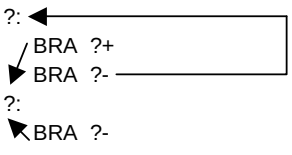
直前または直後に定義されたテンポラリラベルを参照します。

同一ファイル内で定義および参照が可能です。

ファイル内に 65535 個までのテンポラリラベルが定義できます。このとき、ファイル内に ” .INCLUDE ” が記述されている場合は、インクルードファイル内のテンポラリラベルを含み 65535 個までの記述ができます。

アセンブルリストファイルには、テンポラリラベルとして変換された結果が出力されます。

例



矢印のテンポラリラベルを指す

[備考]

テンポラリラベルとして定義したい行に “ ?: ” を記述してください。

直前に定義したテンポラリラベルを参照したい場合は、命令のオペランドに “ ?- ” を記述してください。

直後に定義したテンポラリラベルを参照したい場合は、命令のオペランドに “ ?+ ” を記述してください。

参照できるラベルは、直前と直後のラベルだけです。

```
@
```

マクロ引数、マクロ変数、予約シンボル、制御命令 "..FILE" **の展開ファイル名および指定文字列を連結します。**

[**指定形式**]

```
< 文字列 >@< 文字列 >[@< 文字列 > ...]
```

[**詳細説明**]

マクロ引数、マクロ変数、予約シンボル、制御命令 "..FILE" **の展開ファイル名および指定文字列を連結します。**

例 ファイル名の連結例：

現在処理中のファイル名が sample1.src **の場合、**sample.dat **ファイルにメッセージを出力します。**

```
.ASEERT   "sample" > ..FILE@.dat
```

文字列の連結例：

```
mov_nibble .MACRO p1,src,dest
 MOV.@p1 src,dest
 .ENDM

mov_nibble  W,R1,R2; マクロ呼び出し

 MOV.W R1,R2 ; マクロ展開後コード
```

[**備考**]

本制御命令の前後に記述した空白文字およびタブは、文字列として連結します。

本制御命令の前後には、文字列が記述できます。

@ を文字データ (40H) **として記述する場合は、ダブルクォーテーション** (") **で囲んでください。@ を含む文字列をシングルクォーテーション** (') **で囲った場合は、@ の前後の文字列を連結します。**

一行に複数回記述できます。

連結した文字列を名前とする場合は、本制御命令の前後に空白文字およびタブを記述しないでください。

```
..FILE
```

アセンブラが処理中のファイル名に展開されます（アセンブリ言語ファイル名またはインクルードファイル）。

[**指定形式**]

```
..FILE
```

[**詳細説明**]

アセンブラが処理中のファイル名に展開されます（アセンブリ言語ファイル名またはインクルードファイル）。

例 アセンブリ言語ファイル名が "sample.src" **の場合、**"sample" **ファイルにメッセージを出力します。**

```
.ASSERT "sample" > ..FILE
```

アセンブリ言語ファイル名が "sample.src" **の場合、**"sample.inc" **ファイルをインクルードします。**

```
.INCLUDE ..FILE@.inc
```

上記の行が、"sample.src" **ファイルでインクルードしている** "incl.inc" **内に記述されている場合、通常、**"incl.mes" **に文字列を出力します。**

.ASSERT "sample" >..FILE@.mes

[備考]

制御命令 ".ASSERT" および制御命令 ".INCLUDE" のオペランドに記述できます。

本制御命令で読み込まれるファイル名は、ファイルの拡張子およびパスを除いた部分です。

.STACK

シンボルに対して、Call Walker で表示するスタック使用量を定義します。

[指定形式]

.STACK △ < 名前 >=< 数値 >

[詳細説明]

シンボルに対して、Call Walker で表示するスタック使用量を定義します。

例 .STACK SYMBOL=100H

[備考]

1 つのシンボルに対して定義できるスタック値は 1 度のみ有効とします。

2 度以上指定した場合は、その定義を無効とします。また、指定できるスタック値は、

0H ～ 0FFFFFFFCH の範囲の 4 の倍数のみとし、それ以外を指定した場合はその定義を無効とします。

＜数値＞は定数値とし、かつ前方参照シンボル、外部参照シンボル、相対アドレスシンボルを使わずに指定してください。

.LINE

アセンブラのエラーメッセージあるいはデバッグ時に参照する行番号とファイル名を変更します。

[指定形式]

.LINE △ < ファイル名 >,< 行番号 > .LINE △ < 行番号 >

[詳細説明]

アセンブラのエラーメッセージあるいはデバッグ時に参照する行番号とファイル名を変更します。

プログラム内の最初の .LINE 以降は次の .LINE まで行番号、ファイル名を更新しません。

コンパイラは、デバッグオプションを指定してアセンブリ言語ファイルを出力する時に C 言語ソースファイル行に対応する .LINE を生成します。

ファイル名を省略するとファイル名は変更されず、行番号だけが変更されます。

例 .LINE "C:\asm\test.c",5

.DEFINE

シンボルに文字列を定義します。

[指定形式]

< シンボル名 > △ .DEFINE △ < 文字列 > < シンボル名 > △ .DEFINE △ '< 文字列 >' < シンボル名 > △ .DEFINE △ "< 文字列 >"

[詳細説明]

シンボルに文字列を定義します。
シンボルは再定義が可能です。

例 X_HI.DEFINE R1
MOV.L #0, X_HI

[備考]
空白文字またはタブを含む文字列を定義する場合は、必ずシングルクォーテーション (') または、ダブルクォーテーション (") で囲って記述してください。
本制御命令で定義されたシンボルは、外部参照指定ができません。
本制御命令と .EQU 制御命令で同名のシンボルを宣言した場合、先に記述した方が優先されます。

4.4 マクロ名

オプション指定やバージョンに合わせて、以下のようなプリデファインドマクロが定義されます。

表 4―34 コンパイラのプリデファインドマクロ

オプション	プリデファインドマクロ	
cpu=rx600 cpu=rx200	#define __RX600 #define __RX200	1 1
endian=big endian=little	#define __BIG #define __LIT	1 1
dbl_size=4 dbl_size=8	#define __DBL4 #define __DBL8	1 1
int_to_short	#define __INT_SHORT	1
signed_char unsigned_char	#define __SCHAR #define __UCHAR	1 1
signed_bitfield unsigned_bitfield	#define __SBIT #define __UBIT	1 1
round=zero round=nearest	#define __ROZ #define __RON	1 1
denormalize=off denormalize=on	#define __DOFF #define __DON	1 1
bit_order=left bit_order=right	#define __BITLEFT #define __BITRIGHT	1 1
auto_enum	#define __AUTO_ENUM	1
library=function library=intrinsic	#define __FUNCTION_LIB #define __INTRINSIC_LIB	1 1
fpu	#define __FPU	1
-	#define __RENESAS__ (*1)	1
-	#define __RENESAS_VERSION__ (*1)	0xAABBCC00 (*2)

-	#define __RX (*1)	1
pic	#define __PIC	1
pid	#define __PID	1

注 1. オプションに関わらず常に定義されます。

2. バージョンが AA.BB.CC の場合、__RENESAS_VERSION__ の値は 0xAABBCC00 となります。

例) V2.00.00 の場合、#define __RENESAS_VERSION__ 0x02000000

表 4―35 アセンブラのプリデファインドマクロ

オプション	プリデファインドマクロ	
cpu=rx600 cpu=rx200	__RX600 __RX200	.DEFINE 1 .DEFINE 1
endian=big endian=little	__BIG __LITTLE	.DEFINE 1 .DEFINE 1
-	__RENESAS_VERSION__ (*1)	.DEFINE 020000000H (*2)
-	__RX (*1)	.DEFINE 1

注 1. オプションに関わらず常に定義されます。

2. バージョンが AA.BB.CC の場合、__RENESAS_VERSION__ の値は 0xAABBCC00 となります。

例) V2.00.00 の場合、__RENESAS_VERSION__ .DEFINE 02000000H

4.5 予約語

アセンブラでは、アセンブラ制御命令やニーモニックなどと同一の文字列を予約語として扱います。予約語は特別な機能を持っているため、アセンブリ言語ファイル中でラベル名やシンボル名に使用することはできません。また、予約語は大文字と小文字を区別しません。"ABS" と "abs" は同じ予約語となります。

予約語には以下のものがあります。

(1) アセンブラ制御命令

アセンブラ制御命令と、ピリオド (.) で始まる文字列全てを予約語とします。

(2) ニーモニック

RX ファミリのニーモニック全てを予約語とします。

(3) レジスタ・フラグ名

RX ファミリのレジスタ名およびフラグ名全てを予約語とします。

(4) 演算子

本章で説明している全ての演算子を予約語とします。

(5) システムラベル

アセンブラが生成する、ピリオド 2 つから始まる名前をシステムラベルといい、システムラベルは全て予約語として扱います。

4.6　インストラクション

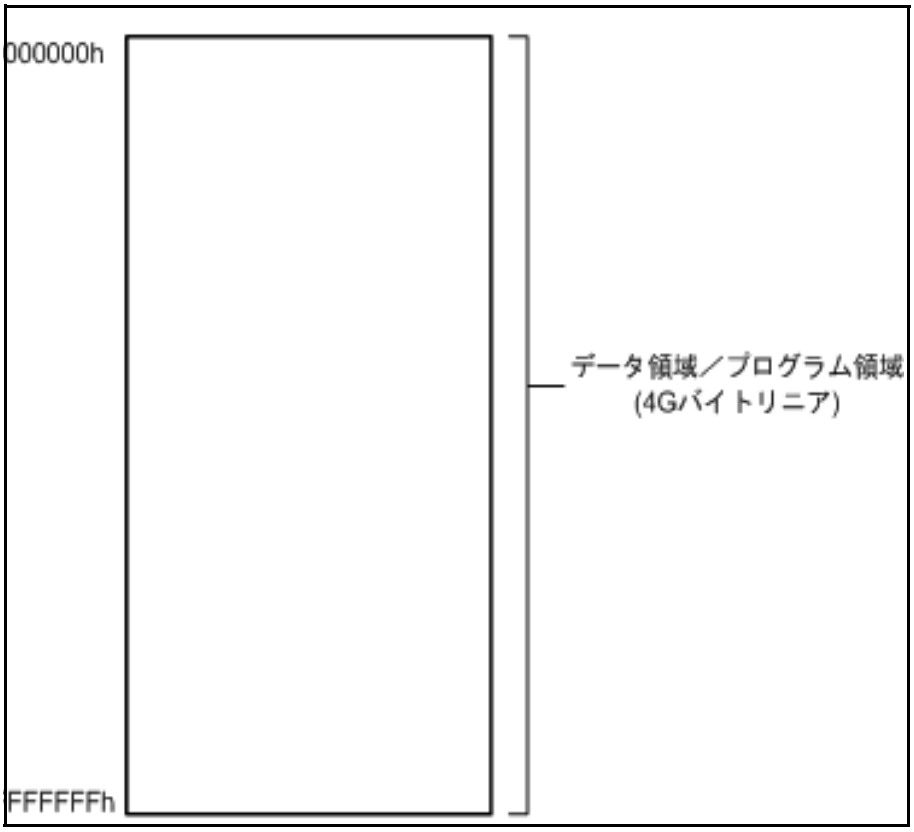
この節では，RX ファミリが持つ各種命令機能を説明します。

RX CPU は、使用頻度の高い命令に短縮フォーマットを用意していますので、少ないメモリ容量で効率の良いプログラムを開発できます。また、1 クロックで実行する命令を持ち、高速な演算処理を実現しました。73 種類の基本命令、8 種類の浮動小数点演算命令、9 種類の DSP 機能命令の合計 90 種類の命令と、10 種類のアドレッシングモードを持ち、レジスターレジスタ間、レジスターメモリ間の演算や、ビットを対象とする演算ができます。また、メモリーメモリ間の転送ができます。乗算器を内蔵していますので、高速な乗算ができます。

4.6.1　アドレス空間

0RX CPU のアドレス空間は、00000000h 番地から FFFFFFFFh 番地までの 4G バイトあります。プログラム領域およびデータ領域合計最大 4G バイトをリニアにアクセス可能です。RX CPU のアドレス空間を図 1.10 に示します。各領域は、各製品、動作モードによって異なります。詳細は、各製品のハードウェアマニュアルを参照してください。

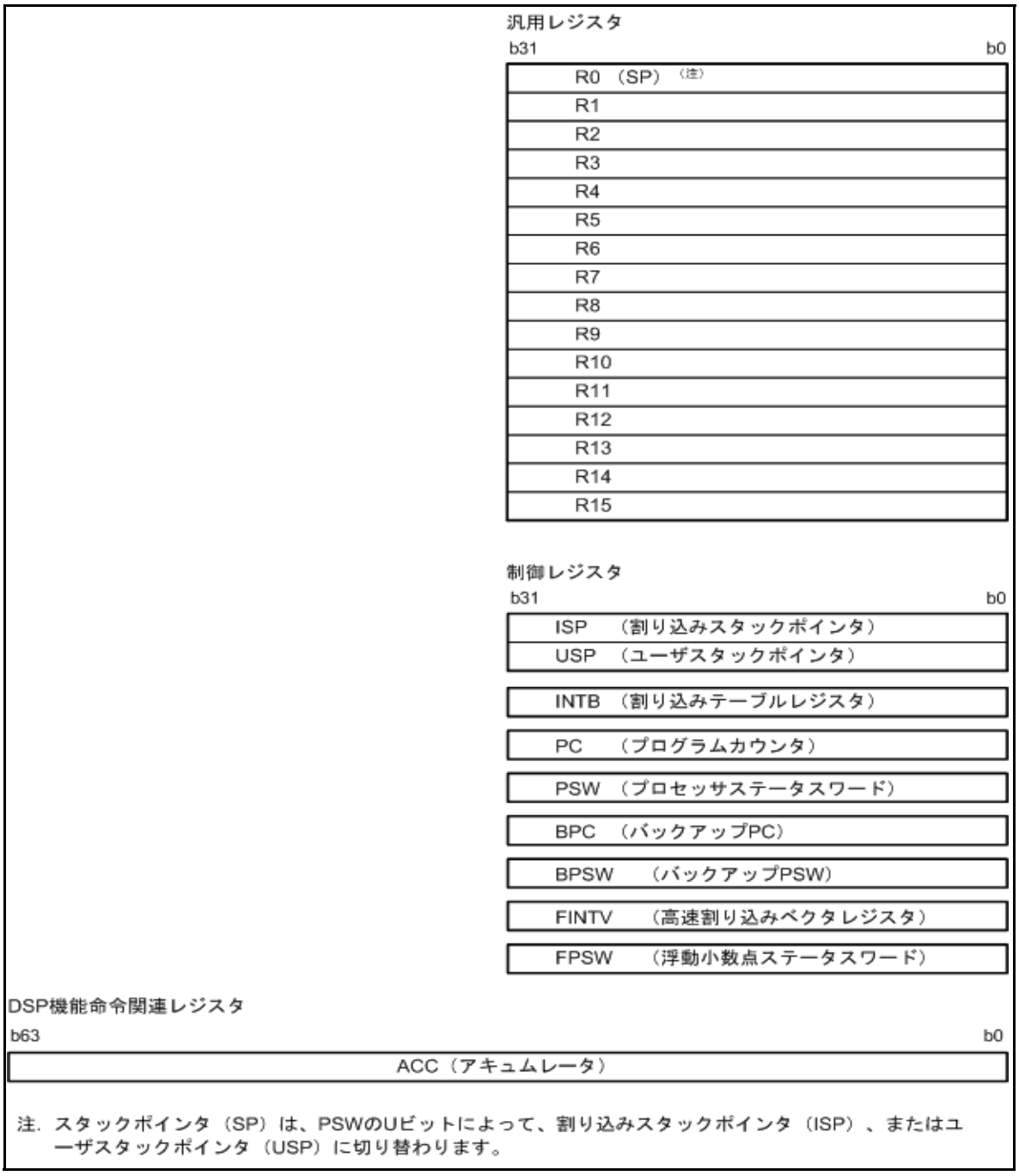
図 4―1　アドレス空間



4.6.2　レジスタ構成

RX CPU のレジスタには、汎用レジスタ（16 本）と、制御レジスタ（9 本）、および DSP 機能命令で使用するアキュムレータ（1 本）があります。

図 4―2　中央演算処理装置のレジスタ構成



(1) 汎用レジスタ（R0 ～ R15）

汎用レジスタは、16 本（R0 ～ R15）あります。汎用レジスタ R0 ～ R15 は、データレジスタやアドレスレジスタとして使用します。

汎用レジスタ R0 には、汎用レジスタとしての機能に加えて、スタックポインタ（SP）としての機能が割り当てられています。SP は、プロセッサステータスワード（PSW）のスタックポインタ指定ビット（U）によって、割り込みスタックポインタ（ISP）、またはユーザスタックポインタ（USP）に切り替わります。

(2) 割り込みスタックポインタ（ISP）／ユーザスタックポインタ（USP）

スタックポインタ（SP）には、割り込みスタックポインタ（ISP）と、ユーザスタックポインタ（USP）の2種類があります。使用するスタックポインタ（ISP/USP）は、プロセッサステータスワード（PSW）のスタックポインタ指定ビット（U）によって切り替えられます。

ISP、USPに4の倍数を設定すると、スタック操作を伴う命令や、割り込みシーケンスのサイクル数が短くなります。

(3) 割り込みテーブルレジスタ（INTB）

割り込みテーブルレジスタ（INTB）には、可変ベクタテーブルの先頭番地を設定してください。

(4) プログラムカウンタ（PC）

プログラムカウンタ（PC）は、実行中の命令の番地を示します。

(5) バックアップPC（BPC）

バックアップPC（BPC）は、割り込み応答を高速化するために設けられたレジスタです。高速割り込みが発生すると、プログラムカウンタ（PC）の内容がBPCに退避されます。

(6) バックアップPSW（BPSW）

バックアップPSW（BPSW）は、割り込み応答を高速化するために設けられたレジスタです。高速割り込みが発生すると、プロセッサステータスワード（PSW）の内容がBPSWに退避されます。BPSWのビットの割り当ては、PSWに対応しています。

(7) 高速割り込みベクタレジスタ（FINTV）

高速割り込みベクタレジスタ（FINTV）は、割り込み応答を高速化するために設けられたレジスタです。高速割り込み発生時の分岐先番地を設定してください。

(8) 浮動小数点ステータスワード（FPSW）

浮動小数点ステータスワード（FPSW）は、浮動小数点演算結果を示します。浮動小数点命令に対応していない製品では、常に“00000000h”が読め、書き込みは無視されます。

例外処理許可ビット（Ej）で例外処理を許可（Ej=“1”）した場合は、例外処理ルーチンで該当するCjフラグをチェックし、例外発生の要因を判断することができます。例外処理を禁止（Ej=“0”）した場合は、一連の処理の最後にFjフラグをチェックし、例外発生の有無を確認することができます。Fjフラグは蓄積フラグです。（j=X、U、Z、O、V）

(9) アキュムレータ（ACC）

アキュムレータ（ACC）は、64ビットのレジスタです。DSP機能命令で使用されます。また、ACCは乗算命令（EMUL、EMULU、FMUL、MUL）、積和演算命令（RMPA）でも使用され、これらの命令実行の際はACCの値が変更されます。

ACCへの書き込みには、MVTACHI命令とMVTACLO命令を使用します。MVTACHI命令は上位側32ビット（b63～b32）に、MVTACLO命令は下位側32ビット（b31～b0）にデータを書きます。

読み出しには、MVFACHI命令とMVFACMI命令を使用します。MVFACHI命令で上位側32ビット（b63～b32）、MVFACMI命令で中央の32ビット（b47～b16）のデータをそれぞれ読みます。

4. 6. 3　プロセッサステータスワード（PSW）

プロセッサステータスワード（PSW）は、命令実行の結果や、CPU の状態を示します。

4. 6. 4　浮動小数点ステータスワード（FPSW）

浮動小数点ステータスワード（FPSW）は、浮動小数点演算結果を示します。浮動小数点命令に対応していない製品では、常に“ 00000000h ”が読め、書き込みは無視されます。

例外処理許可ビット（Ej）で例外処理を許可（Ej=“ 1 ”）した場合は、例外処理ルーチンで該当する Cj フラグをチェックし、例外発生の要因を判断することができます。例外処理を禁止（Ej=“ 0 ”）した場合は、一連の処理の最後に Fj フラグをチェックし、例外発生の有無を確認することができます。Fj フラグは蓄積フラグです。（j=X、U、Z、O、V）

4. 6. 5　リセット解除後の内部状態

リセット後は、スーパバイザモードで動作します。

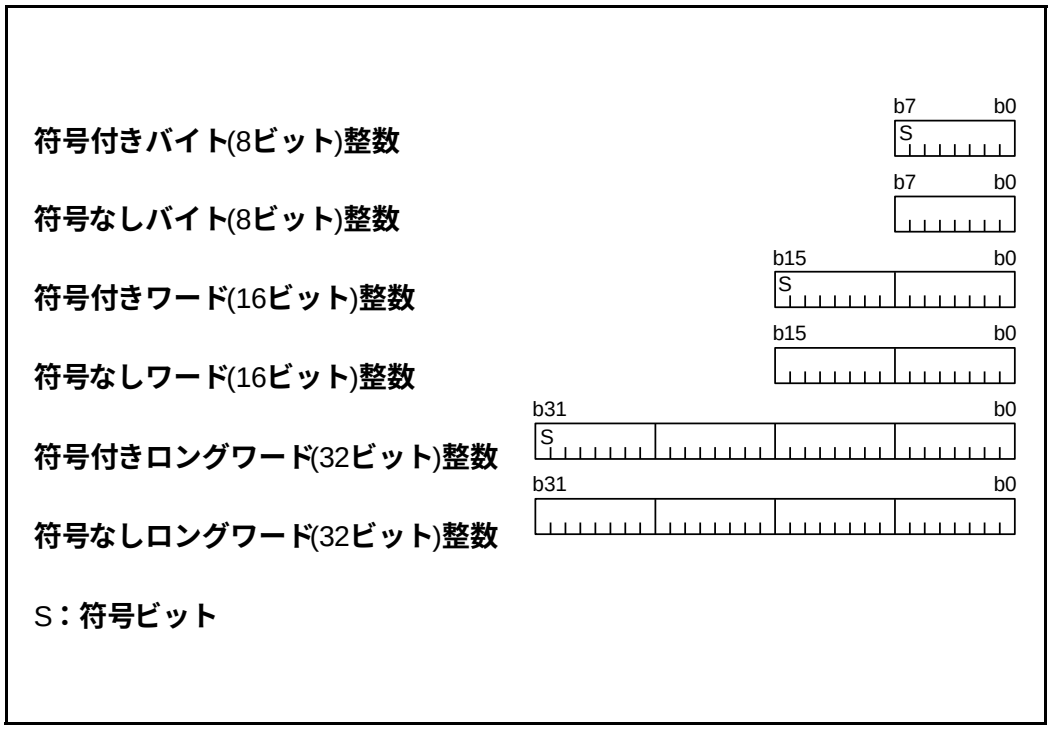
4. 6. 6　データタイプ

整数、浮動小数点数、ビット、ストリングの 4 種類のデータを扱うことができます。

(1) 整数

整数には、符号付きと、符号なしがあります。符号付き整数の負の値は、2 の補数で表現します。

図 4―3　整数データ



(2) 浮動小数点数

浮動小数点数は、IEEE754 で規定されている単精度浮動小数点数に対応しています。浮動小数点数は、浮動小数点演算命令 FADD、FCMP、FDIV、FMUL、FSUB、FTOI、ITOF、ROUND の 8 種類の命令で使用できます。

浮動小数点数は、以下の数値に対応しています。

0 < E < 255（正規化数 - Normal Numbers）

E = 0 かつ F = 0（ゼロ - Signed Zero）

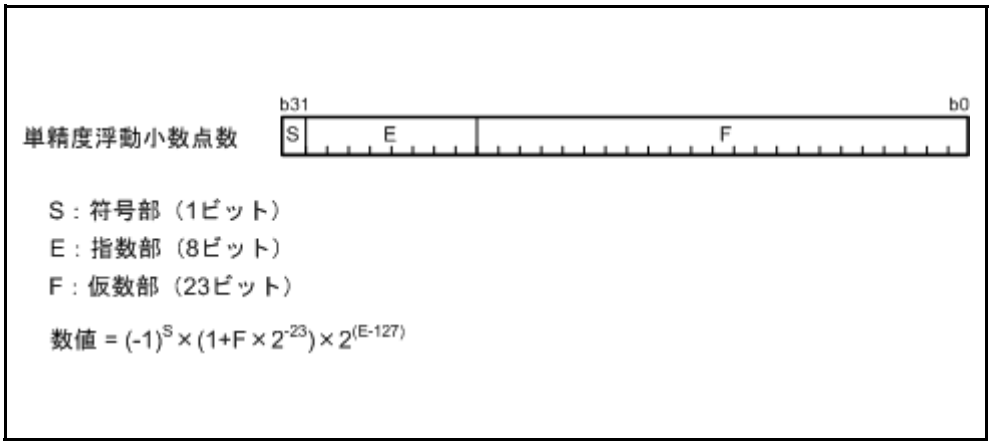
E = 0 かつ F > 0（非正規化数 - Subnormal Numbers）（注）

E = 255 かつ F = 0（無限大 - Infinity）

E = 255 かつ F > 0（非数 - NaN : Not a Number）

FPSW の DN ビットが“1”のときは、0 として扱います。DN ビットが“0”のときは、非実装処理が発生します。

図 4―4　浮動小数点数データ



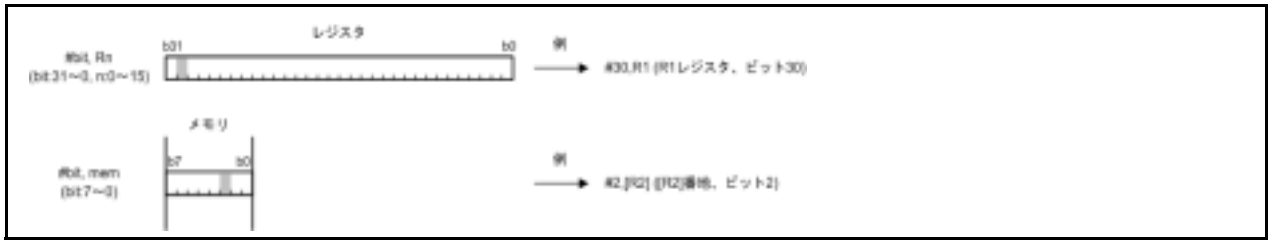
(3) ビット

ビットは、ビット操作命令 BCLR、BMCnd 、BNOT、BSET、BTST の 5 種類の命令で使用できます。

レジスタのビットは、対象とするレジスタと、31 ～ 0 のビット番号で指定します。

メモリのビットは、対象とするアドレスと、7 ～ 0 のビット番号で指定します。アドレス指定に使用できるアドレッシングモードは、レジスタ間接、レジスタ相対の 2 種類です。

図 4―5　レジスタのビット指定

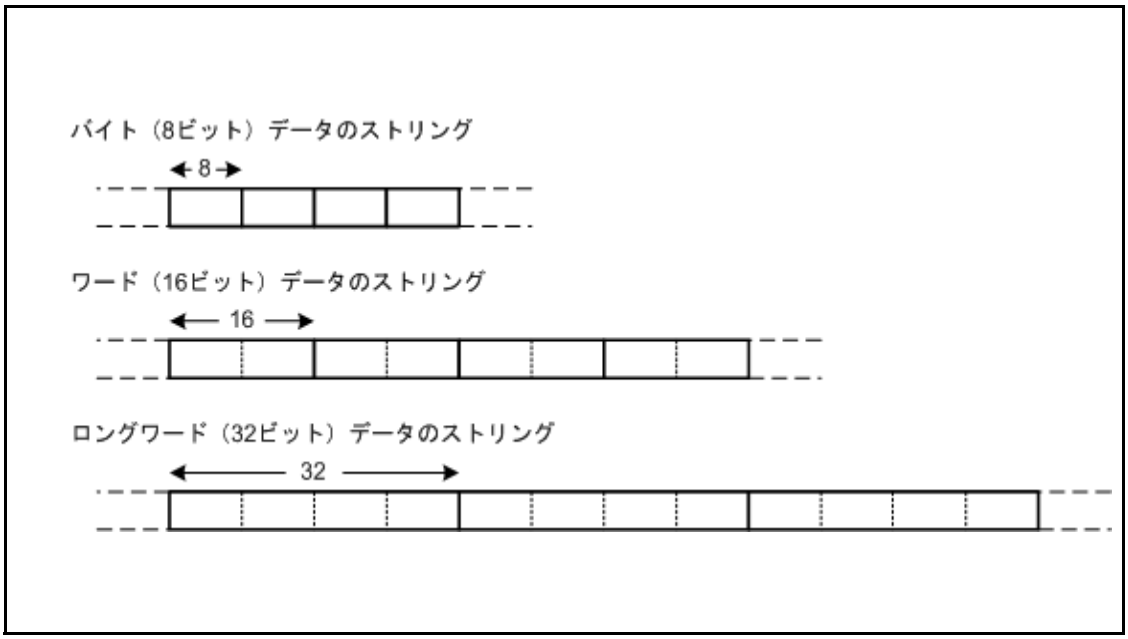


- メモリのビット

(4) ストリング

ストリングとは、バイト（8 ビット）、ワード（16 ビット）、またはロングワード（32 ビット）のデータを任意の数だけ連続して並べたデータタイプです。ストリングは、ストリング操作命令 SCMPU、SMOVB、SMOVF、SMOVU、SSTR、SUNTIL、SWHILE の 7 種類の命令で使用できます。

図 4―6 ストリングデータ

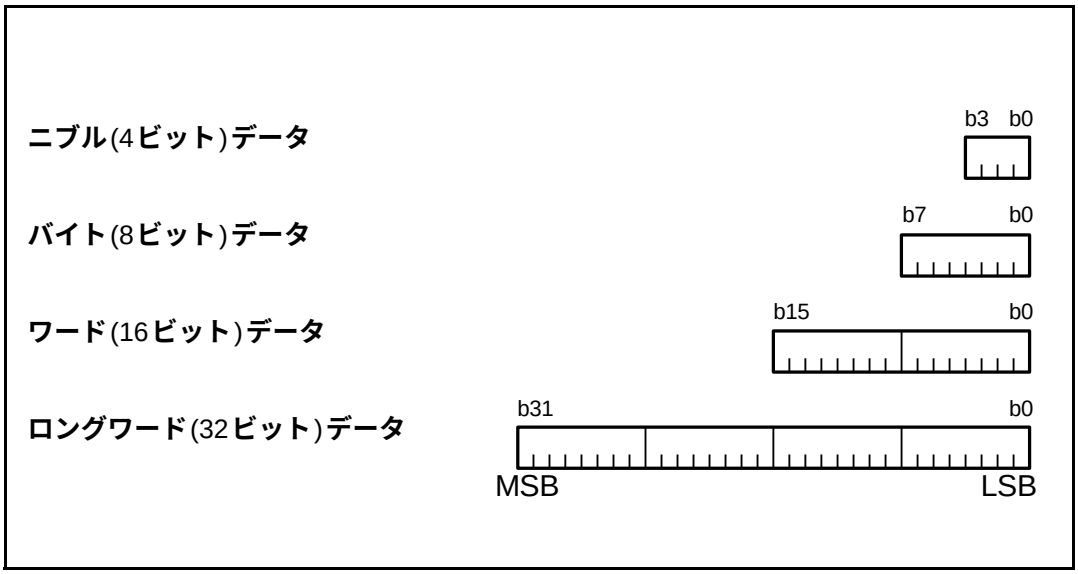


4. 6. 7 データ配置

(1) レジスタのデータ配置

レジスタのデータサイズと、ビット番号の関係を図 1.6 に示します。

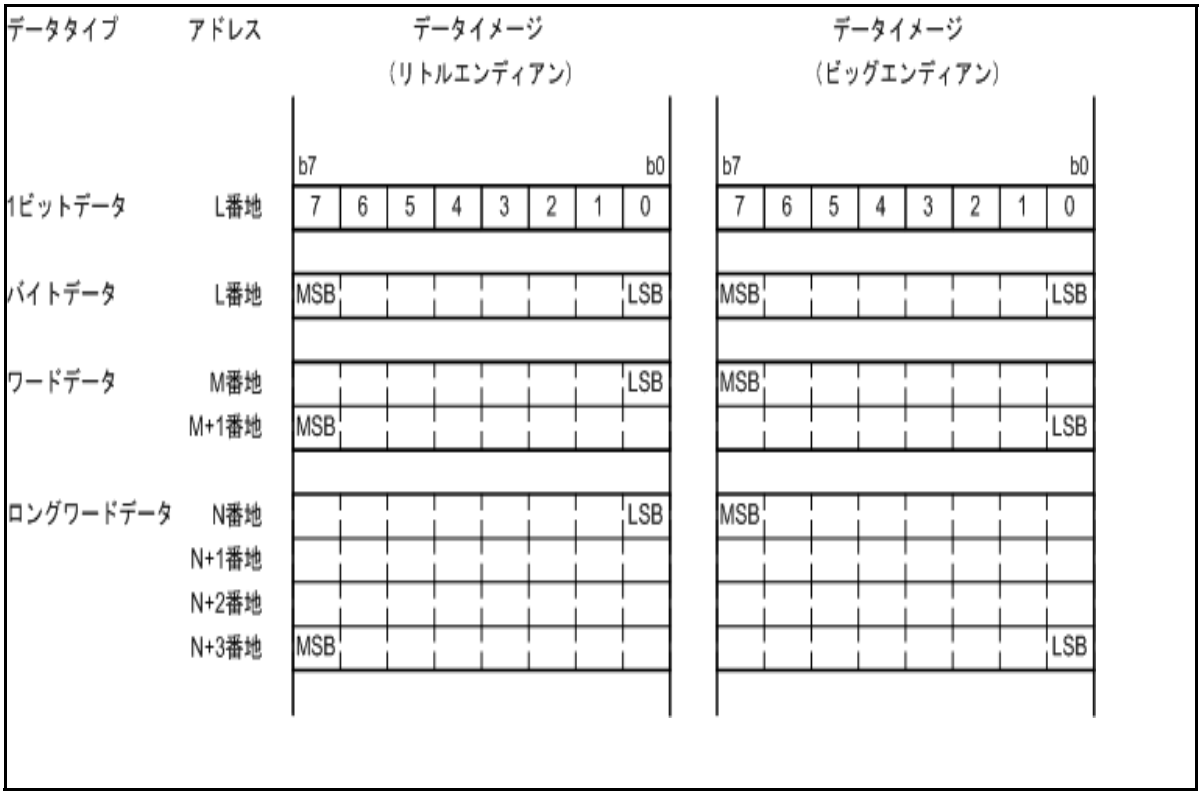
図 4―7 レジスタ上のデータ配置



(2) メモリ上のデータ配置

メモリ上のデータサイズは、バイト（8 ビット）、ワード（16 ビット）、ロングワード（32 ビット）の 3 種類です。データ配置は、リトルエンディアンか、ビッグエンディアンかを選択することができます。メモリ上のデータ配置を以下に示します。

図 4―8 メモリ上のデータ配置



4. 6. 8　ベクタテーブル

ベクタテーブルには、固定ベクタテーブルと可変ベクタテーブルがあります。ベクタテーブルは、1 ベクタあたり 4 バイトで構成されており、各ベクタには対応する例外処理ルーチンの先頭アドレスを設定します。

(1) 固定ベクタテーブル

固定ベクタテーブルは、テーブルの配置アドレスが固定されたベクタテーブルです。FFFFFF80h ～ FFFFFFFFh 番地に、特権命令例外、アクセス例外、未定義命令例外、浮動小数点例外、ノンマスカブル割り込み、リセットの各ベクタを配置しています。図 1.8 に固定ベクタテーブルを示します。

図 4―9　固定ベクタテーブル

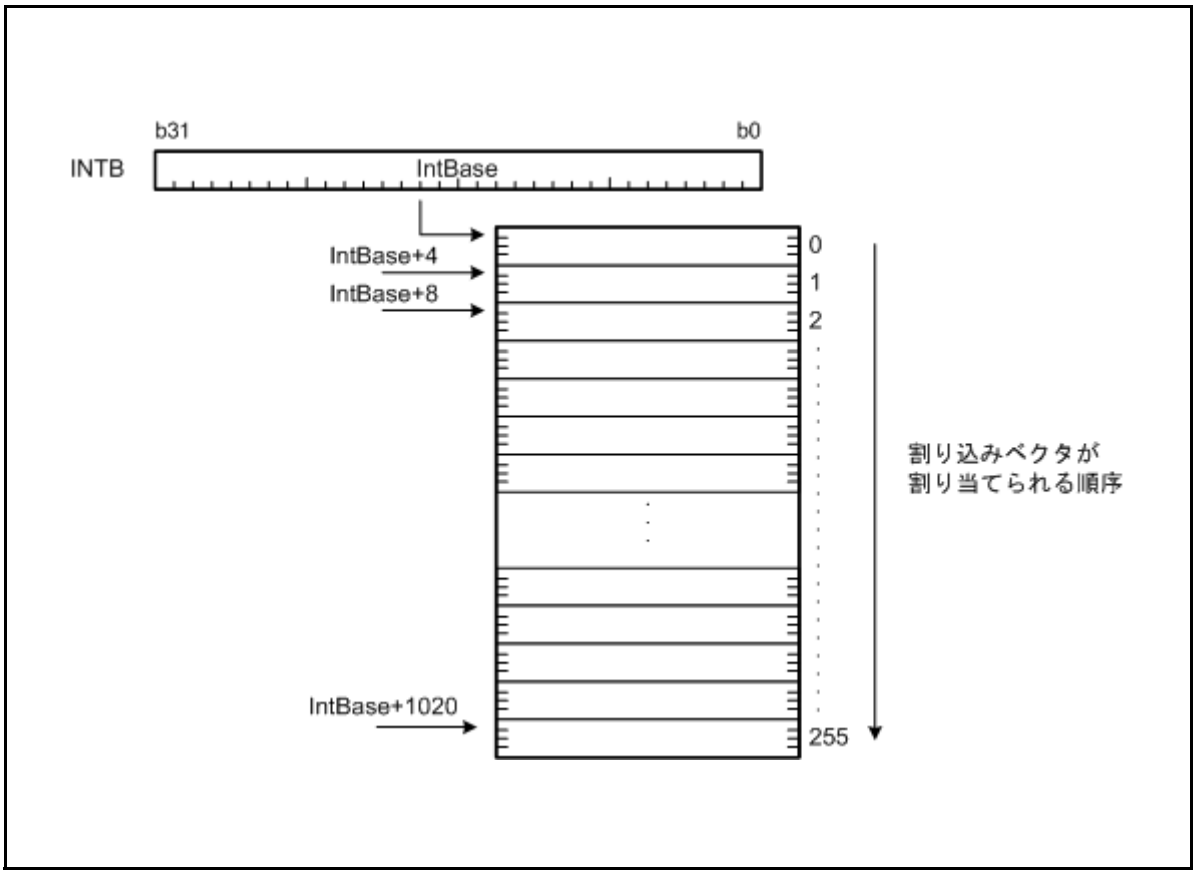
FFFFFF80h	（予約領域）
:	:
FFFFFFCCh	（予約領域）
FFFFFFD0h	特権命令例外
FFFFFFD4h	アクセス例外
FFFFFFD8h	（予約領域）
FFFFFFDCh	未定義命令例外
FFFFFFE0h	（予約領域）
FFFFFFE4h	浮動小数点例外
FFFFFFE8h	（予約領域）
FFFFFFECh	（予約領域）
FFFFFFF0h	（予約領域）
FFFFFFF4h	（予約領域）
FFFFFFF8h	ノンマスカブル割り込み
FFFFFFFCh	リセット

(2) 可変ベクタテーブル

可変ベクタテーブルは、テーブルの配置アドレスを変えることができるベクタテーブルです。割り込みテーブルレジスタ（INTB）の内容で示された値を先頭アドレス（IntBase）とする 1,024 バイトの領域に、無条件トラップ、割り込みの各ベクタを配置しています。図 1.9 に可変ベクタテーブルを示します。

可変ベクタテーブルには、ベクタごとに番号（0 ～ 255）が付けられています。無条件トラップ発生要因の INT 命令では INT 命令番号（0 ～ 255）に対応したベクタが、BRK 命令では番号 0 のベクタが割り当てられています。また、割り込み要因では、製品ごとに決められた番号（0 ～ 255）が割り当てられています。

図 4―10　可変ベクタテーブル



4. 6. 9　アドレッシングモード

本章ではアドレッシングモードを示す記号、動作についてアドレッシングモードごとに説明しています。

アドレッシングモードは、以下に示します。

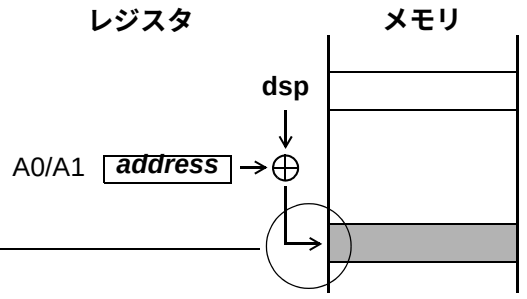
(1) アドレッシング

- 即値
- レジスタ直接
- レジスタ間接
- レジスタ相対
- ポストインクリメントレジスタ間接
- プリデクリメントレジスタ間接
- インデックス付きレジスタ間接
- 制御レジスタ直接
- PSW 直接
- プログラムカウンタ相対

4. 6. 10　本章の見方

本章の見方を以下に実例をあげて示します。

(1)

レジスタ相対		
(2) dsp:5[Rn] (Rn= R0~R7) (3) dsp:8[Rn] (Rn= R0~R15) (4) dsp:16[Rn] (Rn= R0~R15)	ディスプレースメント（dsp）の値を32ビットにゼロ拡張した後、規則に従い（右図参照）、1/2/4倍した値と、レジスタ値を加算した結果の下位32ビットが演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h～FFFFFFFFh です。 dsp:n は、nビット長のディスプレースメントを表します。 dsp:5[Rn] (Rn=R0~R7)、 dsp:8[Rn] (Rn=R0~R15)、 dsp:16[Rn] (Rn=R0~R15) が指定できます。 dsp:5[Rn] (Rn=R0~R7)は、MOV、MOVU命令でのみ使用されます。	

(1) 名称

アドレッシングの名称です。

(2) 記号

アドレッシングモードを示す記号です。

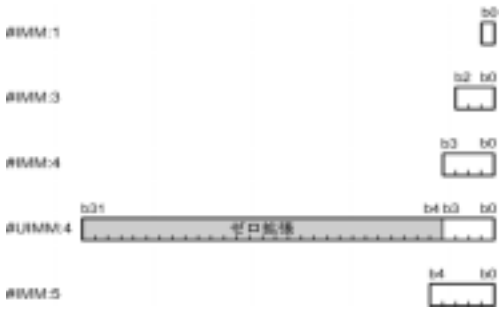
(3) 解説

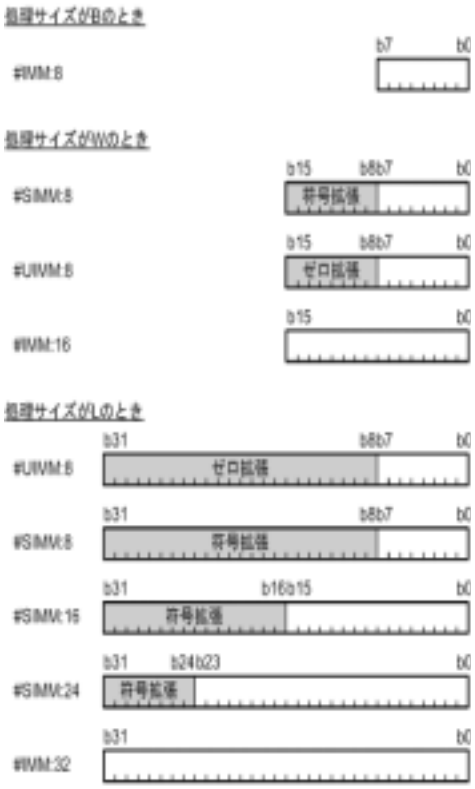
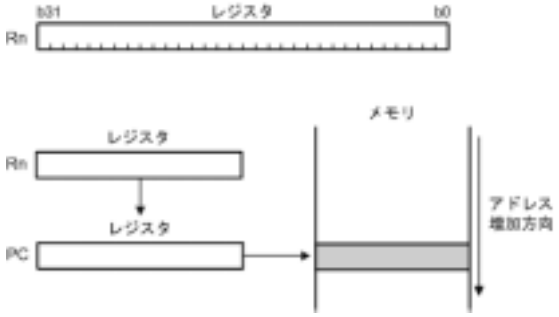
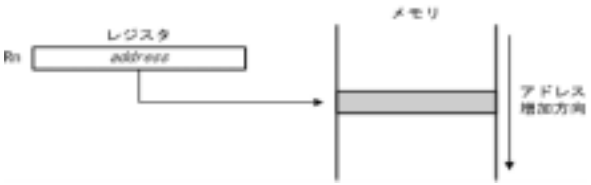
動作、実効アドレスの範囲を説明します。

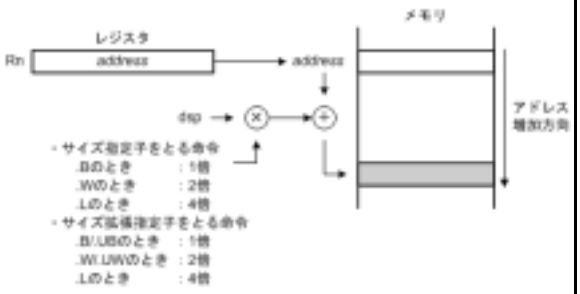
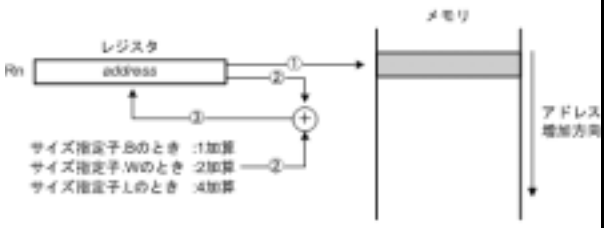
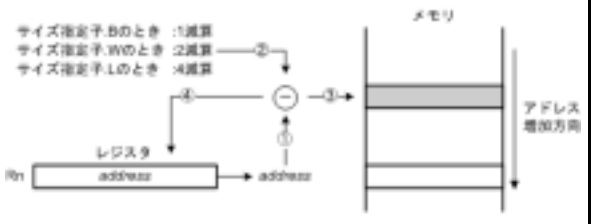
(4) 動作図

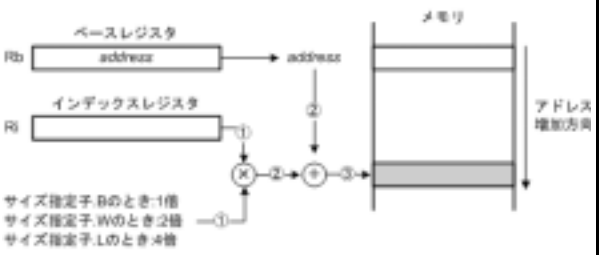
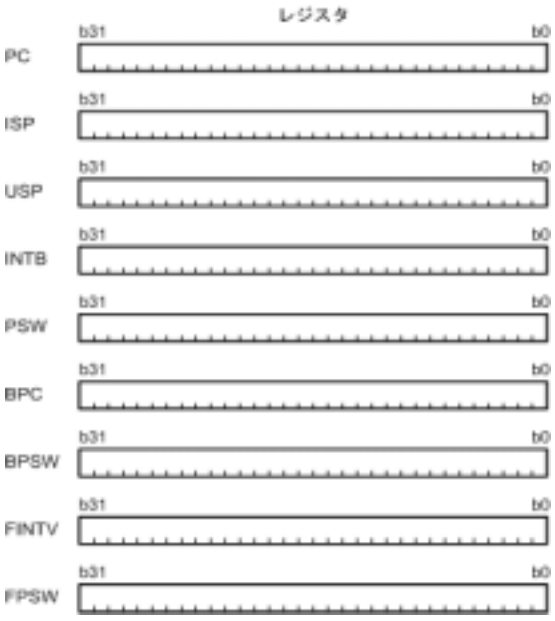
動作を図で説明します。

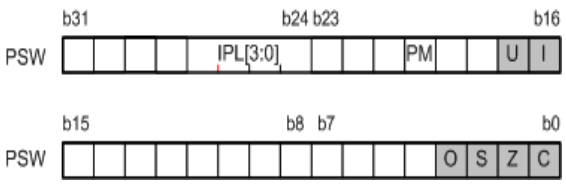
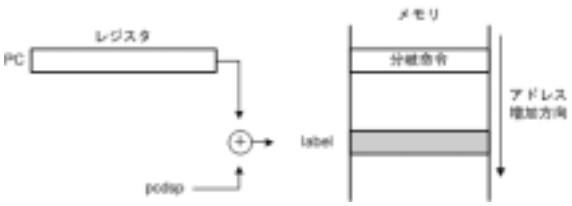
4. 6. 11　アドレッシング

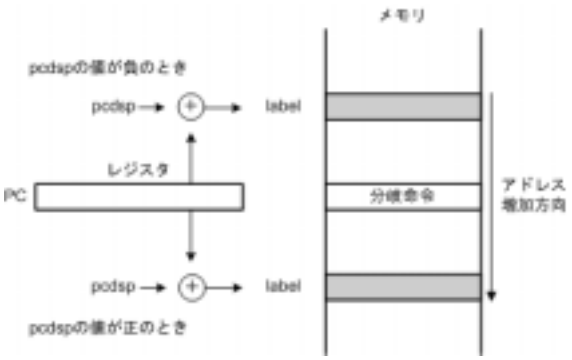
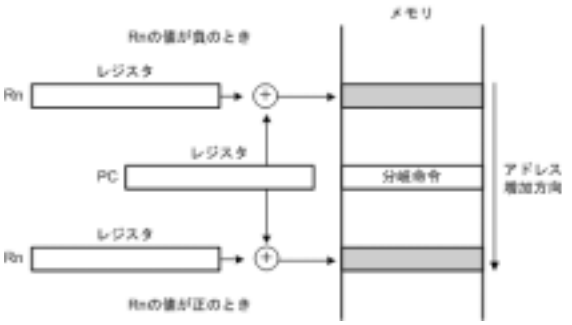
即値		
#IMM **#IMM8** **#IMM16** **#IMM20**	• #IMM:1 #IMM で示した 1 ビット即値が演算の対象となります。このアドレッシングモードは、RACW 命令のソースで使用されます。 • #IMM:3 #IMM で示した 3 ビット即値が演算の対象となります。このアドレッシングモードは、ビット操作命令（BCLR、BMCnd、BNOT、BSET、BTST）のビット番号指定で使用されます。 • #IMM:4 #IMM で示した 4 ビット即値が演算の対象となります。このアドレッシングモードは、MVTIPL 命令の割り込み優先レベル指定で使用されます。 • #UIMM:4 #UIMM で示した 4 ビット即値を 32 ビットにゼロ拡張した結果が演算の対象となります。このアドレッシングモードは、ADD、AND、CMP、MOV、MUL、OR、SUB 命令のソースで使用されます。 • #IMM:5 #IMM で示した 5 ビット即値が演算の対象となります。このアドレッシングモードは、ビット操作命令（BCLR、BMCnd、BNOT、BSET、BTST）のビット番号指定、算術／論理演算命令（SHAR、SHLL、SHLR）のシフト幅指定、および算術／論理演算命令（ROTL、ROTR）のローテート幅指定で使用されます。	

即値		
#IMM:8 **#SIMM:8** **#UIMM:8** **#IMM:16** **#SIMM:16** **#SIMM:24** **#IMM:32**	**即値で指定した値が演算の対象となります。ただし、**#UIMM**で指定した即値は処理サイズにゼロ拡張した結果が、**#SIMM**で指定した即値は処理サイズに符号拡張した結果が演算の対象となります。**#IMM:n、#UIMM:n、#SIMM:n **は、n**ビット長の即値を表します。** IMM **の範囲は、「**2.2.1 IMM **の範囲」を参照してください。**	
レジスタ直接		
Rn **(Rn=** **R0~R15)**	**指定したレジスタが演算の対象となります。** **または** JMP**、**JSR **命令の場合、**Rn **の値をプログラムカウンタ（**PC**）に転送します。** **実効アドレスの範囲は、**00000000h **～**FFFFFFFFh **です。** Rn (Rn=R0~R15)**が指定できます。**	
レジスタ間接		
[Rn] **(Rn=** **R0~R15)**	**レジスタの値が演算対象の実効アドレスとなります。実効アドレスの範囲は、**00000000h **～**FFFFFFFFh **です。** [Rn] (Rn=R0~R15)**が指定できます。**	

レジスタ相対		
dsp:5[Rn] (Rn= R0~R7) **dsp:8[Rn] (Rn= R0~R15)** **dsp:16[Rn] (Rn= R0~R15)**	**ディスプレースメント（**dsp**）の値を**32**ビットにゼロ拡張した後、規則に従い（右図参照）、**1/2/4**倍した値と、レジスタ値を加算した結果の下位**32**ビットが演算対象の実効アドレスとなります。実効アドレスの範囲は、**00000000h**～**FFFFFFFFh**です。**dsp:n**は、**n**ビット長のディスプレースメントを表します。** dsp:5[Rn] (Rn=R0~R7)、 dsp:8[Rn] (Rn=R0~R15)、 dsp:16[Rn] (Rn=R0~R15) **が指定できます。** dsp:5[Rn] (Rn=R0~R7) **は、**MOV**、**MOVU**命令でのみ使用されます。**	
ポストインクリメントレジスタ間接		
[Rn+] (Rn= R0~R15)	**レジスタの値が演算対象の実効アドレスとなります。実効アドレスの範囲は、**00000000h**～**FFFFFFFFh**です。演算実行後、レジスタの値にサイズ指定子**.B/.W/.L**に応じてそれぞれ**1/2/4**を加算します。このアドレッシングモードは、**MOV**、**MOVU**命令で使用されます。**	
プリデクリメントレジスタ間接		
[-Rn] (Rn= R0~R15)	**レジスタの値にサイズ指定子**.B/.W/.L **に応じてそれぞれ**1/2/4**を減算します。減算後の値が演算対象の実効アドレスとなります。実効アドレスの範囲は、**00000000h～FFFFFFFFh**です。このアドレッシングモードは、**MOV**、**MOVU**命令で使用されます。**	

インデックス付きレジスタ間接		
[Ri,Rb] (Ri,Rb= R0~R15)	インデックスレジスタ（Ri）の値をサイズ指定子.B/.W/.L に応じてそれぞれ 1/2/4 倍した値と、ベースレジスタ（Rb）の値を加算した結果の下位 32 ビットが演算対象の実効アドレスとなります。実効アドレスの範囲は、00000000h～FFFFFFFFh です。このアドレッシングモードは、MOV、MOVU 命令で使用されます。	
制御レジスタ直接		
PC ISP USP INTB PSW BPC BPSW FINTV FPSW	指定した制御レジスタが演算の対象となります。このアドレッシングモードは、MVFC、MVTC、POPC、PUSHC 命令で使用されます。 PC は MVFC、PUSHC 命令の src にのみ指定できます。	

PSW直接		
C Z S O I U	指定したフラグ、またはビットが演算の対象となります。このアドレッシングモードは、CLRPSW、SETPSW命令で使用されます。	
プログラムカウンタ相対		
pcdsp:3	**分岐距離指定子が“.S”のとき、プログラムカウンタ（PC）にディスプレースメント（pcdsp）の値を符号なしで加算した結果の下位32ビットが実効アドレスとなります。分岐の範囲は、3～10です。実効アドレスの範囲は、00000000h～FFFFFFFFh です。このアドレッシングモードは、BCnd（Cnd==EQ/Z、NE/NZ のみ）、BRA 命令で使用されます。**	

プログラムカウンタ相対		
pcdsp:8 **pcdsp:16** **pcdsp:24**	**分岐距離指定子が“** .B **”、“** .W **”、または“** .A **”のとき、プログラムカウンタ（**PC**）の値と、ディスプレースメント（**pcdsp**）の値を符号付きで加算した結果が実効アドレスとなります。** **pcdsp の範囲は、** **“** .B **”のとき：**-128≦pcdsp:8≦127 **“** .W **”のとき：**-32768≦pcdsp:16≦32767 **“** .A **”のとき：**-8388608≦pcdsp:24≦8388607 **となります。実効アドレスの範囲は、**00000000h～FFFFFFFFh **です。このアドレッシングモードは、“** .B **”のとき** BCnd、BRA **命令で、“** .W **”のとき** BCnd（Cnd==EQ/Z、NE/NZ **のみ）、**BRA、BSR **命令で、“** .A **”のとき** BRA、BSR **命令で使用されます。**	
プログラムカウンタ相対		
Rn **(Rn=** **R0~R15)**	**プログラムカウンタ（**PC**）の値と、**Rn **の値を符号付きで加算した結果が実効アドレスとなります。**Rn **の値の範囲は、**-2147483648 ～ 2147483647 **です。実効アドレスの範囲は、**00000000h ～ FFFFFFFFh **です。このアドレッシングモードは、**BRA(.L)、BSR(.L) **命令で使用されます。**	

4. 6. 12 命令概要

本項では、構文、オペレーション、機能、選択可能な src/dest、フラグ変化、記述例、関連命令について命令ごとに説明しています。

本項の見方について以下に実例をあげて示します。

(1) 構文

命令の構文を記号で示しています。(:format) を省略した場合、アセンブラが最適な指定子を選択します。

mov.size src.dest

B,W,L → (d)

↓ ↓ ↓

(a) (b) (c)

(a) ニーモニック MOV

ニーモニックを記述します。

(b) サイズ指定子 .size

取り扱うデータサイズを記述します。指定できるサイズを以下に示します。

.Bバイト (8 ビット)

.Wワード (16 ビット)

.Lロングワード (32 ビット)

サイズ指定子をもたない命令もあります。

(c) オペランド src, dest

オペランドを記述します。

(d) (c) で指定できる命令フォーマットを示しています。

(2) フラグ変化

命令実行後のフラグの変化を示します。表中に示す記号の意味は次のとおりです。

“ – ” 変化しません。

“ ○ ” 条件に従って変化します。

(3) 記述例

命令の記述例を示しています。

4. 6. 13 機能

この項では、命令セットについて説明します。

ABS 絶対値 ABS
ABSolute

【構文】

(1)ABS dest

(2)ABS src, dest

【オペレーション】

```
(1)if ( dest < 0 )
  dest = -dest;
(2)if ( src < 0 )
  dest = -src;
else
  dest = src;
```

【機能】

(1)dest **の絶対値をとり、その結果を** dest **に格納します。**

(2)src **の絶対値をとり、その結果を** dest **に格納します。**

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ABS dest	L	-	Rd	2
ABS src, dest	L	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	－	○	○	○

条件

O：－

Z：**演算後の** dest **が** 0 **のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

S：**演算後の** dest **の** MSB **が“** 1 **”のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

C：(1) **演算前の** dest **が** 80000000h **のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

(2) **演算前の** src **が** 80000000h **のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

【記述例】

```
ABS     R2
ABS     R1, R2
```

ADC キャリ付き加算 ADC
ADd with Carry

【構文】

ADC src, dest

【オペレーション】

dest = dest + src + C;

【機能】

- dest と src と C フラグを加算し、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ADC src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	4
	L	dsp:8[Rs].L	Rd	5
	L	dsp:16[Rs].L	Rd	6

注意 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：符号なし演算のオーバフローが発生したとき“1”、それ以外のとき“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

O：符号付き演算のオーバフローが発生したとき“1”、それ以外のとき“0”になります。

【記述例】

```
ADC     #127, R2
ADC     R1, R2
ADC     [R1], R2
```

ADD キャリなしの加算 ADD
ADD

【構文】

(1)ADD src, dest

(2)ADD src, src2, dest

【オペレーション】

(1)dest = dest + src;

(2)dest = src + src2;

【機能】

(1)dest と src を加算し、その結果を dest に格納します。

(2)src と src2 を加算し、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)ADD src,dst	L	#UIMM:4	-	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex	-	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex	-	Rd	4 (memex == UB) 5 (memex != UB)
(2)ADD src,src2,dst	L	#SIMM:8	Rs	Rd	3
	L	#SIMM:16	Rs	Rd	4
	L	#SIMM:24	Rs	Rd	5
	L	#IMM:32	Rs	Rd	6
	L	Rs	Rs	Rd	3

注意　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“.W”または“.UW”のときは 2 の倍数、“.L”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“.W”または“.UW”のとき 0～510（255 × 2）が、“.L”のとき 0～1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“.W”または“.UW”のとき 0～131070

（65535 × 2）が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：符号なし演算のオーバフローが発生したとき“1”、それ以外のとき“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

O：符号付き演算のオーバフローが発生したとき“1”、それ以外のとき“0”になります。

【記述例】

```
ADD     #15, R2
ADD     R1, R2
ADD     [R1], R2
ADD     [R1].UB, R2
ADD     #127, R1, R2
ADD     R1, R2, R3
```

AND 論理積 AND

【構文】

(1)AND src, dest

(2)AND src, src2, dest

【オペレーション】

(1)dest = dest & src;

(2)dest = src & src2;

【機能】

- (1)dest と src の論理積をとり、その結果を dest に格納します。
- (2)src と src2 の論理積をとり、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)AND src,dst	L	#UIMM:4	-	Rd	2
	L	#SIMM:8	-	Rd	3
	L	#SIMM:16	-	Rd	4
	L	#SIMM:24	-	Rd	5
	L	#IMM:32	-	Rd	6
	L	Rs	-	Rd	2
	L	[Rs].memex	-	Rd	2(memex=UB) 3(memex!=UB)
	L	dsp:8[Rs].memex	-	Rd	3(memex==UB) 4(memex==UB)
	L	dsp:16[Rs].memex	-	Rd	4(memex=UB) 5(memex!=UB)

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(2)AND src,src2,dst	L	Rs	Rs2	Rd	3

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	－	○	○	－

条件

C：-

Z：**演算後の** dest **が** 0 **のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

S：**演算後の** dest **の** MSB **が“** 1 **”のとき“** 1 **”、それ以外のとき“** 0 **”になります。**

O：-

【記述例】

```
AND     #15, R2
AND     R1, R2
AND     [R1], R2
AND     [R1].UW, R2
AND     R1, R2, R3
```

BCLR　ビットクリア Bit CLeaR　BCLR

【構文】

BCLR　src, dest

【オペレーション】

(1)dest **がメモリの場合**

```
unsigned char dest;
dest &= ~( 1 << ( src & 7 ));
```

(2)dest **がレジスタの場合**

```
register unsigned long dest;
dest &= ~( 1 << ( src & 31 ));
```

【機能】

srcで指定されたdestのビットを“0”にします。
srcのIMMの値はビット番号です。
IMM:3の範囲は、0≦IMM:3≦7です。
IMM:5の範囲は、0≦IMM:5≦31です。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)BCLR　src, dest	B	#IMM:3	-	[Rd].B	2
	B	#IMM:3	-	dsp:8[Rd].B	3
	B	#IMM:3	-	dsp:16[Rd].B	4
	B	Rs	-	[Rd].B	3
	B	Rs	-	dsp:8[Rd].B	4
	B	Rs	-	dsp:16[Rd].B	5

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(2)BCLR　src, dest	L	#IMM:5	-	Rd	2
	L	Rs	-	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BCLR    #7, [R2]
BCLR    R1, [R2]
BCLR    #31, R2
BCLR    R1, R2
```

BCnd 相対条件分岐 BCnd
Branch Conditionally

【構文】

BCnd(.length) src

【オペレーション】

```
if ( Cnd )
  PC = PC + src;
```

【機能】

- Cnd で示す条件の真偽値を判断し、src で示される分岐先へ相対分岐します。真の場合は分岐しますが、偽の場合は分岐しません。
- BCnd には以下の種類があります。

BCnd	条件		式
BGEU, BC	C == 1	等しいまたは大きい／ C フラグが“1”	≦
BEQ, BZ	Z == 1	等しい／ Z フラグが“1”	=
BGTU	C & ~Z == 1	大きい	<
BPZ	S == 0	正またゼロ	0 ≦
BGE	S ^ O == 0	等しい、または符号付きで大きい	≦
BGT	(S ^ O) | Z == 0	符号付きで大きい	<
BO	O == 1	O フラグが“1”	
BLTU, BNC	C == 0	小さい／ C フラグが“0”	>
BNE, BNZ	Z == 0	等しくない／ Z フラグが“0”	≠
BLEU	C & ~Z == 0	等しいまたは小さい	≧
BN	S == 1	負	0 >
BLE	(S ^ O) | Z == 1	等しい、または符号付きで小さい	≧
BLT	S ^ O == 1	符号付きで小さい	>
BNO	O == 0	O フラグが“0”	

【命令フォーマット】

構文	処理サイズ	src	pcdsp の範囲	コードサイズ（バイト）
BEQ.S src	S	pcdsp:3	3 ≦ pcdsp ≦ 10	1
BNE.S src	S	pcdsp:3	3 ≦ pcdsp ≦ 10	1

構文	処理サイズ	src	pcdsp の範囲	コードサイズ（バイト）
BCnd.B　src	B	pcdsp:8	-128 ≦ pcdsp ≦ 127	2
BEQ.W　src	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3
BNE.W　src	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BC      label1
BC.B     label2
```

弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（pcdsp:3、pcdsp:8、pcdsp:16）は、分岐先のラベルまたは実効アドレスを指定してください。命令コード（pcdsp）には、指定されたアドレスから命令の配置アドレスを引いた値が埋め込まれます。

```
BC      label
BC      1000h
```

BMCnd 条件ビット転送 BMCnd
Bit Move Conditional

【構文】

BMCnd src, dest

【オペレーション】

(1)dest **がメモリの場合**

```
unsigned char dest;
if ( Cnd )
   dest |= ( 1 << ( src & 7 ));
else
   dest &= ~( 1 << ( src & 7 ));
```

(2)dest **がレジスタの場合**

```
register unsigned long dest;
if ( Cnd )
   dest |= ( 1 << ( src & 31 ));
else
   dest &= ~( 1 << ( src & 31 ));
```

【機能】

- Cnd **で示す条件の真偽値を** src **で指定された** dest **のビットに転送します。真の場合は“** 1 **”、偽の場合は“** 0 **”が転送されます。**
- BMCnd **には以下の種類があります。**

BCnd	条件		式
BMGEU, BMC	C == 1	**等しいまたは大きい／** C **フラグが“** 1 **”**	≦
BMEQ, BMZ	Z == 1	**等しい／** Z **フラグが“** 1 **”**	=
BMGTU	C & ~Z == 1	**大きい**	<
BMPZ	S == 0	**正またゼロ**	0 ≦
BMGE	S ^ O == 0	**等しい、または符号付きで大きい**	≦
BMGT	(S ^ O) | Z == 0	**符号付きで大きい**	<
BMO	O == 1	O **フラグが“** 1 **”**	
BMLTU, BMNC	C == 0	**小さい／** C **フラグが“** 0 **”**	>

BCnd	条件		式
BMNE, BMNZ	Z == 0	等しくない／ Z フラグが“0”	≠
BMLEU	C & ~Z == 0	等しいまたは小さい	≦
BMN	S == 1	負	0 >
BMLE	(S ^ O) | Z == 1	等しい、または符号付きで小さい	≦
BMLT	S ^ O == 1	符号付きで小さい	>
BMNO	O == 0	O フラグが“0”	

src の IMM の値はビット番号です。

IMM:3 の範囲は、0 ≦ IMM:3 ≦ 7 です。

IMM:5 の範囲は、0 ≦ IMM:5 ≦ 31 です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
BMCnd　src, dest	B	#IMM:3	[Rd].B	3
	B	#IMM:3	dsp:8[Rd].B	4
	B	#IMM:3	dsp16[Rd].B	5

構文	処理サイズ	src	dest	コードサイズ（バイト）
BMCnd　src, dest	L	#IMM:5	Rd	3

【フラグ変化】

フラグ変換はありません。

【記述例】

```
BMC     #7, [R2]
BMZ     #31, R2
```

BNOT ビット反転 BNOT
Bit NOT

【構文】

BNOT src, dest

【オペレーション】

(1)dest **がメモリの場合**

```
unsigned char dest;
dest ^= ( 1 << ( src & 7 ));
```

(2)dest **がレジスタの場合**

```
register unsigned long dest;
dest ^= ( 1 << ( src & 31 ));
```

【機能】

- src **で指定された** dest **のビットの値を反転し、その結果を元のビットに格納します。**
- src **の** IMM **の値はビット番号です。**
- IMM:3 **の範囲は、**0 ≦ IMM:3 ≦ 7 **です。**
- IMM:5 **の範囲は、**0 ≦ IMM:5 ≦ 31 **です。**

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
(1)BNOT src, dest	B	#IMM:3	[Rd].B	3
	B	#IMM:3	dsp:8[Rd].B	4
	B	#IMM:3	dsp:16[Rd].B	5
	B	Rs	[Rd].B	3
	B	Rs	dsp:8[Rd].B	4
	B	Rs	dsp:16[Rd].B	5

構文	処理サイズ	src	dest	コードサイズ（バイト）
(2)BNOT src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3
	B	Rs	dsp:16[Rd].B	5

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BNOT    #7, [R2]
BNOT    R1, [R2]
BNOT    #31, R2
BNOT    R1, R2
```

BRA 相対無条件分岐 BRA
BRanch Always

【構文】

BRA(.length) src

【オペレーション】

PC = PC + src;

【機能】

- src で示される分岐先に相対分岐します。

【命令フォーマット】

構文	処理サイズ	src	pcdsp/Rs の範囲	コードサイズ（バイト）
BRA(.length) src	S	pcdsp:3	3 ≦ pcdsp ≦ 10	1
	B	pcdsp:8	-128 ≦ pcdsp ≦ 127	2
	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3
	A	pcdsp:24	-8388608 ≦ pcdsp ≦ 8388607	4
	L	Rs	-2147483648 ≦ Rs ≦ 2147483647	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BRA     label1
BRA.A    label2
BRA     R1
BRA.L    R2
```

弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（pcdsp:3、pcdsp:8、pcdsp:16、pcdsp:24）は、分岐先のラベルまたは実効アドレスを指定してください。命令コード（pcdsp）には、指定されたアドレスから命令の配置アドレスを引いた値が埋め込まれます。

```
BRA    label
BRA    1000h
```

BRK　無条件トラップ　BRK
BReak

【構文】

BRK

【オペレーション】

```
tmp0 = PSW;
U = 0;
I = 0;
PM = 0;
tmp1 = PC + 1;
PC = *IntBase;
SP = SP - 4;
*SP = tmp0;
SP = SP - 4;
*SP = tmp1;
```

【機能】

- 番号 0 の無条件トラップが発生します。
- スーパバイザモードに移行し、PSW の PM ビットが “ 0 ” になります。
- PSW の U、I ビットが “ 0 ” になります。
- 実行した BRK 命令の次の命令のアドレスがスタックに退避されます。

【命令フォーマット】

構文	コードサイズ（バイト）
BRK	1

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BRK
```

BSET　Tビットセット　BSET
Bit SEt

【構文】

BSET src,dest

【オペレーション】

(1)destがメモリの場合
unsigned char dest;
dest |= (1 << (src & 7));

(2)destがレジスタの場合
register unsigned long dest;
dest |= (1 << (src & 31));

【機能】

- src で指定された dest のビットを“1”にします。
- src の IMM の値はビット番号です。
- IMM:3 の範囲は、0 ≦ IMM:3 ≦ 7 です。
- IMM:5 の範囲は、0 ≦ IMM:5 ≦ 31 です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
(1)BSET src,dest	B	#IMM:3	[Rd].B	2
	B	#IMM:3	dsp:8[Rd].B	3
	B	#IMM:3	dsp:16[Rd].B	4
	B	Rs	[Rd].B	3
	B	Rs	dsp:8[Rd].B	4
	B	Rs	dsp:16[Rd].B	5

構文	処理サイズ	src	dest	コードサイズ（バイト）
(2)BSET src,dest	L	#IMM:5	Rd	2
	L	Rs	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BSET    #7, [R2]
BSET    R1, [R2]
BSET    #31, R2
BSET    R1, R2
```

BSR 相対サブルーチン BSR
Branch to SubRoutine

【構文】

BSR(.length) src

【オペレーション】

SP = SP - 4;

*SP = (PC + n); **（注）**

PC = PC + src;

1. **(PC + n) は、BSR 命令の次の命令の番地です。**
2. **n は、コードサイズです。コードサイズについては、【命令フォーマット】を参照してください。**

【機能】

- src **で示される分岐先に相対分岐します。**

【命令フォーマット】

構文	**処理サイズ**	src	pcdsp / Rs **の範囲**	**コードサイズ（バイト）**
BSR(.length) src	W	pcdsp:16	-32768 ≦ pcdsp ≦ 32767	3
	A	pcdsp:24	-8388608 ≦ pcdsp ≦ 8388607	4
	L	Rs	-2147483648 ≦ Rs ≦ 2147483647	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
BSR     label1
BSR.A    label2
BSR     R1
BSR.L    R2
```

弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（pcdsp:16、pcdsp:24）は、分岐先のラベルまたは実効アドレスを指定してください。命令コード（pcdsp）には、指定されたアドレスから命令の配置アドレスを引いた値が埋め込まれます。

```
BSR   label
BSR   1000h
```

BTST T ビットテスト Bit TeST BTST

【構文】

BTST src, src3

【オペレーション】

(1)src2 **がメモリの場合**

unsigned char src2;

Z = ~((src2 >> (src & 7)) & 1);

C = ((src2 >> (src & 7)) & 1);

(2)src2 **がレジスタの場合**

register unsigned long src2;

Z = ~((src2 >> (src & 31)) & 1);

C = ((src2 >> (src & 31)) & 1);

【機能】

- rc **で指定した** src2 **のビットの値を反転した結果を** Z **フラグに、**src **で指定した** src2 **のビットの値を** C **フラグに転送します。**
- src **の** IMM **の値はビット番号です。**
- IMM:3 **の範囲は、**0 ≦ IMM:3 ≦ 7 **です。**
- IMM:5 **の範囲は、**0 ≦ IMM:5 ≦ 31 **です。**

【命令フォーマット】

構文	処理サイズ	src	src2	コードサイズ（バイト）
(1)BTST src, src2	B	#IMM:3	[Rs2].B	2
	B	#IMM:3	dsp:8[Rs2].B	3
	B	#IMM:3	dsp:16[Rs2].B	4
	B	Rs	[Rs2].B	3
	B	Rs	dsp:8[Rs2].B	4
	B	Rs	dsp16:[Rs2].B	5

構文	処理サイズ	src	src2	コードサイズ（バイト）
(2)BTST src, src2	L	#IMM:5	Rs2	2
	L	Rs	Rs2	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	－	－

条件

C：指定ビットが“1”のとき“1”、それ以外のとき“0”になります。

Z：指定ビットが“0”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
BTST    #7, [R2]
BTST    R1, [R2]
BTST    #31, R2
BTST    R1, R2
```

CLRPSW PSW のフラグ、ビットのクリア CLRPSW
CLeaR flag in PSW

【構文】

CLRPSW dest

【オペレーション】

dest = 0;

【機能】

- dest で指定された O、S、Z、C フラグ、もしくは U、I ビットを “0” にします。
- ユーザモードでは、U、I ビットへの書き込みは無視されます。スーパバイザモードでは、すべてのフラグとビットへの書き込みが行えます。

【命令フォーマット】

構文	dest	コードサイズ（バイト）
CLRPSW dest	flag	2

【フラグ変化】

フラグ	C	Z	S	O
変化	-(注)	-(注)	-(注)	-(注)

（注）

指定されたフラグが “0” になります。

【記述例】

```
CLRPSW C
CLRPSW Z
```

CMP 比較 CMP
CoMPare

【構文】

CMP src, src2

【オペレーション】

src2 - src;

【機能】

- src2 から src を減算した結果にしたがって、PSW の各フラグが変化します。

【命令フォーマット】

構文	処理サイズ	src	src2	コードサイズ（バイト）
CMP src, src2	L	#UIMM:4	Rs	2
	L	#UIMM:8(注 1)	Rs	3
	L	#SIMM:8(注 1)	Rs	3
	L	#SIMM:16	Rs	4
	L	#SIMM:24	Rs	5
	L	#IMM:32	Rs	6
	L	Rs	Rs2	2
	L	[Rs].memex	Rs2	2(memex == UB) 3(memex != UB)
	L	dsp:8[Rs].memex(注 2)	Rs2	3(memex == UB) 4(memex != UB)
	L	dsp:16[Rs].memex(注 2)	Rs2	4(memex == UB) 5(memex == UB)

(注 1)0 ～ 127 の範囲は、常にゼロ拡張命令コードになります。

(注 2) 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：符号なし演算のオーバフローが発生しなかったとき“1”、それ以外のとき“0”になります。

Z：演算結果が0のとき“1”、それ以外のとき“0”になります。

S：演算結果のMSBが“1”のとき“1”、それ以外のとき“0”になります。

O：符号付き演算のオーバフローが発生したとき“1”、それ以外のとき“0”になります。

【記述例】

```
CMP     #7, R2
CMP     R1, R2
CMP     [R1], R2
```

DIV 符号付き除算 DIV
DIVide

【構文】

DIV src, dest

【オペレーション】

dest = dest / src;

【機能】

- dest を src で符号付き除算し、その商を dest に格納します。商は 0 方向に丸められます。
- 演算は 32 ビットで行い、結果は 32 ビットで格納します。
- 除数（src）が 0 のとき、または演算の結果、オーバフローが発生したときの dest の値は不定です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
DIV src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5(memex == UB) 6(memex != UB)

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	−	−	−	○

条件

O：除数（src）が 0 のとき、または演算が -2147483648 ÷ (-1) のとき “ 1 ”、それ以外のとき “ 0 ” になります。

【記述例】

```
    DIV     #10, R2
DIV     R1, R2
DIV     [R1], R2
DIV     3[R1].B, R2
```

DIVU 符号なし除算 DIVU
DIVide Unsigned

【構文】

DIVU src, dest

【オペレーション】

dest = dest / src;

【機能】

- dest を src で符号なし除算し、その商を dest に格納します。商は 0 方向に丸められます。
- 演算は 32 ビットで行い、結果は 32 ビットで格納します。
- 除数（src）が 0 のときの dest の値は不定です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
DIVU src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs].memex(注)	Rd	5(memex == UB) 6(memex != UB)

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	－	－	－	○

O：除数（src）が0のとき “ 1 ”、それ以外のとき “ 0 ” になります。

【記述例】

```
DIVU    #10, R2
DIVU    R1, R2
DIVU    [R1], R2
DIVU    3[R1].UB, R2
```

EMUL 符号付き乗算 EMUL
Extended MULtiply, signed

【構文】

EMUL src, dest

【オペレーション】

dest2:dest = dest * src;

【機能】

- dest を src で符号付き乗算します。
- src、dest とも 32 ビットで演算し、結果を 64 ビットでレジスタペア dest2:dest（R(n+1):Rn）に格納します。
- dest には Rn（n：0 ～ 14）の 15 種類が指定できます。

注意 アキュムレータ（ACC）を使用します。命令実行後の ACC の値は不定です。

dest で指定するレジスタ	64 ビット拡張で使用されるレジスタ
R0	R1:R0
R1	R2:R1
R2	R3:R2
R3	R4:R3
R4	R5:R4
R5	R6:R5
R6	R7:R6
R7	R8:R7
R8	R9:R8
R9	R10:R9
R10	R11:R10
R11	R12:R11
R12	R13:R12
R13	R14:R13
R14	R15:R14

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
EMUL src, dest	L	#SIMM:8	Rd (Rd=R0~R14)	4
	L	#SIMM:16	Rd (Rd=R0~R14)	5
	L	#SIMM:24	Rd (Rd=R0~R14)	6

構文	処理サイズ	src	dest	コードサイズ（バイト）
	L	#IMM:32	Rd (Rd=R0~R14)	7
	L	Rs	Rd (Rd=R0~R14)	3
	L	[Rs].memex	Rd (Rd=R0~R14)	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd (Rd=R0~R14)	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs]memex(注)	Rd (Rd=R0~R14)	5(memex == UB) 6(memex != UB)

注意　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“ .W ”または“ .UW ”のときは 2 の倍数、“ .L ”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“ .W ”または“ .UW ”のとき 0 ～ 510（255 × 2）が、“ .L ”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“ .W ”または“ .UW ”のとき 0 ～ 131070（65535 × 2）が、“ .L ”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
EMUL    #10, R2
EMUL    R1, R2
EMUL    [R1], R2
EMUL    8[R1].W, R2
```

EMULU　符号なし乗算　EMULU
Extended MULtiply, Unsigned

【構文】

EMULU　src, dest

【オペレーション】

dest2:dest = dest * src;

【機能】

- dest を src で符号なし乗算します。
- src、dest とも 32 ビットで演算し、結果を 64 ビットでレジスタペア dest2:dest（R(n+1):Rn）に格納します。
- dest には Rn（n：0 ～ 14）の 15 種類が指定できます。

注意　アキュムレータ（ACC）を使用します。命令実行後の ACC の値は不定です。

dest で指定するレジスタ	64 ビット拡張で使用されるレジスタ
R0	R1:R0
R1	R2:R1
R2	R3:R2
R3	R4:R3
R4	R5:R4
R5	R6:R5
R6	R7:R6
R7	R8:R7
R8	R9:R8
R9	R10:R9
R10	R11:R10
R11	R12:R11
R12	R13:R12
R13	R14:R13
R14	R15:R14

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
EMUL U　src, dest	L	#SIMM:8	Rd (Rd=R0~R14)	4
	L	#SIMM:16	Rd (Rd=R0~R14)	5
	L	#SIMM:24	Rd (Rd=R0~R14)	6

構文	処理サイズ	src	dest	コードサイズ（バイト）
	L	#IMM:32	Rd (Rd=R0~R14)	7
	L	Rs	Rd (Rd=R0~R14)	3
	L	[Rs].memex	Rd (Rd=R0~R14)	3(memex == UB) 4(memex != UB)
	L	dsp:8[Rs].memex(注)	Rd (Rd=R0~R14)	4(memex == UB) 5(memex != UB)
	L	dsp:16[Rs]memex(注)	Rd (Rd=R0~R14)	5(memex == UB) 6(memex != UB)

注意　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“.W”または“.UW”のときは 2 の倍数、“.L”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 510（255 × 2）が、“.L”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 131070（65535 × 2）が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
EMULU   #10, R2
EMULU   R1, R2
EMULU   [R1], R2
EMULU   8[R1].W, R2
```

FADD 浮動小数点加算 FADD
Floating-point ADD

【構文】

FADD src, dest

【オペレーション】

dest = dest + src;

【機能】

- dest に格納された単精度浮動小数点数と、src に格納された単精度浮動小数点数を加算し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。
- 非正規化数の扱いは、FPSW の DN ビットによって変化します。
- 反対の符号を持つ src、dest の和が正確に 0 であるときは、-∞方向への丸めモードの場合を除いて、結果は +0 になります。-∞方向への丸めモードの場合は、結果は -0 になります。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
FADD src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L（注）	Rd	4
	L	dsp:16[Rs].L（注）	Rd	5

注意 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	−	○	○	○	○	○	○	○	○	−	○	○

条件

Z：演算の結果が “ +0 ” または “ -0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部（ビット 31）が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：無効演算が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CO：オーバフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：アンダフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：非実装処理が発生したとき“1”、それ以外のとき“0”になります。

FV：無効演算が発生したとき“1”、それ以外のときは変化しません。

FO：オーバフローが発生したとき“1”、それ以外のときは変化しません。

FU：アンダフローが発生したとき“1”、それ以外のときは変化しません。

FX：精度異常が発生したとき“1”、それ以外のときは変化しません。

注意　FX、FU、FO、FV フラグは、例外処理許可ビット EX、EU、EO、EV が“1”の場合は変化しません。S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
FADD    R1, R2
FADD    [R1], R2
```

FCMP 浮動小数点比較 FCMP
Floating-point CoMPare

【構文】

FCMP src, src2

【オペレーション】

src2 - src;

【機能】

- src2 に格納された単精度浮動小数点数と、src に格納された単精度浮動小数点数を比較し、その結果にしたがってフラグが変化します。
- 非正規化数の扱いは、FPSW の DN ビットによって変化します。

【命令フォーマット】

構文	処理サイズ	src	src2	コードサイズ（バイト）
FCMP src, src2	L	#IMM:32	Rs2	7
	L	Rs	Rs2	3
	L	[Rs].L	Rs2	3
	L	dsp:8[Rs].L（注）	Rs2	4
	L	dsp:16[Rs].L（注）	Rs2	5

注意 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	○	○	○	○	○	○	○	○	−	−	−	−

条件

Z：src2 == src のとき“1”、それ以外のとき“0”になります。

S：src2 < src のとき“1”、それ以外のとき“0”になります。

O：比較結果が順序化不能のとき“1”、それ以外のとき“0”になります。

CV：無効演算が発生したとき“1”、それ以外のとき“0”になります。

CO：常に“0”になります。

CZ：常に“0”になります。

CU：常に“0”になります。

CX：常に“0”になります。

CE：非実装処理が発生したとき“1”、それ以外のとき“0”になります。

FV：無効演算が発生したとき“1”、それ以外のときは変化しません。

注 FV フラグは、例外処理許可ビット EV が“1”の場合は変化しません。O、S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
FCMP    R1, R2
FCMP    [R1], R2
```

FDIV　浮動小数点除算　FDIV
Floating-point DIVide

【構文】

FDIV　src, dest

【オペレーション】

dest = dest / src;

【機能】

- dest に格納された単精度浮動小数点数を、src に格納された単精度浮動小数点数で除算し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。
- 非正規化数の扱いは、FPSW の DN ビットによって変化します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
FDIV　src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L（注）	Rd	4
	L	dsp:16[Rs].L（注）	Rd	5

注意　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	−	○	○	○	○	○	○	○	○	○	○	○

条件

Z：演算の結果が“+0”または“-0”のとき“1”、それ以外のとき“0”になります。

S：演算の結果、符号部（ビット 31）が“1”のとき“1”、“0”のとき“0”になります。

CV：無効演算が発生したとき“1”、それ以外のとき“0”になります。

CO：オーバフローが発生したとき“1”、それ以外のとき“0”になります。

CZ：ゼロ除算が発生したとき“1”、それ以外のとき“0”になります。

CU：アンダフローが発生したとき“1”、それ以外のとき“0”になります。

CX：精度異常が発生したとき“1”、それ以外のとき“0”になります。

CE：非実装処理が発生したとき“1”、それ以外のとき“0”になります。

FV：無効演算が発生したとき“1”、それ以外のときは変化しません。

FO：オーバフローが発生したとき“1”、それ以外のときは変化しません。

FZ：ゼロ除算が発生したとき“1”、それ以外のときは変化しません。

FU：アンダフローが発生したとき“1”、それ以外のときは変化しません。

FX：精度異常が発生したとき“1”、それ以外のときは変化しません。

【記述例】

```
FDIV    R1, R2
FDIV    [R1], R2
```

FMUL 浮動小数点乗算 FMUL
Floating-point MULtiply

【構文】

FMUL src, dest

【オペレーション】

dest = dest * src;

【機能】

- dest に格納された単精度浮動小数点数と、src に格納された単精度浮動小数点数を乗算し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。
- 非正規化数の扱いは、FPSW の DN ビットによって変化します。

注意 アキュムレータ（ACC）を使用します。命令実行後の ACC の値は不定です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
FMUL src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L（注）	Rd	4
	L	dsp:16[Rs].L（注）	Rd	5

注意 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	−	○	○	○	○	○	○	○	○	−	○	○

条件

Z：演算の結果が “ +0 ” または “ -0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部（ビット 31）が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：無効演算が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CO：オーバフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：アンダフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：非実装処理が発生したとき“1”、それ以外のとき“0”になります。
FV：無効演算が発生したとき“1”、それ以外のときは変化しません。
FO：オーバフローが発生したとき“1”、それ以外のときは変化しません。
FU：アンダフローが発生したとき“1”、それ以外のときは変化しません。
FX：精度異常が発生したとき“1”、それ以外のときは変化しません。

注 FX、FU、FO、FV フラグは、例外処理許可ビット EX、EU、EO、EV が“1”の場合は変化しません。S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
FMUL    R1, R2
FMUL    [R1], R2
```

FSUB　浮動小数点減算　FSUB
Floating-point SUBtract

【構文】

FSUB　src, dest

【オペレーション】

dest = dest - src;

【機能】

- dest に格納された単精度浮動小数点数から、src に格納された単精度浮動小数点数を減算し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。
- 非正規化数の扱いは、FPSW の DN ビットによって変化します。
- 同一の符号を持つ src、dest の差が正確に 0 であるときは、- ∞方向への丸めモードの場合を除いて、結果は +0 になります。- ∞方向への丸めモードの場合は、結果は -0 になります。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
FSUB　src, dest	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L（注）	Rd	4
	L	dsp:16[Rs].L（注）	Rd	5

注意　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	－	○	○	－	○	○	○	○	○	○	○	○	－	○	○

条件

Z：演算の結果が “ +0 ” または “ -0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部 (ビット 31) が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：無効演算が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CO：オーバフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：アンダフローが発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：非実装処理が発生したとき“1”、それ以外のとき“0”になります。

FV：無効演算が発生したとき“1”、それ以外のときは変化しません。

FO：オーバフローが発生したとき“1”、それ以外のときは変化しません。

FU：アンダフローが発生したとき“1”、それ以外のときは変化しません。

FX：精度異常が発生したとき“1”、それ以外のときは変化しません。

【記述例】

```
FSUB    R1, R2
FSUB    [R1], R2
```

FTOI　　浮動小数点数→整数変換　FTOI
Float TO Integer

【構文】

FTOI　src, dest

【オペレーション】

dest = (signed long) src;

【機能】

- src に格納された単精度浮動小数点数を符号付きロングワード（32 ビット）整数に変換し、その結果を dest に格納します。
- 結果は FPSW の RM[1:0] ビットに関係なく、常に 0 方向に丸められます。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
FTOI　src, dest	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L(注)	Rd	4
	L	dsp:16[Rs].L(注)	Rd	5

注　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	–	○	○	–	○	○	○	○	○	○	○	–	–	–	○

条件

Z：演算の結果が “ 0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部（ビット 31）が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：無効演算が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CO：常に “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：常に “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：非実装処理が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

FV：無効演算が発生したとき “ 1 ”、それ以外のときは変化しません。

FX：無効演算が発生したとき “ 1 ”、それ以外のときは変化しません。

注 FX、FV フラグは、例外処理許可ビット EX、EV が“1”の場合は変化しません。S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
FTOI    R1, R2
FTOI    [R1], R2
```

INT ソフトウェア割り込み INT
INTerrupt

【構文】

INT src

【オペレーション】

```
tmp0 = PSW;
U = 0;
I = 0;
PM = 0;
tmp1 = PC + 3;
PC = *(IntBase + src * 4);
SP = SP - 4;
*SP = tmp0;
SP = SP - 4;
*SP = tmp1;
```

【機能】

- src で指定した番号の無条件トラップが発生します。
- src の範囲は、0 ≦ src ≦ 255 です。
- スーパバイザモードに移行し、PSW の PM ビットが “ 0 ” になります。
- PSW の U、I ビットが “ 0 ” になります。

【命令フォーマット】

構文	src	コードサイズ（バイト）
INT src	#IMM:8	3

【フラグ変化】

フラグ変化はありません。

命令実行前の PSW は、スタックに退避されます。

【記述例】

```
INT     #0
```

ITOF　　整数→浮動小数点数変換　ITOF
Integer TO Floating-point

【構文】

ITOF　src, dest

【オペレーション】

dest = (float) src;

【機能】

- src に格納された符号付きロングワード（32 ビット）整数を単精度浮動小数点数に変換し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。00000000h は丸めモードに関係なく、“ +0 ” として扱われます。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ITOF　src, dest	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex（注）	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex（注）	Rd	5 (memex == UB) 6 (memex != UB)

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	−	○	○	○	○	○	○	−	−	−	−	○

条件

Z：演算の結果が “ +0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部（ビット 31）が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：常に “ 0 ” になります。

CO：常に “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：常に “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：常に “ 0 ” になります。

FX：精度異常が発生したとき “ 1 ”、それ以外のときは変化しません。

注 FX フラグは、例外処理許可ビット EX が“1”の場合は変化しません。S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
ITOF    R1, R2
ITOF    [R1], R2
ITOF    16[R1].L, R2
```

JUMP　無条件分岐　JMP
JuMP

【構文】

JMP　src

【オペレーション】

PC = src;

【機能】

- src へ分岐します。

【命令フォーマット】

構文	src	コードサイズ（バイト）
JMP　src	Rs	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
JMP     R1
```

JSR サブルーチン分岐 JSR
Jump SubRoutine

【構文】

JSR src

【オペレーション】

SP = SP - 4;
*SP = (PC + 2);（注）
PC = src;

注 (PC + 2) **は** JSR **命令の次の命令の番地です。**

【機能】

- src **が示すサブルーチンへ分岐します。**

【命令フォーマット】

構文	src	コードサイズ（バイト）
JSR src	Rs	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
JSR     R1
```

MACHI 上位 16 ビット積和演算 MACHI
Multiply-ACcumulate
HIgh-order word

【構文】

MACHI src, src2

【オペレーション】

```
signed short tmp1, tmp2;
signed long long tmp3;
tmp1 = (signed short) (src >> 16);
tmp2 = (signed short) (src2 >> 16);
tmp3 = (signed long) tmp1 * (signed long) tmp2;
ACC = ACC + (tmp3 << 16);
```

【機能】

- src の上位 16 ビットと src2 の上位 16 ビットの乗算を行い、乗算結果とアキュムレータの加算を行います。ただし、乗算結果の最下位ビットはアキュムレータの b16 にあわせて加算します。加算結果はアキュムレータに格納されます。src の上位 16 ビットと src2 の上位 16 ビットは符号付き整数として扱われます。

【命令フォーマット】

構文	src	src2	コードサイズ（バイト）
MACHI src, src2	Rs	Rs2	3

【フラグ変化】：

フラグ変化はありません。

【記述例】

```
MACHI   R1, R2
```

MACLO 下位 16 ビット積和演算 MACLO
Multiply-ACcumulate
LOw-order word

【構文】

MACLO src, src2

【オペレーション】

```
signed short tmp1, tmp2;
signed long long tmp3;
tmp1 = (signed short) src;
tmp2 = (signed short) src2;
tmp3 = (signed long) tmp1 * (signed long) tmp2;
ACC = ACC + (tmp3 << 16);
```

【機能】

- src の下位 16 ビットと src2 の下位 16 ビットの乗算を行い、乗算結果とアキュムレータの加算を行います。ただし、乗算結果の最下位ビットはアキュムレータの b16 にあわせて加算します。加算結果はアキュムレータに格納されます。src の下位 16 ビットと src2 の下位 16 ビットは符号付き整数として扱われます。

【命令フォーマット】

構文	src	src2	コードサイズ（バイト）
MACLO src, src2	Rs	Rs2	3

【フラグ変化】：

フラグ変化はありません。

【記述例】

```
MACLO   R1, R2
```

【関連命令】

EXITD

MAX 最大値選択 MAX
MAXimum value select

【構文】

MAX src, dest

【オペレーション】

```
if ( src > dest )
    dest = src;
```

【機能】

- src と dest を符号付きで比較し、大きい方の値を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
MAX src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex（注）	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex（注）	Rd	5 (memex == UB) 6 (memex != UB)

注意 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“.W”または“.UW”のときは 2 の倍数、“.L”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 510（255 × 2）が、“.L”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 131070（65535 × 2）が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MAX     #10, R2
MAX     R1, R2
MAX     [R1], R2
MAX     3[R1].B, R2
```

MIN 最小値選択 MIN
MINimum value select

【構文】

MIN src, dest

【オペレーション】

```
if ( src < dest )
  dest = src;
```

【機能】

- src と dest を符号付きで比較し、小さい方の値を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
MIX src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex（注）	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex（注）	Rd	5 (memex == UB) 6 (memex != UB)

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“ .W ”または“ .UW ”のときは 2 の倍数、“ .L ”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“ .W ”または“ .UW ”のとき 0 ～ 510（255 × 2）が、“ .L ”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“ .W ”または“ .UW ”のとき 0 ～ 131070（65535 × 2）が、“ .L ”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MIN     #10, R2
MIN     R1, R2
MIN     [R1], R2
MIN     3[R1].B, R2
```

MOV 転送 MOV
MOVe

【構文】

MOV.size src, dest

【オペレーション】

dest = src;

【機能】

- 以下のとおり、src を dest に転送します。

src	dest	機能
即値	レジスタ	即値をレジスタに転送します。32 ビット未満の即値が指定された場合、#UIMM はゼロ拡張を、#SIMM は符号拡張を行いレジスタに転送します。
即値	メモリ	即値を指定したサイズでメモリに転送します。指定したサイズよりもビット幅の小さい即値が指定された場合、#UIMM はゼロ拡張を、#SIMM は符号拡張を行いメモリに転送します。
レジスタ	レジスタ	レジスタ（src）のデータをレジスタ（dest）に転送します。サイズ指定子が .B のときは、レジスタ（src）の LSB 側のバイトデータをロングワードデータに符号拡張し、レジスタ（dest）に転送します。サイズ指定子が .W のときは、レジスタ（src）の LSB 側のワードデータをロングワードデータに符号拡張し、レジスタ（dest）に転送します。
レジスタ	メモリ	レジスタのデータをメモリに転送します。サイズ指定子が .B のときは、レジスタの LSB 側のバイトデータを転送します。サイズ指定子が .W のときは、レジスタの LSB 側のワードデータを転送します。
メモリ	レジスタ	メモリのデータをレジスタに転送します。サイズ指定子が .B または .W のときは、メモリのデータをロングワードデータに符号拡張し、レジスタに転送します。
メモリ	メモリ	指定したサイズでメモリ（src）のデータをメモリ（dest）に転送します。

【命令フォーマット】

構文	size	処理サイズ	src	dest	コードサイズ（バイト）
MOV.size src, dest	B/W/L	size	Rs（Rs=R0 ～ R7）	dsp:5[Rd]（注 1）（Rd=R0 ～ R7）	2
	B/W/L	L	dsp:5[Rs]（注 1）（Rs=R0 ～ R7）	Rd（Rd=R0 ～ R7）	2
	L	L	#UIMM:4	Rd	2
	B	B	#IMM:8	dsp:5[Rd]（注 1）（Rd=R0 ～ R7）	3

構文	size	処理サイズ	src	dest	コードサイズ（バイト）
	W/L	size	#UIMM:8	dsp:5[Rd]（注 1）（Rd=R0 ～ R7）	3
	L	L	#UIMM:8（注 2）	Rd	3
	L	L	#SIMM:8（注 2）	Rd	3
	L	L	#SIMM:16	Rd	4
	L	L	#SIMM:24	Rd	5
	L	L	#IMM:32	Rd	6
	B/W	L	Rs	Rd	2
	L	L	Rs	Rd	3
	B	B	#IMM:8	[Rd]	3
	B	B	#IMM:8	dsp:8[Rd]（注 1）	4
	B	B	#IMM:8	dsp:16[Rd]（注 1）	5
	W	W	#SIMM:8	[Rd]	3
	W	W	#SIMM:8	dsp:8[Rd]（注 1）	4
	W	W	#SIMM:8	dsp:16[Rd]（注 1）	5
	W	W	#IMM:16	[Rd]	4
	W	W	#IMM:16	dsp:8[Rd]（注 1）	5
	W	W	#IMM:16	dsp:16[Rd]（注 1）	6
	L	L	#SIMM:8	[Rd]	3
	L	L	#SIMM:8	dsp:8[Rd]（注 1）	4
	L	L	#SIMM:8	dsp:16[Rd]（注 1）	5
	L	L	#SIMM:16	[Rd]	4
	L	L	#SIMM:16	dsp:8[Rd]（注 1）	5
	L	L	#SIMM:16	dsp:16[Rd]（注 1）	6
	L	L	#SIMM:24	[Rd]	5
	L	L	#SIMM:24	dsp:8[Rd]（注 1）	6
	L	L	#SIMM:24	dsp:16[Rd]（注 1）	7
	L	L	#IMM:32	[Rd]	6
	L	L	#IMM:32	dsp:8[Rd]（注 1）	7
	L	L	#IMM:32	dsp:16[Rd]（注 1）	8
	B/W/L	L	[Rs]	Rd	2
	B/W/L	L	dsp:8[Rs]（注 1）	Rd	3
	B/W/L	L	dsp:16[Rs]（注 1）	Rd	4
	B/W/L	L	[Ri, Rb]	Rd	3
	B/W/L	size	Rs	[Rd]	2
	B/W/L	size	Rs	dsp:8[Rd]（注 1）	3
	B/W/L	size	Rs	dsp:16[Rd]（注 1）	4

構文	size	処理サイズ	src	dest	コードサイズ（バイト）
	B/W/L	size	Rs	[Ri, Rb]	3
	B/W/L	size	[Rs]	[Rd]	2
	B/W/L	size	[Rs]	dsp:8[Rd]（注 1）	3
	B/W/L	size	[Rs]	dsp:16[Rd]（注 1）	4
	B/W/L	size	dsp:8[Rs]（注 1）	[Rd]	3
	B/W/L	size	dsp:8[Rs]（注 1）	dsp:8[Rd]（注 1）	4
	B/W/L	size	dsp:8[Rs]（注 1）	dsp:16[Rd]（注 1）	5
	B/W/L	size	dsp:16[Rs]（注 1）	[Rd]	4
	B/W/L	size	dsp:16[Rs]（注 1）	dsp:8[Rd]（注 1）	5
	B/W/L	size	dsp:16[Rs]（注 1）	dsp:16[Rd]（注 1）	6
	B/W/L	size	Rs	[Rd+]	3
	B/W/L	size	Rs	[-Rd]	3
	B/W/L	L	[Rs+]	Rd	3
	B/W/L	L	[-Rs]	Rd	3

注 1: 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:5、dsp:8、dsp:16）は、サイズ指定子が“.W”のときは 2 の倍数、“.L”のときは 4 の倍数を指定してください。dsp:5 には、サイズ指定子が“.W”のとき 0~62（31 × 2）が、“.L”のとき 0 ～ 124（31 × 4）が指定できます。dsp:8 には、サイズ指定子が“.W”のとき 0 ～ 510（255 × 2）が、“.L”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ指定子が“.W”のとき 0 ～ 131070（65535 × 2）が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

注 2:0 ～ 127 の範囲は、常にゼロ拡張命令コードになります。

注 3: ポストインクリメント付きストア、プリデクリメント付きストアで、Rs と Rd に同じレジスタを指定した場合、アドレス更新前の値がソースとして転送されます。

注 4: ポストインクリメント付きロード、プリデクリメント付きロードで、Rs と Rd に同じレジスタを指定した場合、メモリから転送されてきたデータが Rd に格納されます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MOV.L    #0, R2
MOV.L    #128:8, R2
MOV.L    #-128:8, R2
MOV.L    R1, R2
MOV.L    #0, [R2]
MOV.W    [R1], R2
MOV.W    R1, [R2]
MOV.W    [R1, R2], R3
MOV.W    R1, [R2, R3]
MOV.W    [R1], [R2]
MOV.B    R1, [R2+]
MOV.B    [R1+], R2
MOV.B    R1, [-R2]
MOV.B    [-R1], R2
```

MOVU 符号なしデータ転送 MOVU
MOVe Unsigned data

【構文】

MOVU.size src, dest

【オペレーション】

dest = src;

【機能】

- 以下のとおり、src を dest に転送します。

src	dest	機能
レジスタ	レジスタ	レジスタ（src）の LSB 側のバイトデータまたはワードデータをロングワードデータにゼロ拡張し、レジスタ（dest）に転送します。
メモリ	レジスタ	メモリのバイトデータまたはワードデータをロングワードデータにゼロ拡張し、レジスタに転送します。

【命令フォーマット】

構文	size	処理サイズ	src	dest	コードサイズ（バイト）
MOVU.size src, dest	B/W	L	dsp:5[Rs]（注 1）（Rs=R0 ～ R7）	Rd（Rd=R0 ～ R7）	2
	B/W	L	Rs	Rd	2
	B/W	L	[Rs]	Rd	2
	B/W	L	dsp:8[Rs]（注 1）	Rd	3
	B/W	L	dsp:16[Rs]（注 1）	Rd	4
	B/W	L	[Ri, Rb]	Rd	3
	B/W	L	[Rs+]	Rd	3
	B/W	L	[-Rs]	Rd	3

注 1: 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:5、dsp:8、dsp:16）は、サイズ指定子が “ .W ” のときは 2 の倍数を指定してください。dsp:5 には、サイズ指定子が “ .W ” のとき 0~62（31 × 2）が指定できます。dsp:8 には、サイズ指定子が “ .W ” のとき 0 ～ 510（255 × 2）が指定できます。dsp:16 には、サイズ指定子が “ .W ” のとき 0 ～ 131070（65535 × 2）が指定できます。命令コードには、1/2 した値が埋め込まれます。

注 2: ポストインクリメント付きロード、プリデクリメント付きロードで、Rs と Rd に同じレジスタを指定した場合、メモリから転送されてきたデータが Rd に格納されます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MOVU.W   2[R1], R2
MOVU.W   R1, R2
MOVU.B   [R1+], R2
MOVU.B   [-R1], R2
```

MUL 乗算 MUL
MULtiply

【構文】

(1)MUL src, dest

(2)MUL src, src2, dest

【オペレーション】

(1)dest = src * dest;

(2)dest = src * src2;

【機能】

- (1)src と dest を乗算し、その結果を dest に格納します。
  - 演算は 32 ビットで行い、結果の下位 32 ビットを格納します。
  - 演算結果は、符号付き乗算、符号なし乗算に関係なく同じになります。
- (2)src と src2 を乗算し、その結果を dest に格納します。
  - 演算は 32 ビットで行い、結果の下位 32 ビットを格納します。
  - 演算結果は、符号付き乗算、符号なし乗算に関係なく同じになります。

注 アキュムレータ（ACC）を使用します。命令実行後の ACC の値は不定です。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)MUL src, dest	L	#UIMM:4	—	Rd	2
	L	#SIMM:8	—	Rd	3
	L	#SIMM:16	—	Rd	4
	L	#SIMM:24	—	Rd	5
	L	#IMM:32	—	Rd	6
	L	Rs	—	Rd	2
	L	[Rs].memex	—	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex（注）	—	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex（注）	—	Rd	4 (memex == UB) 5 (memex != UB)
	L	Rs	Rs2	Rd	3

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“ .W ”または“ .UW ”のときは 2 の倍数、“ .L ”のときは 4 の倍数を指定してください。dsp:8 **には、**

サイズ拡張指定子が“.W”または“.UW”のとき0～510（255×2）が、“.L”のとき0～1020（255×4）が指定できます。dsp:16には、サイズ拡張指定子が“.W”または“.UW”のとき0～131070（65535×2）が、“.L”のとき0～262140（65535×4）が指定できます。命令コードには、1/2、1/4した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MUL     #10, R2
MUL     R1, R2
MUL     [R1], R2
MUL     4[R1].W, R2
MUL     R1, R2, R3
```

MULHI 上位 16 ビット乗算 MULHI
MULtiply HIgh-order word

【構文】

MULHI src, src2

【オペレーション】

```
signed short tmp1, tmp2;
signed long long tmp3;
tmp1 = (signed short) (src >> 16);
tmp2 = (signed short) (src2 >> 16);
tmp3 = (signed long) tmp1 * (signed long) tmp2;
ACC = (tmp3 << 16);
```

【機能】

- src の上位 16 ビットと src2 の上位 16 ビットの乗算を行い、その結果をアキュムレータに格納します。ただし、乗算結果の最下位ビットはアキュムレータの b16 にあわせ、アキュムレータの b63 ～ b48 に対応する部分は、符号拡張されます。また、アキュムレータの b15 ～ b0 は、“ 0 ” になります。src の上位 16 ビットと src2 の上位 16 ビットは符号付き整数として扱われます。

【命令フォーマット】

構文	src	src2	コードサイズ（バイト）
MULHI src, src2	Rs	Rs2	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MULHI   R1, R2
```

MULLO 下位16ビット乗算 MULLO
MULtiply LOw-order word

【構文】

MULLO src, src2

【オペレーション】

```
signed short tmp1, tmp2;
signed long long tmp3;
tmp1 = (signed short) src;
tmp2 = (signed short) src2;
tmp3 = (signed long) tmp1 * (signed long) tmp2;
ACC = (tmp3 << 16);
```

【機能】

- src の下位 16 ビットと src2 の下位 16 ビットの乗算を行い、その結果をアキュムレータに格納します。ただし、乗算結果の最下位ビットはアキュムレータの b16 にあわせ、アキュムレータの b63 ～ b48 に対応する部分は、符号拡張されます。また、アキュムレータの b15 ～ b0 は、“ 0 ” になります。src の下位 16 ビットと src2 の下位 16 ビットは符号付き整数として扱われます。

【命令フォーマット】

構文	src	src2	コードサイズ（バイト）
MULLO src, src2	Rs	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MULLO   R1, R2
```

MVFACHI アキュムレータ上位 32 ビットからの転送

MVFACHI MoVe From ACcumulator HIgh-order longword

【構文】

MVFACHI dest

【オペレーション】

dest = (signed long) (ACC >> 32);

【機能】

- アキュムレータの上位 32 ビットの内容を dest に転送します。

【命令フォーマット】

構文	dest	コードサイズ（バイト）
MVFACHI dest	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVFACHI  R1
```

MVFACMI アキュムレータ中央 32 ビットからの転送

MoVe From ACcumulator MIddle-order longword

【構文】

MVFACMI dest

【オペレーション】

dest = (signed long) (ACC >> 16);

【機能】

- アキュムレータの b47 ～ b16 の内容を dest に転送します。

【命令フォーマット】

構文	dest	コードサイズ（バイト）
MVFACMI dest	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVFACMI  R1
```

MVFC 制御レジスタからの転送 MVFC
MoVe From Control register

【構文】

MVFC src, dest

【オペレーション】

dest = src;

【機能】

- src を dest に転送します。
- src に PC を指定した場合、本命令の番地を dest に転送します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
MVFC src, dest	L	Rx(注)	Rd	3

注 選択可能な src：PC、ISP、USP、INTB、PSW、BPC、BPSW、FINTV、FPSW

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVFC    USP, R1
```

MVTACHI　アキュムレータ上位 32 ビットへの転送

MVTACHI
MoVe To ACcumulator HIgh-order longword

【構文】

MVTACHI src

【オペレーション】

ACC = (ACC & 00000000FFFFFFFFh) | ((signed long long)src << 32);

【機能】

- src の内容をアキュムレータの上位 32 ビット（b63 ～ b32）に転送します。

【命令フォーマット】

構文	src	コードサイズ（バイト）
MVTACHI src	Rs	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVTACHI  R1
```

MVTACLO　アキュムレータ下位 32 ビットへの転送　MVTACLO

MoVe To ACcumulator LOw-order longword

【構文】

MVTACLO　src

【オペレーション】

ACC = (ACC & FFFFFFFF00000000h) | src;

注 1.　n は命令のバイト数です。

【機能】

- src **の内容をアキュムレータの下位** 32 **ビット（**b31 ～ b0**）に転送します。**

【命令フォーマット】

構文	src	コードサイズ（バイト）
MVTACLO　src	Rs	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVTACLO  R1
```

MVTC 制御レジスタへの転送 MVTC
MoVe To Control register

【構文】

MVTC src, dest

【オペレーション】

dest = src;

【機能】

- src を dest に転送します。
- ユーザモードでは、ISP、INTB、BPC、BPSW、FINTV と、PSW の IPL[3:0]、PM、U、I ビットへの書き込みは無視されます。スーパバイザモードでは、PSW の PM ビットへの書き込みは無視されます。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
MVTC src, dest	L	#SIMM:8	Rx（注）	7
	L	#SIMM:16	Rx（注）	3
	L	#SIMM:24	Rx（注）	3
	L	#IMM:32	Rx（注）	4
	L	Rs	Rx（注）	5

注 選択可能な dest：ISP、USP、INTB、PSW、BPC、BPSW、FINTV、FPSW

dest に PC を指定することはできません。

【フラグ変化】

フラグ	C	Z	S	O
変化	（注）	（注）	（注）	（注）

注 dest が PSW のときだけ変化します。

【記述例】

```
MVTC    #0FFFFF000h, INTB
MVTC    R1, USP
```

MVTIPL 割り込み優先レベル設定 MVTIPL
MoVe To Interrupt Priority Level

【構文】

MVTIPL src

【オペレーション】

IPL = src;

【機能】

- src を PSW の IPL[3:0] ビットに転送します。
- この命令は特権命令です。ユーザモードで実行すると特権命令例外が発生します。
- src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 15 です。

【命令フォーマット】

構文	src	コードサイズ（バイト）
MVTIPL src	#IMM:32	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
MVTIPL  #2
```

注 RX610 グループでは、MVTIPL 命令は使用できません。プロセッサステータスワード（PSW）のプロセッサ割り込み優先レベル（IPL[2:0]）への書き込みには、MVTC 命令を使用してください。

NEG 符号反転 NEG
NEGate

【構文】

(1)NEG dest

(2)NEG src, dest

【オペレーション】

(1)dest = -dest;

(2)dest = -src;

【機能】

- (1)dest を符号反転し（2 の補数をとり）、その結果を dest に格納します。
- (2)src を符号反転し（2 の補数をとり）、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
(1)NEG dest	L	—	Rd	2
(2)NEG src, dest	L	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：演算後の dest が 0 のとき “ 1 ”、それ以外のとき “ 0 ” になります。

Z：演算後の dest が 0 のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算後の dest の MSB が “ 1 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

O：(1) 演算前の dest が 80000000h のとき “ 1 ”、それ以外のとき “ 0 ” になります。

(2) 演算前の src が 80000000h のとき “ 1 ”、それ以外のとき “ 0 ” になります。

【記述例】

```
NEG     R1
NEG     R1, R2
```

NOP　ノーオペレーション　NOP
No OPeration

【構文】

NOP

【オペレーション】

/* ノーオペレーション */

【機能】

- 処理は何も行いません。次の命令から継続して実行されます。

【命令フォーマット】

構文	コードサイズ（バイト）
NOP	1

【フラグ変化】

フラグ変化はありません。

【記述例】

```
NOP
```

NOT 論理反転 NOT

【構文】

(1)NOT dest

(2)NOT src, dest

【オペレーション】

(1)dest = ~dest;

(2)dest = ~src;

【機能】

- (1)dest を論理反転し、その結果を dest に格納します。
- (2)src を論理反転し、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
(1)NOT dest	L	―	Rd	2
(2)NOT src, dest	L	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	―	○	○	―

条件

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
NOT     R1
NOT     R1, R2
```

OR　論理和　OR

OR

【構文】

(1)OR　src, dest

(2)OR　src, src2, dest

【オペレーション】

(1)dest = dest | src;

(2)dest = src | src2;

【機能】

- (1)dest と src の論理和をとり、その結果を dest に格納します。
- (2)src と src2 の論理和をとり、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)OR　src, dest	L	#UIMM:4	–	Rd	2
	L	#SIMM:8	–	Rd	3
	L	#SIMM:16	–	Rd	4
	L	#SIMM:24	–	Rd	5
	L	#IMM:32	–	Rd	6
	L	Rs	–	Rd	2
	L	[Rs].memex	–	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex（注）	–	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex（注）	–	Rd	4 (memex == UB) 5 (memex != UB)
(2)OR　src, src2, dest	L	Rs	Rs2	Rd	3

注　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）

が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	－	○	○	－

条件

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
OR      #8, R1
OR      R1, R2
OR      [R1], R2
OR      8[R1].L, R2
OR      R1, R2, R3
```

POP スタックからレジスタへの MOV データ復帰 MOVe

【構文】

POP dest

【オペレーション】

tmp = *SP;

SP = SP + 4;

dest = tmp;

【機能】

- スタックから復帰したデータを dest に転送します。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。

【命令フォーマット】

構文	処理サイズ	dest	コードサイズ（バイト）
POP dest	L	Rd	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
POP     R1
```

POPC 制御レジスタの復帰 POPC
POP Control register

【構文】

POPC dest

【オペレーション】

tmp = *SP;

SP = SP + 4;

dest = tmp;

【機能】

- スタックから復帰したデータを、dest で示される制御レジスタに転送します。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。
- ユーザモードでは、ISP、INTB、BPC、BPSW、FINTV と、PSW の IPL[3:0]、PM、U、I ビットへの書き込みは無視されます。スーパバイザモードでは、PSW の PM ビットへの書き込みは無視されます。

【命令フォーマット】

構文	処理サイズ	dest	コードサイズ（バイト）
POPC dest	L	Rx(注)	2

注 選択可能な dest：ISP、USP、INTB、PSW、BPC、BPSW、FINTV、FPSW

dest に PC を指定することはできません。

【フラグ変化】

フラグ	C	Z	S	O
変化	（注）	（注）	（注）	（注）

注 dest が PSW のときだけ変化します。

【記述例】

```
POPC    PSW
```

POPM 複数レジスタの復帰 POPM
POP Multiple registers

【構文】

POPM dest-dest2

【オペレーション】

```
signed char i;
for ( i = register_num(dest); i <= register_num(dest2); i++ ) {
   tmp = *SP;
   SP = SP + 4;
   register(i) = tmp;
}
```

【機能】

- dest と dest2 で範囲指定したレジスタを一括してスタックから復帰します。
- 範囲は、先頭レジスタ番号と最終レジスタ番号で指定します。ただし、（先頭レジスタのレジスタ番号 < 最終レジスタのレジスタ番号）となっている必要があります。
- R0 を指定することはできません。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。
- スタックから復帰する順序は以下のとおりです。

【命令フォーマット】

構文	処理サイズ	dest	dest2	コードサイズ（バイト）
POPM dest-dest2	L	Rd （Rd=R1 ～ R14）	Rd2 （Rd2=R2 ～ R15）	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
POPM    R1-R3
POPM    R4-R8
```

PUSH　スタックデータ退避　PUSH
PUSH data onto the stack

【構文】

PUSH.size　src

【オペレーション】

tmp = src;

SP = SP - 4;（注）

*SP = tmp;

備考　サイズ指定子（.size）が“ .B ”、“ .W ”でも SP は 4 減算されます。“ .B ”のときの上位 24 ビット、“ .W ”のときの上位 16 ビットは不定になります。

【機能】

- src をスタックに退避します。
- src がレジスタ、かつサイズ指定子が“ .B ”または“ .W ”のとき、それぞれレジスタの LSB 側のバイトデータ、またはワードデータを退避します。
- スタックへの転送サイズは、ロングワードで行います。サイズ指定子が .B のときの上位 24 ビット、.W のときの上位 16 ビットは不定になります。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。

【命令フォーマット】

構文	size	処理サイズ	src	コードサイズ（バイト）
PUSH.size　src	B/W/L	L	Rs	2
	B/W/L	L	[Rs]	2
	B/W/L	L	dsp:8[Rs]（注）	3
	B/W/L	L	dsp:16[Rs]（注）	4

注　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ指定子が“ .W ”のときは 2 の倍数、“ .L ”のときは 4 の倍数を指定してください。dsp:8 には、サイズ指定子が“ .W ”のとき 0 ～ 510（255 × 2）が、“ .L ”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ指定子が“ .W ”のとき 0 ～ 131070（65535 × 2）が、“ .L ”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
PUSH.B   R1
PUSH.L   [R1]
```

PUSHC 制御レジスタの退避 PUSHC
PUSH Control register

【構文】

PUSHC src

【オペレーション】

```
tmp = src;
SP = SP - 4;
*SP = tmp;
```

【機能】

- src で示される制御レジスタをスタックに退避します。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。
- src に PC を指定した場合、本命令の番地をスタックに退避します。

【命令フォーマット】

構文	処理サイズ	src	コードサイズ（バイト）
PUSHC src	L	Rx(注)	2

注 選択可能な src：PC、ISP、USP、INTB、PSW、BPC、BPSW、FINTV、FPSW

【フラグ変化】

フラグ変化はありません。

【記述例】

```
PUSHC   PSW
```

PUSHM 複数レジスタの退避 PUSHM
PUSH Multiple registers

【構文】

PUSHM src-src2

【オペレーション】

```
signed char i;
for ( i = register_num(src2); i >= register_num(src); i-- ) {
  tmp = register(i);
  SP = SP - 4;
  *SP = tmp;
}
```

【機能】

- src と src2 で範囲指定したレジスタを一括してスタックに退避します。
- 範囲は、先頭レジスタ番号と最終レジスタ番号で指定します。ただし、（先頭レジスタのレジスタ番号 < 最終レジスタのレジスタ番号）となっている必要があります。
- R0 を指定することはできません。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。
- スタックに退避する順序は R15 から退避します。

【命令フォーマット】

構文	処理サイズ	src	src2	コードサイズ（バイト）
PUSHM src-src2	L	Rs （Rs=R1 ～ R14）	Rs2 （Rs2=R2 ～ R15）	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
PUSHM   R1-R3
PUSHM   R4-R8
```

RACW　　16 ビット符号付きアキュム　RACW

レータ丸め処理
Round ACcumulator Word

【構文】

RACW　src

【オペレーション】

```
signed long long tmp;
tmp = (signed long long) ACC << src;
tmp = tmp + 0000000080000000h;
if (tmp > (signed long long) 00007FFF00000000h)
   ACC = 00007FFF00000000h;
else if (tmp < (signed long long) FFFF800000000000h)
   ACC = FFFF800000000000h;
else
   ACC = tmp & FFFFFFFF00000000h;
```

【機能】

- アキュムレータの値に対してワードサイズで丸めを行い、その結果をアキュムレータに格納します。

【命令フォーマット】

構文	src	コードサイズ（バイト）
RACW　src	#IMM:1（注） (IMM:1 = 1 ～ 2)	3

注　弊社の「RX ファミリ アセンブラ」では、即値（IMM:1）は、1 ～ 2 を指定してください。命令コードには、-1 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
RACW    #1
RACW    #2
```

REVL エンディアン変換 REVL
REVerse Longword data

【構文】

REVL src, dest

【オペレーション】

Rd = { Rs[7:0], Rs[15:8], Rs[23:16], Rs[31:24] }

【機能】

- src で指定した 32 ビットデータをバイト単位でエンディアン変換します。その結果を dest に格納します。

【命令フォーマット】

構文	src	dest	コードサイズ（バイト）
REVL src, dest	Rs	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
REVL    R1, R2
```

REVW　エンディアン変換　REVW
REVerse Word data

【構文】

REVW　src, dest

【オペレーション】

Rd = { Rs[23:16], Rs[31:24], Rs[7:0], Rs[15:8] }

【機能】

- src で指定した上位 16 ビットデータと下位 16 ビットデータそれぞれで、バイト単位でエンディアン変換します。その結果を dest に格納します。

【命令フォーマット】

構文	src	dest	コードサイズ（バイト）
REVW　src, dest	Rs	Rd	3

【フラグ変化】

フラグ変化はありません。

【記述例】

```
REVW    R1, R2
```

RMPA 積和演算 RMPA

Repeated MultiPly and Accumulate

【構文】

RMPA.size

【オペレーション】

```
while ( R3 != 0 ) {
  R6:R5:R4 = R6:R5:R4 + *R1 * *R2;
  R1 = R1 + n;
  R2 = R2 + n;
  R3 = R3 - 1;
}
```

注 1. R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

注 2. n：サイズ指定子 (.size) が“ .B ”のとき 1、“ .W ”のとき 2、“ .L ”のとき 4 になります。

【機能】

- R1 を被乗数番地、R2 を乗数番地、R3 を回数とする積和演算を行います。演算は符号付きで行い、その結果を R6:R5:R4 の 80 ビットに格納します。ただし、R6 の上位 16 ビットには、下位 16 ビットを符号拡張した値が格納されます。
- R3 に設定可能な最大値は 00010000h です。
- 命令終了時の R1、R2 の内容は不定となります。
- 命令実行前に R6:R5:R4 には初期値を設定してください。また、R6 には R5:R4 が負のときは“ FFFFFFFFh ”を、正のときは“ 00000000h ”を設定してください。
- 命令実行中に割り込み要求があった場合は、演算を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3、R4、R5、R6 と PSW を退避／復帰してください。
- 命令実行時は、R1 で示される被乗数番地と R2 で示される乗数番地から、それぞれデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。

注 アキュムレータ（ACC）を使用します。命令実行後の ACC の値は不定です。

【命令フォーマット】

構文	size	size	コードサイズ（バイト）
RMPA.size	B/W/L	size	2

【フラグ変化】

フラグ	C	Z	S	O
変化	–	–	○	○

条件

S：R6 の MSB が“1”のとき“1”、それ以外のとき“0”になります。

O：R6:R5:R4 の内容が 2 - 1 または -2 を超えると“1”、それ以外のとき“0”になります。

【記述例】

```
RMPA.W
```

ROLC キャリ付き左回転 ROLC
ROtate Left with Carry

【構文】

ROLC dest

【オペレーション】

```
dest <<= 1;
if ( C == 0 ) { dest &= FFFFFFFEh; }
else { dest |= 00000001h; }
```

【機能】

- C フラグを含めて、dest を 1 ビット左へ回転します。

注 1. U フラグで示すスタックポインタが対象となります。

【命令フォーマット】

構文	処理サイズ	dest	コードサイズ（バイト）
ROLC dest	L	Rd	2

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	–

条件

C：シフトアウトしたビットが“1”のとき“1”、それ以外のとき“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
ROLC    R1
```

RORC キャリ付き右回転 RORC
ROtate Right with Carry

【構文】

RORC dest

【オペレーション】

```
dest >>= 1;
if ( C == 0 ) { dest &= 7FFFFFFFh; }
else { dest |= 80000000h; }
```

【機能】

- C フラグを含めて、dest を 1 ビット右へ回転します。

【命令フォーマット】

構文	処理サイズ	dest	コードサイズ（バイト）
RORC dest	L	Rd	2

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	−

条件

C：シフトアウトしたビットが“1”のとき“1”、それ以外のとき“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
RORC    R1
```

ROTL 左回転 ROTL
ROTate Left

【構文】

ROTL src, dest

【オペレーション】

```
unsigned long tmp0, tmp1;
tmp0 = src & 31;
tmp1 = dest << tmp0;
dest = (( unsigned long ) dest >> ( 32 - tmp0 )) | tmp1;
```

【機能】

- dest を src で指定されたビット数分だけ左回転します。MSB から溢れたビットは LSB と C フラグに転送します。
- src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。
- src がレジスタのとき、LSB 側 5 ビットのみ有効です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ROTL src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	−

条件

C：演算後の dest の LSB と同じになります。src が 0 のときも、演算後の dest の LSB と同じになります。

Z：演算後の dest が 0 のとき “1”、それ以外のとき “0” になります。

S：演算後の dest の MSB が “1” のとき “1”、それ以外のとき “0” になります。

【記述例】

```
ROTL    #1, R1
ROTL    R1, R2
```

ROTR 右回転 ROTR
ROTate Right

【構文】

ROTR src, dest

【オペレーション】

```
unsigned long tmp0, tmp1;
tmp0 = src & 31;
tmp1 = ( unsigned long ) dest >> tmp0;
dest = ( dest << ( 32 - tmp0 )) | tmp1;
```

【機能】

- dest を src で指定されたビット数分だけ右回転します。LSB から溢れたビットは MSB と C フラグに転送します。
- src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。
- src がレジスタのとき、LSB 側 5 ビットのみ有効です。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ROTR src, dest	L	#IMM:5	Rd	3
	L	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	−

条件

C：演算後の dest の MSB と同じになります。src が 0 のときも、演算後の dest の MSB と同じになります。

Z：演算後の dest が 0 のとき “1”、それ以外のとき “0” になります。

S：演算後の dest の MSB が “1” のとき “1”、それ以外のとき “0” になります。

【記述例】

```
ROTR    #1, R1
ROTR    R1, R2
```

ROUND 浮動小数点数→整数変換 ROUND

ROUND floating-point to integer

【構文】

ROUND src, dest

【オペレーション】

dest = (signed long) src;

【機能】

- src に格納された単精度浮動小数点数を符号付きロングワード（32 ビット）整数に変換し、その結果を dest に格納します。結果は FPSW の RM[1:0] ビットにしたがって丸められます。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
ROUND src, dest	L	Rs	Rd	3
	L	[Rs].L	Rd	3
	L	dsp:8[Rs].L（注）	Rd	4
	L	dsp:16[Rs].L（注）	Rd	4

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O	CV	CO	CZ	CU	CX	CE	FV	FO	FZ	FU	FX
変化	−	○	○	−	○	○	○	○	○	○	○	−	−	−	○

条件

Z：演算の結果が “ 0 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算の結果、符号部（ビット 31）が “ 1 ” のとき “ 1 ”、“ 0 ” のとき “ 0 ” になります。

CV：無効演算が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CO：常に “ 0 ” になります。

CZ：常に “ 0 ” になります。

CU：常に “ 0 ” になります。

CX：精度異常が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

CE：非実装処理が発生したとき “ 1 ”、それ以外のとき “ 0 ” になります。

FV：無効演算が発生したとき “ 1 ”、それ以外のときは変化しません。

FX：精度異常が発生したとき “ 1 ”、それ以外のときは変化しません。]

注 FX、FV フラグは、例外処理許可ビット EX、EV が“1”の場合は変化しません。S、Z フラグは、例外処理が発生した場合は変化しません。

【記述例】

```
ROUND   R1, R2
ROUND   [R1], R2
```

RTE 例外からの復帰 RTE
ReTurn from Exception

【構文】

RTE

【オペレーション】

PC = *SP;

SP = SP + 4;

tmp = *SP;

SP = SP + 4;

PSW = tmp;

【機能】

- 例外が受け付けられたときに退避した PC と PSW を復帰し、例外処理ルーチンから戻ります。
- この命令は特権命令です。ユーザモードで実行すると特権命令例外が発生します。
- ユーザモードに移行する場合、PSW の U ビットは“1”になります。

【命令フォーマット】

構文	コードサイズ（バイト）
RTE	2

【フラグ変化】

フラグ	C	Z	S	O
変化	（注）	（注）	（注）	（注）

注意　スタック上の値になります。

【記述例】

```
RTE
```

RTFI 高速割り込みからの復帰 RTFI
ReTurn from Fast Interrupt

【構文】

RTFI

【オペレーション】

```
PSW = BPSW;
PC = BPC;
```

【機能】

- 高速割り込み要求を受け付けたときに退避した PC と PSW を、それぞれ BPC、BPSW から復帰し、高速割り込みハンドラから戻ります。
- この命令は特権命令です。ユーザモードで実行すると特権命令例外が発生します。
- ユーザモードに移行する場合、PSW の U ビットは“1”になります。
- 命令終了時の BPC、BPSW の値は不定になります。

【命令フォーマット】

構文	コードサイズ（バイト）
RTFI	2

【フラグ変化】

フラグ	C	Z	S	O
変化	（注）	（注）	（注）	（注）

注意 BPSW の値になります。

【記述例】

```
RTFI
```

RTS サブルーチンからの復帰 RTS
ReTurn from Subroutine

【構文】

RTS

【オペレーション】

PC = *SP;
SP = SP + 4;

【機能】

- サブルーチンから復帰します。

【命令フォーマット】

構文	コードサイズ（バイト）
RTS	1

【フラグ変化】

フラグ変化はありません。

【記述例】

```
RTS
```

RTSD スタックフレームの解放と サブルーチンからの復帰 RTSD

ReTurn from Subroutine and Deallocate stack frame

【構文】

(1)RTSD src

(2)RTSD src, dest-dest2

【オペレーション】*1

```
(1)SP = SP + src;
PC = *SP;
SP = SP + 4;

(2)signed char i;
SP = SP + ( src - ( register_num(dest2) - register_num(dest) +1 ) * 4 );
for ( i = register_num(dest); i <= register_num(dest2); i++ ) {
  tmp = *SP;
  SP = SP + 4;
  register(i) = tmp;
}
PC = *SP;
SP = SP + 4;
```

【機能】

- スタックフレームを解放後、サブルーチンから復帰します。
  - src がスタックフレーム（自動変数領域）のサイズになるように指定してください。
- スタックフレームの解放とレジスタの復帰を行った後、サブルーチンから復帰します。
  - src がスタックフレーム（自動変数領域とレジスタ退避領域）のサイズになるように指定してください。
- dest と dest2 で範囲指定したレジスタを一括してスタックから復帰します。
- 範囲は先頭レジスタ番号と最終レジスタ番号で指定します。ただし、（先頭レジスタのレジスタ番号≦最終レジスタのレジスタ番号）となっている必要があります。
- R0 を指定することはできません。
- 使用されるスタックポインタは、PSW の U ビットで示すスタックポインタになります。
- スタックから復帰する順序は R1 から復帰します。

【命令フォーマット】

構文	src	dest	dest2	コードサイズ（バイト）
(1)RTSD src	#UIMM:8（注）	–	–	2

構文	src	dest	dest2	コードサイズ（バイト）
(2)RTSD　src, dest-dest2	#UIMM:8 (注)	Rd (Rd=R1~R15)	Rd2 (Rd=R1~R15)	3

注　弊社の「RX ファミリ アセンブラ」では、即値は、4 の倍数を指定してください。UIMM:8 **には、**0 ～ 1020（255 × 4）**が指定できます。命令コードには、**1/4 **した値が埋め込まれます。**

【フラグ変化】

フラグ変化はありません。

【記述例】

```
RTSD    #4
RTSD    #16, R5-R7
```

SAT 32 ビット符号付き飽和処理 SAT
SATurate signed 32-bit data

【構文】

SAT dest

【オペレーション】

```
if ( O == 1 && S == 1 )
  dest = 7FFFFFFFh;
else if ( O == 1 && S == 0 )
  dest = 80000000h;
```

【機能】

- 32 ビット符号付きで飽和処理を行います。
- O フラグが “ 1 ” かつ S フラグが “ 1 ” のとき、演算結果が 7FFFFFFFh になり、その結果を dest に格納します。O フラグが “ 1 ” かつ S フラグが “ 0 ” のとき、演算結果が 80000000h になり、その結果を dest に格納します。それ以外のとき、dest は変化しません。

【命令フォーマット】

構文	処理サイズ	dest	コードサイズ（バイト）
SAT dest	L	Rd	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SAT     R1
```

SATR RMPA 命令用 64 ビット符号付 RORC き飽和処理

SATuRate signed 64-bit data for RMPA

【構文】

SATR

【オペレーション】

```
if ( O == 1 && S == 0 )
  R6:R5:R4 = 000000007FFFFFFFFFFFFFFFh;
else if ( O == 1 && S == 1 )
  R6:R5:R4 = FFFFFFFF8000000000000000h;
```

【機能】

- 64 ビット符号付きで飽和処理を行います。
- O フラグが “1” かつ S フラグが “0” のとき、演算結果が 000000007FFFFFFFFFFFFFFFh になり、その結果を R6:R5:R4 に格納します。O フラグが “1” かつ S フラグが “1” のとき、演算結果が FFFFFFFF8000000000000000h になり、その結果を R6:R5:R4 に格納します。それ以外のとき、R6:R5:R4 は変化しません。

【命令フォーマット】

構文	コードサイズ（バイト）
SATR	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SATR
```

SBB ボロー付き減算 SBB
SuBtract with Borrow

【構文】

SBB src, dest

【オペレーション】

dest = dest - src - !C;

【機能】

- dest から src と C フラグの反転（ボロー）を減算し、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
SBB src, dest	L	Rs	Rd	3
	L	[Rs].L	Rd	4
	L	dsp:8[Rs].L（注）	Rd	5
	L	dsp:16[Rs].L（注）	Rd	6

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、4 の倍数を指定してください。dsp:8 には、0 ～ 1020（255 × 4）が指定できます。dsp:16 には、0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：符号なし演算のオーバフローが発生しなかったとき “1”、それ以外のとき “0” になります。

Z：演算後の dest が 0 のとき “1”、それ以外のとき “0” になります。

S：演算後の dest の MSB が “1” のとき “1”、それ以外のとき “0” になります。

O：符号付き演算のオーバフローが発生したとき “1”、それ以外のとき “0” になります。

【記述例】

```
SBB     R1, R2
SBB     [R1], R2
```

SCCnd 条件設定 SCCnd
Store Condition Conditionally

【構文】

SCCnd.size dest

【オペレーション】

```
if ( Cnd )
  dest = 1;
else
  dest = 0;
```

【機能】

- Cnd で示す条件の真偽値を dest に設定します。真の場合は“1”を、偽の場合は“0”を設定します。
- SCCnd には次の種類があります。

BCnd	条件		式
SCGEU, SCC	C == 1	等しいまたは大きい／ C フラグが“1”	≦
SCEQ, SCZ	Z == 1	等しい／ Z フラグが“1”	=
SCGTU	C & ~Z == 1	大きい	<
SCPZ	S == 0	正またゼロ	0 ≦
SCGE	S ^ O == 0	等しい、または符号付きで大きい	≦
SCGT	(S ^ O) | Z == 0	符号付きで大きい	<
SCO	O == 1	O フラグが“1”	
SCLTU, SCNC	C == 0	小さい／ C フラグが“0”	>
SCNE, SCNZ	Z == 0	等しくない／ Z フラグが“0”	≠
SCLEU	C & ~Z == 0	等しいまたは小さい	≧
SCN	S == 1	負	0 >
SCLE	(S ^ O) | Z == 1	等しい、または符号付きで小さい	≧
SCLT	S ^ O == 1	符号付きで小さい	>
SCNO	O == 0	O フラグが“0”	

【命令フォーマット】

構文	size	処理サイズ	dest	コードサイズ（バイト）
SCCnd.size dest	L	L	Rd	3

構文	size	処理サイズ	dest	コードサイズ（バイト）
	B/W/L	size	[Rd]	3
	B/W/L	size	dsp:8[Rd]（注）	4
	B/W/L	size	dsp:16[Rd]（注）	5

注　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ指定子が“.W”のときは2の倍数、“.L”のときは4の倍数を指定してください。dsp:8には、サイズ指定子が“.W”のとき0～510（255 × 2）が、“.L”のとき0～1020（255 × 4）が指定できます。dsp:16には、サイズ指定子が“.W”のとき0～131070（65535 × 2）が、“.L”のとき0～262140（65535 × 4）が指定できます。命令コードには、1/2、1/4した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SCC.L    R2
SCNE.W   [R2]
```

SCMPU ストリング比較 SCMPU

String CoMPare Until not equal

【構文】

SCMPU

【オペレーション】

```
unsigned char *R2, *R1, tmp0, tmp1;
unsigned long R3;
while ( R3 != 0 ) {
   tmp0 = *R1++;
   tmp1 = *R2++;
   R3--;
   if ( tmp0 != tmp1 || tmp0 == '\0' ) {
      break;
   }
}
```

注 R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

【機能】

- R1 で示される比較元番地と R2 で示される比較先番地のデータを、比較の結果が不一致になるか、Null キャラクタ ‘ \0 ’ (=00h) が検出されるまで、R3 で指定されたバイト数を上限として、アドレスの加算方向にストリング比較を行います。
- 命令実行時は、R1 で示される比較元番地と R2 で示される比較先番地から、それぞれデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- 命令終了時の R1、R2 は不定になります。
- 命令実行中に割り込み要求があった場合は、演算を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	処理サイズ	コードサイズ（バイト）
SCMPU	B	2

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	–	–

条件

C：(*R1 - *R2）を符号なしで演算した結果、0 以上のとき“1”、それ以外のとき“0”になります。

Z：双方のストリングが一致していたとき“1”、それ以外のとき“0”になります。

【記述例】

```
SCMPU
```

SETPSW PSW のフラグ、ビットのセット SETPSW
SET flag of PSW

【構文】

SETPSW dest

【オペレーション】

dest = 1;

【機能】

- dest で指定された O、S、Z、C フラグ、もしくは U、I ビットを “1” にします。
- ユーザモードでは、PSW の U、I ビットへの書き込みは無視されます。スーパバイザモードでは、すべてのフラグとビットへの書き込みが行えます。

【命令フォーマット】

構文	dest	コードサイズ（バイト）
SETPSW dest	flag	2

【フラグ変化】

フラグ	C	Z	S	O
変化	（注）	（注）	（注）	（注）

注 指定されたフラグが ”1” になります。

【記述例】

```
SETPSW  C
SETPSW  Z
```

SHAR　　算術シフト　　SHAR
SHift Arithmetic Right

【構文】

(1)SHAR　src, dest

(2)SHAR　src, src2, dest

【オペレーション】

(1)dest = (signed long) dest >> (src & 31);

(2)dest = (signed long) src2 >> (src & 31);

【機能】

- dest を src で指定されたビット数分、算術右シフトし、その結果を dest に格納します。
  - LSB から溢れたビットは C フラグに転送します。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。
  - src がレジスタのとき、LSB 側 5 ビットのみ有効です。
- src2 を dest に転送後、dest を src で指定されたビット数分、算術右シフトし、その結果を dest に格納します。
  - LSB から溢れたビットは C フラグに転送します。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)SHAR　src, dest	L	#IMM:5	–	Rd	2
	L	Rs	–	Rd	3
(1)SHAR　src, src2, dest	L	#IMM:5	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：シフトアウトしたビットが“1”のとき“1”、それ以外のとき“0”になります。ただし、src が 0 のときは“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

O：“0”になります。

【記述例】

```
SHAR    #3, R2
SHAR    R1, R2
SHAR    #3, R1, R2
```

SHLL 論理 / 算術左シフト SHLL

SHift Logical and arithmetic Left

【構文】

(1)SHLL src, dest

(2)SHLL src, src2, dest

【オペレーション】

(1)dest = dest << (src & 31);

(2)dest = src2 << (src & 31);

【機能】

- dest を src で指定されたビット数分、論理左シフトし、その結果を dest に格納します。
  - MSB から溢れたビットは C フラグに転送します。
  - src がレジスタのとき、LSB 側 5 ビットのみ有効です。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。
- src2 を dest に転送後、dest を src で指定されたビット数分、論理左シフトし、その結果を dest に格納します。
  - MSB から溢れたビットは C フラグに転送します。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)SHLL src, dest	L	#IMM:5	–	Rd	2
	L	Rs	–	Rd	3
(1)SHLL src, src2, dest	L	#IMM:5	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：シフトアウトしたビットが“1”のとき“1”、それ以外のとき“0”になります。ただし、src が 0 のときは“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

O：演算結果の MSB とシフトアウトしたビットがすべて同じ値のとき（シフト中に符号が変化しなかったとき）“0”、それ以外のとき“1”になります。ただし、src が 0 のときは“0”になります。

【記述例】

```
SHLL    #3, R2
SHLL    R1, R2
SHLL    #3, R1, R2
```

SHLR　論理右シフト　SHLR
SHift Logical Right

【構文】

(1)SHLR　src, dest

(2)SHLR　src, src2, dest

【オペレーション】

(1)dest = (unsigned long) dest >> (src & 31);

(2)dest = (unsigned long) src2 >> (src & 31);

【機能】

- dest を src で指定されたビット数分、論理右シフトし、その結果を dest に格納します。
  - LSB から溢れたビットは C フラグに転送します。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。
  - src がレジスタのとき、LSB 側 5 ビットのみ有効です。
- src2 を dest に転送後、dest を src で指定されたビット数分、論理右シフトし、その結果を dest に格納します。
  - LSB から溢れたビットは C フラグに転送します。
  - src の値は符号なし整数です。src の範囲は、0 ≦ src ≦ 31 です。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)SHLR　src, dest	L	#IMM:5	–	Rd	2
	L	Rs	–	Rd	3
(1)SHLR　src, src2, dest	L	#IMM:5	Rs	Rd	3

【フラグ変化】

フラグ	C	Z	S	O
変化	–	○	○	○

条件

C：シフトアウトしたビットが“1”のとき“1”、それ以外のとき“0”になります。ただし、src が 0 のときは“0”になります。

Z：演算後の dest が 0 のとき“1”、それ以外のとき“0”になります。

S：演算後の dest の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
SHLR    #3, R2
SHLR    R1, R2
SHLR    #3, R1, R2
```

SMOVB 逆方向のストリング転送 SMOVB
String MOVe Backward

【構文】

SMOVB

【オペレーション】

```
unsigned char *R1, *R2;
unsigned long R3;
while ( R3 != 0 ) {
  *R1-- = *R2--;
  R3 = R3 - 1;
}
```

【機能】

- R3 で指定されたバイト数分、R2 で示される転送元番地から R1 で示される転送先番地へ、アドレス減算方向にストリング転送を行います。
- 命令実行時は、R2 で示される転送元番地からデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- R2 で示される転送元番地からデータプリフェッチされる範囲に R1 で示される転送先番地が含まれない条件で使用してください。
- 命令終了時の R1、R2 は、最後に転送したデータの次の番地を示します。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	処理サイズ	コードサイズ（バイト）
SMOVB	B	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SMOVB
```

SMOVF 順方向ストリング転送 SMOVF
Strings MOVe Forward

【構文】

SMOVF

【オペレーション】

```
unsigned char *R1, *R2;
unsigned long R3;
while ( R3 != 0 ) {
  *R1++ = *R2++;
  R3 = R3 - 1;
}
```

注意 R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

【機能】

- R3 で指定されたバイト数分、R2 で示される転送元番地から R1 で示される転送先番地へ、アドレス加算方向にストリング転送を行います。
- 命令実行時は、R2 で示される転送元番地からデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- R2 で示される転送元番地からデータプリフェッチされる範囲に R1 で示される転送先番地が含まれない条件で使用してください。
- 命令終了時の R1、R2 は、最後に転送したデータの次の番地を示します。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	処理サイズ	コードサイズ（バイト）
SMOVF	B	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SMOVF
```

SMOVU ストリング転送 SMOVU
Strings MOVe while Unequal to zero

【構文】

SMOVU

【オペレーション】

```
unsigned char *R1, *R2, tmp;
unsigned long R3;
while ( R3 != 0 ) {
   tmp = *R2++;
   *R1++ = tmp;
   R3--;
   if ( tmp == '\0' ) {
      break;
   }
}
```

注 R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

【機能】

- R2 で示される転送元番地から R1 で示される転送先番地へ Null キャラクタ‘ \0 ’(=00h) が検出されるまで、R3 で指定されたバイト数を上限として、アドレス加算方向にストリング転送を行います。転送は Null キャラクタ転送後に終了します。
- 命令実行時は、R2 で示される転送元番地からデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- R2 で示される転送元番地からデータプリフェッチされる範囲に R1 で示される転送先番地が含まれない条件で使用してください。
- 命令終了時の R1、R2 は、不定となります。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	処理サイズ	コードサイズ（バイト）
SMOVU	B	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SMOVU
```

SSTR ストリングストア Strings SToRe SSTR

【構文】

SSTR.size

【オペレーション】

```
unsigned { char | short | long } *R1, R2;
unsigned long R3;
while ( R3 != 0 ) {
  *R1++ = R2;
  R3 = R3 - 1;
}
```

注　R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

注　R1++：サイズ指定子（.size）が“ .B ”のとき 1、“ .W ”のとき 2、“ .L ”のとき 4 が加算されます。

注　R2：サイズ指定子（.size）が“ .B ”のとき R2 の LSB 側バイトデータ、“ .W ”のとき R2 の LSB 側ワードデータ、“ .L ”のとき R2 のロングワードデータがストアされます。

【機能】

- R3 で示される回数分、R2 の内容を R1 で示される転送先番地へ、アドレス加算方向にストリングストアを行います。
- 命令終了時の R1 は、最後に転送したデータの次の番地を示します。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	size	処理サイズ	コードサイズ（バイト）
SSTR.size	B/W/L	L	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
SSTR.W
```

STNZ 条件付き転送 STNZ
STore on Not Zero

【構文】

STNZ src, dest

【オペレーション】

```
if ( Z == 0 )
  dest = src;
```

【機能】

- Z フラグが“0”のとき、src を dest に転送します。“1”のとき dest は変化しません。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
STNZ src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7

【フラグ変化】

フラグ変化はありません。

【記述例】

```
STNZ    #1, R2
```

STZ 条件付き転送 STZ
STore on Zero

【構文】

STZ src, dest

【オペレーション】

```
if ( Z == 1 )
  dest = src;
```

【機能】

- Z フラグが“1”のとき、src を dest に転送します。“0”のとき、dest は変化しません。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
STZ src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7

【フラグ変化】

フラグ変化はありません。

【記述例】

```
STZ     #1, R2
```

SUB ボローなし減算 SUB
SUBtract

【構文】

(1)SUB src, dest

(2)SUB src, src2, dest

【オペレーション】

(1)destからsrcを減算し、その結果をdestに格納します。
(2)src2からsrcを減算し、その結果をdestに格納します。

【機能】

- (1)dest から src を減算し、その結果を dest に格納します。
- (2)src2 から src を減算し、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	src2	dest	コードサイズ（バイト）
(1)SUB src, dest	L	#UIMM:4	—	Rd	2
	L	Rs	—	Rd	2
	L	[Rs].memex	—	Rd	2 (memex == UB) 3 (memex != UB)
	L	dsp:8[Rs].memex（注）	—	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:16[Rs].memex（注）	—	Rd	4 (memex == UB) 5 (memex != UB)
(2)SUB src, src2, dest	L	Rs	Rs2	Rd	3

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“.W”または“.UW”のときは2の倍数、“.L”のときは4の倍数を指定してください。dsp:8には、サイズ拡張指定子が“.W”または“.UW”のとき0～510（255 × 2）が、“.L”のとき0～1020（255 × 4）が指定できます。dsp:16には、サイズ拡張指定子が“.W”または“.UW”のとき0～131070（65535 × 2）が、“.L”のとき0～262140（65535 × 4）が指定できます。命令コードには、1/2、1/4した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	○	○

条件

C：符号なし演算のオーバフローが発生しなかったとき“1”、それ以外のとき“0”になります。

Z：演算後の dest が 0 のとき “1”、それ以外のとき “0” になります。

S：演算後の dest の MSB が “1” のとき “1”、それ以外のとき “0” になります。

O：符号付き演算のオーバフローが発生したとき “1”、それ以外のとき “0” になります。

【記述例】

```
SUB     #15, R2
SUB     R1, R2
SUB     [R1], R2
SUB     1[R1].B, R2
SUB     R1, R2, R3
```

SUNTIL ストリングサーチ SUNTIL
Search UNTIL equal string

【構文】

SUNTIL.size

【オペレーション】

```
unsigned { char | short | long } *R1;
unsigned long R2, R3, tmp;
while ( R3 != 0 ) {
   tmp = ( unsigned long ) *R1++;
   R3--;
   if ( tmp == R2 ) {
      break;
   }
}
```

注 1. R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

注 2. R1++：サイズ指定子（.size）が“ .B ”のとき 1、“ .W ”のとき 2、“ .L ”のとき 4 が加算されます。

【機能】

- R1 で示される比較先番地からアドレスの加算方向に、R2 の内容と一致するデータが現れるまで、R3 で指定される回数を上限として検索を行います。サイズ指定子（.size）が“ .B ”または“ .W ”のときは、メモリのバイトデータまたはワードデータをロングワードデータにゼロ拡張し、R2 の内容と比較を行います。
- 命令実行時は、R1 で示される比較先番地からデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- フラグは「*R1-R2」の演算結果にしたがって変化します。
- 命令終了時の R1 は、一致したデータの次の番地を示します。すべて一致しなかったときは、最後に転送したデータの次の番地を示します。
- 命令終了後の R3 は、「初期値 - 比較回数」となります。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	size	処理サイズ	コードサイズ（バイト）
SUNTIL.size	B/W/L	L	2

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	−	−

条件

C：符号なし比較の結果、0以上のとき“1”、それ以外のとき“0”になります。

Z：一致したとき“1”、それ以外のとき“0”になります。

【記述例】

```
SUNTIL.W
```

SWHILE ストリングサーチ SWHILE
Search WHILE unequal string

【構文】

SWHILE.size

【オペレーション】

```
unsigned { char | short | long } *R1;
unsigned long R2, R3, tmp;
while ( R3 != 0 ) {
   tmp = ( unsigned long ) *R1++;
   R3--;
   if ( tmp != R2 ) {
      break;
   }
}
```

注 1. R3 に 0 を設定して実行したとき、本命令は無視され、レジスタおよびフラグは変化しません。

注 2. R1++：サイズ指定子（.size）が“ .B ”のとき 1、“ .W ”のとき 2、“ .L ”のとき 4 が加算されます。

【機能】

- R1 で示される比較先番地からアドレスの加算方向に、R2 の内容と一致しないデータが現れるまで、R3 で指定される回数を上限として検索を行います。サイズ指定子（.size）が“ .B ”または“ .W ”のときは、メモリのバイトデータまたはワードデータをロングワードデータにゼロ拡張し、R2 の内容と比較を行います。
- 命令実行時は、R1 で示される比較先番地からデータプリフェッチが行われる場合があります。ただし、R3 で指定された範囲を超えるデータプリフェッチは行いません。プリフェッチされるデータサイズについては、各製品のハードウェアマニュアルを参照してください。
- フラグは「*R1-R2」の演算結果にしたがって変化します。
- 命令終了時の R1 は、一致しなかったデータの次の番地を示します。すべて一致したときは、最後に転送したデータの次の番地を示します。
- 命令終了後の R3 は、「初期値 - 比較回数」となります。
- 命令実行中に割り込み要求があった場合は、命令途中で転送を中断して割り込みを受け付けます。割り込みルーチンからの復帰後、中断されていた処理を継続して実行します。本命令を使用する際には、割り込み時、R1、R2、R3 と PSW を退避／復帰してください。

【命令フォーマット】

構文	size	処理サイズ	コードサイズ（バイト）
SWHILE.size	B/W/L	L	2

【フラグ変化】

フラグ	C	Z	S	O
変化	○	○	−	−

条件

C：符号なし比較の結果、0以上のとき“1”、それ以外のとき“0”になります。

Z：すべて一致したとき“1”、それ以外のとき“0”になります。

【記述例】

```
SWHILE.W
```

TST　テスト　TST
TeST

【構文】

TST　src, src2

【オペレーション】

src2 & src;

【機能】

- src2 と src の論理積をとった結果にしたがって、PSW の各フラグが変化します。

【命令フォーマット】

構文	処理サイズ	src	src2	コードサイズ（バイト）
TST　src, src2	L	#SIMM:8	Rs	4
	L	#SIMM:16	Rs	5
	L	#SIMM:24	Rs	6
	L	#IMM:32	Rs	7
	L	Rs	Rs2	3
		[Rs].memex	Rs2	3 (memex == UB) 4 (memex != UB)
		dsp:8[Rs].memex（注）	Rs2	4 (memex == UB) 5 (memex != UB)
		dsp:16[Rs].memex（注）	Rs2	5 (memex == UB) 6 (memex != UB)

注　弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が“.W”または“.UW”のときは 2 の倍数、“.L”のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 510（255 × 2）が、“.L”のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が“.W”または“.UW”のとき 0 ～ 131070（65535 × 2）が、“.L”のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	–	○	○	–

条件

Z：演算結果が 0 のとき“1”、それ以外のとき“0”になります。

S：演算結果の MSB が“1”のとき“1”、それ以外のとき“0”になります。

【記述例】

```
TST     #7, R2
TST     R1, R2
TST     [R1], R2
TST     1[R1].UB, R2
```

WAIT ウェイト WAIT

【構文】

WAIT

【オペレーション】

【機能】

- プログラムの実行を停止します。ノンマスカブル割り込み、割り込み、またはリセットが発生するとプログラムの実行を開始します。
- この命令は特権命令です。ユーザモードで実行すると特権命令例外が発生します。
- PSW の I ビットが“1”になります。
- 割り込み発生時に退避される PC は、WAIT 命令の次のアドレスになります。

注 プログラムの実行を停止した状態での低消費電力状態については、各製品のハードウェアマニュアルを参照してください。

【命令フォーマット】

構文	コードサイズ（バイト）
WAIT	2

【フラグ変化】

フラグ変化はありません。

【記述例】

```
WAIT
```

XCHG 交換 XCHG
eXCHanGe

【構文】

XCHG src, dest

【オペレーション】

tmp = src;

src = dest;

dest = tmp;

【機能】

- 以下のとおり、src と dest の内容を交換します。

src	dest	機能
レジスタ	レジスタ	レジスタ（src）のデータとレジスタ（dest）のデータを交換します。
メモリ	レジスタ	メモリのデータとレジスタのデータを交換します。サイズ拡張指定子が .B および .UB のときは、レジスタの LSB 側のバイトデータとメモリのデータを交換します。サイズ拡張指定子が .W および .UW のときは、レジスタの LSB 側のワードデータとメモリのデータを交換します。サイズ拡張指定子が .L 以外のときは、指定した拡張方法でメモリのデータをロングワードデータに拡張し、レジスタに転送します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
XCHG src, dest	L	Rs	Rd	3
	L	[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
	L	dsp:8[Rs].memex（注）	Rd	4 (memex == UB) 5 (memex != UB)
	L	dsp:16[Rs].memex（注）	Rd	5 (memex == UB) 6 (memex != UB)

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ変化はありません。

【記述例】

```
XCHG    R1, R2
XCHG    [R1].W, R2
```

XOR 排他的論理和 XOR
eXclusive OR logical

【構文】

XOR src, dest

【オペレーション】

dest = dest ^ src;

【機能】

- dest と src の排他的論理和をとり、その結果を dest に格納します。

【命令フォーマット】

構文	処理サイズ	src	dest	コードサイズ（バイト）
XOR src, dest	L	#SIMM:8	Rd	4
	L	#SIMM:16	Rd	5
	L	#SIMM:24	Rd	6
	L	#IMM:32	Rd	7
	L	Rs	Rd	4
		[Rs].memex	Rd	3 (memex == UB) 4 (memex != UB)
		dsp:8[Rs].memex（注）	Rd	4 (memex == UB) 5 (memex != UB)
		dsp:16[Rs].memex（注）	Rd	5 (memex == UB) 6 (memex != UB)

注 弊社の「RX ファミリ アセンブラ」では、ディスプレースメントの値（dsp:8、dsp:16）は、サイズ拡張指定子が “ .W ” または “ .UW ” のときは 2 の倍数、“ .L ” のときは 4 の倍数を指定してください。dsp:8 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 510（255 × 2）が、“ .L ” のとき 0 ～ 1020（255 × 4）が指定できます。dsp:16 には、サイズ拡張指定子が “ .W ” または “ .UW ” のとき 0 ～ 131070（65535 × 2）が、“ .L ” のとき 0 ～ 262140（65535 × 4）が指定できます。命令コードには、1/2、1/4 した値が埋め込まれます。

【フラグ変化】

フラグ	C	Z	S	O
変化	−	○	○	−

条件

Z：演算後の dest が 0 のとき “ 1 ”、それ以外のとき “ 0 ” になります。

S：演算後の dest の MSB が “ 1 ” のとき “ 1 ”、それ以外のとき “ 0 ” になります。

【記述例】

```
XOR     #8, R1
XOR     R1, R2
XOR     [R1], R2
XOR     16[R1].L, R2
```

第 5 章　リンク・ディレクティブ仕様

この章では、リンクに必要となる項目や、リンクの設定方法について説明します。

5. 1　配置方法

セクションの配置については、次の項目を参照してください。

RX ビルド編 / B.1.3 オプション / (3) 最適化リンケージエディタ（rlink）・オプション / セクションオプション / -start

5. 2　セクションの種類

セクションの種類については、「3. 5　セクション名一覧」を参照してください。

第 6 章　関数仕様

この章では，CCRX が提供するライブラリ関数について説明します。

6. 1　提供ライブラリ

CCRX で提供しているライブラリは、C 標準ライブラリ、C99 標準ライブラリ、および EC++ ライブラリです。

6. 1. 1　ライブラリ関数の説明で使用する用語

(1) ストリーム入出力

データの入出力において、1 文字ごとの入出力関数の呼び出しのたびに入出力装置を駆動したり、OS の機能を呼び出していたのでは、効率が悪くなります。そこで、通常はバッファと呼ばれる記憶域を用意しておき、バッファ内のデータに対して一括して入出力を行います。

一方、プログラムの側から見ると、1 文字ごとに入出力関数を呼び出せた方が便利です。

ライブラリ関数では、バッファの管理を自動的に行うことにより、プログラム内でバッファの状態を意識することなしに、1 文字単位の入出力を効率よく行うことができます。

このように、データの入出力を効率よく実現するために詳細な手段を意識せず、入出力をひとつのデータの流れ (ストリーム) と考えてプログラムを作ることのできる機能をストリーム入出力といいます。

(2) FILE 構造体およびファイルポインタ

ストリーム入出力に必要なバッファやその他の情報は、一つの構造体の中に記憶されており、標準インクルードファイル <stdio.h> の中で FILE という名前で定義されています。

ストリーム入出力においては、ファイルはすべて FILE 構造体のデータ構造を持つものとして扱います。このようなファイルをストリームファイルと呼びます。このファイル構造体へのポインタをファイルポインタと呼び、入出力ファイルを指定するために用います。

ファイルポインタは、

```
FILE *fp;
```

と定義します。

fopen 関数等でファイルをオープンすると、ファイルポインタが得られますが、オープン処理が失敗すると NULL が返ってきます。NULL ポインタを、他のストリーム入出力関数に指定すると、その関数は異常終了しますので、注意が必要です。ファイルをオープンした時は、必ずファイルポインタの値をチェックするようにしてください。

(3) 関数とマクロ

ライブラリ関数の実現方法としては、関数とマクロの二通りがあります。

関数は、通常のユーザ作成の関数と同じインタフェースを持ち、リンク時に取り込みます。

マクロは、その関数に関連した標準インクルードファイルの中で #define 文を用いて定義されています。

マクロについては、以下の点に注意する必要があります。

- マクロは、プリプロセッサによって自動的に展開されてしまうので、ユーザが同じ名前の関数を宣言してもマクロを無効にすることはできません。
- マクロのパラメータとして副作用のある式 (代入式、インクリメント、デクリメント) を指定した時、その効果は保証しません。

例：

パラメータの絶対値を求める MACRO を以下のようにマクロ定義します。

```
#define MACRO(a) ((a)>=0?(a):-(a))
```

と定義されている時、

```
X=MACRO(a++)
```

がプログラム内にあると、

```
X=((a++)>=0?(a++):-(a++))
```

と展開され、a は 2 回インクリメントされることになり、また結果の値も最初の a の値の絶対値とは異なります。

(4) EOF

getc 関数、getchar 関数、fgetc 関数等のファイルからデータを入力する関数において、ファイル終了 (End Of File) 時に返される値です。EOF は、標準インクルードファイル <stdio.h> の中で定義されています。

(5) NULL

ポインタが何も指していない時の値です。NULL は、標準インクルードファイル <stddef.h> の中で定義されています。

(6) ヌル文字

C/C++ 言語における文字列の終わりは、文字 "\0" によって示されることになっています。
ライブラリ関数における文字列のパラメータも、すべてこの約束に従っていなければなりません。
この文字列の終わりを示す文字 "\0" を以下ヌル文字と呼びます。

(7) リターンコード

ライブラリ関数の中には、リターン値によって、指定された処理が成功したか、失敗したか等の結果を判断するものがあります。
このような場合のリターン値を特にリターンコードと呼びます。

(8) テキストファイルとバイナリファイル

多くのシステムでは、データを格納するために特殊なファイル形式を持っています。
これをサポートするために、ライブラリ関数にはテキストファイルとバイナリファイルの 2 種類のファイル形式があります。

- テキストファイル

  テキストファイルは、通常のテキストを格納するためのファイルで、行の集まりとして構成されています。テキストファイルの入力の時、行の区切りとして改行文字 ("\n") が入力されます。また、出力の時、改行文字を出力することにより、現在の行の出力を終了します。テキストファイルは、処理系ごとの標

準的なテキストを格納するファイルの入出力を行うためのファイルです。テキストファイルでは、ライブラリ関数で入出力する文字は必ずしもファイル内の物理的なデータの並びと対応していません。

- バイナリファイル

バイナリファイルは、バイトデータの列として構成されているファイルです。ライブラリ関数で入出力するデータは、ファイル内の物理的なデータの並びと対応しています。

(9) 標準入出力ファイル

入出力のライブラリ関数で、ファイルのオープン等の準備を行わずに標準的に使用できるファイルを標準入出力ファイルといいます。標準入出力ファイルには、標準入力ファイル (stdin)、標準出力ファイル (stdout)、標準エラー出力ファイル (stderr) があります。

- 標準入力ファイル (stdin)

プログラムへの入力となる標準的なファイルです。

- 標準出力ファイル (stdout)

プログラムからの出力となる標準的なファイルです。

- 標準エラー出力ファイル (stderr)

プログラムからのエラーメッセージ等の出力を行うための標準的なファイルです。

(10) 浮動小数点型

浮動小数点型は、実数を近似して表現したものです。C/C++ 言語のソースプログラム上では浮動小数点型を 10 進数で表現していますが、計算機の内部では通常 2 進数で表現されます。

2 進数の場合の浮動小数点型の表現は次のようになります。

$2^n \times m$ (n：整数、m：2 進小数)

ここで n を浮動小数点型の指数部、m を仮数部といいます。浮動小数点型を一定のデータサイズで表現するために、n と m のビット数は通常固定されています。

以下、浮動小数点型に関する用語を説明します。

- 基数

浮動小数点型が何進数で表現されているかを示す整数値です。通常、基数は 2 です。

- 丸め

浮動小数点型よりも精度の高い演算の途中結果を浮動小数点型に格納する場合に丸めが行われます。丸めには、切り上げ、切り捨て、四捨五入 (2 進小数の場合は、0 捨 1 入となります) があります。

- 正規化

浮動小数点型を、$2^n \times m$ の形式で表現する場合、同一の数値を表す異なる表現が可能です。

例：

$2^5 \times 1.0_{(2)}$ ($_{(2)}$ は 2 進数を示します)

$2^6 \times 0.1_{(2)}$

どちらも同じ値です。

通常は、有効桁数を確保するために、先頭の桁が 0 でないような表現を用います。これを正規化された浮動小数点型といい、浮動小数点型をこのような表現に変換する操作を正規化といいます。

- ガードビット

浮動小数点型の演算の途中結果を保持する場合、通常は、丸めを行うために実際の浮動小数点型よりも 1 ビット多いデータを用意します。しかし、これだけでは桁落ち等が生じた時に正確な結果を求めるこ

とができません。このために、もう 1 ビット設けて演算の途中結果を保持する手法があります。このビットをガードビットといいます。

(11) ファイルアクセスモード

ファイルをオープンする時にどのような処理をファイルに行うかを示す文字列です。文字列の種類には表 6.1 に示す 12 種類があります。

表 6―1 ファイルアクセスモードの種類

	アクセスモード	意味
1	“r”	テキストファイルを読み込み用にオープンします。
2	“w”	テキストファイルを書き出し用にオープンします。
3	“a”	テキストファイルを追加用にオープンします。
4	“rb”	バイナリファイルを読み込み用にオープンします。
5	“wb”	バイナリファイルを書き出し用にオープンします。
6	“ab”	バイナリファイルを追加用にオープンします。
7	“r+”	テキストファイルを読み込み用でかつ更新用にオープンします。
8	“w+”	テキストファイルを書き出し用でかつ更新用にオープンします。
9	“a+”	テキストファイルを追加用でかつ更新用にオープンします。
10	“r+b”	バイナリファイルを読み込み用でかつ更新用にオープンします。
11	“w+b”	バイナリファイルを書き出し用でかつ更新用にオープンします。
12	“a+b”	バイナリファイルを追加用でかつ更新用にオープンします。

(12) 処理系定義

コンパイラが異なることによって定義が異なるという意味です。

(13) エラー指示子、ファイル終了指示子

ストリームファイルごとに、ファイルの入出力の際にエラーが生じたかどうかを示すエラー指示子と、入力ファイルが終了したかどうかを示すファイル終了指示子というデータを保持しています。

これらのデータは、それぞれ ferror 関数、feof 関数によって参照することができます。

ストリームファイルを扱う関数のうち、そのリターン値だけからではエラーの発生やファイルの終了の情報が得られないものがあります。エラー指示子とファイル終了指示子は、このような関数の実行後にファイルの状態を調べるために使用することができます。

(14) 位置指示子

ディスク上のファイル等、ファイル内の任意の位置からの読み書きができるストリームファイルは、現在読み書きしているファイル内の位置を示すデータを保持しています。これを位置指示子といいます。端末装置等、ファイル内の読み書きの位置を変更できないストリームファイルでは、位置指示子は使用しません。

6. 1. 2 ライブラリ使用時の注意事項

ライブラリの中で定義されているマクロの内容は、コンパイラごとに異なります。

ライブラリを使用する場合、これらのマクロの内容を再定義した場合、動作は保証しません。

ライブラリは、すべての場合についてエラーを検出しているわけではありません。以降の説明に示す以外の形式でライブラリ関数を呼び出した場合、動作は保証しません。

6.2 ヘッダ・ファイル

RX でライブラリを使用するときに必要となるヘッダ・ファイルの一覧は次のとおりです。なお，各ファイルにはマクロ定義，関数宣言が記述されています。

表 6―2 ライブラリの種類と対応する標準インクルードファイル

	ライブラリの種類	内容	標準インクルードファイル
1	プログラム診断用ライブラリ	プログラムの診断情報の出力を行うライブラリです。	<assert.h>
2	文字操作用ライブラリ	文字の操作およびチェックを行うライブラリです。	<ctype.h>
3	数値計算用ライブラリ	三角関数等の数値計算を行うライブラリです。	<math.h> <mathf.h>
4	プログラムの制御移動用ライブラリ	関数間の制御の移動をサポートするライブラリです。	<setjmp.h>
5	可変個の実引数アクセス用ライブラリ	可変個の実引数を持つ関数に対し、その実引数へのアクセスをサポートするライブラリです。	<stdarg.h>
6	入出力用ライブラリ	入出力操作を行うライブラリです。	<stdio.h>
7	標準処理用ライブラリ	記憶域管理等の C プログラムでの標準的処理を行うライブラリです。	<stdlib.h>
8	文字列操作用ライブラリ	文字列の比較、複写等を行うライブラリです。	<string.h>
9	複素数計算ライブラリ	複素数の計算を行うライブラリです。	<complex.h>
10	浮動小数点環境ライブラリ	浮動小数点環境のライブラリです。	<fenv.h>
11	整数型の書式変換	最大幅の整数の操作、変換を行うライブラリです。	<inttypes.h>
12	多バイト文字、ワイド文字ライブラリ	多バイト文字の操作を行うライブラリです。	<wchar.h> <wctype.h>

また、以上の標準インクルードファイルの他にプログラムの作成作業の効率向上を図るため表 6.3 に示すマクロ名の定義だけからなる標準インクルードファイルがあります。

表 6―3 マクロ名定義からなる標準インクルードファイル

	標準インクルードファイル	内容
1	<stddef.h>	各標準インクルードファイルで共通に使用するマクロ名を定義します。
2	<limits.h>	コンパイラの内部処理に関する各種制御値を定義します。
3	<errno.h>	ライブラリ関数においてエラーが発生した時に errno に設定する値を定義します。
4	<float.h>	浮動小数点型の限界に関する各種制御値を定義します。
5	<iso646.h>	代替つづりのマクロ名を定義します。
6	<stdbool.h>	論理型、および論理値に関するマクロを定義します。
7	<stdint.h>	指定した幅の整数型を宣言してマクロを定義します。
8	<tgmath.h>	型総称マクロを定義します。

6.3 リエントラント性

標準ライブラリ構築ツールで reent オプションを指定して作成したライブラリは、rand、srand 関数を除いてすべてリエントラントに実行できます。

reent オプションを指定していない場合について、表 6.4 にリエントラントライブラリー覧を示します。表中、△で示した関数は、errno 変数を設定しますので、プログラム中で errno を参照していなければリエントラントに実行できます。

リエントラント欄 ○：リエントラント ×：ノンリエントラント △：errno を設定

表 6―4 リエントラントライブラリー覧

標準インクルードファイル	関数名	リエントラント	標準インクルードファイル	関数名	リエントラント
stddef.h	offsetof	○	math.h	frexp	△
assert.h	assert	×		ldexp	△
ctype.h	isalnum	○		log	△
	isalpha	○		log10	△
	iscntrl	○		modf	△
	isdigit	○		pow	△
	isgraph	○		sqrt	△
	islower	○		ceil	△
	isprint	○		fabs	△
	ispunct	○		floor	△
	isspace	○		fmod	△
	isupper	○	mathf.h	acosf	△
	isxdigit	○		asinf	△
	tolower	○		atanf	△
	toupper	○		atan2f	△
math.h	acos	△		cosf	△
	asin	△		sinf	△
	atan	△		tanf	△
	atan2	△		coshf	△
	cos	△		sinhf	△
	sin	△		tanhf	△
	tan	△		expf	△
	cosh	△		frexpf	△
	sinh	△		ldexpf	△
	tanh	△		logf	△
	exp	△		log10f	△

標準インクルードファイル	関数名	リエントラント
mathf.h	modff	△
	powf	△
	sqrtf	△
	ceilf	△
	fabsf	△
	floorf	△
	fmodf	△
setjmp.h	setjmp	○
	longjmp	○
stdarg.h	va_start	○
	va_arg	○
	va_end	○
stdio.h	fclose	×
	fflush	×
	fopen	×
	freopen	×
	setbuf	×
	setvbuf	×
	fprintf	×
	fscanf	×
	printf	×
	scanf	×
	sprintf	△
	sscanf	△
	vfprintf	×
	vprintf	×
	vsprintf	△
	fgetc	×
	fgets	×
	fputc	×
	fputs	×
	getc	×
	getchar	×
stdio.h	gets	×
	putc	×
	putchar	×
	puts	×
	ungetc	×
	fread	×
	fwirte	×
	fseek	×
	ftell	×
	rewind	×
	clearerr	×
	feof	×
	ferror	×
	perror	×
stdlib.h	atof	△
	atoi	△
	atol	△
	atoll	△
	strtod	△
	strtol	△
	strtoul	△
	strtoll	△
	strtoull	△
	rand	×
	srand	×
	calloc	×
	free	×
	malloc	×
	realloc	×
	bsearch	○
	qsort	○
	abs	○
	div	△

標準インクルードファイル	関数名	リエントラント		標準インクルードファイル	関数名	リエントラント
string.h	labs	○		string.h	memchr	○
	llabs	○			strchr	○
	ldiv	△			strcspn	○
	lldiv	△			strpbrk	○
	memcpy	○			strrchr	○
	strcpy	○			strspn	○
	strncpy	○			strstr	○
	strcat	○			strtok	×
	strncat	○			memset	○
	memcmp	○			strerror	○
	strcmp	○			strlen	○
	strncmp	○			memmove	○

6.4 ライブラリ関数

この節では、ライブラリ関数について説明します。

6.4.1 <stddef.h>

標準インクルードファイルの中で共通に使用されるマクロ名を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
型（マクロ）	ptrdiff_t	2 つのポインタを減算した結果の型です。
	size_t	sizeof 演算子による演算結果の型です。
定数（マクロ）	NULL	ポインタが何も指していない時の値です。 これは、0 と等値演算子 (==) による比較結果が真になるような値です。
変数（マクロ）	errno	ライブラリ関数の処理中にエラーが発生した場合、そのライブラリごとに定義されたエラーコードがこの errno に設定されます。 ライブラリ関数を呼び出す前に errno に 0 を設定しておき、ライブラリ関数の処理終了後に errno に設定されているコードを調べることによってライブラリ関数の処理中に発生したエラーをチェックすることができます。
関数（マクロ）	offsetof	構造体メンバの構造体先頭からのオフセット値をバイト単位で求めます。
型（マクロ）	wchar_t	拡張文字を表す型です。

処理系定義仕様

	項目	
1	マクロ NULL の値	0（ただし、void 型へのポインタ型）
2	マクロ ptrdiff_t に適合する型	long 型
3	wchar_t に適合する型	short 型

6. 4. 2 <assert.h>

プログラム中に診断機能を付け加えます。

種別	定義名	説明
関数（マクロ）	assert	プログラム中に診断機能を付け加えます。

<assert.h> で定義される診断機能を無効にするためには、<assert.h> を取り込む前に NDEBUG というマクロ名を #define 文で定義してください (#define NDEBUG)。

注 assert というマクロ名に対して #undef 文を使用すると、それ以降の assert の呼び出しの効果は保証しません。

assert

プログラム中に診断機能を付け加えます。

[指定形式]

#include <assert.h>

void assert(long expression) ;

[引数]

expression 評価する式

[戻り値]

ー

[備考]

assert マクロは、expression が真の時は値を返さずに処理を終了します。expression が偽の時は、診断情報をコンパイラによって定義された書式で標準エラーファイルに出力し、その後 abort 関数を呼び出します。

診断情報の中には、パラメータのプログラムテキスト、ソースファイル名、ソース行番号が含まれています。

処理系定義仕様assert(expression) において、expression が偽の時メッセージを出力します。

なお、コンパイル時の lang オプションにより表示は変化します。

(1) -lang=c99 がないとき (C(C89)、C++、EC++ 言語の場合):

ASSERTION FAILED:Δ 式 ΔFILEΔ< ファイル名 >,LINEΔ< 行番号 >

(2) -lang=c99 があるとき (C(C99) 言語の場合):

ASSERTION FAILED:Δ 式 ΔFILEΔ< ファイル名 >,LINEΔ< 行番号 >ΔFUNCNAMEΔ< 関数名 >

6. 4. 3 <ctype.h>

文字に対して、その種類の判定や変換を行います。

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

上記の関数において、入力パラメータの値が unsigned char 型で表現できる範囲に含まれず、なおかつ EOF でない場合、その関数の動作は保証しません。

文字の種類の一覧を表 6.5 に示します。

表 6―5 文字の種類

	文字の種類	内容
1	英大文字	以下の 26 文字のいずれかの文字です。 'A'、'B'、'C'、'D'、'E'、'F'、'G'、'H'、'I'、'J'、'K'、'L'、'M'、 'N'、'O'、'P'、'Q'、'R'、'S'、'T'、'U'、'V'、'W'、'X'、'Y'、'Z'
2	英小文字	以下の 26 文字のいずれかの文字です。 'a'、'b'、'c'、'd'、'e'、'f'、'g'、'h'、'i'、'j'、'k'、'l'、'm'、 'n'、'o'、'p'、'q'、'r'、's'、't'、'u'、'v'、'w'、'x'、'y'、'z'
3	英字	英大文字と英小文字のいずれかの文字です。
4	10 進数字	以下の 10 文字のいずれかの文字です。 '0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'
5	印字文字	空白 (' ') を含む、ディスプレイ上に表示される文字のことです。 ASCII コードの 0x20 ～ 0x7E に対応します。
6	制御文字	印字文字以外の文字のことです。
7	空白類文字	以下の 6 文字のいずれかの文字です。 空白 (' ')、書式送り ('\f')、改行 ('\n')、復帰 ('\r')、水平タブ ('\t')、垂直タブ ('\v')
8	16 進数字	以下の 22 文字のいずれかの文字です。 '0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'、 'A'、'B'、'C'、'D'、'E'、'F'、'a'、'b'、'c'、'd'、'e'、'f'
9	特殊文字	空白 (' ')、英字、および 10 進数字を除く任意の印字文字のことです。
10	ブランク文字	以下の 2 文字のいずれかの文字です。 空白 (' ')、水平タブ ("\t')

処理系定義仕様

	項目	コンパイラの仕様
1	isalnum 関数、isalpha 関数、iscntrl 関数、islower 関数、isprint 関数、isupper 関数で判定される文字集合	unsigned char 型で表現できる文字集合です。判定の結果、真になる文字を表 6.6 に示します。

表 6―6 真となる文字の集合

	関数名	真となる文字
1	isalnum	'0' ～ '9', 'A' ～ 'Z', 'a' ～ 'z'
2	isalpha	'A' ～ 'Z', 'a' ～ 'z'
3	iscntrl	'\x00' ～ '\x1f', '\x7f'
4	islower	'a' ～ 'z'
5	isprint	'\x20' ～ '\x7E'
6	isupper	'A' ～ 'Z'

isalnum

文字が英字または 10 進数字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long isalnum(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が英字または 10 進数字の時: 0 以外

文字 c が英字または 10 進数字以外の時: 0

[備考]

―

isalpha

文字が英字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long isalpha(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が英字の時: 0 以外

文字 c が英字以外の時: 0

[備考]

iscntrl

文字が制御文字であるかどうか判定します。

[**指定形式**]

#include <ctype.h>

long iscntrl(long c);

[**引数**]

c **判定する文字**

[**戻り値**]

文字 c **が制御文字の時:** 0 **以外**

文字 c **が制御文字以外の時:** 0

[**備考**]

−

isdigit

文字が 10 **進数字であるかどうか判定します。**

[**指定形式**]

#include <ctype.h>

long isdigit(long c);

[**引数**]

c **判定する文字**

[**戻り値**]

文字 c **が** 10 **進数字の時:** 0 **以外**

文字 c **が** 10 **進数字以外の時:** 0

[**備考**]

−

isgraph

文字が空白 (' ') **を除く任意の印字文字かどうかを判定します。**

[**指定形式**]

#include <ctype.h>

long isgraph(long c);

[**引数**]

c **判定する文字**

[**戻り値**]

文字 c **が空白を除く印字文字の時:** 0 **以外**

文字 c **が空白を除く印字文字以外の時:** 0

[**備考**]

−

islower

文字が英小文字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long islower(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が英小文字の時：0 以外

文字 c が英小文字以外の時：0

[備考]

－

isprint

文字が空白文字 (' ') を含む印字文字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long isprint(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が空白文字を含む印字文字の時：0 以外

文字 c が空白文字を含む印字文字以外の時：0

[備考]

－

ispunct

文字が特殊文字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long ispunct(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が特殊文字の時：0 以外

文字 c が特殊文字以外の時：0

[備考]

－

isspace

文字が空白類文字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long isspace(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が空白類文字の時：0 以外

文字 c が空白類文字以外の時：0

[備考]

－

isupper

文字が英大文字であるかどうか判定します。

[指定形式]

#include <ctype.h>

long isupper(long c);

[引数]

c 判定する文字

[戻り値]

文字 c が英大文字の時：0 以外

文字 c が英大文字以外の時：0

[備考]

－

isxdigit

文字が 16 進数字かどうか判定します。

[指定形式]

#include <ctype.h>

long isxdigit(long c)

[引数]

c 判定する文字

[戻り値]

文字 c が 16 進数字の時：0 以外

文字 c が 16 進数字以外の時：0

[備考]

－

tolower

英大文字を対応する英小文字に変換します。

[指定形式]

#include <ctype.h>

long tolower(long c);

[引数]

c 変換する文字

[戻り値]

文字 c が英大文字の時：文字 c に対応する英小文字

文字 c が英大文字以外の時：文字 c

[**備考**]

－

toupper

英小文字を対応する英大文字に変換します。

[**指定形式**]

#include <ctype.h>

long toupper(long c);

[**引数**]

c **変換する文字**

[**戻り値**]

文字 c が英小文字の時：文字 c に対応する英大文字

文字 c が英小文字以外の時：文字 c

[**備考**]

－

isblank

空白文字またはタブ文字か判定します。

[**指定形式**]

#include <ctype.h>

long isblank(long c);

[**引数**]

c **判定する文字**

[**戻り値**]

文字 c が空白文字またはタブ文字の時：0 以外

文字 c が空白文字でもタブ文字でもない時：0

[**備考**]

－

6. 4. 4 <float.h>

浮動小数点型の内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数（マクロ）	FLT_RADIX	2	指数部表現における基数です。
	FLT_ROUNDS	1	加算演算結果を丸めるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・演算結果を丸める場合：正の値 ・演算結果を切り捨てる場合：0 ・特に規定しない場合：-1 丸め、切り捨ての方法は、処理系定義です。
	FLT_GUARD	1	乗算演算結果においてガードビットを用いるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・ガードビットを用いる場合：1 ・ガードビットを用いない場合：0
	FLT_NORMALIZE	1	浮動小数点値を正規化するかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・正規化する場合：1 ・正規化しない場合：0
	FLT_MAX	3.4028235677973364e+38F	float 型が浮動小数点値として表現できる最大値です。
	DBL_MAX	1.7976931348623158e+308	double 型が浮動小数点値として表現できる最大値です。
	LDBL_MAX	1.7976931348623158e+308	long double 型が浮動小数点値として表現できる最大値です。
	FLT_MAX_EXP	127	float 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	DBL_MAX_EXP	1023	double 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	LDBL_MAX_EXP	1023	long double 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	FLT_MAX_10_EXP	38	float 型が浮動小数点値として表現できる 10 のべき乗の最大値です。

種別	定義名	定義値	説明
定数（マクロ）	DBL_MAX_10_EXP	308	double 型が浮動小数点値として表現できる 10 のべき乗の最大値です。
	LDBL_MAX_10_EXP	308	long double 型が浮動小数点値として表現できる 10 のべき乗の最大値です。
	FLT_MIN	1.175494351e-38F	float 型が浮動小数点値として表現できる正の値での最小値です。
	DBL_MIN	2.2250738585072014e-308	double 型が浮動小数点値として表現できる正の値での最小値です。
	LDBL_MIN	2.2250738585072014e-308	long double 型が浮動小数点値として表現できる正の値での最小値です。
	FLT_MIN_EXP	-149	float 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	DBL_MIN_EXP	-1074	double 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	LDBL_MIN_EXP	-1074	long double 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	FLT_MIN_10_EXP	-44	float 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	DBL_MIN_10_EXP	-323	double 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	LDBL_MIN_10_EXP	-323	long double 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	FLT_DIG	6	float型の浮動小数点値の10進精度の最大桁数です。
	DBL_DIG	15	double型の浮動小数点値の10進精度の最大桁数です。
	LDBL_DIG	15	long double型の浮動小数点値の10進精度の最大桁数です。
	FLT_MANT_DIG	24	float 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。
	DBL_MANT_DIG	53	double 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。

種別	定義名	定義値	説明
定数（マクロ）	LDBL_MANT_DIG	53	long double 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。
	FLT_EXP_DIG	8	float 型の浮動小数点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 型の浮動小数点値を基数に合わせて表現した時の指数部の最大桁数です。
	FLT_POS_EPS	5.9604648328104311e-8F	float 型において、1.0＋x≠1.0 である最小の浮動小数点値 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値 x を示します。
	LDBL_POS_EPS	1.1102230246251567e-16	long double 型において、1.0 ＋ x ≠ 1.0 である最小の浮動小数点値 x を示します。
	FLT_NEG_EPS	2.9802324164052156e-8F	float 型において、1.0－x≠1.0 である最小の浮動小数点値 x を示します。
	DBL_NEG_EPS	5.5511151231257834e-17	double 型において、1.0－x≠1.0 である最小の浮動小数点値 x を示します。
	LDBL_NEG_EPS	5.5511151231257834e-17	long double 型において、1.0 － x ≠ 1.0 である最小の浮動小数点値 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。
	FLT_NEG_EPS_EXP	-24	float 型において、1.0－(基数)^n≠1.0 となる最小の整数 n を示します。
	DBL_NEG_EPS_EXP	-53	double 型において、1.0－(基数)^n≠1.0 となる最小の整数 n を示します。
	LDBL_NEG_EPS_EXP	-53	long double 型において、1.0 － (基数)^n ≠ 1.0 となる最小の整数 n を示します。

種別	定義名	定義値	説明
定数 (マクロ)	DECIMAL_DIG	10	浮動小数点型数値の 10 進精度の最大桁数です。
	FLT_EPSILON	1E-5	float 型で表現可能な 1 より大きい最小の値と１との差を示します。
	DBL_EPSILON	1E-9	double 型で表現可能な 1 より大きい最小の値と１との差を示します。
	LDBL_EPSILON	1E-9	long double 型で表現可能な 1 より大きい最小の値と１との差を示します。

(1) <limits.h>

整数型データの内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数（マクロ）	CHAR_BIT	8	char 型が何ビットから構成されるかを示します。
	CHAR_MAX	127	char 型の変数が値として持つことができる最大値です。
		255 *1	
	CHAR_MIN	-128	char 型の変数が値として持つことができる最小値です。
		0 *1	
	SCHAR_MAX	127	signed char 型の変数が値として持つことができる最大値です。
	SCHAR_MIN	-128	signed char 型の変数が値として持つことができる最小値です。
	UCHAR_MAX	255U	unsigned char 型の変数が値として持つことができる最大値です。
	SHRT_MAX	32767	short 型の変数が値として持つことができる最大値です。
	SHRT_MIN	-32768	short 型の変数が値として持つことができる最小値です。
	USHRT_MAX	65535U	unsigned short 型の変数が値として持つことができる最大値です。
	INT_MAX	2147483647	int 型の変数が値として持つことができる最大値です。
		32767*2	
	INT_MIN	-2147483647-1	int 型の変数が値として持つことができる最小値です。
		-32768*2	
	UINT_MAX	4294967295U	unsigned int 型の変数が値として持つことができる最大値です。
		65535U*2	
	LONG_MAX	2147483647L	long型の変数が値として持つことができる最大値です。
	LONG_MIN	-2147483647L-1L	long型の変数が値として持つことができる最小値です。

種別	定義名	定義値	説明
定数（マクロ）	ULONG_MAX	4294967295U	unsigned long 型の変数が値として持つことができる最大値です。
	LLONG_MAX	9223372036854775807LL	long long 型の変数が値として持つことができる最大値です。
	LLONG_MIN	–9223372036854775807LL-1LL	long long 型の変数が値として持つことができる最小値です。
	ULLONG_MAX	18446744073709551615ULL	unsigned long long 型の変数が値として持つことができる最大値です。

注 1. signed_char オプションを指定した場合の変数が値として持つことができる値になります。
2. int_to_short オプションを指定した場合の変数が値として持つことができる値になります。

6. 4. 5 <errno.h>

ライブラリ関数においてエラーが発生したときに errno に設定する値を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
変数 (マクロ)	errno	int 型変数です。ライブラリ関数においてエラーが発生したときにエラー番号が設定されます。
定数 (マクロ)	ERANGE	「11.3 C 標準ライブラリ関数のエラーメッセージ」を参照してください。
	EDOM	
	ESTRN	
	PTRERR	
	ECBASE	
	ETLN	
	EEXP	
	EEXPN	
	EFLOATO	
	EFLOATU	
	EDBLO	
	EDBLU	
	ELDBLO	
	ELDBLU	
	NOTOPN	
	EBADF	
	ECSPEC	
	EFIXEDO	
	EFIXEDU	
	EACCUMO	
	EACCUMU	
	ELFIXEDO	
	ELFIXEDU	
	ELACCUMO	
	ELACCUMU	
	EILSEQ	

6. 4. 6 <math.h>

各種の数値計算を行います。

以下の定数 (マクロ) はすべて処理系定義です。

種別	定義名	説明
定数 (マクロ)	EDOM	関数に入力するパラメータの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が double 型の値として表せない時、あるいはオーバフロー / アンダフローとなった時、errno に設定する値です。
	HUGE_VAL HUGE_VALF HUGE_VALL	関数の計算結果がオーバフローした時に、関数のリターン値として返す値です。
	INFINITY	正または符号なしの無限大を表す float 型の定数式に展開します。
	NAN	float 型の qNaN をサポートしている場合に定義されます。
	FP_INFINITE FP_NAN FP_NORMAL FP_SUBNORMAL FP_ZERO	浮動小数点数の値の排他的な種類を表します。
	FP_FAST_FMA FP_FAST_FMAF FP_FAST_FMAFL	Fma 関数が double 型のオペランドを持つ 1 回の乗算と加算が、同等以上の速度で実行される場合に定義されます。
	FP_ILOGB0 FP_ILOGBNAN	それぞれ 0 または非数の場合に ilogb で返される値の整数定数式に展開します。
	MATH_ERRNO MATH_ERREXCEPT	それぞれ整数定数 1 および 2 に展開します。
	math_errhandling	Int 型で値が、MATH_ERRNO,MATH_ERREXCEPT のビット単位の論理和の式に展開します。
型	float_t double_t	それぞれ float 型、double 型と同じ幅を持つ浮動小数点型です。
関数 (マクロ)	fpclassify	実引数の値を非数、無限大、正規化数、非正規化数、０に分類します。
	isfinite	実引数が有限の値か判定します。
	isinf	実引数が無限大か判定します。
	isnan	実引数が非数か判定します。
	isnormal	実引数が正規化数か判定します。
	signbit	実引数の符号が負か判定します。
	isgreater	最初の引数が２番目の引数より大きいかどうかを判定します。
	isgreaterequal	最初の引数が２番目の引数以上かどうかを判定します。

種別	定義名	説明
関数（マクロ）	isless	最初の引数が２番目の引数より小さいかどうかを判定します。
	Islessequal	最初の引数が２番目の引数以下かどうかを判定します。
	Islessgreater	最初の引数が２番目の引数より小さいまたは大きいを判定します。
	Isunordered	順序付けられていないかどうかを判定します。
関数	acos acosf acosl	浮動小数点値の逆余弦を計算します。
	asin asinf asinl	浮動小数点値の逆正弦を計算します。
	atan atanf atanl	浮動小数点値の逆正接を計算します。
	atan2 atan2f atan2l	浮動小数点値どうしを除算した結果の値の逆正接を計算します。
	cos cosf cosl	浮動小数点値のラジアン値の余弦を計算します。
	sin sinf sinl	浮動小数点値のラジアン値の正弦を計算します。
	tan tanf tanl	浮動小数点値のラジアン値の正接を計算します。
	cosh coshf coshl	浮動小数点値の双曲線余弦を計算します。
	sinh sinhf sinhl	浮動小数点値の双曲線正弦を計算します。
	tanh tanhf tanhl	浮動小数点値の双曲線正接を計算します。
	exp expf expl	浮動小数点値の指数関数を計算します。

種別	定義名	説明
関数	frexp frexpf frexpl	浮動小数点値を [0.5,1.0] の値と 2 のべき乗の積に分解します。
	ldexp ldexpf ldexpl	浮動小数点値と 2 のべき乗の乗算を計算します。
	log logf logl	浮動小数点値の自然対数を計算します。
	log10 log10f log10l	浮動小数点値の 10 を底とする対数を計算します。
	modf modff modfl	浮動小数点値を整数部分と小数部分に分解します。
	pow powf powl	浮動小数点値のべき乗を計算します。
	sqrt sqrtf sqrtl	浮動小数点値の正の平方根を計算します。
	ceil ceilf ceill	浮動小数点値の小数点以下を切り上げた整数値を求めます。
	fabs fabsf fabsl	浮動小数点値の絶対値を計算します。
	floor floorf floorl	浮動小数点値の小数点以下を切り捨てた整数値を求めます。
	fmod fmodf fmodl	浮動小数点値どうしを除算した結果の余りを計算します。
	acosh acoshf acoshl	浮動小数点値の双曲線逆余弦を計算します。

種別	定義名	説明
関数	asinh asinhf asinhl	浮動小数点値の双曲線逆正弦を計算します。
	atanh atanhf atanhl	浮動小数点値の双曲線逆正接を計算します。
	exp2 exp2f exp2l	浮動小数点値の 2 の x 乗を計算します。
	expm1 expm1f expm1l	自然対数の x 乗から 1 を引いた値を計算します。
	ilogb ilogbf ilogbl	符号あり int の値として x の指数を抽出します。
	log1p log1pf log1pl	実引数に 1 を加えた値の自然対数を計算します。
	log2 log2f log2l	2 を底とする対数を計算します。
	logb logbf logbl	符号あり整数の値として x の指数を抽出します。
	scalbn scalbnf scalbnl scalbln scalblnf scalblnl	X × FLT_RADIXn を計算します。
	cbrt cbrtf cbrtl	浮動小数点値の立方根を計算します。
	hypot hypotf hypotl	浮動小数点値の引数ごとに２乗し、その和を計算します。

種別	定義名	説明
関数	erf erff erfl	誤差関数を計算します。
	erfc erfcf erfcl	余誤差関数を計算します。
	lgamma lgammaf lgammal	ガンマ関数の絶対値の自然対数を計算します。
	tgamma tgammaf tgammal	ガンマ関数を計算します。
	nearbyint nearbyintf nearbyintl	浮動小数点値を丸め方向にしたがって、浮動小数点形式の整数値に丸めます。
	rint rintf rintl	nearbyint に対して、浮動小数点例外を生成することがあります。
	lrint lrintf lrintl llrint llrintf llrintl	丸め方向に従って、最も近い整数値に丸めます。
	round roundf roundl	浮動小数点形式の最も近い整数値に丸めます。
	lround lroundf lroundl llround llroundf llroundl	最も近い整数値に丸めます。
	trunc truncf truncl	浮動小数点形式の最も近い整数値に丸めます。

種別	定義名	説明
関数	remainder remainderf remainderl	IEEE60559 の剰余 x REM y を計算します。
	remquo remquof remquol	x/y と同符号で、商の絶対値を 2^n を法として合同である絶対値を計算します。
	copysign copysignf copysignl	絶対値、および符号が同じ値を生成します。
	nan nanf nanl	nan(“n 文字列 ”) は、strtod(“NAN(n 文字列)”, (char**) NULL) と等価です。
	nextafter nextafterf nextafterl	関数の型に変換して、実軸上の次に表現可能な値を求めます。
	nexttoward nexttowardf nexttowardl	2 番目の引数型が long double, 引数同士が等しい場合に、2 番目の引数をその関数の型に変換して返す以外は、nextafter 関数群と同じです。
	fdim fdimf fdiml	正の差を計算します。
	fmax fmaxf fmaxl	大きい方の値を求めます。
	fmin fminf fminl	小さいほうの値を求めます。
	fma fmaf fmal	(x × y)+z をひとつの 3 項演算としてまとめて計算します。

エラーが発生した時の動作を以下に説明します。

(1) 定義域エラー

関数に入力するパラメータの値が関数内で定義している値の範囲を超えている時、定義域エラーが発生します。この時 errno には EDOM の値が設定されます。また、関数のリターン値は、処理系定義です。

(2) 範囲エラー

関数における計算結果が double 型の値として表せない時には範囲エラーが発生します。この時、errno には ERANGE の値が設定されます。また、計算結果がオーバフローの時は、正しく計算が行われた時と同様の符号の HUGE_VAL、HUGE_VALF あるいは HUGE_VALL の値をリターン値として返します。逆に計算結果がアンダフローの時は、0 をリターン値として返します。

注 1. <math.h> **の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず** errno **をチェックしてから用いてください。**

例

```
.
.
.
1 x=asin(a);
2 if (errno==EDOM)
3 printf("error\n");
4 else
5 printf("result is : %lf\n",x);
.
.
.
```

1 **行目で、**asin **関数を使って逆正弦値を求めます。このとき、実引数** a **の値が、**asin **関数の定義域** [-1.0, 1.0] **の範囲を超えていると、**errno **に値** EDOM **が設定されます。**2 **行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、**3 **行目で、**error **を出力します。定義域エラーが生じなければ** 5 **行目で、逆正弦値を出力します。**

2. **範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点型の内部表現形式によって異なります。例えば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように** <math.h> **のライブラリ関数を実現することができます。**

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力実引数が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「9.1.3 浮動小数点型の仕様」を参照してください。
2	数学関数でアンダフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmod 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

acos/ acosf/ acosl

浮動小数点値の逆余弦を計算します。

[**指定形式**]

```
#include <math.h>
double acos(double d);
float acosf(float d);
long double acosl(long double d);
```

[**引数**]

d **逆余弦を求める浮動小数点値**

[**戻り値**]

正常：d **の逆余弦値**

異常：定義域エラーの時は、非数を返します

[**備考**]

d **の値が** [-1.0,1.0] **の範囲を超えている時、定義域エラーになります。**

acos **関数のリターン値の範囲は** [0,π] **です。**

asin/asinf/asinl

浮動小数点値の逆正弦を計算します。

[**指定形式**]

#include <math.h>

double asin(double d);

float asinf(float d);

long double asinl(long double);

[**引数**]

d **逆正弦を求める浮動小数点値**

[**戻り値**]

正常：d **の逆正弦値**

異常：定義域エラーの時は、非数を返します

[**備考**]

d **の値が** [-1.0,1.0] **の範囲を超えている時、定義域エラーになります。**

asin **関数のリターン値の範囲は** [-π/2,π/2] **です。**

atan/ atanf/ atanl

浮動小数点値の逆正接を計算します。

[**指定形式**]

#include <math.h>

double atan(double d);

float atanf(float d);

long double atanl(long double d);

[**引数**]

d **逆正接を求める浮動小数点値**

[**戻り値**]

d **の逆正接値**

[**備考**]

atan **関数のリターン値の範囲は** (-π/2,π/2) **です。**

atan2/ atan2f/ atan2l

浮動小数点値どうしを除算した結果の値の逆正接を計算します。

[**指定形式**]

#include <math.h>

double atan2(double y, double x);

float atan2f(float y, float x);

long double atan2l(long double y, long double x);

[引数]

x 除数

y 被除数

[戻り値]

正常：y を x で除算したときの逆正接値

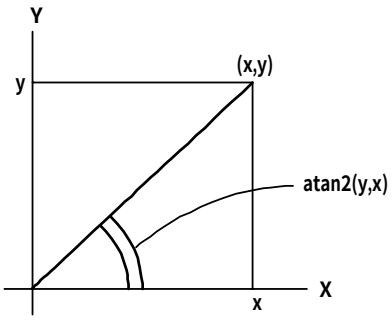
異常：定義域エラーの時は、非数を返します

[備考]

x, y の値がともに 0.0 の時、定義域エラーになります。

atan2 関数のリターン値の範囲は (-π, π) です。atan2 関数の示す意味を図 6.1 に示します。図に示すように、atan2 関数の結果は、点 (x, y) と原点を通る直線と x 軸をなす角を求めます。y ＝ 0.0 で x が負の時、結果は π、x ＝ 0.0 の時、y の値の正負に従って結果は± π/2 となります。

図 6―1 atan2 関数の意味



cos/ cosf/ cosl

浮動小数点値のラジアン値の余弦を計算します。

[指定形式]

#include <math.h>

double cos(double d);

float cosf(float d);

long double cosl(long double d);

[引数]

d 余弦を求めるラジアン値

[戻り値]

d の余弦値

[備考]

sin/ sinf/ sinl

浮動小数点値のラジアン値の正弦を計算します。

[指定形式]

#include <math.h>

double sin(double d);

float sinf(float d);

long double sinl(long double d);

[**引数**]

d **正弦を求めるラジアン値**

[**戻り値**]

d **の正弦値**

[**備考**]

－

tan/ tanf/ tanl

浮動小数点値のラジアン値の正接を計算します。

[**指定形式**]

#include <math.h>

double tan(double d);

float tanf(float d);

long double tanl(long double d);

[**引数**]

d **正接を求めるラジアン値**

[**戻り値**]

d **の正接値**

[**備考**]

－

cosh/ coshf/ coshl

浮動小数点値の双曲線余弦を計算します。

[**指定形式**]

#include <math.h>

double cosh(double d);

float coshf(float d);

long double coshl(long double d);

[**引数**]

d **双曲線余弦を求める浮動小数点値**

[**戻り値**]

d **の双曲線正弦値**

[**備考**]

－

sinh / sinhf / sinhl

浮動小数点値の双曲線正弦を計算します。

[**指定形式**]

#include <math.h>

double sinh(double d);

float sinhf(float d);

long double sinhl(long double d);

[**引数**]

d　**双曲線正弦を求める浮動小数点値**

[**戻り値**]

d **の双曲線正弦値**

[**備考**]

－

tanh/ tanhf/ tanhl

浮動小数点値の双曲線正接を計算します。

[**指定形式**]

#include <math.h>

double tanh(double d);

float tanhf(float d);

long double tanhl(long double d)

[**引数**]

d　**双曲線正接を求める浮動小数点値**

[**戻り値**]

d **の双曲線正接値**

[**備考**]

－

exp/ expf/ expl

浮動小数点値の指数関数を計算します。

[**指定形式**]

#include <math.h>

double exp(double d);

float expf(float d);

long double expl(long double d);

[**引数**]

d　**指数関数を求める浮動小数点値**

[**戻り値**]

d **の指数関数値**

[**備考**]

－

frexp/ frexpf/ frexpl

浮動小数点値を [0.5,1.0) **の値と** 2 **のべき乗の積に分解します。**

[**指定形式**]

#include <math.h>

double frexp(double value, long *exp);

float frexpf(float value, long *exp);

long double frexpl(long double value, long *exp);

[引数]

value [0.5,1.0) の値と 2 のべき乗の積に分解する浮動小数点値

exp 2 のべき乗値を格納する記憶域へのポインタ

[戻り値]

value が 0.0 の時：0.0

value が 0.0 でない時：ret*2exp の指している領域の値 =value で定義される ret の値

[備考]

frexp 関数は、value を [0.5,1.0) の値と 2 のべき乗の積に分解します。exp の指す領域には、分解した結果の 2 のべき乗値を設定します。

リターン値 ret の値の範囲は [0.5,1.0) または 0.0 になります。

value が 0.0 ならば、exp の指す int 型の記憶域の内容と ret の値は 0.0 になります。

ldexp/ ldexpf/ ldexpl

浮動小数点値と 2 のべき乗の積を計算します。

[指定形式]

#include <math.h>

double ldexp(double e, long f);

float ldexpf(float e, long f);

long double ldexpl(long double e, long f);

[引数]

e 2 のべき乗値を求める浮動小数点値

f 2 のべき乗値

[戻り値]

$e*2^f$ の演算結果の値

[備考]

ー

log/ logf/ logl

浮動小数点値の自然対数を計算します。

[指定形式]

#include <math.h>

double log(double d);

float logf(float d);

long double logl(long double d);

[引数]

d 自然対数を求める浮動小数点値

[戻り値]

正常：d の自然対数の値

異常：定義域エラーの時は、非数を返します

[**備考**]

d **の値が負の値の時、定義域エラーになります。**

d **の値が** 0.0 **の時、範囲エラーになります。**

log10/ log10f/ log10l

浮動小数点値の 10 **を底とする対数を計算します。**

[**指定形式**]

```
#include <math.h>
double log10(double d);
float log10f(float d);
long double log10l(long double d);
```

[**引数**]

d 10 **を底とする対数を求める浮動小数点値**

[**戻り値**]

正常：d **は** 10 **を底とする対数値**

異常：定義域エラーの時は、非数を返します

[**備考**]

d **の値が負の値の時、定義域エラーになります。**

d **の値が** 0.0 **の時、範囲エラーになります。**

modf/ modff/ modfl

浮動小数点値を整数部分と小数部分に分解します。

[**指定形式**]

```
#include <math.h>
double modf(double a, double *b) ;
float modff(float a, float *b);
long double modfl(long double a, long double *b);
```

[**引数**]

a **整数部分と小数部分に分解する浮動小数点値**

b **整数部分を格納する記憶域を指すポインタ**

[**戻り値**]

a **の小数部分**

[**備考**]

－

pow/ powf/ powl

浮動小数点値のべき乗を計算します。

[**指定形式**]

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
```

long double powl(long double x, long double y);

[引数]

x べき乗される値

y べき乗する値

[戻り値]

正常：x の y 乗の値

異常：定義域エラーの時は、非数を返します

[備考]

エラー条件x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

sqrt/ sqrtf/ sqrtl

浮動小数点値の正の平方根を計算します。

[指定形式]

#include <math.h>

double sqrt(double d);

float sqrtf(float d);

long double sqrtl(long double d);

[引数]

d 正の平方根を求める浮動小数点値

[戻り値]

正常：d の正の平方根の値

異常：定義域エラーの時は、非数を返します

[備考]

d の値が負の値の時、定義域エラーになります。

ceil/ ceilf/ ceill

浮動小数点値の小数点以下を切り上げた整数値を求めます。

[指定形式]

#include <math.h>

double ceil(double d);

float ceilf(float d);

long double ceill(long double d);

[引数]

d 小数点以下を切り上げる浮動小数点値

[戻り値]

d の小数点以下を切り上げた整数値

[備考]

ceil 関数は、d の値より大きいかまたは等しい最小の整数値を double 型の値として返す関数です。したがって d の値が負の値の時は小数点以下を切り捨てた時の値を返します。

fabs/ fabsf/ fabsl

浮動小数点値の絶対値を計算します。

[指定形式]

#include <math.h>

double fabs(double d);

float fabsf(float d);

long double fabsl(long double d);

[引数]

d 絶対値を求める浮動小数点値

[戻り値]

d の絶対値

[備考]

－

floor/ floorf/ floorl

浮動小数点値の小数点以下を切り捨てた整数値を求めます。

[指定形式]

#include <math.h>

double floor(double d);

float floorf(float d);

long double floorl(long double d);

[引数]

d 小数点以下を切り捨てる浮動小数点値

[戻り値]

d の小数点以下を切り捨てた整数値

[備考]

floor 関数は、d の値を超えない範囲の整数の最大値を、double 型の値として返す関数です。したがって d の値が負の値の時は小数点以下を切り上げた時の値を返します。

fmod/ fmodf/ fmodl

浮動小数点値どうしを除算した結果の余りを計算します。

[指定形式]

#include <math.h>

double fmod(double x, double y);

float fmodf(float x, float y);

long double fmodl(long double x, long double y);

[引数]

x 被除数

y 除数

[戻り値]

y の値が 0.0 の時：x

y の値が 0.0 でない時：x を y で除算した結果の余り

[備考]

fmod 関数では、引数 x、y、リターン値 ret の間には、次に示す関係が成立します。

x=y*i+ret （ ただし i は整数値 ）

また、リターン値 ret の符号は x の符号と同じ符号になります。x/y の商を表現できない場合、結果の値は保証しません。

acosh/ acoshf/ acoshl

双曲線逆余弦を計算します。

[指定形式]

#include <math.h>

double acosh(double d);

float acoshf(float d);

long double acoshl(long double d);

[引数]

d 双曲線逆余弦を求める浮動小数点値

[戻り値]

正常：d の双曲線逆余弦値

異常：定義域エラーの時は、非数を返します

[備考]

d の値が 1.0 未満の時、定義域エラーになります。

acosh 関数のリターン値の範囲は [0,+ ∞] です。

asinh/ asinhf/ asinhl

双曲線逆正弦を計算します。

[指定形式]

#include <math.h>

double asinh(double d);

float asinhf(float d);

long double asinhl(long double d);

[引数]

d 双曲線逆正弦を求める浮動小数点値

[戻り値]

d の双曲線逆正弦値

[備考]

ー

atanh/ atanhf/ atanhl

双曲線逆正接を計算します。

[指定形式]

#include <math.h>

double atanh(double d);

float atanhf(float d);

long double atanhl(long double d);

[引数]

d 双曲線逆正接を求める浮動小数点値

[戻り値]

正常： d の双曲線逆正接値

異常： 定義域エラーの場合は関数に応じて HUGE_VAL, HUGE_VALF, HUGE_VALL
範囲エラーの場合は非数

[備考]

d の値が [-1,+1] の範囲にない場合は定義域エラーになります。

d の値が -1 または 1 に等しい場合、範囲エラーになる可能性があります。

exp2/ exp2f/ exp2l

2 の d 乗を計算します。

[指定形式]

#include <math.h>

double exp2(double d);

float exp2f(float d);

long double exp2l(long double d);

[引数]

d 指数関数を求める浮動小数点数

[戻り値]

正常： 2 の指数関数値

異常： 範囲エラーの場合は 0 又は関数に応じて +HUGE_VAL, +HUGE_VALF, +HUGE_VALL

[備考]

d の絶対値が大きすぎる場合、範囲エラーになります。

expm1/ expm1f/ expm1l

自然対数の底 e の d 乗から 1 を引いた値を計算します。

[指定形式]

#include <math.h>

double expm1(double d);

float expm1f(float d);

long double expm1l(long double d);

[引数]

d 自然対数の底 e の指数となる値

[戻り値]

正常：自然対数の底 e の d 乗から 1 を引いた値

異常：範囲エラーの場合は関数に応じて -HUGE_VAL, -HUGE_VALF, -HUGE_VALL

[備考]

d の値が 0 に近い場合でも expm1(d) は exp(x)-1 よりも正確に計算できます。

ilogb/ ilogbf/ ilogbl

d の指数を抽出します。

[指定形式]

```
#include <math.h>
long ilogb(double d);
long ilogbf(float d);
long ilogbl(long double d);
```

[引数]

d 指数を抽出する値

[戻り値]

正常： d の指数関数値

d が∞の場合は INT_MAX

d が非数の場合は FP_ILOGBNAN

d が 0 の場合は FP_ILOGBNAN

異常： d が 0 で範囲エラーの場合は FP_ILOGB0

[備考]

d の値が 0 の場合、範囲エラーになることがあります。

log1p/ log1pf/ log1pl

d に 1 を加えた値の e を底とする自然対数を計算します。

[指定形式]

```
#include <math.h>
double log1p(double d);
float log1pf(float d);
long double log1pl(long double d);
```

[引数]

d 引数に 1 を加えた値の自然対数を計算する値

[戻り値]

正常： d に 1 を加えた値の自然対数

異常： 定義域エラーの場合は非数

範囲エラーの場合は関数に応じて -HUGE_VAL, -HUGE_VALF, -HUGE_VALL

[備考]

d の値が -1 より小さい場合、定義域エラーになります。

d の値が -1 の場合、範囲エラーになります。

d の値が 0 に近い場合でも log1p(d) は log(1+d) より正確な計算ができます。

log2/ log2f/ log2l

d の 2 を底とする対数を計算します。

[指定形式]

```
#include <math.h>
double log2(double d);
```

float log2f(float d);

long double log2l(long double d);

[引数]

d 対数を計算する値

[戻り値]

正常： d の 2 を底とする対数

異常： 定義域エラーの場合は非数

[備考]

d の値が負の値の場合、定義域エラーになります。

logb/ logbf/ logbl

d の浮動小数点数の内部表現における指数部を浮動小数点値として抽出します。

[指定形式]

#include <math.h>

double logb(double d);

float logbf(float d);

long double logbl(long double d);

[引数]

d 指数を抽出する値

[戻り値]

正常： d の符号付き指数

異常： 範囲エラーの場合は関数に応じて -HUGE_VAL, -HUGE_VALF, -HUGE_VALL

[備考]

d の値が 0 の場合、範囲エラーになることがあります。

d は常に正規化されているものとして処理します。

scalbn/ scalbnf/ scalbnl/ scalbln/ scalblnf/ scalblnl

浮動小数点数に整数である基数の累乗を計算します。

[指定形式]

#include <math.h>

double scalbn(double d, long e);

float scalbnf(float d, long e);

long double scalbnl(long double d, long e);

double scalbln(double d, long e);

float scalblnf(float d, long int e);

long double scalblnl(long double d, long int e);

[引数]

d FLT_RADIX を e 乗した値と乗算する値

e FLT_RADIX を累乗する際に指数となる値

[戻り値]

正常： d と FLT_RADIX を乗算した値と等価の値

異常： 範囲エラーの場合は関数に応じて -HUGE_VAL, -HUGE_VALF, -HUGE_VALL

[備考]

d の値が 0 の場合、範囲エラーになることがあります。

実際に e を指数とした FLT_RADIX の累乗は計算しません。

cbrt/ cbrtf/ cbrtl

浮動小数点値の立方根を計算します。

[指定形式]

```
#include <math.h>
double cbrt(double d);
float cbrtf(float d);
long double cbrtl(long double d);
```

[引数]

d 立方根を求める値

[戻り値]

d の立方根値

[備考]

ー

hypot/ hypotf/ hypotl

浮動小数点値の 2 乗の和の平方根を計算します。

[指定形式]

```
#include <math.h>
double hypot(double d, double e) ;
float hypotf(float d, double e);
long double hypotl(long double d, double e);
```

[引数]

d, e 2 乗の和の平方根を求める値

[戻り値]

正常： d の 2 乗と e の 2 乗の和の平方根関数値

異常： 範囲エラーの場合は関数に応じて HUGE_VAL, HUGE_VALF, HUGE_VALL

[備考]

結果がオーバーフローする場合に範囲エラーになることがあります。

erf/ erff/ erfl

浮動小数点値の誤差関数を計算します。

[指定形式]

```
#include <math.h>
double erf(double d);
float erff(float d);
long double erfl(long double d);
```

[引数]

d 誤差関数値を求める値

[戻り値]

d の誤差関数値

[備考]

－

erfc/ erfcf/ erfcl

浮動小数点値の余誤関数を計算します。

[指定形式]

#include <math.h>

double erfc(double d);

float erfcf(float d);

long double erfcl(long double d);

[引数]

d 余誤関数値を求める値

[戻り値]

d の余誤関数値

[備考]

d の絶対値が大きすぎる場合、範囲エラーが発生します。

lgamma/ lgammaf/ lgammal

浮動小数点値のガンマ関数の対数を計算します。

[指定形式]

#include <math.h>

double lgamma(double d);

float lgammaf(float d);

long double lgammal(long double d);

[引数]

d ガンマ関数の対数値を求める値

[戻り値]

正常な場合 : d のガンマ関数の対数値。

定義域エラーの場合 : 数学的に正しい符号が付与された HUGE_VAL, HUGE_VALF, HUGE_VALL

範囲エラーの場合 : +HUGE_VAL, +HUGE_VALF, +HUGE_VALL

[備考]

d の絶対値が大きすぎる又は小さすぎる場合、範囲エラーを設定します。

d の値が負の整数又は 0 で、計算結果が表現できない場合、定義域エラーが発生します。

tgamma/ tgammaf/ tgammal

浮動小数点値のガンマ関数を計算します。

[指定形式]

#include <math.h>

double tgamma(double d);

float tgammaf(float d);

long double tgammal(long double d);

[引数]

d ガンマ関数値を求める値

[戻り値]

正常な場合 : d のガンマ関数値。

定義域エラーの場合 : d と同じ符号が付与された HUGE_VAL, HUGE_VALF, HUGE_VALL

範囲エラーの場合 : 0 又は関数に応じて数学的に正しい符号が付与された +HUGE_VAL, +HUGE_VALF, +HUGE_VALL

[備考]

d の絶対値が大きすぎる又は小さすぎる場合、範囲エラーを設定します。

d の値が負の整数又は 0 で、計算結果が表現できない場合、定義域エラーが発生します。

nearbyint/ nearbyintf/ nearbyintl

浮動小数点値を丸め方向にしたがって、浮動小数点形式の整数値に丸めます。

[指定形式]

#include <math.h>

double nearbyint(double d);

float nearbyintf(float d);

long double nearbyintl(long double d);

[引数]

d 浮動小数点形式の整数値に丸める値

[戻り値]

d の浮動小数点形式の整数値に丸めた値

[備考]

nearbyint 関数群は " 不正確結果 " 浮動小数点例外を生成しません。

rint/ rintf/ rintl

浮動小数点値を丸め方向にしたがって、浮動小数点形式の整数値に丸めます。

[指定形式]

#include <math.h>

double rint(double d);

float rintf(float d);

long double rintl(long double d);

[引数]

d 浮動小数点形式の整数値に丸める値

[戻り値]

d の浮動小数点形式の整数値に丸めた値

[備考]

rint 関数群は " 不正確結果 " 浮動小数点例外を生成する場合があるという点のみで nearbyint 関数群と違います。

lrint/ lrintf/ lrintl/ llrint/ llrintf/ llrintl

浮動小数点値を丸め方向にしたがって、最も近い整数値に丸めます。

[指定形式]

```
#include <math.h>
long int lrint (double d);
long int lrintf(float d);
long int lrintl(long double d);
long long int llrint (double d);
long long int llrintf(float d);
long long int llrintl(long double d);
```

[引数]

d 整数に丸める値

[戻り値]

正常： d を整数値に丸めた値

異常： 範囲エラーの場合は不定

[備考]

d の絶対値が大きすぎる場合、範囲エラーが発生する場合があります。

丸めた値がリターン値型の範囲外である場合のリターン値は未規定とします。

round/ roundf/ roundl/ lround/ lroundf/ lroundl/ llround/ llroundf/ llroundl

浮動小数点値を最も近い整数値に丸めます

[指定形式]

```
#include <math.h>
double round(double d);
float roundf(float d);
long double roundl(long double d);
long int lround(double d);
long int lroundf(float d);
long int lroundl(long double d);
long long int llround (double d);
long long int llroundf(float d);
long long int llroundl(long double d);
```

[引数]

d 整数に丸める値

[戻り値]

正常： d を整数値に丸めた値

異常： 範囲エラーの場合は不定

[備考]

d の絶対値が大きすぎる場合、範囲エラーが発生する場合があります。

lround 関数群は d の値が中間にある場合、その時点の丸め方向とは関係なく 0 から遠い方向を選んで丸めます。

丸めた値がリターン値型の範囲外である場合のリターン値は未規定とします。

trunc/ truncf/ truncl

浮動小数点値を最も近い浮動小数点形式の整数値に丸めます。

[**指定形式**]

#include <math.h>

double trunc(double d);

float truncf(float d);

long double truncl(long double d);

[**引数**]

d **浮動小数点形式の整数に丸める値**

[**戻り値**]

d **を切り捨てた浮動小数点形式の整数値**

[**備考**]

trunc **関数群は丸めた値の絶対値が** d **の絶対値より大きくならないようにします。**

remainder/ remainderf/ remainderl

浮動小数点数同士の剰余を計算します。

[**指定形式**]

#include <math.h>

double remainder(double d1, double d2) ;

float remainderf(float d1, float d2);

long double remainderl(long double d1, long double d2);

[**引数**]

d1,d2 **剰余を求める値** (d1 / d2)

[**戻り値**]

d1 **と** d2 **の剰余値**

[**備考**]

remainder **関数群の剰余計算は** IEEE 60559 **の規定に沿っています。**

remquo/ remquof/ remquol

浮動小数点値を最も近い整数値に丸めます。

[**指定形式**]

#include <math.h>

double remquo(double d1, double d2, long *q);

float remquof(float d1, float d2, long *q);

long double remquol(long double d1, long double d2, long *q);

[**引数**]

d1,d2 **整数に丸める値** (d1 / d2)

q **剰余計算結果の商を格納する場所を指す値**

[**戻り値**]

d1 **と** d2 **の剰余値**

[**備考**]

q に格納される値は x/y と同じ符号と、2n(n は 3 以上の処理系定義整数値) を法とする x/y の整数の商を持ちます。

copysign/ copysignf/ copysignl

絶対値が d1 に等しく、符号ビットが d2 に等しい値を生成します。

[指定形式]

```
#include <math.h>
double copysign(double d1, double d2) ;
float copysignf(float d1, float d2);
long double copysignl(long double d1, long double d2);
```

[引数]

d1生成する絶対値の値

d2生成する符号

[戻り値]

正常： d1 の絶対値、d2 の符号の値

異常： 範囲エラーの場合は不定

[備考]

copysign 関数群は d1 が非数の場合 d2 の符号ビットを持った非数を生成します。

nan/ nanf/ nanl

非数を返します。

[指定形式]

```
#include <math.h>
double nan(const char *c)
float nanf(const char *c)
long double nanl(const char *c);
```

[引数]

c 文字列ポインタ

[戻り値]

c が示す内容をもつ qNaN または 0(qNaN 未サポート時)

[備考]

nan("c 文字列 ") の呼出しは，strtod("NAN(c 文字列)", (char**) NULL) と等価です。nanf 及び nanl の呼出しは，strtof 及び strtold のそれぞれに対応する呼出しと等価です。

nextafter/ nextafterf/ nextafterl

実軸上で d1 から見て d2 に向かう方向で d1 のすぐ次の浮動小数点数表現を計算します。

[指定形式]

```
#include <math.h>
double nextafter(double d1, double d2);
float nextafterf(float d1, float d2);
long double nextafterl(long double d1, long double d2);
```

[引数]

d1 実軸上の浮動小数点値

d2 d1 から見て表現可能な浮動小数点値の存在する方向を示す値

[戻り値]

正常： 表現可能な浮動小数点値

異常： 範囲エラーの場合、関数に応じて数学的に正しい符号が付与された HUGE_VAL, HUGE_VALF, HUGE_VALL

[備考]

d1 がその型で表現できる最大の有限な値であり，かつリターン値が無限大又はその型で表現できない場合，範囲エラーが発生することがあります。

nextafter 関数群は d1 と d2 が等しい場合、d2 を返します。

nexttoward/ nexttowardf/ nexttowardl

実軸上で d1 から見て d2 に向かう方向で d1 のすぐ次の浮動小数点数表現を計算します。

[指定形式]

```
#include <math.h>
double nexttoward(double d1, long double d2);
float nexttowardf(float d1, long double d2);
long double nexttowardl(long double d1, long double d2);
```

[引数]

d1 実軸上の浮動小数点値

d2 d1 から見て表現可能な浮動小数点値の存在する方向を示す値

[戻り値]

正常： 表現可能な浮動小数点値

異常： 範囲エラーの場合、関数に応じて数学的に正しい符号が付与された HUGE_VAL, HUGE_VALF, HUGE_VALL

[備考]

d1 がその型で表現できる最大の有限な値であり，かつリターン値が無限大又はその型で表現できない場合，範囲エラーが発生することがあります。

nexttoward 関数群は d2 の値が long double であり、d1 と d2 が等しい場合は d2 を関数に応じて変換して返すという点以外は nextafter 関数群と等価です。

fdim/ fdimf/ fdiml

２引数間の正の差分を求めます。

[指定形式]

```
#include <math.h>
double fdim(double d1, double d2);
float fdimf(float d1, float d2);
long double fdiml(long double d1, long double d2);
```

[引数]

d1,d2 正の差を求める値 (|d1 - d2|)

[戻り値]

正常：　2引数間の正の差分

異常：　範囲エラーの場合は HUGE_VAL, HUGE_VALF, HUGE_VALL

[**備考**]

リターン値がオーバーフローした場合に範囲エラーが発生することがあります。

fmax/ fmaxf/ fmaxl

2引数の大きい方を求めます。

[**指定形式**]

```
#include <math.h>
double fmax(double d1, double d2) ;
float fmaxf(float d1, float d2);
long double fmaxl(long double d1, long double d2);
```

[**引数**]

d1,d2 **大きさを比較する値**

[**戻り値**]

2引数の大きい方

[**備考**]

fmax **関数群は非数を，データが欠けているものとして認識します。一方の引数が非数で，もう一方が数値の場合，数値の値を返します。**

fmin/ fminf/ fminl

2引数の小さい方を求めます。

[**指定形式**]

```
#include <math.h>
double fmin(double d1, double d2);
float fminf(float d1, float d2);
long double fminl(long double d1, long double d2);
```

[**引数**]

d1,d2 **大きさを比較する値**

[**戻り値**]

2引数の小さい方

[**備考**]

fmin **関数群は非数を，データが欠けているものとして認識します。一方の引数が非数で，もう一方が数値の場合，数値の値を返します。**

fma/ fmaf/ fmal

(d1*d2)+d3 **を一つの** 3 **項演算としてまとめて計算します。**

[**指定形式**]

```
#include <math.h>
double fma(double d1, double d2, double d3) ;
float fmaf(float d1, float d2, float d3);
long double fmal(long double d1, long double d2, long double d3);
```

[引数]

d1, d2, d3浮動小数点値

[戻り値]

(d1*d2)+d3 を 3 項演算としてまとめて計算した結果

[備考]

fma 関数群は計算結果を無限の精度であるものとして計算し、FLT_ROUNDS の値が示す丸めモードに従って、1 回だけ丸めます。

6. 4. 7 <mathf.h>

各種の数値計算を行います。

<mathf.h> では ANSI 規格規定外の単精度形式の数学関数の宣言とマクロの定義をしています。

各関数は float 型の実引数を受け取り、float 型の値を返します。

以下の定数 (マクロ) はすべて処理系定義です。

種別	定義名	説明
定数 (マクロ)	EDOM	関数に入力するパラメータの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が float 型の値として表せない時、あるいはオーバフロー / アンダフローとなった時、errno に設定する値です。
	HUGE_VALF	関数の計算結果がオーバフローした時に、関数のリターン値として返す値です。

種別	定義名	説明
関数	acosf	浮動小数点値の逆余弦を計算します。
	asinf	浮動小数点値の逆正弦を計算します。
	atanf	浮動小数点値の逆正接を計算します。
	atan2f	浮動小数点値どうしを除算した結果の値の逆正接を計算します。
	cosf	浮動小数点値のラジアン値の余弦を計算します。
	sinf	浮動小数点値のラジアン値の正弦を計算します。
	tanf	浮動小数点値のラジアン値の正接を計算します。
	coshf	浮動小数点値の双曲線余弦を計算します。
	sinhf	浮動小数点値の双曲線正弦を計算します。
	tanhf	浮動小数点値の双曲線正接を計算します。
	expf	浮動小数点値の指数関数を計算します。
	frexpf	浮動小数点値を [0.5, 1.0) の値と 2 のべき乗の積に分解します。
	ldexpf	浮動小数点値と 2 のべき乗の乗算を計算します。
	logf	浮動小数点値の自然対数を計算します。
	log10f	浮動小数点値の 10 を底とする対数を計算します。
	modff	浮動小数点値を整数部分と小数部分に分解します。
	powf	浮動小数点値のべき乗を計算します。
	sqrtf	浮動小数点値の正の平方根を計算します。
関数	ceilf	浮動小数点値の小数点以下を切り上げた整数値を求めます。
	fabsf	浮動小数点値の絶対値を計算します。
	floorf	浮動小数点値の小数点以下を切り捨てた整数値を求めます。
	fmodf	浮動小数点値どうしを除算した結果の余りを計算します。

エラーが発生した時の動作を以下に説明します。

(1) 定義域エラー

関数に入力するパラメータの値が関数内で定義している値の範囲を超えている時、定義域エラーが発生します。この時 errno には EDOM の値が設定されます。また、関数のリターン値は、処理系定義です。

(2) 範囲エラー

関数における計算結果が float 型の値として表せない時には範囲エラーが発生します。この時、errno には ERANGE の値が設定されます。また、計算結果がオーバフローの時は、正しく計算が行われた時と同様の符号の HUGE_VALF の値をリターン値として返します。逆に計算結果がアンダフローの時は、0 をリターン値として返します。

注 1. <mathf.h> の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず errno をチェックしてから用いてください。

例

```
.
.
.
1 x=asinf(a);
2 if (errno==EDOM)
3 printf("error\n");
4 else
5 printf("result is : %f\n",x);
.
.
.
```

1 行目で、asinf 関数を使って逆正弦値を求めます。このとき、実引数 a の値が、asinf 関数の定義域 [-1.0,1.0] の範囲を超えていると、errno に値 EDOM が設定されます。2 行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、3 行目で、error を出力します。定義域エラーが生じなければ 5 行目で、逆正弦値を出力します。

2. 範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点型の内部表現形式によって異なります。例えば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように <mathf.h> のライブラリ関数を実現することができます。

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力実引数が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「9.1.3 浮動小数点型の仕様」を参照してください
2	数学関数でアンダフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmodf 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

acosf

浮動小数点値の逆余弦を計算します。

[**指定形式**]

```
#include <mathf.h>
float acosf(float f);
```

[**引数**]

f　**逆余弦を求める浮動小数点値**

[**戻り値**]

正常：f **の逆余弦値**

異常：定義域エラーの時は、非数を返します

[**備考**]

f **の値が** [-1.0,1.0] **の範囲を超えている時、定義域エラーになります。**

acosf **関数のリターン値の範囲は** [0, π] **です。**

asinf

浮動小数点値の逆正弦を計算します。

[指定形式]

#include <mathf.h>

float asinf(float f);

[引数]

f 逆正弦を求める浮動小数点値

[戻り値]

正常：f の逆正弦値

異常：定義域エラーの時は、非数を返します

[備考]

f の値が [-1.0,1.0] の範囲を超えている時、定義域エラーになります。

asinf 関数のリターン値の範囲は [-π/2,π/2] です。

atanf

浮動小数点値の逆正接を計算します。

[指定形式]

#include <mathf.h>

float atanf(float f);

[引数]

f 逆正接を求める浮動小数点値

[戻り値]

f の逆正接値

[備考]

atanf 関数のリターン値の範囲は (-π/2,π/2) です。

atan2f

浮動小数点値どうしを除算した結果の値の逆正接を計算します。

[指定形式]

#include <mathf.h>

float atan2f(float y, float x);

[引数]

x 除数

y 被除数

[戻り値]

正常：y を x で除算したときの逆正接値

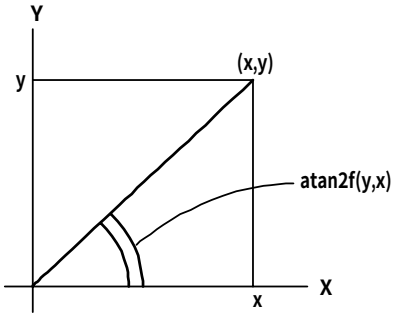
異常：定義域エラーの時は、非数を返します

[備考]

x, y の値がともに 0.0 の時、定義域エラーになります。

atan2f 関数のリターン値の範囲は (-π,π) です。atan2f 関数の示す意味を図 6.2 に示します。図に示すように、atan2f 関数の結果は、点 (x,y) と原点を通る直線と x 軸をなす角を求めます。y ＝ 0.0 で x が負の時、結果は π、x ＝ 0.0 の時、y の値の正負に従って結果は± π/2 となります。

図 6―2 atan2f 関数の意味



cosf

(機能)

[指定形式]

#include <mathf.h>

float cosf(float f);

[引数]

f **余弦を求めるラジアン値**

[戻り値]

f **の余弦値**

[備考]

―

sinf

浮動小数点値のラジアン値の正弦を計算します。

[指定形式]

#include <mathf.h>

float sinf(float f);

[引数]

f **正弦を求めるラジアン値**

[戻り値]

f **の正弦値**

[備考]

―

tanf

浮動小数点値のラジアン値の正接を計算します。

[指定形式]

#include <mathf.h>

float tanf(float f);

[引数]

f 正接を求めるラジアン値

[戻り値]

f の正接値

[備考]

―

coshf

浮動小数点値の双曲線余弦を計算します。

[指定形式]

#include <mathf.h>

float coshf(float f);

[引数]

f 双曲線余弦を求める浮動小数点値

[戻り値]

f の双曲線余弦値

[備考]

―

sinhf

浮動小数点値の双曲線正弦を計算します。

[指定形式]

#include <mathf.h>

float sinhf(float f);

[引数]

f 双曲線正弦を求める浮動小数点値

[戻り値]

f の双曲線正弦値

[備考]

―

tanhf

浮動小数点値の双曲線正接を計算します。

[指定形式]

#include <mathf.h>

float tanhf(float f)

[引数]

f 双曲線正接を求める浮動小数点値

[戻り値]

f の双曲線正接値

[備考]

―

expf

浮動小数点値の指数関数を計算します。

[**指定形式**]

#include <mathf.h>

float expf(float f);

[**引数**]

f **指数関数を求める浮動小数点値**

[**戻り値**]

f **の指数関数値**

[**備考**]

－

frexpf

浮動小数点値を [0.5,1.0) **の値と** 2 **のべき乗の積に分解します。**

[**指定形式**]

#include <mathf.h>

float frexpf(float value, long *exp);

[**引数**]

value[0.5,1.0) **の値と** 2 **のべき乗の積に分解する浮動小数点値**

exp2 **のべき乗値を格納する記憶域へのポインタ**

[**戻り値**]

value **が** 0.0 **の時：** 0.0

value **が** 0.0 **でない時：** ret*2^{exp} **の指している領域の値** =value **で定義される** ret **の値**

[**備考**]

frexpf **関数は、**value **を** [0.5,1.0) **の値と** 2 **のべき乗の積に分解します。**exp **の指す領域には、分解した結果の** 2 **のべき乗値を設定します。**

リターン値 ret **の値の範囲は** [0.5,1.0) **または** 0.0 **になります。**

value **が** 0.0 **ならば、**exp **の指す** int **型の記憶域の内容と** ret **の値は** 0.0 **になります。**

ldexpf

浮動小数点値と 2 **のべき乗の積を計算します。**

[**指定形式**]

#include <mathf.h>

float ldexpf(float e, long f);

[**引数**]

e 2 **のべき乗との積を求める浮動小数点値**

f 2 **のべき乗値**

[**戻り値**]

e*2^{f} **の演算結果の値**

[**備考**]

－

logf

浮動小数点値の自然対数を計算します。

[**指定形式**]

#include <mathf.h>

float logf(float f);

[**引数**]

f **自然対数を求める浮動小数点値**

[**戻り値**]

正常：f **の自然対数の値**

異常：定義域エラーの時は、非数を返します

[**備考**]

f **の値が負の時、定義域エラーになります。**

f **の値が** 0.0 **の時、範囲エラーになります。**

log10f

浮動小数点値の 10 **を底とする対数を計算します。**

[**指定形式**]

[**引数**]

f 10 **を底とする対数を求める浮動小数点値**

[**戻り値**]

正常：f **は** 10 **を底とする対数値**

異常：定義域エラーの時は、非数を返します

[**備考**]

f **の値が負の値の時、定義域エラーになります。**

f **の値が** 0.0 **の時、範囲エラーになります。**

modff

浮動小数点値を整数部分と小数部分に分解します。

[**指定形式**]

#include <mathf.h>

float modff(float a, float *b);

[**引数**]

a **整数部分と小数部分に分解する浮動小数点値**

b **整数部分を格納する記憶域を指すポインタ**

[**戻り値**]

a **の小数部分**

[**備考**]

－

powf

浮動小数点値のべき乗を計算します。

[指定形式]

#include <mathf.h>

float powf(float x, float y);

[引数]

x べき乗される値

y べき乗する値

[戻り値]

正常：x の y 乗の値

異常：定義域エラーの時は、非数を返します

[備考]

x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

sqrtf

浮動小数点値の正の平方根を計算します。

[指定形式]

#include <mathf.h>

float sqrtf(float f);

[引数]

f 正の平方根を求める浮動小数点値

[戻り値]

正常：f の正の平方根の値

異常：定義域エラーの時は、非数を返します

[備考]

f の値が負の値の時、定義域エラーになります。

ceilf

浮動小数点値の小数点以下を切り上げた整数値を求めます。

[指定形式]

#include <mathf.h>

float ceilf(float f);

[引数]

f 小数点以下を切り上げる浮動小数点値

[戻り値]

f の小数点以下を切り上げた整数値

[備考]

ceilf 関数は、f の値より大きい、または等しい最小の整数値を float 型の値として返す関数です。したがって f の値が負の値の時は小数点以下を切り捨てた時の値を返します。

fabsf

浮動小数点値の絶対値を計算します。

[指定形式]

#include <mathf.h>

float fabsf(float f);

[引数]

f 絶対値を求める浮動小数点値

[戻り値]

f の絶対値

[備考]

－

floorf

浮動小数点値の小数点以下を切り捨てた整数値を求めます。

[指定形式]

#include <mathf.h>

float floorf(float f);

[引数]

f 小数点以下を切り捨てる浮動小数点値

[戻り値]

f の小数点以下を切り捨てた整数値

[備考]

floorf 関数は、f の値を超えない範囲の整数の最大値を、float 型の値として返す関数です。したがって f の値が負の値の時は小数点以下を切り上げた時の値を返します。

fmodf

浮動小数点値どうしを除算した結果の余りを計算します。

[指定形式]

#include <mathf.h>

float fmodf(float x, float y);

[引数]

x 被除数

y 除数

[戻り値]

y の値が 0.0 の時：x

y の値が 0.0 でない時：x を y で除算した結果の余り

[備考]

fmodf 関数では、引数 x、y、リターン値 ret の間には、次に示す関係が成立します。

x=y*i+ret（ただし i は整数値）

また、リターン値 ret の符号は x の符号と同じ符号になります。

x/y の商を表現できない場合、結果の値は、保証しません。

6. 4. 8 <setjmp.h>

関数間の制御の移動をサポートします。

以下のマクロは、処理系定義です。

種別	定義名	説明
型 (マクロ)	jmp_buf	関数間の制御の移動を可能とする情報を保存しておくための記憶域に対応する型名です。
関数	setjmp	現在実行中の関数の jmp_buf で定義した実行環境を指定した記憶域に退避します。
	longjmp	setjmp 関数で退避していた関数の実行環境を回復し、setjmp 関数を呼び出したプログラムの位置に制御を移動します。

setjmp 関数は現在の関数の実行環境を退避します。その後 longjmp 関数を呼び出すことにより、setjmp 関数を呼び出したプログラム上の位置に戻ることができます。

以下に setjmp、longjmp 関数を使用して関数間の制御の移動をサポートした例を示します。

```
1  #include <stdio.h>
2  #include <setjmp.h>
3  jmp_buf env;
4  void sub();
5  void main()
6  {
7
8       if (setjmp(env)!=0){
9          printf("return from longjmp\n");
10         exit(0);
11      }
12      sub();
13 }
14
15  void sub()
16 {
17      printf("subroutine is running \n");
18      longjmp(env,1);
19 }
```

【説明】

8 行目で setjmp 関数を呼んでいます。この時、setjmp 関数の呼び出された環境を、jmp_buf 型の変数 env に退避します。この時のリターン値は 0 なので、次に関数 sub が呼び出されます。

関数 sub の中で呼び出される longjmp 関数により、変数 env に退避した環境を回復します。その結果、プログラムは、あたかも 8 行目の setjmp 関数からリターンしたかのようにふるまいます。ただし、この時のリターン値は longjmp 関数の第 2 実引数で指定した値 (1) になります。

その結果、9 行目以降が実行されます。

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。

[指定形式]

#include <setjmp.h>

long setjmp(jmp_buf env);

[引数]

env実行環境を退避する記憶域へのポインタ

[戻り値]

setjmp 関数を呼び出した時: 0

longjmp 関数からのリターン時: 0 以外

[備考]

setjmp 関数により退避された実行環境は、longjmp 関数において使用されます。

setjmp 関数として呼び出された時のリターン値は 0 ですが、longjmp 関数からリターンしてきた時のリターン値は、longjmp 関数で指定した第 2 引数の値となります。

setjmp 関数を複雑な式から呼び出す場合、式の評価の途中結果等の現在の実行環境の一部が失われる可能性があります。setjmp 関数は setjmp 関数の結果と定数式の比較という形だけで使用し、複雑な式の中では呼び出さないようにしてください。

setjmp 関数へのポインタを使った間接呼び出しはしないでください。

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。

[指定形式]

#include <setjmp.h>

void longjmp(jmp_buf env, long ret);

[引数]

env 実行環境を退避した記憶域へのポインタ

retsetjmp 関数へのリターンコード

[戻り値]

ー

[備考]

longjmp 関数は、同じプログラム中で最後に呼び出された setjmp 関数によって退避された関数の実行環境を第 1 引数 env で指定された記憶域から回復し、その setjmp 関数を呼び出したプログラムの位置に制御を移します。この時 longjmp 関数の第 2 引数 ret が setjmp 関数のリターン値として返ります。ただし、ret が 0 の時は setjmp 関数へのリターン値としては 1 が返ります。

setjmp 関数が呼び出されていない時、あるいは setjmp 関数を呼び出した関数がすでに return 文を実行している時は、longjmp 関数の動作は保証しません。

6. 4. 9 <stdarg.h>

可変個の引数を持つ関数に対し、その引数の参照を可能にします。

以下はすべて処理系定義です。

種別	定義名	説明
型 (マクロ)	va_list	可変個の引数を参照するために、va_start, va_arg, va_end マクロで共通に使用される変数の型です。
関数 (マクロ)	va_start	可変個の引数の参照を行うため、初期設定処理を行います。
	va_arg	可変個の引数を持つ関数に対して、現在参照中引数の次の引数への参照を可能とします。
	va_end	可変個の引数を持つ関数の引数への参照を終了させます。
	va_copy	可変個の引数をコピーします。

本標準インクルードファイルで定義しているマクロを使用したプログラムの例を以下に示します。

```
1   #include <stdio.h>
2   #include <stdarg.h>
3
4   extern void prlist(int count,...);
5
6   void main()
7   {
8       prlist(1, 1);
9       prlist(3, 4, 5, 6);
10      prlist(5, 1, 2, 3, 4, 5);
11  }
12
13  void prlist(int count,...)
14  {
15      va_list ap;
16      int i;
17
18      va_start(ap,count);
19      for(i=0;i<count;i++)
20          printf("%d",va_arg(ap,int));
21      putchar('\n');
22      va_end(ap);
23  }
```

【説明】

この例では、第 1 引数に出力するデータの数を指定し、以下の引数をその数だけ出力する関数 prlist を実現しています。

18 行目で、可変個の引数への参照を va_start で初期化します。その後引数を一つ出力するたびに、va_arg マクロによって次の引数を参照します (20 行目)。va_arg マクロでは、引数の型名 (この場合は int 型) を第 2 引数に指定します。

引数の参照が終了したら、va_end マクロを呼び出します (22 行目)。

va_start

可変個の実引数への参照を行うため、初期設定処理を行います。

[指定形式]

#include <stdarg.h>

void va_start(va_list ap, parmN)；

[引数]

ap 可変個の引数にアクセスするための変数

parmN 最右端の引数の識別子

[戻り値]

ー

[備考]

va_start マクロは、va_arg、va_end マクロによって使用される ap の初期化を行います。また、parmN には、外部関数定義における引数の並びの最右端の引数の識別子、すなわち「,...」の直前の識別子を指定します。

可変個の名前のない引数を参照するためには、va_start マクロ呼び出しを一番初めに実行する必要があります。

va_arg

可変個の実引数を持つ関数に対して、現在参照中の引数の次の引数への参照を可能とします。

[指定形式]

#include <stdarg.h>

type va_arg(va_list ap, type) ;

[引数]

ap　可変個の引数にアクセスするための変数

type　アクセスする引数の型

[戻り値]

引数の値

[備考]

va_start マクロで初期化した va_list 型の変数を第 1 引数に指定します。

ap の値は va_arg を使用する度に更新され、結果として可変個の引数が順次本マクロのリターン値として返されます。

第 2 引数 type は、参照する型を指定してください。

ap は va_start によって初期化された ap と同じでなければなりません。

type に char 型、unsigned char 型、short 型、unsigned short 型、float 型のように関数の引数に指定した時に型変換によってサイズが変わる型を指定した場合、正しく引数を参照することができなくなります。このような型を指定すると動作は保証しません。

va_end

可変個の実引数を持つ関数の引数への参照を終了させます。

[指定形式]

#include <stdarg.h>

void va_end(va_list ap) ;

[引数]

ap 可変個の引数を参照するための変数

[戻り値]

－

[備考]

ap は va_start によって初期化された ap と同じでなければなりません。また、関数からの return 前に va_end マクロが呼び出されない時は、その関数の動作は保証しません。

va_copy

可変個の実引数を持つ関数に対して、現在参照中の引数の複製を作ります。

[指定形式]

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src) ;
```

[引数]

dest 可変個の引数を参照するための変数の複製

src 可変個の引数を参照するための変数

[戻り値]

－

[備考]

va_start マクロで初期化され、va_arg で使用された可変個の引数の状態を持つ第 2 引数 src に対し、第 1 引数 dest に複製を作ります。

src は va_start によって初期化された src と同じでなければなりません。

dest は、これ以降の va_arg マクロで可変個の引数を表す引数として使用することができます。

6. 4. 10 <stdio.h>

ストリーム入出力用ファイルの入出力に関する処理を行います。

以下の定数 (マクロ) はすべて処理系定義です。

種別	定義名	説明
定数 (マクロ)	FILE	ストリーム入出力処理で必要とするバッファへのポインタやエラー指示子、終了指示子などの各種制御情報を保存しておく構造体の型です。
	_IOFBF	バッファ領域の使用方法として、入出力処理はすべてバッファ領域を使用することを示しています。
	_IOLBF	バッファ領域の使用方法として、入出力処理は行単位でバッファ領域を使用することを示しています。
	_IONBF	バッファ領域の使用方法として、入出力処理はバッファ領域を使用しないことを示しています。
	BUFSIZ	入出力処理において必要とするバッファの大きさです。
	EOF	ファイルの終わり (End Of File) すなわちファイルからそれ以上の入力が無いことを示しています。
	L_tmpnam*	tmpnam 関数によって生成される一時ファイル名の文字列を格納するのに十分な大きさの配列のサイズです。
	SEEK_CUR	ファイルの現在の読み書き位置を現在の位置からのオフセットに移すことを示しています。
	SEEK_END	ファイルの現在の読み書き位置をファイルの終了位置からのオフセットに移すことを示しています。
	SEEK_SET	ファイルの現在の読み書き位置をファイルの先頭位置からのオフセットに移すことを示しています。
	SYS_OPEN*	処理系が同時にオープンすることができることを保証するファイルの数です。
	TMP_MAX*	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です。
	stderr	標準エラーファイルに対するファイルポインタです。
	stdin	標準入力ファイルに対するファイルポインタです。
	stdout	標準出力ファイルに対するファイルポインタです。
関数	fclose	ストリーム入出力用ファイルをクローズします。
	fflush	ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。
	fopen	ストリーム入出力用ファイルを指定したファイル名によってオープンします。
	freopen	現在オープンされているストリーム入力出用ファイルをクローズし、新しいファイルを指定したファイル名で再オープンします。
	setbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。

種別	定義名	説明
関数	setvbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。
	fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。
	vfprintf	可変個の引数リストを書式に従って指定したストリーム入出力用ファイルに出力します。

注 * 本処理系では、定義されません。

種別	定義名	説明
関数	printf	データを書式に従って変換し、標準出力ファイル (stdout) へ出力します。
	vprintf	可変個の引数リストを書式に従って標準出力ファイル (stdout) に出力します。
	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。
	snprintf	データを書式に従って変換し、配列に書き込みます。
	vsnprintf	可変個数の実引数並びを va_list で置き換えた snprintf と等価です。
	vfscanf	可変個数の実引数並びを va_list で置き換えた fscanf と等価です。
	vscanf	可変個数の実引数並びを va_list で置き換えた scanf と等価です。
	vsscanf	可変個数の実引数並びを va_list で置き換えた sscanf と等価です。
	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。
	scanf	標準入力ファイル (stdin) からデータを入力し、書式に従って変換します。
	vsprintf	可変個の引数リストを書式に従って指定した領域に出力します。
	fgetc	ストリーム入出力用ファイルから 1 文字入力します。
	fgets	ストリーム入出力用ファイルから文字列を入力します。
	fputc	ストリーム入出力用ファイルへ 1 文字出力します。
	fputs	ストリーム入出力用ファイルへ文字列を出力します。
	getc	(マクロ) ストリーム入出力用ファイルから 1 文字入力します。
	getchar	(マクロ) 標準入力ファイルから 1 文字入力します。
	gets	標準入力ファイルから文字列を入力します。
	putc	(マクロ) ストリーム入出力用ファイルへ 1 文字出力します。
	putchar	(マクロ) 標準出力ファイルへ 1 文字出力します。
	puts	標準出力ファイルへ文字列を出力します。
	ungetc	ストリーム入出力用ファイルへ 1 文字をもどします。
	fread	ストリーム入出力用ファイルから指定した記憶域にデータを入力します。
	fwrite	記憶域からストリーム入出力用ファイルにデータを出力します。
	fseek	ストリーム入出力用ファイルの現在の読み書き位置を移動させます。

種別	定義名	説明
関数	ftell	ストリーム入出力用ファイルの現在の読み書き位置を求めます。
	rewind	ストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
	clearerr	ストリーム入出力用ファイルのエラー状態をクリアします。
	feof	ストリーム入出力用ファイルが終わりであるかどうかを判定します。
	ferror	ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
	perror	標準エラーファイル (stderr) に、エラー番号に対応したエラーメッセージを出力します。
型	fpos_t	ファイル中の任意の位置を指定可能な型です。
定数（マクロ）	FOPEN_MAX	同時にオープン可能なファイル数です。
	FILENAME_MAX	保持可能なファイル名の最大長です。

処理系定義仕様

	項目	コンパイラの仕様
1	入力テキストの最終の行が終了を示す改行文字を必要とするかどうか	規定しません。低水準インタフェースルーチンの仕様によります。
2	改行文字の直前にかき出された空白文字は、読み込み時に読み込まれるかどうか	
3	バイナリファイルに書かれたデータに付加されるヌル文字の数	
4	追加モード時のファイル位置指定子の初期値	
5	テキストファイルへの出力によってそれ以降のファイルのデータが失われるかどうか	
6	ファイルバッファリングの仕様	
7	長さ 0 のファイルが存在するかどうか	
8	正当なファイル名の構成規則	
9	同時に同じファイルをオープンできるかどうか	
10	fprintf 関数における %p 書式変換の出力形式	16 進数出力となります
11	fscanf 関数における %p 書式変換の入力形式 fscanf 関数での変換文字 ¢ − £ の意味	16 進数入力となります。 先頭、最後あるいは「^」の直後でない場合、直前の文字と直後の範囲を示します。
12	fgetpos,ftell 関数で設定される errno の値	fgetpos 関数はサポートしていません。 ftell 関数については規定しません。 低水準インタフェースルーチンの仕様によります。
13	perror 関数が生成するメッセージ出力形式	メッセージの出力形式を (a) に示します。

(a) perror 関数の出力形式は、

< 文字列 >：<error に設定したエラー番号に対応するエラーメッセージ >

となります。

(b) printf 関数、fprintf 関数で、浮動小数点の無限大および非数を表示するときの形式を表 6.7 に示します。

表 6―7 無限大および非数の表示形式

	値	表示形式
1	正の無限大	＋＋＋＋＋＋
2	負の無限大	－－－－－－
3	非数	＊＊＊＊＊＊

ストリーム入出力用ファイルに対する一連の入出力処理を行ったプログラムの例を以下に示します。

```
1    #include <stdio.h>
2
3    void main()
```

```
4   {
5       int c;
6       FILE *ifp, *ofp;
7
8       if ((ifp=fopen("INPUT.DAT","r"))==NULL){
9           fprintf(stderr,"cannot open input file\n");
10          exit(1);
11      }
12      if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
13          fprintf(stderr,"cannot open output file\n");
14          exit(1);
15      }
16      while ((c=getc(ifp))!=EOF)
17          putc(c, ofp);
18      fclose(ifp);
19      fclose(ofp);
20  }
```

【説明】

ファイル INPUT.DAT **の内容をファイル** OUTPUT.DAT **へコピーするプログラムです。**

8 **行目の** fopen **関数で入力ファイル** INPUT.DAT **を、**12 **行目の** fopen **関数で出力ファイル** OUTPUT.DAT **をオープンします。オープンに失敗した場合、**fopen **関数のリターン値として** NULL **が返されますので、エラーメッセージを出力してプログラムを終了させます。**

fopen **関数が正常に終了した時、オープンしたファイルの情報を格納するデータ** (FILE **型**) **へのポインタが返されますので、これらを変数** ifp**、**ofp **に設定します。**

オープンが成功した後は、これらの FILE **型のデータを用いて入出力を行います。**

ファイルの処理が終了したら、fclose **関数でファイルをクローズします。**

fclose

ストリーム入出力用ファイルをクローズします。

[**指定形式**]

```
#include <stdio.h>
long fclose(FILE *fp);
```

[**引数**]

fp **ファイルポインタ**

[**戻り値**]

正常：0

異常：0 **以外**

[**備考**]

fclose **関数は、ファイルポインタ** fp **の示すストリーム入出力用ファイルをクローズします。**

fclose **関数は、ストリーム入出力用ファイルの出力ファイルがオープンされており、まだ出力されていないデータがバッファに残っている時は、それをファイルに出力してからクローズします。**

また、入出力用のバッファがシステムによって自動的に割り付けられていた場合は、それを解放します。

fflush

ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。

[指定形式]

#include <stdio.h>

long fflush(FILE *fp);

[引数]

fp ファイルポインタ

[戻り値]

正常：0

異常：0 以外

[備考]

fflush 関数は、ストリーム入出力用ファイルの出力ファイルがオープンされている時、ファイルポインタ fp で指定されたストリーム入出力用ファイルのバッファの未出力内容をファイルに出力します。また、入力ファイルがオープンされている時、ungetc 関数の指定を無効にします。

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。

[指定形式]

#include <stdio.h>

FILE *fopen(const char *fname, const char *mode);

[引数]

fname ファイル名を示す文字列へのポインタ

mode ファイルアクセスモードを示す文字列へのポインタ

[戻り値]

正常：オープンしたファイルのファイル情報を指すファイルポインタ

異常：NULL

[備考]

fopen 関数は、fname が指す文字列をファイル名とするストリーム入出力用ファイルをオープンします。書き出しモードあるいは追加モードで存在しないファイルをオープンしようとした時は、可能な限り新しいファイルを作成します。また既存のファイルに対して書き出しモードでオープンした時は、ファイルの先頭から書き込みが行われ、以前に書き込まれていたファイルの内容は消去されます。

追加モードでオープンしたファイルは、そのファイルの終わりの位置から書き出しの処理が行われます。更新モードでオープンしたファイルは、このファイルに対して入力と出力の両方の処理を行うことができます。

ただし、出力処理は後に fflush、fseek、rewind 関数が実行されることなしに入力処理を続けることはできません。

また同様に入力処理の後に fflush、fseek、rewind 関数が実行されることなしに出力処理を続けることはできません。また、ファイルアクセスモードを示す文字列の後にオープンの方法を指示する文字が付くこともあります。

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。

[指定形式]

#include <stdio.h>

FILE *freopen(const char *fname, const char *mode, FILE *fp);

[引数]

fname 新しいファイル名を示す文字列へのポインタ

mode ファイルアクセスモードを示す文字列へのポインタ

fp 現在オープンされているストリーム入出力用ファイルのファイルポインタ

[戻り値]

正常：fp

異常：NULL

[備考]

freopen 関数は、まず、ファイルポインタ fp の示すストリーム入出力用ファイルをクローズします (このクローズ処理が正しく行われない時でも以下の処理は続けます)。

次に、その fp の指す FILE 構造体を再使用して、ファイル名 fname で示すファイルを、ストリーム入出力用にオープンします。

freopen 関数は一時にオープンするファイル数が限られているときなどに有効です。

freopen 関数は通常、fp と同じ値を返しますが、エラーが発生した時は、NULL を返します。

setbuf

ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。

[指定形式]

#include <stdio.h>

void setbuf(FILE *fp, char buf[BUFSIZ]);

[引数]

fp ファイルポインタ

buf バッファ領域へのポインタ

[戻り値]

ー

[備考]

setbuf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対して buf の指す記憶域を入出力用のバッファ領域として使用するように定義します。この結果、大きさが BUFSIZ のバッファ領域を使用した入出力処理が行われます。

setvbuf

ストリーム入出力用のバッファ領域をユーザプログラムの側で定義して設定します。

[指定形式]

#include <stdio.h>

long setvbuf(FILE *fp, char *buf, long type, size_t size);

[引数]

fp ファイルポインタ

buf バッファ領域へのポインタ

type バッファの管理方式

size バッファ領域の大きさ

[戻り値]

正常：0

異常：0 以外

[備考]

setvbuf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対して buf の指す記憶域を入出力用のバッファ領域として使用するように定義します。

このバッファ領域の使用方法としては、以下の三通りの方法があります。

（a）type に _IOFBF を指定した時

入出力処理はすべてバッファ領域を使用して行います。

（b）type に _IOLBF を指定した時

入出力処理は行単位でバッファ領域を使用して行います。すなわち、入出力データは、改行文字が書かれた時、バッファ領域が一杯になった時、入力が要求された時にバッファ領域から取り出されることになります。

（c）type に _IONBF を指定した時

入出力処理はバッファ領域を使用せず行います。

setvbuf 関数は通常 0 を返しますが、type あるいは size に不正な値が与えられた時、あるいはバッファ領域の使用方法等の要求が受け入れられなかった時には 0 以外の値を返します。

バッファ領域は、オープンされているストリーム入出力用ファイルがクローズされる前に解放してはいけません。また、setvbuf 関数は、ストリーム入出力用ファイルがオープンされてから入出力用処理が行われるまでの間で使用してください。

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。

[指定形式]

```
#include <stdio.h>
long fprintf(FILE *fp, const char *control [, arg] …);
```

[引数]

fp ファイルポインタ

control 書式を示す文字列へのポインタ

arg,... 書式に従って出力されるデータの並び

[戻り値]

正常：変換し出力した文字数

異常：負の値

[備考]

fprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、ファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。

fprintf 関数は、通常は変換し出力したデータの個数を返しますが、エラー発生時には負の値を返します。

書式の仕様は以下のとおりです。

【書式の概要】

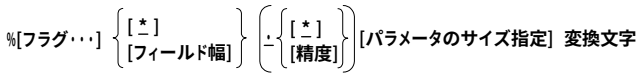
書式を表す文字列は、2 種類の文字列から構成されます。

- 通常の文字

  次の変換仕様を示す文字列以外の文字はそのまま出力されます。

- 変換仕様

  変換仕様は、% で始まる文字列で、後に続く引数の変換方法を指定します。変換仕様の形式は次の規則に従います。



  この変換仕様に対して、実際に出力する引数が無い時は、その動作は保証しません。また、変換仕様よりも実際に出力する引数の個数が多い時は、余分な引数はすべて無視されます。

【変換仕様の説明】

(d) フラグ

符号を付けるなどの出力するデータに対する修飾を指定します。指定できるフラグの種類と意味を表 6.8 に示します。

表 6―8 フラグの種類と意味

	種類	意味
1	-	変換したデータの文字数が指定したフィールド幅より少ない時、そのデータをフィールド内で左詰めにして出力します。
2	+	符号付きのデータに変換する時、そのデータの符号に従って、変換したデータの先頭にプラスあるいはマイナス符号を付けます。
3	空白	符号付きのデータの変換において、変換したデータの先頭に符号が付かない時、そのデータの先頭に空白を付けます。 「+」と共に使用した時、本フラグは無視されます。
4	#	表 6.10 で説明する変換の種類に従って、変換後のデータに修飾を行います。 1. c，d，i，s，u 変換の時 本フラグは無視されます。 2. o 変換の時 変換したデータの先頭に 0 を付けます。 3. x(あるいは X) 変換の時 変換したデータの先頭に 0x(あるいは 0X) を付けます。 4. e，E，f，g，G 変換の時 変換したデータに小数点以下がない時でも、小数点を出力します。 また、g，G 変換の時は、変換後のデータの後に付く 0 は取り除きません。

(e) フィールド幅

変換したデータを出力する文字数を任意の 10 進数で指定します。

変換したデータの文字数がフィールド幅より少ない時、フィールド幅までそのデータの先頭に空白が付けられます (ただし、フラグとして '-' を指定した時は、データの後に空白が付けられます)。

もし、変換したデータの文字数がフィールド幅より大きい時は、フィールド幅は、変換結果を出力できる幅に拡張されます。

また、フィールド幅指定の先頭が 0 で始まっている時は、出力するデータの先頭には空白ではなく文字「0」が付けられます。

(f) 精度

表 6.10 で説明する変換の種類に従って変換したデータの精度を指定します。

精度は、ピリオド (.) の後に 10 進整数を続ける形式で指定します。10 進整数を省略した時は、0 を指定したものと仮定します。

精度を指定した結果、フィールド幅の指定との間に矛盾が生じれば、フィールド幅の指定を無効とします。

各変換の種類と精度指定の意味を以下に示します。

- d，i，o，u，x，X 変換の時

  変換したデータの最小の桁数を示します。

- e，E，f 変換の時

  変換したデータの小数点以下の桁数を示します。

- g，G 変換の時

  変換したデータの最大有効桁数を示します。

- s 変換の時

印字される最大文字数を示します。

(g) パラメータのサイズ指定

d，i，o，u，x，X，e，E，f，g，G 変換の時 (表 6.10 参照)

変換するデータのサイズ (short 型、long 型、long long 型、long double 型) を指定します。これ以外の変換の時は、本指定を無視します。表 6.9 にサイズ指定の種類とその意味を示します。

表 6―9 パラメータのサイズ指定の種類とその意味

	種類	意味
1	h	d，i，o，u，x，X 変換において、変換するデータが short 型あるいは unsigned short 型であることを指定します。
2	l	d，i，o，u，x，X 変換において、変換するデータが long 型、unsigned long 型あるいは、double 型であることを指定します。
3	L	e，E，f，g，G 変換において、変換するデータが long double 型であることを指定します。
4	ll	d，i，o，u，x，X 変換において、変換するデータが long long 型あるいは、unsigned long long 型であることを指定します。n 変換において、変換するデータが long long 型へのポインタ型であることを指定します。

(h) 変換文字

変換するデータをどのような形式に変換するかを指定します。

もし、変換するデータが構造体や配列型の時や、それらの型を指すポインタの時は、s 変換で文字の配列を変換する時、p 変換でポインタを変換する時を除いてその動作は保証しません。

表 6.10 に変換文字と変換方式を示します。この表に述べられていない英文字を変換文字として指定した時は、その動作は保証しません。また、それ以外の文字を指定した時の動作はコンパイラによって異なります。

表 6―10 変換文字と変換の方式

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
1	d	d 変換	int型データを符号付き10進数の文字列に変換します。d 変換と i 変換は同じ仕様です。	int 型	精度指定は、最低で何文字出力されるかを示しています。もし、変換後の文字数が精度の値より少ない時は、文字列の先頭に 0 が付きます。また、精度を省略した時は、1 が仮定されます。さらに、0 の値を持つデータを精度に 0 を指定して変換し出力しようとした時は、何も出力されません。
2	i	i 変換		int 型	
3	o	o 変換	int 型データを符号なしの 8 進数の文字列に変換します。	int 型	
4	u	u 変換	int型データを符号なしの10進数の文字列に変換します。	int 型	
5	x	x 変換	int型データを符号なしの16進数に変換します。16 進文字には a，b，c，d，e，f を用います。	int 型	
6	X	X 変換	int型データを符号なしの16進数に変換します。16 進文字には A，B，C，D，E，F を用います。	int 型	
7	f	f 変換	double 型データを「[-]ddd.ddd」の形式の 10 進数の文字列に変換します。	double 型	精度の指定は、小数点以降の桁数を表します。小数点以降の文字が存在する時には、必ず小数点の前に 1 桁の数字が出力されます。精度を省略した時は、6 が仮定されます。また、精度に 0 を指定した時は、小数点と小数点以降の文字は出力しません。出力するデータは丸められます。
8	e	e 変換	double 型データを「[-]d.ddde±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	精度の指定は、小数点以降の桁数を表します。変換した文字は小数点の前に 1 桁の数字が出力され、小数点以降に精度に等しい桁数の数字が出力される形式となります。精度を省略した時は 6 が仮定されます。また、精度に 0 を指定した時は、小数点以降の文字は出力しません。出力するデータは丸められます。
9	E	E 変換	double 型データを「[-]d.dddE±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
10	g	g変換(あるいは G 変換)	変換する値と有効桁数を指定する精度の値から f 変換の形式で出力するか e 変換 (あるいは E 変換) の形式で出力するかを決め double 型データを出力します。もし、変換されたデータの指数が -4 より小さいか、有効桁数を指定する精度より大きい時には e 変換 (あるいは E 変換) の形式に変換します。	double 型	精度の指定は、変換されたデータの最大有効桁数を示します。
11	G			double 型	
12	c	c 変換	int 型のデータを unsigned char 型データとし、そのデータに対応する文字に変換します。	int 型	精度の指定は無効です。
13	s	s 変換	char 型へのポインタ型データが指す文字列を文字列の終了を示すヌル文字まで、あるいは、精度で指定された文字数分出力します (ただしヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません)。	char型へのポインタ型	精度の指定は出力する文字数を示します。もし、精度が省略された時は、データが指す文字列のヌル文字までの文字が出力されます (ただし、ヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません)。
14	p	p 変換	データをポインタとして、コンパイラごとに定義された印字可能な文字列に変換します。	void 型へのポインタ	精度の指定は無効です。
15	n	データの変換は生じません。	データは int 型へのポインタ型とみなされ、このデータが指す記憶域にいままで、出力したデータの文字数を設定します。	int 型へのポインタ型	
16	%	データの変換は生じません。	% を出力します。	なし	

(i) フィールド幅あるいは精度に対する * 指定

フィールド幅あるいは精度指定の値として * を指定することができます。この時は、この変換仕様に対応するパラメータの値がフィールド幅あるいは精度指定の値として使用されます。このパラメータが負のフィールド幅を持つ時は、正のフィールド幅にフラグ - が指定されたと解釈します。また、負の精度を持つ時は、精度が省略されたものと解釈します。

snprintf

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。

[指定形式]

#include <stdio.h>

long snprintf(char *restrict s, size_t n, const char *restrict control [, arg] ...);

[引数]

s データを出力する記憶域へのポインタ

n 出力する文字数

control 書式を示す文字列へのポインタ

arg,... 書式に従って出力されるデータ

[戻り値]

変換した文字数

[備考]

snprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、s の指す記憶域へ出力します。

変換して出力した文字列の最後には、ヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。書式の仕様の詳細は fprintf 関数を参照してください。

vsnprintf

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。

[指定形式]

#include <stdarg.h>

#include <stdio.h>

long vsnprintf(char *restrict s, size_t n, const char *restrict control, va_list arg);

[引数]

s データを出力する記憶域へのポインタ

n 出力する文字数

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

変換した文字数

[備考]

vsnprintf 関数は，可変個引数を arg で置き換えた snprintf と等価です。

vsnprintf 関数の呼出し前に，va_start マクロで arg を初期化してください。

vsnprintf 関数は，va_end マクロを呼び出しません。

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。

[指定形式]

#include <stdio.h>

long fscanf(FILE *fp, const char *control[, ptr] ...);

[引数]

fp ファイルポインタ

control 書式を示す文字列へのポインタ

ptr,... 入力したデータを格納する記憶域へのポインタ

[戻り値]

正常 : 入力変換に成功したデータの個数

異常 : 入力データの変換を行う前に入力データが終了した時：EOF

[備考]

fscanf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルからデータを入力し、control が指す書式を文字列に従って変換、編集して、その結果を ptr の指す記憶域へ格納します。

データを入力するための書式の仕様を以下に示します。

【書式の概要】

書式を表す文字列は、以下の 3 種類の文字列から構成されます。

- 空白文字

  空白 (' ')、水平タブ ('\t') あるいは改行文字 ('\n') を指定すると、入力データを次の空白類文字でない文字まで読み飛ばす処理を行います。

- 通常の文字

  上の空白文字でも % でもない文字を指定すると、入力データを 1 文字入力します。ここで入力した文字は書式を表す文字列の中に指定した文字と一致していなければなりません。

- 変換仕様

  変換仕様は、% で始まる文字列で、書式を表す文字列の後に続く引数の指す領域に入力データを変換して格納する方法を指定します。変換仕様の形式は次の規則に従います。

  %[*][フィールド幅][変換後のデータのサイズ] 変換文字

  書式中の変換仕様に対して入力したデータを格納する記憶域へのポインタがない時は、その動作は保証しません。また、書式が終了したにもかかわらず、入力データを格納する記憶域へのポインタが残っている時は、そのポインタは無視されます。

【変換仕様の説明】

* 指定

入力したデータを引数が指す記憶域に格納することを抑止します。

フィールド幅

入力するデータの最大文字数を 10 進数字で指定します。

変換後のデータのサイズ

d，i，o，u，x，X，e，E，f 変換の時 (表 6.12 参照)、変換後のデータのサイズ (short 型、long 型、long long 型、long double 型) を指定します。これ以外の変換の時は、本指定を無視します。表 6.11 にサイズ指定の種類とその意味を示します。

表 6―11 変換後のデータのサイズ指定の種類とその意味

	種類	意味
1	h	d，i，o，u，x，X 変換において、変換後のデータは short 型であることを指定します。
2	l	d，i，o，u，x，X 変換において、変換後のデータは long 型であることを指定します。 また、e，E，f 変換において、変換後のデータは double 型であることを指定します。
3	L	e，E，f 変換において、変換後のデータは、long double 型であることを指定します。
4	ll	d，i，o，u，x，X 変換において、変換後のデータは long long 型であることを指定します。

- 変換文字

入力するデータは、各変換文字が指定する変換の種類に従って変換します。ただし、空白類文字を読み込んだ場合、変換の対象として許されていない文字を読み込んだ場合、あるいは指定されたフィールド幅を超えた場合は処理を終了します。

表 6―12 変換文字と変換の内容

	変換文字	変換の種類	変換の方式	対応するパラメータの型名
1	d	d 変換	10 進数字の文字列を整数型データに変換します。	整数型
2	i	i 変換	先頭に符号が付いている 10 進数字の文字列、あるいは最後に u(U) または l(L) が付いている 10 進数字の文字列を整数型データに変換します。また、先頭が 0x(あるいは 0X) で始まっている文字列は、16 進数字として解釈し、文字列を int 型データに変換します。さらに、先頭が 0 で始まっている文字列は、8 進数字として解釈し文字列を int 型データに変換します。	整数型
3	o	o 変換	8 進数字の文字列を整数型データに変換します。	整数型
4	u	u 変換	符号なしの 10 進数字の文字列を整数型データに変換します。	整数型
5	x	x 変換	16 進数字の文字列を整数型データに変換します。 x 変換と X 変換に意味の違いはありません。	整数型
6	X	X 変換		
7	s	s 変換	空白、水平タブ、改行文字を読み込むまでをひとつの文字列として変換します。文字列の最後にはヌル文字を付加します (変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です)。	文字型
8	c	c 変換	1 文字を入力します。この時、入力する文字が空白類文字であっても読み飛ばすことはしません。もし、空白類文字以外の文字だけを読み込む時は、%1s と指定してください。また、フィールド幅が指定されている時は、その指定分の文字が読み込まれます。したがって、この時、変換したデータを格納する記憶域は、指定分の大きさが必要です。	char 型

	変換文字	変換の種類	変換の方式	対応するパラメータの型名
9	e	e 変換	浮動小数点型を示す文字列を浮動小数点型データに変換します。e 変換と E 変換、g 変換と G 変換にそれぞれ意味の違いはありません。入力形式は strtod 関数で表現できる浮動小数点型です。	浮動小数点型
10	E	E 変換		
11	f	f 変換		
12	g	g 変換		
13	G	G 変換		
14	p	p 変換	fprintf 関数において、p 変換で変換される形式の文字列をポインタ型データに変換します。	void 型へのポインタ型
15	n	データの変換は生じません。	データの入力は行わず、いままでに入力したデータの文字数が設定されます。	整数型
16	[	[変換	[の後に文字の集合、その後に] を指定します。この文字集合は、文字列を構成する文字の集合を定義しています。もし、文字集合の最初の文字が ^ でない時は、入力データはこの文字集合にない文字が最初に読み込まれるまでをひとつの文字列として入力します。もし、最初の文字が ^ の時は、 ^ を除いた文字集合の文字が最初に読み込まれるまでをひとつの文字列として入力します。入力した文字列の最後には自動的にヌル文字を付加します (変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です)。	文字型
17	%	データの変換は生じません。	% を読み込みます。	なし

変換文字が表 6.12 に示す文字以外の英文字の時は、その動作は保証しません。また、その他の文字の時は、その動作は処理系定義です。

printf

データを書式に従って変換し、標準出力ファイル (stdout) へ出力します。

[指定形式]

```
#include <stdio.h>
long printf(const char *control[, arg]...);
```

[引数]

control 書式を示す文字列へのポインタ

arg,... 書式に従って出力されるデータ

[戻り値]

正常：変換し出力した文字数

異常：負の値

[備考]

printf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、標準出力ファイル (stdout) へ出力します。

書式の仕様の詳細は fprintf 関数を参照してください。

vfscanf

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。

[指定形式]

#include <stdarg.h>

#include <stdio.h>

long vfscanf(FILE *restrict fp, const char *restrict control , va_list arg);

[引数]

fp ファイルポインタ

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：入力データの変換を行う前に入力データが終了した時：EOF

[備考]

vfscanf 関数は可変個引数並びを arg で置き換えた fscanf と等価です。

vfscanf 関数の呼出し前に，va_start マクロで arg を初期化してください。

vfscanf 関数は va_end マクロを呼び出しません。

scanf

標準入力ファイル (stdin) からデータを入力し、書式に従って変換します。

[指定形式]

#include <stdio.h>

long scanf(const char *control[, ptr] …);

[引数]

control 書式を示す文字列へのポインタ

ptr,... 入力変換したデータを格納する記憶域へのポインタ

[戻り値]

正常：入力変換に成功したデータの個数

異常：EOF

[備考]

scanf 関数は、標準入力ファイル (stdin) からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。

scanf 関数は、入力変換に成功したデータの個数をリターン値として返します。最初の変換の前に標準入力ファイルが終了した時には EOF を返します。

書式の仕様の詳細は fscanf 関数を参照してください。

%e 変換では、double 型の場合は l、long double 型の場合は L で指定します。デフォルトの型は float 型です。

vscanf

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。

[指定形式]

#include <stdarg.h>

#include <stdio.h>

long vscanf(const char *restrict control , va_list arg);

[引数]

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：入力データの変換を行う前に入力データが終了した時：EOF

[備考]

vscanf 関数は，可変個数引数を arg で置き換えた scanf と等価です。

vscanf 関数の呼出し前に，va_start マクロで arg を初期化してください。

vscanf 関数は va_end マクロを呼びません。

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。

[指定形式]

#include <stdio.h>

long sprintf(char *s, const char *control [, arg] …);

[引数]

s データを出力する記憶域へのポインタ

control 書式を示す文字列へのポインタ

arg,... 書式に従って出力されるデータ

[戻り値]

変換した文字数

[備考]

sprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、s の指す記憶域へ出力します。

変換して出力した文字列の最後には、ヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。

書式の仕様の詳細は fprintf 関数を参照してください。

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。

[指定形式]

#include <stdio.h>

long sscanf(const char *s, const char *control [, ptr] …);

[引数]

s 入力するデータがある記憶域

control 書式を示す文字列へのポインタ

ptr,... 入力変換したデータを格納する記憶域へのポインタ

[戻り値]

正常：入力変換に成功したデータの個数

異常：EOF

[備考]

sscanf 関数は、s の指す記憶域からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。

sscanf 関数は、入力変換に成功したデータの個数を返します。また、最初の変換の前に入力するデータが終了した時には EOF を返します。

書式の仕様の詳細は fscanf 関数を参照してください。

vsscanf

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。

[指定形式]

```
#include <stdarg.h>
#include <stdio.h>
long vsscanf(const char *restrict s, const char *restrict control , va_list arg);
```

[引数]

s 入力するデータがある記憶域

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：入力データの変換を行う前に入力データが終了した時：EOF

[備考]

vsscanf 関数は，可変個数引数を arg で置き換えた sscanf と等価です。

vsscanf 関数の呼出し前に，va_start マクロで arg を初期化してください。

vsscanf 関数は va_end マクロを呼びません。

vfprintf

可変個の引数リストを書式に従って、指定したストリーム入出力用ファイルに出力します。

[指定形式]

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

[引数]

fp ファイルポインタ

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

正常：変換し出力した文字数

異常：負の値

[備考]

vfprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、fp の示すストリーム入出力用ファイルへ出力します。

vfprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。

また、vfprintf 関数では va_end マクロは呼び出しません。

書式の仕様の詳細は fprintf 関数を参照してください。

引数リストを示す arg は、va_start(およびそれに続く 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。

vprintf

可変個の引数リストを書式に従って標準出力ファイル (stdout) に出力します。

[指定形式]

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

[引数]

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

正常：変換し出力した文字数

異常：負の値

[備考]

vprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、標準出力ファイルへ出力します。

vprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。

また、vprintf 関数では va_end マクロは呼び出しません。

書式の仕様の詳細は fprintf 関数を参照してください。

引数リストを示す arg は、va_start(およびそれに続く 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。

vsprintf

可変個の引数リストを書式に従って、指定した記憶域に出力します。

[指定形式]

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

[引数]

s データを出力する記憶域へのポインタ

control 書式を示す文字列へのポインタ

arg 引数リスト

[戻り値]

正常：変換した文字数

異常：負の数

[備考]

vsprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、s により指される記憶域へ出力します。

変換して出力した文字列の最後にヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。

書式の仕様の詳細は fprintf 関数を参照してください。

引数リストを示す arg は、va_start(およびそれに続く 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。

fgetc

ストリーム入出力用ファイルから 1 文字入力します。

[指定形式]

```
#include <stdio.h>
long fgetc(FILE *fp);
```

[引数]

fp ファイルポインタ

[戻り値]

正常：ファイルの終了の時: EOF

ファイルの終了でない時: 入力した文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してのエラー指示子が設定されます。

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

fgetc 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は、EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

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。

[指定形式]

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

[引数]

s 文字列を入力する記憶域へのポインタ

n 文字列を入力する記憶域のバイト数

fp ファイルポインタ

[戻り値]

正常：ファイルの終了の時: NULL

ファイルの終了でない時: s

異常：NULL

[備考]

fgets 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから、ポインタ s の指す記憶域に文字列を入力します。

fgets 関数は、n-1 文字まであるいは改行文字を入力するまで、またはファイルの終わりになるまで文字を入力し、入力文字列の最後にヌル文字を付け加えます。

fgets 関数は通常、文字列を入力する記憶域へのポインタ s を返しますが、ファイルが終了した時やエラー発生の時は NULL を返します。

ファイルが終了した時は、s が指す記憶域の内容は変化しませんが、エラー発生の時は、s が指す記憶域の内容は保証しません。

fputc

ストリーム入出力用ファイルへ 1 文字出力します。

[指定形式]

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

[引数]

c 出力する文字

fp ファイルポインタ

[戻り値]

正常：出力した文字

異常：EOF

[備考]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

fputc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力ファイルへ出力します。

fputc 関数は、通常出力した文字 c を返しますが、エラー発生の時は、EOF を返します。

fputs

ストリーム入出力用ファイルへ文字列を出力します。

[指定形式]

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

[引数]

s 出力する文字列へのポインタ

fp ファイルポインタ

[戻り値]

正常：0

異常：0 以外

[備考]

fputs 関数は、s の指すヌル文字の直前までの文字列をファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。この時、文字列の終了を示すヌル文字は出力されません。

fputs 関数は、通常 0 を返しますが、エラー発生の時は、0 以外の値を返します。

getc

ストリーム入出力用ファイルから 1 文字入力します。

[指定形式]

#include <stdio.h>

long getc(FILE *fp);

[引数]

fp ファイルポインタ

[戻り値]

正常：ファイルの終了の時: EOF

ファイルの終了でない時: 入力した文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

getc 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから 1 文字入力します。

getc 関数は、通常入力した 1 文字を返しますがファイルの終了やエラー発生の時は、EOF を返します。またファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

getchar

標準入力ファイル (stdin) から、1 文字入力します。

[指定形式]

#include <stdio.h>

long getchar(void);

[引数]

ー

[戻り値]

正常：ファイルの終了の時: EOF

ファイルの終了でない時: 入力した文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

getchar 関数は標準入力ファイル (stdin) から 1 文字入力します。

getchar 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

gets

標準入力ファイル (stdin) から文字列を入力します。

[指定形式]

#include <stdio.h>

char *gets(char *s);

[引数]

s 文字列を入力する記憶域へのポインタ

[戻り値]

正常：ファイルの終了の時： NULL

ファイルの終了でない時： s

異常：NULL

[**備考**]

gets **関数は、標準入力ファイル** (stdin) **から、**s **で始まる記憶域へ文字列を入力します。**

gets **関数は、ファイルの終了か、改行文字を入力するまで文字を入力し、改行文字の代わりにヌル文字を付け加えます。**

gets **関数は、通常文字列を入力する記憶域へのポインタ** s **を返しますが、標準入力ファイルの終了やエラー発生の時は、**NULL **を返します。**

標準入力ファイルが終了した時は、s **が指す記憶域の内容は変化しませんが、エラー発生の時は** s **が指す記憶域の内容は保証しません。**

putc

ストリーム入出力用ファイルへ 1 **文字出力します。**

[**指定形式**]

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

[**引数**]

c **出力する文字**

fp **ファイルポインタ**

[**戻り値**]

正常：出力した文字

異常：EOF

[**備考**]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

putc **関数は、文字** c **をファイルポインタ** fp **の示すストリーム入出力ファイルへ出力します。**

putc **関数は、通常出力した文字** c **を返しますが、エラー発生の時は** EOF **を返します。**

putchar

標準出力ファイル (stdout) **へ** 1 **文字出力します。**

[**指定形式**]

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

[**引数**]

c **出力する文字**

[**戻り値**]

正常：出力した文字

異常：EOF

[**備考**]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

putchar 関数は、文字 c を標準出力ファイル (stdout) へ出力します。putchar マクロは、通常出力した文字 c を返しますが、エラー発生の時は EOF を返します。

puts

標準出力ファイル (stdout) へ文字列を出力します。

[指定形式]

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

[引数]

s 出力する文字列へのポインタ

[戻り値]

正常：0

異常：0 以外

[備考]

puts 関数は、s の指す文字列を標準出力ファイル (stdout) へ出力します。この時、文字列の終了を示す文字は出力されず、代わりに改行文字を出力します。

puts 関数は、通常 0 を返しますが、エラー発生の時は 0 以外の値を返します。

ungetc

ストリーム入出力用ファイルへ 1 文字を戻します。

[指定形式]

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

[引数]

c 戻す文字

fp ファイルポインタ

[戻り値]

正常：戻した文字

異常：EOF

[備考]

ungetc 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へ戻します。

また、ここで戻された文字は、fflush，fseek，rewind 関数を呼び出さなければ次の入力データとなります。

ungetc 関数は、通常戻した文字 c を返しますが、エラー発生の時は EOF を返します。

ungetc 関数が fflush，fseek，rewind 関数を実行することなく 2 回以上呼び出された時の動作は保証しません。また、ungetc 関数が実行されるとファイルに対する現在の位置指示子が一つ戻されますが、この位置指示子がすでにファイルの先頭に位置している時は、位置指示子は保証しません。

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。

[指定形式]

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

[引数]

ptr データを入力する記憶域へのポインタ

size 1 メンバのバイト数

n 入力するメンバの数

fp ファイルポインタ

[戻り値]

size もしくは n が 0 の時: 0

size,n がともに 0 でない時: 入力に成功したメンバ数

[備考]

fread 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから ptr が指す記憶域に size で指定したバイト数を 1 メンバとしたデータを n メンバ入力します。この時、ファイルに対する位置指示子は入力したバイト数分進められます。

fread 関数は、実際に入力に成功したメンバ数を返しますので、通常 n と同じ値になります。しかし、ファイルが終了した時やエラー発生の時は、それまで入力に成功したメンバ数を返しますので、n より小さな値となります。ファイルの終了かエラー発生かの区別は、ferror，feof 関数を用いて行ってください。

size もしくは n が 0 の時、リターン値として 0 を返し、ptr の指す記憶域の内容は変化しません。また、エラーが発生した時、または、メンバの途中までしか入力できなかった時は、そのファイルの位置指示子は保証しません。

fwrite

メモリ領域からストリーム入出力用ファイルにデータを出力します。

[指定形式]

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

[引数]

ptr 出力するデータを格納している記憶域へのポインタ

size 1 メンバのバイト数

n 出力するメンバの数

fp ファイルポインタ

[戻り値]

出力に成功したメンバ数

[備考]

fwrite 関数は、ptr の指す記憶域から、ファイルポインタ fp の示すストリーム入出力用ファイルに、size で指定したバイト数を 1 メンバとしたデータを n メンバ出力します。

この時、ファイルに対する位置指示子は出力したバイト数進められます。

fwrite 関数は、実際に出力に成功したメンバ数を返しますので、通常 n と同じ値になります。しかし、エラー発生の時はそれまで出力に成功したメンバ数を返しますので、n より小さな値となります。

エラー発生の時、そのファイルの位置指示子は保証しません。

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。

[指定形式]

#include <stdio.h>

long fseek(FILE *fp, long offset, long type);

[引数]

fp ファイルポインタ

offset オフセットの種類で指定された位置からのオフセット

type オフセットの種類

[戻り値]

正常：0

異常：0 以外

[備考]

fseek 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をオフセットの種類 type で指定した場所から offset バイト先の位置に移動します。

オフセットの種類を表 6.13 に示します。

fseek 関数は、通常は 0 を返しますが、不適当な要求に対しては 0 以外の値を返します。

表 6―13 オフセットの種類

	オフセットの種類	意味
1	SEEK_SET	ファイルの先頭から offset バイト先の位置に移動します。この時、offset で指定する値は 0 か正でなければなりません。
2	SEEK_CUR	ファイルの現在位置から offset バイト先の位置に移動します。この時、offset で指定する値が正ならばファイルの後方に、負ならばファイルの先頭に向かって移動します。
3	SEEK_END	ファイルの終わりから offset バイト先の位置に移動します。この時 offset で指定する値は 0 か負でなければなりません。

テキストファイルの時は、オフセットの種類は SEEK_SET で、かつ offset は 0 かそのファイルに対する ftell 関数によって返された値でなければなりません。また、fseek 関数を呼び出すことによって ungetc 関数の効果はなくなりますので注意が必要です。

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。

[指定形式]

#include <stdio.h>

long ftell(FILE *fp);

[引数]

fp ファイルポインタ

[戻り値]

現在の位置指示子の位置 (テキストファイル)

ファイルの先頭から現在位置までのバイト数 (バイナリファイル)

[備考]

ftell 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置を求めます。

ftell 関数は、バイナリファイルの時、ファイルの先頭から現在位置までのバイト数を返しますが、テキストファイルの時は、ここで返した値が fseek 関数で使用できるように処理系定義の値を位置指示子の位置として返します。

ftell 関数を 2 回テキストファイルに適用した時、そのリターン値の差が実際のファイル上の隔たりを表すことにはなりません。

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。

[指定形式]

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

[引数]

fp ファイルポインタ

[戻り値]

－

[備考]

rewind 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。

また、rewind 関数は、そのファイルに対する終了指示子とエラー指示子をクリアします。

rewind 関数を呼び出すことによって、ungetc 関数の効果はなくなりますので、注意が必要です。

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。

[指定形式]

```
#include <stdio.h>
```

[引数]

fp ファイルポインタ

[戻り値]

－

[備考]

clearerr 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対するエラー指示子と終了指示子をクリアします。

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。

[指定形式]

```
#include <stdio.h>
long feof(FILE *fp);
```

[引数]

fp ファイルポインタ

[戻り値]

ファイルが終わりの時：0 以外

ファイルが終わりでない時：0

[備考]

feof 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルが終了したかどうかを判定します。

feof 関数は、指定したストリーム入出力用ファイルに対するファイル終了指示子を調べ、設定されていればファイルが終わりであるとして、0 以外の値を返します。設定されていなければ、ファイルはまだ終わりではないとして 0 を返します。

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。

[指定形式]

```
#include <stdio.h>
long ferror(FILE *fp);
```

[引数]

fp ファイルポインタ

[戻り値]

ファイルがエラー状態の時: 0 以外

ファイルがエラー状態でない時: 0

[備考]

ferror 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルがエラー状態であるかどうかを判定します。

ferror 関数は、指定したストリーム入出力用ファイルに対するエラー指示子を調べ、設定されていれば、エラー状態にあるとして 0 以外の値を返します。設定されていなければ、エラー状態ではないとして 0 を返します。

perror

標準エラーファイル (stderr) に、エラー番号に対応したエラーメッセージを出力します。

[指定形式]

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

[引数]

s エラーメッセージへのポインタ

[戻り値]

－

[備考]

perror 関数は標準エラーファイル (stderr) へ s で示されるエラーメッセージと errno とを対応させ出力します。

出力するメッセージは、もし、s が NULL でなく、s の指す文字列がヌル文字でなければ、s の指す文字列にコロンと空白とその後に処理系定義のエラーメッセージを続け、最後に改行文字を付けた形式で出力されます。

6. 4. 11 < stdlib.h>

C プログラムでの標準的処理を行う関数を定義しています。

以下のマクロは、処理系定義です。

種別	定義名	説明
型（マクロ）	onexit_t	onexit 関数で登録する関数の返す型および onexit 関数のリターン値の型です。
	div_t	div 関数のリターン値の構造体の型です。
	ldiv_t	ldiv 関数のリターン値の構造体の型です。
	lldiv_t	lldiv 関数のリターン値の構造体の型です。
定数（マクロ）	RAND_MAX	rand 関数において生成する擬似乱数整数の最大値です。
	EXIT_SUCCESS	成功終了状態を表します。
関数	atof	数を表現する文字列を double 型の浮動小数点値に変換します。
	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。
	atoll	10 進数を表現する文字列を long long 型の整数値に変換します。
	strtod	数を表現する文字列を double 型の浮動小数点値に変換します。
	strtof	数を表現する文字列を float 型の浮動小数点値に変換します。
	strtold	数を表現する文字列を long double 型の浮動小数点値に変換します。
	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。
	strtoll	数を表現する文字列を long long 型の整数値に変換します。
	strtoull	数を表現する文字列を unsigned long long 型の整数値に変換します。
	rand	0 から RAND_MAX の間の擬似乱数整数を生成します。
	srand	rand 関数で生成する擬似乱数列の初期値を設定します。
	calloc	記憶域を割り当てて、すべての割り当てられた記憶域を 0 で初期化します。
	free	指定された記憶域を解放します。
	malloc	記憶域を割り当てます。
	realloc	記憶域の大きさを指定された大きさに変更します。
	bsearch	2 分割検索を行います。
	qsort	ソートを行います。
	abs	int 型整数の絶対値を計算します。
	div	int 型整数の除算の商と余りを計算します。
	labs	long 型整数の絶対値を計算します。
	ldiv	long 型整数の除算の商と余りを計算します。
	llabs	long long 型整数の絶対値を計算します。
	lldiv	long long 型整数の除算の商と余りを計算します。
	mbstowcs	多バイト文字列をワイド文字列に変換します。
	wcstombs	ワイド文字列を多バイト文字列に変換します。

処理系定義仕様

	項目	コンパイラの仕様
1	calloc,malloc,realloc 関数でサイズが 0 のときの動作	NULL を返します。

atof

数を表現する文字列を、double 型の浮動小数点値に変換します。

[指定形式]

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

[引数]

nptr 変換する数を表現する文字列のポインタ

[戻り値]

変換された double 型の浮動小数点値

[備考]

変換後の値がオーバフロー / アンダフローをおこした時は errno を設定します。

変換は、浮動小数点型の形式に合わない最初の文字までに対して行います。

atof 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtod 関数を使用してください。

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。

[指定形式]

```
#include <stdlib.h>
long atoi(const char *nptr);
```

[引数]

nptr 変換する数を表現する文字列のポインタ

[戻り値]

変換された int 型の整数値

[備考]

変換後の値がオーバフローをおこした時は errno を設定します。

変換は、10 進数の形式に合わない最初の文字までに対して行います。

atoi 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtol 関数を使用してください。

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。

[指定形式]

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

[引数]

nptr 変換する数を表現する文字列のポインタ

[戻り値]

変換された long 型の整数値

[備考]

変換後の値がオーバフローをおこした時は errno を設定します。

変換は、10 進数の形式に合わない最初の文字までに対して行います。

atol 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtol 関数を使用してください。

atoll

10 進数を表現する文字列を、long long 型の整数値に変換します。

[指定形式]

```
#include <stdlib.h>
long long atoll (const char *nptr);
```

[引数]

nptr 変換する数を表現する文字列のポインタ

[戻り値]

変換された long long 型の整数値

[備考]

変換後の値がオーバフローをおこした時は errno を設定します。

変換は、10 進数の形式に合わない最初の文字までに対して行います。

atoll 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtoll 関数を使用してください。

strtod

数を表現する文字列を double 型の浮動小数点値に変換します。

[指定形式]

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

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 浮動小数点値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

[戻り値]

正常：nptr が指している文字列が浮動小数点型を構成しない文字で始まっている時：0

nptr が指している文字列が浮動小数点型を構成する文字で始まっている時: 変換された double 型の浮動小数点値

異常：変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつ HUGE_VAL

変換後の値がアンダフローの時：0

[備考]

変換後の値がオーバフロー / アンダフローをおこした時は errno を設定します。

strtod 関数は、最初の数字もしくは小数点から浮動小数点値を構成しない文字の直前までを double 型の浮動小数点値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮

定されます。endptr の指す領域には、浮動小数点型を構成しない最初の文字へのポインタを設定します。数字を読み込む前に浮動小数点型を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

strtof

数を表現する文字列を float 型の浮動小数点値に変換します。

[指定形式]

```
#include <stdlib.h>
float strtof(const char *nptr, char **endptr);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 浮動小数点値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

[戻り値]

正常：nptr が指している文字列が浮動小数点型を構成しない文字で始まっている時：0

nptr が指している文字列が浮動小数点型を構成する文字で始まっている時: 変換された float 型の浮動小数点値

異常：変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつ HUGE_VALF

変換後の値がアンダフローの時：0

[備考]

変換後の値がオーバフロー / アンダフローをおこした時は errno を設定します。

strtof 関数は、最初の数字もしくは小数点から浮動小数点値を構成しない文字の直前までを float 型の浮動小数点値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、浮動小数点型を構成しない最初の文字へのポインタを設定します。数字を読み込む前に浮動小数点型を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

strtold

数を表現する文字列を long double 型の浮動小数点値に変換します。

[指定形式]

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 浮動小数点値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

[戻り値]

正常：nptr が指している文字列が浮動小数点型を構成しない文字で始まっている時：0

nptr が指している文字列が浮動小数点型を構成する文字で始まっている時: 変換された long double 型の浮動小数点値

異常：変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつ HUGE_VALL

変換後の値がアンダフローの時：0

[備考]

変換後の値がオーバフロー / アンダフローをおこした時は errno を設定します。

strtold 関数は、最初の数字もしくは小数点から浮動小数点値を構成しない文字の直前までを long double 型の浮動小数点値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、浮動小数点型を構成しない最初の文字へのポインタを設定します。数字を読み込む前に浮動小数点型を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

strtol

数を表現する文字列を long 型の整数値に変換します。

[指定形式]

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr, long base);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時: 変換された long 型の整数値

異常：変換後の値がオーバフローの時：変換する文字列の符号に従って LONG_MAXあるいは LONG_MIN

[備考]

変換後の値がオーバフローをおこした時は、errno を設定します。

strtol 関数は、最初の数字から整数を構成しない最初の文字の前までを long 型の整数値に変換します。

endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。

base の値が 0 の時は、「3.1.3(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A) から z(もしくは Z) までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X) も無視されます。

strtoul

数を表現する文字列を unsigned long 型の整数値に変換します。

[指定形式]

```
#include <stdlib.h>
unsigned long strtoul (const char *nptr, char **endptr, long base);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時：変換された unsigned long 型の整数値

異常：変換後の値がオーバフローの時：ULONG_MAX

[備考]

変換後の値がオーバフローをおこした時は、errno を設定します。

strtoul 関数は、最初の数字から整数を構成しない最初の文字の前までを unsigned long 型の整数値に変換します。

endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。

base の値が 0 の時は、「3.1.3(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A) から z(もしくは Z) までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X) も無視されます。

strtoll

数を表現する文字列を long long 型の整数値に変換します。

[指定形式]

```
#include <stdlib.h>
long long strtoll (const char *nptr, char **endptr, long base);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時：変換された long long 型の整数値

異常：変換後の値がオーバフローの時：変換する文字列の符号に従って LLONG_MAXあるいは LLONG_MIN

[備考]

変換後の値がオーバフローをおこした時は、errno を設定します。

strtoll 関数は、最初の数字から整数を構成しない最初の文字の前までを long long 型の整数値に変換します。

endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。

base の値が 0 の時は、「3.1.3(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A) から z(もしくは Z) までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X) も無視されます。

strtoull

数を表現する文字列を unsigned long long 型の整数値に変換します。

[指定形式]

#include <stdlib.h>

unsigned long long strtoull (const char *nptr, char **endptr, long base);

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時：変換された unsigned long long 型の整数値

異常：変換後の値がオーバフローの時：ULLONG_MAX

[備考]

変換後の値がオーバフローをおこした時は、errno を設定します。

strtoull 関数は、最初の数字から整数を構成しない最初の文字の前までを unsigned long long 型の整数値に変換します。

endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。

base の値が 0 の時は、「3.1.3(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A) から z(もしくは Z) までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X) も無視されます。

rand

0 から RAND_MAX の間の擬似乱数整数を生成します。

[指定形式]

#include <stdlib.h>

long rand(void);

[引数]

－

[戻り値]

擬似乱数整数値

[備考]

－

srand

rand 関数で生成する擬似乱数列の初期値を設定します。

[指定形式]

#include <stdlib.h>

void srand(unsigned long seed)

[引数]

seed 擬似乱数列生成の初期値

[戻り値]

－

[備考]

srand 関数は、rand 関数が擬似乱数列を生成するための初期値を設定します。したがって、rand 関数で擬似乱数値を生成している時に、再度 srand 関数で、同じ値の初期値を設定すると、擬似乱数列はくり返し生成されることになります。

rand 関数が srand 関数より先に呼ばれた時は、擬似乱数列の生成の初期値として 1 が設定されます。

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 で初期化します。

[指定形式]

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

[引数]

nelem 要素の数

elsize 一つの要素の占めるバイト数

[戻り値]

正常：割り当てられた記憶域の先頭のアドレス

異常：記憶域の割り当てができなかった時、または引数のいずれかが 0 の時：NULL

[備考]

elsize バイト単位の記憶域を nelem 個記憶域に割り当てます。また、その割り当てられた記憶域のすべてのビットは 0 で初期化されます。

free

指定された記憶域を解放します。

[指定形式]

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

[引数]

ptr 解放する記憶域のアドレス

[戻り値]

－

[備考]

ptr が指す記憶域を解放し、再度割り当てて使用することを可能とします。ptr が NULL であれば何もしません。

解放しようとした記憶域が、calloc、malloc、realloc 関数で割り当てられた記憶域でない時、または、すでに free、realloc 関数によって解放されていた時の動作は保証しません。また、解放された後の記憶域を参照した時の動作も保証しません。

malloc

記憶域を割り当てます。

[指定形式]

#include <stdlib.h>

void *malloc(size_t size);

[引数]

size 割り当てる記憶域のバイト数

[戻り値]

正常：割り当てられた記憶域の先頭アドレス

異常：記憶域の割り当てができなかった時、または size が 0 の時：NULL

[備考]

size で示されるバイトの分だけ記憶域を割り当てます。

realloc

記憶域の大きさを指定された大きさに変更します。

[指定形式]

#include <stdlib.h>

void *realloc(void *ptr, size_t size);

[引数]

ptr 変更する記憶域の先頭アドレス

size 変更後の記憶域のバイト数

[戻り値]

正常：変更した記憶域の先頭アドレス

異常：記憶域の割り当てができなかった時、または size が 0 の時：NULL

[備考]

ptr の指す記憶域の大きさを size で示されるバイト分の大きさの記憶域に変更します。もし、新しく割り当てられた記憶域の大きさが、変更前の記憶域の大きさより小さい時は、新しく割り当てられた記憶域の大きさまでの内容は変化しません。

ptr が calloc、malloc、realloc 関数で割り当てられた記憶域へのポインタでない時、またはすでに free、realloc 関数によって解放されている記憶域へのポインタの時、動作はされません。

bsearch

二分割検索を行います。

[指定形式]

#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))

[引数]

key 検索するデータへのポインタ

base 検索対象となるテーブルへのポインタ

nmemb 検索対象のメンバの数

size 検索対象のメンバのバイト数

compar 比較を行う関数へのポインタ

[戻り値]

一致するメンバが検索できた時: 一致したメンバへのポインタ

一致するメンバが検索できなかった時: NULL

[備考]

key の指すデータと一致するメンバを、base の指すテーブルの中で二分割検索法によって検索します。比較を行う関数は、比較する 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 の時、負の値を返します。

*p1==*p2 の時、0 を返します。

*p1>*p2 の時、正の値を返します。

検索対象となる各メンバは、昇順に並んでいる必要があります。

qsort

ソートを行います。

[指定形式]

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

[引数]

base ソート対象となるテーブルへのポインタ

nmemb ソート対象のメンバの数

size ソート対象のメンバのバイト数

compar 比較を行う関数へのポインタ

[戻り値]

ー

[備考]

base の指すテーブルのデータをソートします。データの並べる順序は、比較を行う関数へのポインタによって指定します。この関数は、比較する 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 の時、負の値を返します。

*p1==*p2 の時、0 を返します。

*p1>*p2 の時、正の値を返します。

abs

int 型整数の絶対値を求めます。

[指定形式]

```
#include <stdlib.h>
long abs(long i);
```

[引数]

i 絶対値を求める整数

[戻り値]

i の絶対値

[備考]

i の絶対値を求めた結果、int 型整数値として表現できない時の動作は保証しません。

div

int 型整数の除算の商と余りを計算します。

[指定形式]

#include <stdlib.h>

div_t div(long numer, long denom);

[引数]

numer 被除数

denom 除数

[戻り値]

numer を denom で除算した結果の商と余り

[備考]

－

labs

long 型整数の絶対値を計算します。

[指定形式]

#include <stdlib.h>

long labs(long j);

[引数]

j　絶対値を求める整数

[戻り値]

j の絶対値

[備考]

j の絶対値を求めた結果、long 型の整数値として表現できない時の動作は保証しません。

ldiv

long 型整数の除算の商と余りを計算します。

[指定形式]

#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);

[引数]

numer 被除数

denom 除数

[戻り値]

numer を denom で除算した結果の商と余り

[備考]

－

llabs

long long 型整数の絶対値を計算します。

[指定形式]

#include <stdlib.h>

long long llabs(long long j);

[引数]

j 絶対値を求める整数

[戻り値]

j の絶対値

[備考]

j の絶対値を求めた結果、long long 型の整数値として表現できない時の動作は保証しません。

lldiv

long long 型整数の除算の商と余りを計算します。

[指定形式]

#include <stdlib.h>

lldiv_t lldiv(long long numer,long long denom);

[引数]

numer 被除数

denom 除数

[戻り値]

numer を denom で除算した結果の商と余り

[備考]

－

mbstowcs

多バイト文字列をワイド文字列に変換します。

[指定形式]

#include <stdlib.h>

size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);

[引数]

pwcs ワイド文字列へのポインタ

s 多バイト文字列へのポインタ

n ワイド文字列へ格納されるワイド文字数

[戻り値]

正常： ワイド文字列へ書き込まれた文字数

異常： (size_t)(-1)不正な多バイト文字の並びに遭遇した場合

[備考]

mbstowcs 関数は，s が指す配列中の初期シフト状態で始まる多バイト文字の並びを，対応するワイド文字の並びに変換し，n 個以下のワイド文字を pwcs が指す配列に格納します。

ナル文字を発見した場合はナルワイド文字に変換し、変換処理を終了します。各多バイト文字は，mbtowc 関数の変換状態が影響を受けないことを除いて，mbtowc 関数の呼出しによる場合と同じ規則で変換します。領域の重なり合うオブジェクト間でコピーが行われる場合の動作は未定義とします。

正常なリターン値であっても終端文字分のバイトは含みません。

リターン値が n のとき，配列はナル文字で終わっていません。

wcstombs

ワイド文字列を多バイト文字列に変換します。

[指定形式]

#include <stdlib.h>

size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);

[引数]

s 多バイト文字列へのポインタ

pwcsワイド文字列へのポインタ

n 多バイト文字列へ書き込むバイト数

[戻り値]

正常： 多バイト文字列へ書き込まれたバイト数

異常： (size_t)(-1)不正な多バイト文字の並びに遭遇した場合

[備考]

wcstombs 関数は、pwcs が指す配列中のワイド文字の列を、初期シフト状態から始まる対応する多バイト文字の並びに変換し，s が指す配列に格納します。ただし，多バイト文字が合計で n バイトの上限を超えるとき，又はナル文字が格納されたとき，配列への格納を終了します。各ワイド文字は，wctomb 関数の変換状態が影響を受けないことを除いて，wctomb 関数の呼出しによる場合と同じ規則で変換します。

領域の重なり合うオブジェクト間でコピーが行われた場合の動作は未定義とします。

正常なリターン値であっても終端文字分のバイトは含みません。

リターン値が n のとき，配列はナル文字で終わっていません。

6. 4. 12 < string.h>

文字配列の操作に必要な種々の関数を定義します。

種別	定義名	説明
関数	memcpy	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
	strcpy	複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。
	strncpy	複写元の文字列を指定された文字数分、複写先の記憶域に複写します。
	strcat	文字列の後に、文字列を連結します。
	strncat	文字列に文字列を指定した文字数分、連結します。
	memcmp	指定された 2 つの記憶域の比較を行います。
	strcmp	指定された 2 つの文字列を比較します。
	strncmp	指定された 2 つの文字列を指定された文字数分まで比較します。
	memchr	指定された記憶域において、指定された文字が最初に現われる位置を検索します。
	strchr	指定された文字列において、指定された文字が最初に現われる位置を検索します。
	strcspn	指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。
	strpbrk	指定された文字列において、別に指定された文字列中の文字が最初に現われる位置を検索します。
	strrchr	指定された文字列において指定された文字が最後に現われる位置を検索します。
	strspn	指定された文字列を先頭から調べ別に指定した文字列中の文字が先頭から何文字続くかを求めます。
	strstr	指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。
	strtok	指定した文字列をいくつかの字句に切り分けます。
	memset	指定された記憶域の先頭から指定された文字を指定された文字数分設定します。
	strerror	エラーメッセージを設定します。
	strlen	文字列の文字数を計算します。
	memmove	複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。複写元と複写先の記憶域が重なっていても、正しく複写されます。

処理系定義仕様

	項目	コンパイラの仕様
1	strerror 関数が返すエラーメッセージの内容	「11.3 C 標準ライブラリ関数のエラーメッセージ」を参照してください。

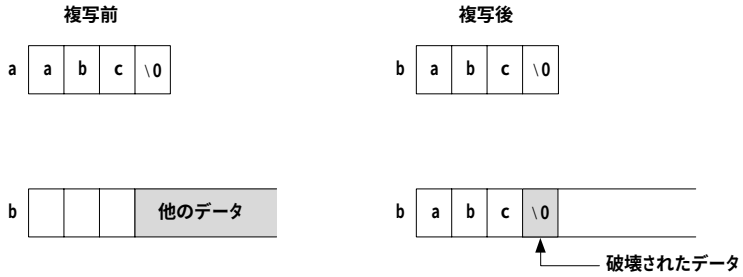
本標準インクルードファイル内で定義されている関数を使用する時は、以下の 2 つの事項に注意する必要があります。

(1) 文字列の複写を行う時、複写先の領域が複写元の領域よりも小さい場合、動作は保証しませんので注意が必要です。

例

```
char a[]="abc";
char b[3];
  :
  :
strcpy(b,a);
```

この場合、配列 a のサイズは (ヌル文字を含めて)4 バイトです。したがって、strcpy 関数によって複写を行うと、配列 b の領域以外のデータを書き換えることになります。

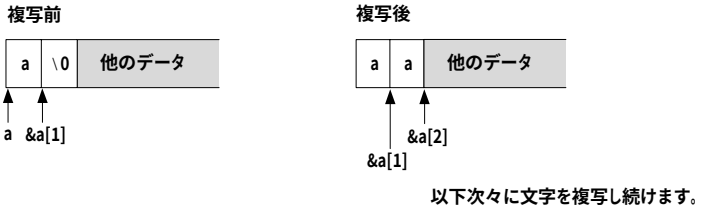


(2) 文字列の複写を行う時、複写元の領域と複写先の領域が重なっていると正しい動作は保証しませんので注意が必要です。

例

```
int a[]="a";
  :
  :
strcpy(&a[1], a);
  :
  :
```

この場合、複写元の文字列がヌル文字に達する以前に、ヌル文字の上に文字 'a' を書き込むことになります。したがって、複写元の文字列のデータに続くデータを書き換えることになります。



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。

[**指定形式**]

#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);

[**引数**]

s1 **複写先の記憶域へのポインタ**

s2 **複写元の記憶域へのポインタ**

n **複写する文字数**

[**戻り値**]

s1 **の値**

[**備考**]

－

strcpy

複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。

[**指定形式**]

#include <string.h>

char *strcpy(char *s1, const char *s2);

[**引数**]

s1 **複写先の記憶域へのポインタ**

s2 **複写元の文字列へのポインタ**

[**戻り値**]

s1 **の値**

[**備考**]

－

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。

[**指定形式**]

#include <string.h>

char *strncpy(char *s1, const char *s2, size_t n);

[**引数**]

s1 **複写先の記憶域へのポインタ**

s2 **複写元の文字列へのポインタ**

n **複写する文字数**

[**戻り値**]

s1 **の値**

[**備考**]

s2 **で指された文字列の先頭から最高** n **文字を** s1 **で指される記憶域に複写します。**s2 **で指定された文字列の文字数が** n **文字より短い時は、**n **文字になるまでヌル文字が付加されます。**

s2 で指された文字列の文字数が n 文字より長い時は、s1 に複写された文字列はヌル文字で終了しないことになります。

strcat

文字列の後に、文字列を連結します。

[指定形式]

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

[引数]

s1 連結される文字列へのポインタ

s2 連結する文字列へのポインタ

[戻り値]

s1 の値

[備考]

s1 で指された文字列の最後に、s2 で指された文字列を連結します。この時、s2 の指す文字列の最後を示すヌル文字も複写します。また、s1 で指された文字列の最後のヌル文字は削除されます。

strncat

文字列に文字列を指定した文字数分連結します。

[指定形式]

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

[引数]

s1 連結される文字列へのポインタ

s2 連結する文字列へのポインタ

n 連結する文字数

[戻り値]

s1 の値

[備考]

s2 で指された文字列の先頭から最高 n 文字を s1 で指された文字列の最後に付加します。s1 で指された文字列の最後のヌル文字は s2 の先頭文字で置き換えられます。

また、連結された後の文字列の最後には、必ずヌル文字が付加されます。

memcmp

指定された 2 つの記憶域の内容を比較します。

[指定形式]

```
#include <string.h>
long memcmp(const void *s1, const void *s2, size_t n);
```

[引数]

s1 比較される記憶域へのポインタ

s2 比較する記憶域へのポインタ

n 比較する記憶域の文字数

[戻り値]

s1 で指された記憶域>s2 で指された記憶域の時：正の値

s1 で指された記憶域==s2 で指された記憶域の時：0

s1 で指された記憶域<s2 で指された記憶域の時：負の値

[備考]

s1 で指された記憶域と s2 で指された記憶域の、最初の n 文字分の内容を比較します。

この比較は処理系定義です。

strcmp

指定された 2 つの文字列の内容を比較します。

[指定形式]

```
#include <string.h>
long strcmp(const char *s1, const char *s2);
```

[引数]

s1 比較される文字列へのポインタ

s2 比較する文字列へのポインタ

[戻り値]

s1 で指された文字列>s2 で指された文字列の時：正の値

s1 で指された文字列==s2 で指された文字列の時：0

s1 で指された文字列<s2 で指された文字列の時：負の値

[備考]

s1 で指された文字列と、s2 で指された文字列の内容を比較し、その結果をリターン値として設定します。

この比較は処理系定義です。

strncmp

指定された 2 つの文字列を指定された文字分まで比較します。

[指定形式]

```
#include <string.h>
long strncmp(const char *s1, const char *s2, size_t n);
```

[引数]

s1 比較される文字列へのポインタ

s2 比較する文字列へのポインタ

n 比較する文字数の最大値

[戻り値]

s1 で指された文字列>s2 で指された文字列の時：正の値

s1 で指された文字列==s2 で指された文字列の時：0

s1 で指された文字列<s2 で指された文字列の時：負の値

[備考]

s1 で指された文字列と、s2 で指された文字列を最初の n 文字以内の範囲で、その内容を比較します。

この比較は処理系定義です。

memchr

指定された記憶域において、指定された文字が最初に現われる位置を検索します。

[指定形式]

#include <string.h>

void *memchr(const void *s, long c, size_t n);

[引数]

s 検索を行う記憶域へのポインタ

c 検索する文字

n 検索を行う文字数

[戻り値]

検索の結果見つかった時: 見つけられた文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

s で指定された記憶域の先頭から n 文字の中で最初に現われた c の文字と同一文字の位置へのポインタをリターン値として返します。

strchr

指定された文字列において、指定された文字が最初に現われる位置を検索します。

[指定形式]

#include <string.h>

char *strchr(const char *s, long c);

[引数]

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時: 見つけられた文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

s で指定された文字列中で最初に現われた c の文字と同一文字へのポインタをリターン値として返します。

s によって指される文字列の終了を現わすヌル文字も検索の対象として含まれます。

strcspn

指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。

[指定形式]

#include <string.h>

size_t strcspn(const char *s1, const char *s2);

[引数]

s1 調べられる文字列へのポインタ

s2 s1 を調べるための文字列へのポインタ

[戻り値]

s2 が指す文字列を構成する文字以外の文字が構成される文字列 s1 の先頭からの長さ

[備考]

s2 が指す文字列を構成する文字以外の文字が、文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の文字数をリターン値として返します。

s2 によって指される文字列の終了を表すヌル文字は、s2 で指された文字列の一部とはみなされません。

strpbrk

指定された文字列内において、別に指定された文字列中の文字が最初に現われる位置を検索します。

[指定形式]

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

[引数]

s1 検索を行う文字列へのポインタ

s2 s1 内で検索する文字を示す文字列へのポインタ

[戻り値]

検索の結果見つかった時: 見つかった文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

s1 で指された文字列において、s2 で指された文字列中の文字の一つが最初に現われる所を検索し、そのポインタをリターン値として返します。

strrchr

指定された文字列において、指定された文字が最後に現われる位置を検索します。

[指定形式]

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

[引数]

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時: 見つかった文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

s で指された文字列の中で c で指定する文字と同一の文字が最後に現われた位置へのポインタをリターン値として返します。

s によって指される文字列の終了を表すヌル文字も検索の対象として含まれます。

strspn

指定された文字列を先頭から調べ、別に指定した文字列中の文字が先頭から何文字続くかを求めます。

[指定形式]

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

[引数]

s1 調べられる文字列へのポインタ

s2 s1 を調べるための文字列へのポインタ

[戻り値]

s1 の先頭から、s2 で指定した文字が続いている文字数

[備考]

s2 が指す文字列を構成する文字が文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の文字数をリターン値として返します。

strstr

指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。

[指定形式]

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

[引数]

s1 検索を行う文字列へのポインタ

s2 検索する文字列へのポインタ

[戻り値]

検索の結果見つかったとき：見つけられた文字へのポインタ

検索の結果見つからなかったとき：NULL

[備考]

s1 で指された文字列において、s2 で指された文字列が最初に現われる所を検索し、そのポインタをリターン値として返します。

strtok

指定した文字列をいくつかの字句に切り分けます。

[指定形式]

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

[引数]

s1 いくつかの字句に切り分ける文字列へのポインタ

s2 文字列を切り分けるための文字からなる文字列へのポインタ

[戻り値]

字句に切り分けられた時：切り分けた字句の先頭へのポインタ

字句に切り分けられなかった時：NULL

[備考]

strtok 関数は文字列を切り分けるために連続的に呼び出されます。

(a) 最初の呼び出し時

s1 で指された文字列を先頭から s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。

(b) 2 回目以降の呼び出し時

以前に切り分けられた字句の次の文字から、s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。

2 回目以降の呼び出しの時は、第 1 引数には NULL を指定します。また、s2 で指された文字列は呼び出しのたびに異なっていてもかまいません。切り出された字句の最後にはヌル文字が付きます。

strtok 関数の使用例を以下に示します。

例

```
1 #include <string.h>
2 static char s1[]="a@b,@c/@d";
3 char *ret;
4
5 ret=strtok(s1,"@");
6 ret=strtok(NULL,",@");
7 ret=strtok(NULL,"/@");
8 ret=strtok(NULL,"@");
```

【説明】

この例は、文字列「"a@b,@c/@d"」を strtok 関数を用いて a,b,c,d という字句に切り分けるプログラムを示しています。

2 行目で文字列 s1 に初期値として、文字列 "a@b,@c/@d" を設定しています。

5 行目では、「@」を区切り文字として字句を切り分けるため、strtok 関数を呼び出します。この結果、文字 'a' へのポインタがリターン値として得られ、文字 'a' の次の最初の区切り文字である「@」にヌル文字を埋め込みます。この結果、文字列 "a" が切り出されます。

以下、同一の文字列から次々に字句を切り出すために第 1 引数に NULL を指定して strtok 関数を呼び出します。

この結果、文字列 "b"、"c"、"d" が次々に切り出されます。

memset

指定された記憶域の先頭から、指定された文字を指定された文字数分設定します。

[指定形式]

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

[引数]

s 文字が設定される記憶域へのポインタ

c 設定する文字

n 設定する文字数

[戻り値]

s の値

[備考]

s で指された記憶域に n 文字分、文字 c を設定します。

strerror

エラー番号を指定して、それに対するエラーメッセージを返します。

[**指定形式**]

#include <string.h>

char *strerror(long s);

[**引数**]

s **エラー番号**

[**戻り値**]

エラー番号に対応するエラーメッセージ (文字列) へのポインタ

[**備考**]

エラー番号 s に対応するエラーメッセージへのポインタをリターン値として返します。

エラーメッセージの内容に関しては処理系定義です。

リターン値として返されたエラーメッセージを修正した時、動作は保証しません。

strlen

文字列の文字数を計算します。

[**指定形式**]

#include <string.h>

size_t strlen(const char *s);

[**引数**]

s **長さを求める文字列へのポインタ**

[**戻り値**]

文字列の文字数

[**備考**]

s が指す文字列の終了を表すヌル文字は、文字列の文字数としては計算に入れません。

memmove

複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。また、複写元と複写先の記憶域が、重なっている部分があっても、複写元の重なっている部分を上書きする前に複写するので正しく複写されます。

[**指定形式**]

#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);

[**引数**]

s1 **複写先の記憶域へのポインタ**

s2 **複写元の記憶域へのポインタ**

n **複写する文字数**

[**戻り値**]

s1 **の値**

[**備考**]

ー

6. 4. 13 < complex.h>

各種の複素数計算を行います。float 型の複素数の場合は、定義名の最後に ’ f ’、long double 型の複素数の場合は、定義名の最後に ’ l ’、double 型の複素数の場合は、定義名が関数名になります。

種別	定義名	説明
関数	cacos	複素数逆余弦を計算します。
	casin	複素数逆正弦を計算します。
	catan	複素数逆正接を計算します。
	ccos	複素数余弦を計算します。
	csin	複素数正弦を計算します。
	ctan	複素数正接を計算します。
	cacosh	複素数逆双曲線余弦を計算します。
	casinh	複素数逆双曲線正弦を計算します。
	catanh	複素数逆双曲線正接を計算します。
	ccosh	複素数双曲線余弦を計算します。
	csinh	複素数双曲線正弦を計算します。
	ctanh	複素数双曲線正接を計算します。
	cexp	複素数自然対数の底 e の z 乗を計算します。
	clog	複素数自然対数を計算します。
	cabs	複素数絶対値を計算します。
	cpow	複素数べき乗を計算します。
	csqrt	複素数平方根を計算します。
	carg	偏角を計算します。
	cimag	虚部を計算します。
	conj	虚部の符号を反転させて複素共役を計算します。
	cproj	リーマン球面上への射影を計算します。
	creal	実部を計算します。

cacosf/ cacos/ cacosl

複素数逆余弦を計算します。

[**指定形式**]

```
#include <complex.h>
float complex cacosf(float complex z);
double complex cacos(double complex z);
long double complex cacosl(long double complex z);
```

[**引数**]

z **複素数逆余弦を求める複素数**

[**戻り値**]

正常：z の逆余弦値

異常：定義域エラーの時は、非数を返します

[備考]

z の値が [-1.0,1.0] の範囲を超えている時、定義域エラーになります。

acos 関数のリターン値の実軸方向の範囲は [0,π]、虚軸方向の範囲は無限の区間です。

casinf/ casin/ casinl

複素数逆正弦を計算します。

[指定形式]

```
#include <complex.h>
float complex casinf(float complex z);
double complex casin(double complex z);
long double complex casinl(long double complex z);
```

[引数]

z 複素数逆正弦を求める複素数

[戻り値]

正常：z の複素数逆正弦値

異常：定義域エラーの時は、非数を返します

[備考]

z の値が [-1.0,1.0] の範囲を超えている時、定義域エラーになります。

casin 関数のリターン値の実軸方向の範囲は [-π/2,π/2]、虚軸方向の範囲は無限の空間です。

catanf/ catan/ catanl

複素数逆正接を計算します。

[指定形式]

```
#include <complex.h>
float complex catanf(float complex z);
double complex catan(double complex z);
long double complex catanl(long double complex z);
```

[引数]

z 複素数逆正接を求める複素数

[戻り値]

正常：z の複素数逆正接値

[備考]

catan 関数のリターン値の実軸方向の範囲は [-π/2,π/2]、虚軸方向の範囲は無限の空間です。

ccosf/ ccos/ ccosl

複素数余弦を計算します。

[指定形式]

```
#include <complex.h>
float complex ccosf(float complex z);
double complex ccos(double complex z);
long double complex ccosl(long double complex z);
```

[引数]

z 複素数余弦を求める複素数

[戻り値]

z の複素数余弦値

[備考]

―

csinf/ csin/ csinl

複素数正弦を計算します。

[指定形式]

#include <complex.h>

float complex csinf(float complex z);

double complex csin(double complex z);

long double complex csinl(long double complex z);

[引数]

z 複素数正弦を求める複素数

[戻り値]

z の複素数正弦値

[備考]

―

ctanf/ ctan/ ctanl

複素数正接を計算します。

[指定形式]

#include <complex.h>

float complex ctanf(float complex z);

double complex ctan(double complex z) ;

long double complex ctanl(long double complex z);

[引数]

z 複素数正接を求める複素数

[戻り値]

z の複素数正接値

[備考]

―

cacoshf/ cacosh/ cacoshl

複素数逆双曲線余弦を計算します。

[指定形式]

#include <complex.h>

float complex cacoshf(float complex z);

double complex cacosh(double complex z);

long double complex cacoshl(long double complex z);

[引数]

z 複素数逆双曲線余弦を求める複素数

[戻り値]

正常： z の複素数逆双曲線余弦値

異常： 定義域エラーの時は、非数を返します。

[備考]

z の値が [-1.0, 1.0] の範囲を超えている時、定義域エラーになります。

cacoshf 関数群のリターン値の範囲は [0, π] です。

casinhf/ casinh/ casinhl

複素数逆双曲線正弦を計算します。

[指定形式]

#include <complex.h>

float complex casinhf(float complex z);

double complex casinh(double complex z);

long double complex casinhl(long double complex z);

[引数]

z 複素数逆双曲線正弦を求める複素数

[戻り値]

z の複素数逆双曲線正弦値

[備考]

―

catanhf/ catanh/ catanhl

複素数逆双曲線正接を計算します。

[指定形式]

#include <complex.h>

float complex catanhf(float complex z);

double complex catanh(double complex z);

long double complex catanhl(long double complex z);

[引数]

z 複素数逆双曲線正接を求める複素数

[戻り値]

z の複素数逆双曲線正接値

[備考]

―

ccoshf/ ccosh/ ccoshl

複素数双曲線余弦を計算します。

[指定形式]

#include <complex.h>

float complex ccoshf(float complex z);

double complex ccosh(double complex z);

long double complex ccoshl(long double complex z);

[**引数**]

z **双曲線余弦を求める複素数**

[**戻り値**]

z **の複素数双曲線余弦値**

[**備考**]

－

csinhf/ csinh/ csinhl

複素数双曲線正弦を計算します。

[**指定形式**]

#include <complex.h>

float complex csinhf(float complex z);

double complex csinh(double complex z);

long double complex csinhl(long double complex z);

[**引数**]

z **双曲線正弦を求める複素数**

[**戻り値**]

z **の複素数双曲線正弦値**

[**備考**]

－

ctanhf/ ctanh/ ctanhl

複素数双曲線正接を計算します。

[**指定形式**]

#include <complex.h>

float complex ctanhf(float complex z);

double complex ctanh(double complex z);

long double complex ctanhl(long double complex z);

[**引数**]

z **双曲線正接を求める複素数**

[**戻り値**]

z **の複素数双曲線正接値**

[**備考**]

－

cexpf/ cexp/ cexpl

複素数の指数関数を計算します。

[**指定形式**]

#include <complex.h>

float complex cexpf(float complex z);

double complex cexp(double complex z);

long double complex cexpl(long double complex z);

[引数]

z 指数関数を求める複素数

[戻り値]

z の指数関数値

[備考]

ー

clogf/ clog/ clogl

複素数の自然対数を計算します。

[指定形式]

#include <complex.h>

float complex clogf(float complex z);

double complex clog(double complex z);

long double complex clogl(long double complex z);

[引数]

z 複素数自然対数を求める複素数

[戻り値]

正常： z の複素数自然対数値

異常： 定義域エラーの時は、非数を返します。

[備考]

z の値が負の時、定義域エラーになります。

z の値が 0.0 の時、範囲エラーになります。

clog 関数群のリターン値の実軸方向の範囲は無限の区間、虚軸方向の範囲は [-iπ,+iπ] です。

cabsf/ cabs/ cabsl

複素数絶対値を計算します。

[指定形式]

#include <complex.h>

float cabsf(float complex z);

double cabs(double complex z);

long double cabsl(long double complex z);

[引数]

z 複素数絶対値を求める複素数

[戻り値]

z の複素数絶対値

[備考]

ー

cpowf/ cpow/ cpowl

複素数べき乗を計算します。

[指定形式]

#include <complex.h>

float complex cpowf(float complex x, float complex y);

double complex cpow(double complex x, double complex y);

long double complex cpowl(long double complex x, long double complex y);

[引数]

x べき乗される値

y べき乗する値

[戻り値]

正常： x の y 乗の値

異常： 定義域エラーの時は、非数を返します。

[備考]

x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

cpow 関数群の第 1 仮引数に対する分岐切断線は負の実軸に沿っています。

csqrtf/ csqrt/ csqrtl

複素数平方根を計算します。

[指定形式]

#include <complex.h>

float complex csqrtf(float complex z);

double complex csqrt(double complex z);

long double complex csqrtl(long double complex z);

[引数]

z 平方根関数値を求める複素数

[戻り値]

正常： z の複素数平方根値

異常： 定義域エラーの時は、非数を返します。

[備考]

z の値が負の値の時、定義域エラーになります。

csqrt 関数群の分岐分断線は負の実軸に沿っています。

csqrt 関数群のリターン値の領域は虚軸を含む右半平面です。

cargf/ carg/ cargl

偏角を計算します。

[指定形式]

#include <complex.h>

float cargf(float complex z);

double carg(double complex z);

long double cargl(long double complex z);

[引数]

z 偏角値を求める複素数

[戻り値]

z の偏角値

[備考]

carg 関数群の分岐切断線はは負の実軸に沿っています。

carg 関数群のリターン値の範囲は区間 [-π,+π] です。

cimagf/ cimag/ cimagl

虚部を計算します。

[指定形式]

```
#include <complex.h>
float cimagf(float complex z);
double cimag(double complex z);
long double cimagl(long double complex z);
```

[引数]

z 虚部を求める複素数

[戻り値]

実数としての z の虚部値

[備考]

―

conjf/ conj/ conjl

虚部の符号を反転させて複素共役を計算します。

[指定形式]

```
#include <complex.h>
float complex conjf(float complex z);
double complex conj(double complex z);
long double complex conjl(long double complex z);
```

[引数]

z 複素共役値を求める複素数

[戻り値]

z の複素共役値

[備考]

―

cprojf/ cproj/ cprojl

リーマン球面上への射影を計算します。

[指定形式]

```
#include <complex.h>
float complex cprojf(float complex z);
double complex cproj(double complex z);
long double complex cprojl(long double complex z);
```

[引数]

z リーマン球面上への射影値を求める複素数

[戻り値]

リーマン球面上への z の射影値

[備考]

ー

crealf/ creal/ creall

実部を計算します。

[指定形式]

```
#include <complex.h>
float crealf(float complex z);
double creal(double complex z);
long double creall(long double complex z);
```

[引数]

z 実部値を求める複素数

[戻り値]

z の実部値

[備考]

ー

6. 4. 14 < fenv.h>

浮動小数点環境へアクセスします。

以下は、すべて処理系定義です。

種別	定義名	説明
型 (マクロ)	fenv_t	浮動小数点環境全体の型です。
	fexcept_t	浮動小数点状態フラグの型です。
定数 (マクロ)	FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW FE_UNDERFLOW FE_ALL_EXCEPT	浮動小数点例外をサポートするときに定義されるマクロです。
定数 (マクロ)	FE_DOWNWARD FE_TONEAREST FE_TOWARDZERO FE_UPWARD	浮動小数点数の丸め方向のマクロです。
定数 (マクロ)	FE_DFL_ENV	プログラム既定の浮動小数点環境です。
関数	feclearexcept	浮動小数点例外のクリアを試みます。
	fegetexceptflag	浮動小数点フラグの状態のオブジェクトへの格納を試みます。
	feraiseexcept	浮動小数点例外の生成を試みます。
	fesetexceptflag	浮動小数点フラグのセットを試みます。
	fetestexcept	浮動小数点フラグがセットされているか確認します。
	fegetround	丸め方向を取得します。
	fesetround	丸め方向を設定します。
	fegetenv	浮動小数点環境の取得を試みます。
	feholdexcept	浮動小数点環境を保存し、浮動小数点状態フラグをクリアし、浮動小数点例外について無停止モードに設定します。
	fesetenv	浮動小数点環境の設定を試みます。
	feupdateenv	浮動小数点例外の自動記憶域への保存、浮動小数点環境の設定、保存していた浮動小数点例外の生成を試みます。

feclearexcept

浮動小数点例外のクリアを試みます。

[指定形式]

```
#include <fenv.h>
long feclearexcept(long e);
```

[引数]

e 浮動小数点例外

[戻り値]

正常： 0

異常： 0 以外

[備考]

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。
使用してもリターン値として異常を表す 0 以外を返します。

fegetexceptflag

例外フラグの状態を取得します。

[指定形式]

```
#include <fenv.h>
long fegetexceptflag(fexcept_t *f, long e);
```

[引数]

f 例外フラグ状態の格納先を指すポインタ

e 状態を取得する例外フラグを表す値

[戻り値]

正常： 0

異常： 0 以外

[備考]

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。
使用してもリターン値として異常を表す 0 以外を返します。

feraiseexcept

浮動小数点例外の生成を試みます。

[指定形式]

```
#include <fenv.h>
long feraiseexcept(long e);
```

[引数]

e 生成を試みる例外を指す値

[戻り値]

正常： 0

異常： 0 以外

[備考]

feraiseexcept 関数が，“ オーバフロー ” 浮動小数点例外又は “ アンダフロー ” 浮動小数点例外を生成する際に “ 不正確結果 ” 浮動小数点例外を生成するかどうかは，処理系定義とします。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。
使用してもリターン値として異常を表す 0 以外を返します。

fesetexceptflag

例外フラグの状態を設定します。

[指定形式]

#include <fenv.h>

long fesetexceptflag(const fexcept_t *f, long e);

[引数]

f 例外フラグ状態の取得元を指すポインタ

e 状態を設定する例外フラグを表す値

[戻り値]

正常： 0

異常： 0 以外

[備考]

fesetexceptflag 関数を呼ぶ前にフラグ状態の取得元を fegetexceptflag 関数にて設定してください。

fesetexceptflag 関数は浮動小数点例外を生成せず、フラグの状態だけを設定します。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

fetestexcept

例外フラグの状態を判定します。

[指定形式]

#include <fenv.h>

long fetestexcept(long e);

[引数]

e 状態を判定するフラグ (複数可) を表す値

[戻り値]

e と浮動小数点例外マクロのビット単位の論理和

[備考]

fetestexcept 関数は１回の呼び出しで複数個の浮動小数点例外を判定することができます。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

fegetround

その時点の丸め方向を取得します。

[指定形式]

#include <fenv.h>

long fegetround(void);

[引数]

－

[戻り値]

正常： 0

異常： 丸め方向マクロ値が存在しない
又は丸め方向を決めることができない場合は負の値

[備考]

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

fesetround

その時点の丸め方向を設定します。

[指定形式]

#include <fenv.h>

#include <assert.h>

long fesetround(long rnd);

[引数]

ー

[戻り値]

成功した場合のみに 0

[備考]

fesetround 関数に丸め方向マクロの値と等しくない変更を要求した場合

丸め方向を変更しません。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

fegetenv

浮動小数点環境を取得します。

[指定形式]

#include <fenv.h>

long fegetenv(fenv_t *f);

[引数]

f 浮動小数点環境格納先を指すポインタ

[戻り値]

正常： 0

異常： 0 以外

[備考]

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

feholdexcept

浮動小数点環境を保存します。

[指定形式]

#include <fenv.h>

long feholdexcept(fenv_t *f);

[引数]

f 浮動小数点環境を指すポインタ

[戻り値]

成功した場合のみに 0

[備考]

feholdexcept 関数は浮動小数点関数環境保存時に浮動小数点状態フラグをクリアし、すべての浮動小数点例外について、無停止 (non-stop) モードを設定します。無停止モード設定時は浮動小数点例外発生時も実行を継続します。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

fesetenv

浮動小数点環境を設定します。

[指定形式]

```
#include <fenv.h>
long fesetenv(const fenv_t *f);
```

[引数]

f　浮動小数点環境を指すポインタ

[戻り値]

正常：　0

異常：　0 以外

[備考]

設定する環境は fegetenv 関数または feholdexcept 関数にて設定した環境か、浮動小数点環境マクロと等しい環境を指定していください。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

feupdateenv

既出の例外を残したまま浮動小数点環境を設定します。

[指定形式]

```
#include <fenv.h>
long feupdateenv(const fenv_t *f);
```

[引数]

f　設定する浮動小数点環境を指すポインタ

[戻り値]

正常：　0

異常：　0 以外

[備考]

設定する浮動小数点環境は、fegetenv 関数または feholdexcept 関数の呼出しによって設定されたオブジェクトを指すか，又は浮動小数点環境マクロに等しいものにしてください。

本関数は、コンパイルオプション nofpu が選択されている場合は使用しないでください。

使用してもリターン値として異常を表す 0 以外を返します。

6. 4. 15 < inttypes.h>

整数型を拡張します。

以下は、すべて処理系定義です。

種別	定義名	説明
型 (マクロ)	Imaxdiv_t	imaxdiv 関数の返す値の型です。
変数 (マクロ)	PRIdN PRIdLEASTN PRIdFASTN PRIdMAX PRIdPTR PRIiN PRIiLEASTN PRIiFASTN PRIiMAX PRIiPTR PRIoN PRIoLEASTN PRIoFASTN PRIoMAX PRIoPTR PRIuN PRIuLEASTN PRIuFASTN PRIuMAX PRIuPTR PRIxN PRIxLEASTN PRIxFASTN PRIxMAX PRIxPTR PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXPTR	

種別	定義名	説明
変数 (マクロ)	SCNdN SCNdLEASTN SCNdFASTN SCNdMAX SCNdPTR SCNiN SCNiLEASTN SCNiFASTN SCNiMAX SCNiPTR SCNoN SCNoLEASTN SCNoFASTN SCNoMAX SCNoPTR SCNuN SCNuLEASTN SCNuFASTN SCNuMAX SCNuPTR SCNxN SCNxLEASTN SCNxFASTN SCNxMAX SCNxPTR	
関数	imaxabs	絶対値を計算する。
	imaxdiv	商、剰余を計算する。
	strtoimax strtoumax	文字列最初の部分を intmax_t 型および uintmax_t 型表現に変換する以外は、strtol, strtoll, strtoul および strtoull 関数と等価。
	wcstoimax wcstoumax	ワイド文字列最初の部分を intmax_t 型および uintmax_t 型表現に変換する以外は、wcstol, wcstoll, wcstoul および wcstoull 関数と等価。

imaxabs

絶対値を計算します。

[**指定形式**]

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t a);
```

[**引数**]

a **絶対値を求める値**

[**戻り値**]

a の絶対値

[備考]

―

imaxdiv

除算を行います。

[指定形式]

#include <inttypes.h>

intmaxdiv_t imaxdiv(intmax_t n, intmax_t d);

[引数]

n 除算をする値

d

[戻り値]

商と剰余から成る除算結果

[備考]

―

strtoimax/ strtoumax

数を表現する文字列を intmax_t 型の整数に変換します。

[指定形式]

#include <inttypes.h>

intmax_t strtoimax(const char *nptr, char **endptr, long base);

uintmax_t strtoumax(const char *nptr, char **endptr, long base);

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時: 変換された intmax_t 型の整数値

異常：変換後の値がオーバフローの時：INTMAX_MAX, INTMAX_MIN または UINTMAX_MAX

[備考]

変換後の値がオーバフローをおこした時は、errno に ERANGE を設定します。

strtoimax 関数及び strtoumax 関数は文字列の最初の部分をそれぞれ intmax_t 型および uintmax_t 型整数に変換するという点を除いて、strtol 関数、strtoll 関数、strtoul 関数及び strtoull 関数と等価とします。

wcstoimax/ wcstoumax

数を表現する文字列を intmax_t 型または uintmax_t 型の整数に変換します。

[指定形式]

#include <stddef.h>

#include <inttypes.h>

intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);

uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時: 変換された intmax_t 型の整数値

異常：変換後の値がオーバフローの時：INTMAX_MAX, INTMAX_MIN または UINTMAX_MAX

[備考]

変換後の値がオーバフローをおこした時は、errno に ERANGE を設定します。

wcstrtoimax 関数及び wcstrtoumax 関数は文字列の最初の部分をそれぞれ intmax_t 型および uintmax_t 型整数に変換するという点を除いて、wcstol 関数、wcstoll 関数、wcstoul 関数及び wcstoull 関数と等価とします。

6. 4. 16 < iso646.h>

以下は、すべてマクロ定義です。

種別	定義名	説明
マクロ	and	&&
	and_eq	&=
	bitand	&
	bitor	|
	compl	~
	not	!
	not_eq	!=
	or	||
	or_eq	|=
	xor	^
	xor_eq	^=

6. 4. 17 < stdbool.h>

以下は、すべてマクロ定義です。

種別	定義名	説明
マクロ（変数）	bool	_Bool に展開します。
マクロ（定数）	true	1 に展開します。
	false	0 に展開します。
	__bool_true_false_are_defined	1 に展開します。

6. 4. 18 < stdint.h>

以下は、すべてマクロ定義です。

種別	定義名	説明
マクロ	int_least8_t uint_least8_t int_least16_t uint_least16_t int_least32_t uint_least32_t int_least64_t uint_least64_t	8,16,32 および 64 ビットに対する、それぞれの符号あり / なし整数型を少なくとも格納できる大きさを持つ型です。
	int_fast8_t uint_fast8_t int_fast16_t uint_fast16_t int_fast32_t uint_fast32_t int_fast64_t uint_fast64_t	8,16,32 および 64 ビットに対する、それぞれの符号あり / なし整数型を最速で演算できる型です。
	intptr_t uintptr_t	void へのポインタを相互変換可能な符号あり / なし整数型です。
	intmax_t uintmax_t	すべての符号あり / なし整数型のすべての値を表現可能な符号あり / なし整数型です。
	int*N*_t uint*N*_t	N ビットの幅をもつ符号あり / なし整数型です。
	INT*N*_MIN INT*N*_MAX UINT*N*_MAX	幅指定符号あり整数型の最小値です。 幅指定符号あり整数型の最大値です。 幅指定符号なし整数型の最大値です。
	INT_LEAST*N*_MIN INT_LEAST*N*_MAX UINT_LEAST*N*_MAX	最小幅指定符号あり整数型の最小値です。 最小幅指定符号あり整数型の最大値です。 最小幅指定符号なし整数型の最大値です。
	INT_FAST*N*_MIN INT_FAST*N*_MAX UINT_FAST*N*_MAX	最速最小幅指定符号あり整数型の最小値です。 最速最小幅指定符号あり整数型の最大値です。 最速最小幅指定符号なし整数型の最大値です。
	INTPTR_MIN INTPTR_MAX UINTPTR_MAX	ポインタ保持可能な符号あり整数型の最小値です。 ポインタ保持可能な符号あり整数型の最大値です。 ポインタ保持可能な符号なし整数型の最大値です。

種別	定義名	説明
マクロ	INTMAX_MIN INTMAX_MAX UINTMAX_MAX	最大幅符号あり整数型の最小値です。 最大幅符号あり整数型の最大値です。 最大幅符号なし整数型の最大値です。
	PTRDIFF_MIN PTRDIFF_MAX	-65535 +65535
	SIG_ATOMIC_MIN SIG_ATOMIC_MAX	-127 +127
	SIZE_MAX	65535
	WCHAR_MIN WCHAR_MAX	0 65535U
	WINT_MIN WINT_MAX	0 4294967295U
関数 (マクロ)	INT*N*_C UINT*N*_C	Int_least*N*_t に対応する整数定数式に展開します。 uInt_least*N*_t に対応する整数定数式に展開します。
	INT_MAX_C UINT_MAX_C	intmax_t の整数定数式に展開します。 uintmax_t の整数定数式に展開します。

6. 4. 19 < tgmath.h>

以下は、すべてマクロ定義です。

型総称マクロ	<math.h> の関数	<complex.h> の関数
acos	acos	cacos
asin	asin	casin
atan	atan	catan
acosh	acosh	cacosh
asinh	asinh	casinh
atanh	atanh	catanh
cos	cos	ccos
sin	sin	csin
tan	tan	ctan
cosh	cosh	ccosh
sinh	sinh	csinh
tanh	tanh	ctanh
exp	exp	cexp
log	log	clog
pow	pow	cpow
sqrt	sqrt	csqrt
fabs	fabs	cfabs
atan2	atan2	-
cbrt	cbrt	-
ceil	ceil	-
copysign	copysign	-
erf	erf	-
erfc	erfc	-
exp2	exp2	-
expm1	expm1	-
fdim	fdim	-
floor	floor	-
fma	fma	-
fmax	fmax	-
fmin	fmin	-
fmod	fmod	-
frexp	frexp	-
hypot	hypot	-
ilogb	ilogb	-

型総称マクロ	<math.h> の関数	<complex.h> の関数
ldexp	ldexp	-
lgamma	lgamma	-
llrint	llrint	-
llround	llround	-
log10	log10	-
log1p	log1p	-
log2	log2	-
logb	logb	-
lrint	lrint	-
lround	lround	-
nearbyint	nearbyint	-
nextafter	nextafter	-
nexttoward	nexttoward	-
remainder	remainder	-
remquo	remquo	-
rint	rint	-
round	round	-
scalbn	scalbn	-
scalbln	scalbln	-
tgamma	tgamma	-
trunc	trunc	-
carg	-	carg
cimag	-	cimag
conj	-	conj
cproj	-	cproj
creal		creal

6. 4. 20 < wchar.h>

以下は、すべてマクロ定義です。

種別	定義名	説明
マクロ	mbstate_t	多バイト文字の並びとワイド文字の並びの間に必要な変換の状態を保持する型です。
	wint_t	拡張文字を保持する型です。
定数（マクロ）	WEOF	ファイルの終わりを表します。

種別	定義名	説明
関数	fwprintf	出力形式を変換して、ストリームへ出力します。
	vfwprintf	可変個数の実引数並びを va_list で置き換えた fwprintf と等価です。
	swprintf	出力形式を変換してワイド文字の配列に書き込みます。
	vswprintf	可変個数の実引数並びを va_list で置き換えた swprintf と等価です。
	wprintf	与えられた実引数の前に stdout を実引数として付加した fwprintf と等価です。
	vwprintf	可変個数の実引数並びを va_list で置き換えた wprintf と等価です。
	fwscanf	ワイド文字列の制御に従ってストリームから入力して変換し、オブジェクトに代入します。
	vfwscanf	可変個数の実引数並びを va_list で置き換えた fwscanf と等価です。
	swscanf	ワイド文字列の制御に従って変換し、オブジェクトに代入します。
	vswscanf	可変個数の実引数並びを va_list で置き換えた swscanf と等価です。
	wscanf	与えられた実引数の前に stdin を実引数として付加した fwscanf と等価です。
	vwscanf	可変個数の実引数並びを va_list で置き換えた wscanf と等価です。
	fgetwc	wchar_t 型として取り込み wint_t 型に変換します。
	fgetws	ワイド文字の列を配列に格納します。
	fputwc	ワイド文字を書き込みます。
	fputws	ワイド文字列を書き込みます。
	fwide	入出力の単位を設定します。
	getwc	fgetwc と等価です。
	getwchar	実引数に stdin を指定した getwc と等価です。
	putwc	fputwc と等価です。
	putwchar	第２引数に stdout を指定した putwc と等価です。
	ungetwc	ワイド文字をストリームに戻します。
	wcstod wcstof wcstold	ワイド文字列の最初の部分を double, float および long double 型の表現に変換します。

種別	定義名	説明
関数	wcstol wcstoll wcstoul wcstoull	ワイド文字列の最初の部分を long int, long long int, unsigned long int および unsigned long long int 型の表現に変換します。
	wcscpy	ワイド文字列をコピーします。
	wcsncpy	n 個以下のワイド文字をコピーします。
	wmemcpy	n ワイド文字をコピーします。
	wmemmove	n ワイド文字をコピーします。
	wcscat	ワイド文字列をコピーし、ワイド文字列の最後に付加します。
	wcsncat	n 個以下のワイド文字列をコピーし、ワイド文字列の最後に付加します。
	wcscmp	ワイド文字列同士を比較します。
	wcsncmp	n ワイド文字以下の配列を比較します。
	wmemcmp	n ワイド文字を比較します。
	wcschr	ワイド文字列の中でワイド文字を検索します。
	wcscspn	ワイド文字列の中で、ワイド文字列が含まれているかを検索します。
	wcspbrk	ワイド文字列の中で、ワイド文字列が含まれている最初の位置を検索します。
	wcsrchr	ワイド文字列の中でワイド文字が最後に現れる位置を検索します。
	wcsspn	ワイド文字列の中から、ワイド文字を含む先頭部分の最大の長さを計算します。
	wcsstr	ワイド文字列の中からワイド文字の並びをが最初に現れる位置を検索します。
	wcstok	ワイド文字列をワイド文字で区切られる字句の列に分割します。
	wmemchr	オブジェクトの先頭から n ワイド文字の中でワイド文字が最初に現れる位置を検索します。
	wcslen	ワイド文字列の長さを計算します。
	wmemset	n ワイド文字をコピーします。
	wctob	多バイト文字表現が１バイトに可能か判定します。
	mbsinit	初期変換状態か判定します。
	mbrlen	多バイト文字を構成するバイト数を計算します。
	mbrtowc	多バイト文字をワイド文字に変換します。
	wcrtomb	ワイド文字を多バイト文字に変換します。
	mbsrtowcs	多バイト文字の並びを対応するワイド文字の並びに変換します。
	wcsrtombs	ワイド文字の並びを対応する多バイト文字の並びに変換します。

fwprintf

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。

[**指定形式**]

#include <stdio.h>

#include <wchar.h>

long fwprintf(FILE *restrict fp, const wchar_t *restrict control [, arg] …);

[**引数**]

fp **ファイルポインタ**

control **書式を示すワイド文字列へのポインタ**

arg,... **書式に従って出力されるデータの並び**

[**戻り値**]

正常： 変換し出力したワイド文字列数

異常： 負の値

[**備考**]

fwprintf **関数は** fprintf **関数のワイド文字対応版です。**

vfwprintf

可変個の引数リストを書式に従って、指定したストリーム入出力用ファイルに出力します。

[**指定形式**]

#include <stdarg.h>

#include <stdio.h>

#include <wchar.h>

long vfwprintf(FILE *restrict fp, const char *restrict control, va_list arg);

[**引数**]

fp **ファイルポインタ**

control **書式を示すワイド文字列へのポインタ**

arg **引数リスト**

[**戻り値**]

正常：変換し出力した文字数

異常：負の値

[**備考**]

vfwprintf **関数は** vfprintf **関数のワイド文字対応版です。**

swprintf

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。

[**指定形式**]

#include <stdio.h>

#include <wchar.h>

long swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control [, arg] …)

[**引数**]

s **データを出力する記憶域へのポインタ**

n 出力するワイド文字数

control 書式を示すワイド文字列へのポインタ

arg,... 書式に従って出力されるデータ

[戻り値]

正常： 変換した文字数

異常： 表現形式エラー又は

n 個以上のワイド文字の書き込みが要求された場合は負の値

[備考]

mbrtowc() 関数に不正な多バイト文字列を渡した場合、表現形式エラーが発生します。

swprintf 関数は sprintf 関数のワイド文字対応版です。

vswprintf

可変個の引数リストを書式に従って、指定した記憶域に出力します。

[指定形式]

```
#include <stdarg.h>
#include <wchar.h>
long vswprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict control, va_list arg)
```

[引数]

s データを出力する記憶域へのポインタ

n 出力するワイド文字数

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：変換した文字数

異常：負の数

[備考]

vswprintf 関数は、vsprintf 関数のワイド文字対応版です。

wprintf

データを書式に従って変換し、標準出力ファイル (stdout) へ出力します。

[指定形式]

```
#include <stdio.h>
    #include <wchar.h>
long wprintf(const wchar_t *restrict control [, arg] …);
```

[引数]

control 書式を示す文字列へのポインタ

arg,... 書式に従って出力されるデータ

[戻り値]

正常：変換し出力したワイド文字数

異常：負の値

[備考]

wprintf 関数は printf 関数のワイド文字対応版です。

vwprintf

可変個の引数リストを書式に従って標準出力ファイル (stdout) に出力します。

[指定形式]

```
#include <stdarg.h>
#include <wchar.h>
long vwprintf(const wchar_t *restrict control, va_list arg);
```

[引数]

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：変換し出力した文字数

異常：負の値

[備考]

vwprintf 関数は vprintf 関数のワイド文字対応版です。

fwscanf

ストリーム入出力ファイルからデータを入力し、書式に従って変換します。

[指定形式]

```
#include <stdio.h>
#include <wchar.h>
long fwscanf(FILE *restrict fp, const wchar_t *restrict control [, ptr] ,,,);
```

[引数]

fp ファイルポインタ

control 書式を示すワイド文字列へのポインタ

ptr 入力したデータを格納する記憶域へのポインタ

[戻り値]

正常： 入力変換に成功したデータの個数

異常： 入力データの変換を行う前に入力データが終了した時：EOF

[備考]

fwscanf 関数は、fscanf 関数のワイド文字対応版です。

vfwscanf

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。

[指定形式]

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
long vfwscanf(FILE *restrict fp, const wchar_t *restrict control , va_list arg);
```

[引数]

fp ファイルポインタ

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：入力データの変換を行う前に入力データが終了した時：EOF

[備考]

vfwscanf 関数は vfscanf 関数のワイド文字対応版です。

swscanf

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。

[指定形式]

```
#include <stdio.h>
#include <wchar.h>
long swscanf(const wchar_t *restrict s, const wchar_t *restrict control [, ptr] …);
```

[引数]

s 入力するデータがある記憶域

control 書式を示すワイド文字列へのポインタ

ptr,... 入力変換したデータを格納する記憶域へのポインタ

[戻り値]

正常：入力変換に成功したデータの個数

異常：EOF

[備考]

swscanf 関数は sscanf 関数のワイド文字対応版です。

vswscanf

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。

[指定形式]

```
#include <stdarg.h>
#include <wchar.h>
long vswscanf(const wchar_t *restrict s, const wchar_t *restrict control, va_list arg)
```

[引数]

s 入力するデータがある記憶域

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：EOF

[備考]

－

wscanf

標準入力ファイル (stdin) からデータを入力し、書式に従って変換します。

[指定形式]

#include <wchar.h>

long wscanf(const wchar_t *control [, ptr] …);

[引数]

control 書式を示すワイド文字列へのポインタ

ptr,... 入力変換したデータを格納する記憶域へのポインタ

[戻り値]

正常：入力変換に成功したデータの個数

異常：EOF

[備考]

wscanf 関数は scanf 関数のワイド文字対応版です。

vwscanf

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。

[指定形式]

#include <stdarg.h>

#include <wchar.h>

long vwscanf(const wchar_t *restrict control , va_list arg);

[引数]

control 書式を示すワイド文字列へのポインタ

arg 引数リスト

[戻り値]

正常：入力変換に成功したデータの個数

異常：入力データの変換を行う前に入力データが終了した時：EOF

[備考]

vwscanf 関数は、vscanf 関数をワイド文字列の書式を使えるようにした関数です。

fgetwc

ストリーム入出力用ファイルから 1 つのワイド文字を入力します。

[指定形式]

#include <stdio.h>

#include <wchar.h>

wint_t fgetwc(FILE *fp);

[引数]

fp ファイルポインタ

[戻り値]

正常：ファイルの終了の時: EOF

ファイルの終了でない時: 入力したワイド文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してのエラー指示子が設定されます。

fgetwc 関数は fgetc 関数をワイド文字が入力できるようにした関数です。

fgetws

ストリーム入出力用ファイルからワイド文字列を入力します。

[**指定形式**]

#include <stdio.h>

#include <wchar.h>

wchar_t *fgetws(wchar_t *restrict s, long n, FILE *fp);

[**引数**]

s **ワイド文字列を入力する記憶域へのポインタ**

n **ワイド文字列を入力する記憶域のバイト数**

fp **ファイルポインタ**

[**戻り値**]

正常：ファイルの終了の時： NULL

ファイルの終了でない時： s

異常：NULL

[**備考**]

fgetws 関数は fgets 関数をワイド文字列が入力できるように対応した関数です。

fputwc

ストリーム入出力用ファイルへ 1 **つのワイド文字を出力します。**

[**指定形式**]

#include <stdio.h>

#include <wchar.h>

wint_t fputwc(wchar_t c, FILE *fp);

[**引数**]

c **出力する文字**

fp **ファイルポインタ**

[**戻り値**]

正常：出力したワイド文字

異常：EOF

[**備考**]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

fputwc 関数は fputc 関数のワイド文字対応版です。

fputws

ストリーム入出力用ファイルへワイド文字列を出力します。

[**指定形式**]

#include <stdio.h>

#include <wchar.h>

long fputws(const wchar_t *restrict s, FILE *restrict fp);

[**引数**]

s 出力するワイド文字列へのポインタ

fp ファイルポインタ

[戻り値]

正常：0

異常：EOF

[備考]

fputws 関数は fputs 関数のワイド文字対応版です。

fwide

ファイルへの入力単位を設定します。

[指定形式]

```
#include <stdio.h>
#include <wchar.h>
long fwide(FILE *fp, long mode);
```

[引数]

fp ファイルポインタ

mode 入力単位を表す値

[戻り値]

ワイド文字単位が設定された場合は 0 より大きい値

バイト単位の場合は 0 より小さい値

入出力単位をもたない場合は 0

[備考]

fwide 関数はストリーム入出力単位が既に決定されている場合、それを変更しません。

getwc

ストリーム入出力用ファイルから 1 つのワイド文字を入力します。

[指定形式]

```
#include <stdio.h>
#include <wchar.h>
long getwc(FILE *fp);
```

[引数]

fp ファイルポインタ

[戻り値]

正常：ファイルの終了の時: WEOF

ファイルの終了でない時: 入力した文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

getwc 関数は fgetwc と等価ですが、マクロとして実装されているため、fp を 2 回以上評価することがあります。したがって、fp は副作用を伴わない式にしてください。

getwchar

標準入力ファイル (stdin) から、1 つのワイド文字を入力します。

[指定形式]

#include <wchar.h>

long getwchar(void);

[引数]

―

[戻り値]

正常：ファイルの終了の時: WEOF

ファイルの終了でない時: 入力したワイド文字

異常：EOF

[備考]

読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

getwchar 関数は getchar 関数のワイド文字対応版です。

putwc

ストリーム入出力用ファイルへ１つのワイド文字を出力します。

[指定形式]

#include <stdio.h>

#include <wchar.h>

wint_t putwc(wchar_t c, FILE *fp);

[引数]

c 出力するワイド文字

fp ファイルポインタ

[戻り値]

正常：出力したワイド文字

異常：WEOF

[備考]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

putwc 関数は fputwc と等価ですが、マクロとして実装されているため、fp を 2 回以上評価することがあります。したがって、fp は副作用を伴わない式にしてください。

putwchar

標準出力ファイル (stdout) へ 1 つのワイド文字を出力します。

[指定形式]

#include <wchar.h>

wint_t putwchar(wchar_t c);

[引数]

c 出力するワイド文字

[戻り値]

正常：出力したワイド文字

異常：WEOF

[備考]

書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

putwchar 関数は putchar 関数のワイド文字対応版です。

ungetwc

ストリーム入出力用ファイルへ 1 つのワイド文字を戻します。

[指定形式]

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t c, FILE *fp) ;
```

[引数]

c 戻すワイド文字

fp ファイルポインタ

[戻り値]

正常：戻したワイド文字

異常：WEOF

[備考]

ungetwc 関数は、ungetc 関数のワイド文字対応版です。

wcstod/ wcstof/ wcstold

ワイド文字列の最初の部分を所定の型の浮動小数点値に変換します。

[指定形式]

```
#include <wchar.h>
double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
long double wcstold(const wchar_t *restrict nptr, wchar_t **restrict endptr);
```

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 浮動小数点値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

[戻り値]

正常：nptr が指している文字列が浮動小数点型を構成しない文字で始まっている時：0

nptr が指している文字列が浮動小数点型を構成する文字で始まっている時: 変換された型の浮動小数点値

異常：変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつHUGE_VAL, HUGE_VALF, HUGE_VALL

変換後の値がアンダフローの時：0

[備考]

変換後の値がオーバフロー / アンダフローをおこした時は errno を設定します。

wcstod 関数群は strtod 関数群のワイド文字対応版です。

wcstol/ wcstoll/ wcstoul/ wcstoull

ワイド文字列の最初の部分を所定の型の整数値に変換します。

[指定形式]

#include <wchar.h>

long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, long base);

long long int wcstoll(const wchar_t * restrict nptr,wchar_t ** restrict endptr,long base);

unsigned long int wcstoul(const wchar_t * restrict nptr,wchar_t ** restrict endptr,long base);

unsigned long long int wcstoull(const wchar_t * restrict nptr,wchar_t ** restrict endptr,long base);

[引数]

nptr 変換する数を表現する文字列へのポインタ

endptr 整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ

base 変換の基数 (0 又は 2 ～ 36)

[戻り値]

正常：nptr が指している文字列が整数を構成しない文字で始まっている時：0

nptr が指している文字列が整数を構成する文字で始まっている時: 変換された型の整数値

異常：変換後の値がオーバフローの時：変換する文字列の符号に従って LONG_MIN，LONG_MAX，LLONG_MIN，LLONG_MAX，ULONG_MAX 又は ULLONG_MAX

[備考]

変換後の値がオーバフローをおこした時は、errno を設定します。

wcstol 関数群は strtol 関数群のワイド文字対応版です。

wcscpy

複写元のワイド文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。

[指定形式]

#include <wchar.h>

wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);

[引数]

s1 複写先の記憶域へのポインタ

s2 複写元の文字列へのポインタ

[戻り値]

s1 の値

[備考]

wcscpy 関数群は strcpy 関数群のワイド文字対応版です。

wcsncpy

複写元のワイド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。

[指定形式]

#include <wchar.h>

wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);

[引数]

s1 複写先の記憶域へのポインタ

s2 複写元の文字列へのポインタ

n 複写する文字数

[戻り値]

s1 の値

[備考]

wcsnpy 関数は strncpy 関数のワイド文字対応版です。

wmemcpy

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。

[指定形式]

#include <wchar.h>

wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);

[引数]

s1 複写先の記憶域へのポインタ

s2 複写元の記憶域へのポインタ

n 複写する文字数

[戻り値]

s1 の値

[備考]

wmemcpy 関数は memcpy 関数のワイド文字対応版です。

wmemmove

複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。また、複写元と複写先の記憶域が、重なっている部分があっても、複写元の重なっている部分を上書きする前に複写するので正しく複写されます。

[指定形式]

#include <wchar.h>

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);

[引数]

s1 複写先の記憶域へのポインタ

s2 複写元の記憶域へのポインタ

n 複写する文字数

[戻り値]

s1 の値

[備考]

wmemmove 関数は memmove 関数のワイド文字対応版です。

wcscat

文字列の後に、文字列を連結します。

[指定形式]

#include <wchar.h>

wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);

[引数]

s1 連結される文字列へのポインタ

s2 連結する文字列へのポインタ

[戻り値]

s1 の値

[備考]

wcscat 関数は strcat 関数のワイド文字対応版です。

wcsncat

文字列に文字列を指定した文字数分連結します。

[指定形式]

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

[引数]

s1 連結される文字列へのポインタ

s2 連結する文字列へのポインタ

n 連結する文字数

[戻り値]

s1 の値

[備考]

wcsncat 関数は strncat 関数のワイド文字対応版です。

wcscmp

指定された 2 つの文字列の内容を比較します。

[指定形式]

```
#include <wchar.h>
long wcscmp(const wchar_t *s1, const wchar_t *s2);
```

[引数]

s1 比較される文字列へのポインタ

s2 比較する文字列へのポインタ

[戻り値]

s1 で指された文字列>s2 で指された文字列の時：正の値

s1 で指された文字列==s2 で指された文字列の時：0

s1 で指された文字列<s2 で指された文字列の時：負の値

[備考]

wcscmp 関数は strcmp 関数のワイド文字対応版です。

wcsncmp

指定された 2 つの文字列を指定された文字分まで比較します。

[指定形式]

```
#include <wchar.h>
long wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

[引数]

s1 比較される文字列へのポインタ

s2 比較する文字列へのポインタ

n 比較する文字数の最大値

[戻り値]

s1 で指された文字列>s2 で指された文字列の時：正の値

s1 で指された文字列==s2 で指された文字列の時：0

s1 で指された文字列<s2 で指された文字列の時：負の値

[備考]

wcsncmp 関数は strncmp 関数のワイド文字対応版です。

wmemcmp

指定された 2 つの記憶域の内容を比較します。

[指定形式]

```
#include <wchar.h>
long wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n);
```

[引数]

s1 比較される記憶域へのポインタ

s2 比較する記憶域へのポインタ

n 比較する記憶域の文字数

[戻り値]

s1 で指された記憶域>s2 で指された記憶域の時：正の値

s1 で指された記憶域==s2 で指された記憶域の時：0

s1 で指された記憶域<s2 で指された記憶域の時：負の値

[備考]

wmemcmp 関数は memcmp 関数のワイド文字対応版です。

wcschr

指定された文字列において、指定された文字が最初に現われる位置を検索します。

[指定形式]

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

[引数]

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時: 見つけられた文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

wcschr 関数は strchr 関数のワイド文字対応版です。

wcscspn

指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。

[指定形式]

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

[引数]

s1 調べられる文字列へのポインタ

s2 s1 を調べるための文字列へのポインタ

[戻り値]

s2 が指す文字列を構成する文字以外の文字が構成される文字列 s1 の先頭からの長さ

[備考]

wcscspn 関数は strcspn 関数のワイド文字対応版です。

wcspbrk

指定された文字列内において、別に指定された文字列中の文字が最初に現われる位置を検索します。

[指定形式]

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

[引数]

s1 検索を行う文字列へのポインタ

s2 s1 内で検索する文字を示す文字列へのポインタ

[戻り値]

検索の結果見つかった時: 見つかった文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

wcspbrk 関数は strpbrk 関数のワイド文字対応版です。

wcsrchr

指定された文字列において、指定された文字が最後に現われる位置を検索します。

[指定形式]

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

[引数]

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時: 見つかった文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

ー

wcsspn

指定された文字列を先頭から調べ、別に指定した文字列中の文字が先頭から何文字続くかを求めます。

[指定形式]

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

[引数]

s1 調べられる文字列へのポインタ

s2 s1 を調べるための文字列へのポインタ

[戻り値]

s1 の先頭から、s2 で指定した文字が続いている文字数

[備考]

wcsspn 関数は strspn 関数のワイド文字対応版です。

wcsstr

指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。

[指定形式]

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

[引数]

s1 検索を行う文字列へのポインタ

s2 検索する文字列へのポインタ

[戻り値]

検索の結果見つかったとき: 見つけられた文字へのポインタ

検索の結果見つからなかったとき: NULL

[備考]

ー

wcstok

指定した文字列をいくつかの字句に切り分けます。

[指定形式]

```
#include <wchar.h>
wchar_t* wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);
```

[引数]

s1 いくつかの字句に切り分ける文字列へのポインタ

s2 文字列を切り分けるための文字からなる文字列へのポインタ

ptr 次の関数呼び出し時に検索を始める文字列へのポインタ

[戻り値]

字句に切り分けられた時: 切り分けた字句の先頭へのポインタ

字句に切り分けられなかった時: NULL

[備考]

wcstok 関数は strtok 関数のワイド文字対応版です。

同じ文字列に対して 2 回目以降の検索をする場合は s1 に NULL を設定し、ptr には

前回の同文字列に対する関数呼び出しで取得した値を設定してください。

wmemchr

指定された記憶域において、指定された文字が最初に現われる位置を検索します。

[指定形式]

```
#include <wchar.h>
```

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);

[引数]

s 検索を行う記憶域へのポインタ

c 検索する文字

n 検索を行う文字数

[戻り値]

検索の結果見つかった時: 見つけられた文字へのポインタ

検索の結果見つからなかった時: NULL

[備考]

wmemchr 関数は memchr 関数のワイド文字対応版です。

wcslen

終端ナルワイド文字を除くワイド文字列の長さを計算します。

[指定形式]

#include <wchar.h>

size_t wcslen(const wchar_t *s);

[引数]

s 長さをワイド文字列へのポインタ

[戻り値]

ワイド文字列の文字数

[備考]

wcslen 関数は strlen 関数のワイド文字対応版です。

wmemset

指定された記憶域の先頭から、指定された文字を指定された文字数分設定します。

[指定形式]

#include <stdio.h>

#include <wchar.h>

wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);

[引数]

s 文字が設定される記憶域へのポインタ

c 設定する文字

n 設定する文字数

[戻り値]

s の値

[備考]

wmemset 関数は memset 関数のワイド文字対応版です。

wctob

ワイド文字を 1 バイト表現に変換します。

[指定形式]

#include <stdio.h>

#include <wchar.h>

long wctob(wint_t c);

[引数]

c ワイド文字

[戻り値]

正常： ワイド文字の 1 バイト値

異常： EOF

[備考]

ワイド文字が１バイトで表現できない場合は EOF を返します。

wctob 関数は，c が拡張文字集合の要素であり，かつ初期シフト状態では多バイト文字表現が 1 バイトになるものに対応するかどうかを判定します。

mbsinit

指定された mbstate_t オブジェクトが初期変換状態かどうか判定します。

[指定形式]

#include <wchar.h>

long mbsinit(const mbstate_t *ps);

[引数]

ps mbstate_t オブジェクトへのポインタ

[戻り値]

初期化状態の場合 0 以外の値

それ以外の状態の場合は 0

[備考]

－

mbrlen

指定された多バイト文字のバイト数を取得します。

[指定形式]

#include <wchar.h>

size_t mbrlen(const char * restrict s, size_t n, mbstate_t *restrict ps);

[引数]

s 多バイト文字列へのポインタ

n 認識する多バイト文字の最大バイト数

ps mbstate_t オブジェクトへのポインタ

[戻り値]

0 n 個以下のバイトによってナルワイド文字を認識した場合

1 以上 n 以下n 個以下のバイトによって多バイト文字を認識した場合

(size_t)(-2)n 個のバイトだけでは完全な多バイト文字を認識できない場合

(size_t)(-1)不正な多バイト文字の並びに遭遇した場合

[備考]

－

mbrtowc

多バイト文字をワイド文字に変換します。

[**指定形式**]

#include <wchar.h>

size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);

[**引数**]

pwc **取得したワイド文字を格納するワイド文字列へのポインタ**

s **多バイト文字列へのポインタ**

n **認識する多バイト文字の最大バイト数**

ps mbstate_t **オブジェクトへのポインタ**

[**戻り値**]

0 n **個以下のバイトによってナルワイド文字を認識した場合**

1 **以上** n **以下**n **個以下のバイトによって多バイト文字を認識した場合**

(size_t)(-2)n **個のバイトだけでは完全な多バイト文字を認識できない場合**

(size_t)(-1)**不正な多バイト文字の並びに遭遇した場合**

[**備考**]

不正な多バイト文字の並びに遭遇した場合、マクロ EILSEQ **の値を** errno **に格納し、変換状態は未規定とします。**

wcrtomb

ワイド文字を多バイト文字に変換します。

[**指定形式**]

#include <wchar.h>

size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);

[**引数**]

s **多バイト文字列へのポインタ**

wc **変換するワイド文字**

ps mbstate_t **オブジェクトへのポインタ**

[**戻り値**]

正常： 多バイト文字のバイト数

異常： (size_t)(-1)**不正な多バイト文字の並びに遭遇した場合**

[**備考**]

不正な多バイト文字の並びに遭遇した場合、マクロ EILSEQ **の値を** errno **に格納し、変換状態は未規定とします。**

wcrtomb **関数が決定した多バイト文字のバイト数にはシフトシーケンスを含みます。バイト数は** MB_CUR_MAX **を超えません。変換結果がナルワイド文字であった場合は初期変換状態となりますが、必要であれば初期シフト状態に戻すためのシフトシーケンスをワイド文字の前に格納します。**

6.5 EC++ ライブラリ関数

この節では、C++ **プログラムから標準的に利用できる** EC++ **クラスライブラリの仕様について説明します。ここでは、クラスライブラリの種類と対応する標準インクルードファイルについて説明します。以降では、ライブラリの構成に従って各クラスライブラリの仕様について説明します。**

- ライブラリの種類

表 6.14 にクラスライブラリの種類と対応する標準インクルードファイルを示します。

表 6―14 クラスライブラリの種類と標準インクルードファイルの対応

	ライブラリの種類	内容	標準 インクルードファイル
1	ストリーム入出力用クラスライブラリ	入出力操作を行うライブラリです。	<ios>,<streambuf>, <istream>,<ostream>, <iostream>,<iomanip>
2	メモリ操作用ライブラリ	メモリの確保・解放を行うライブラリです。	<new>
3	複素数計算用クラスライブラリ	複素数データ演算を行うライブラリです。	<complex>
4	文字列操作用クラスライブラリ	文字列操作を行うライブラリです。	<string>

6. 5. 1 ストリーム入出力用クラスライブラリ

ストリーム入出力用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <ios>

  入出力用書式設定、入出力状態管理を行うデータメンバおよび関数メンバを定義します。

  ios クラスの他に、Init クラス、ios_base クラスを定義します。

- <streambuf>

  ストリームバッファに対する関数を定義します。

- <istream>

  入力ストリームからの入力関数を定義します。

- <ostream>

  出力ストリームへの出力関数を定義します。

- <iostream>

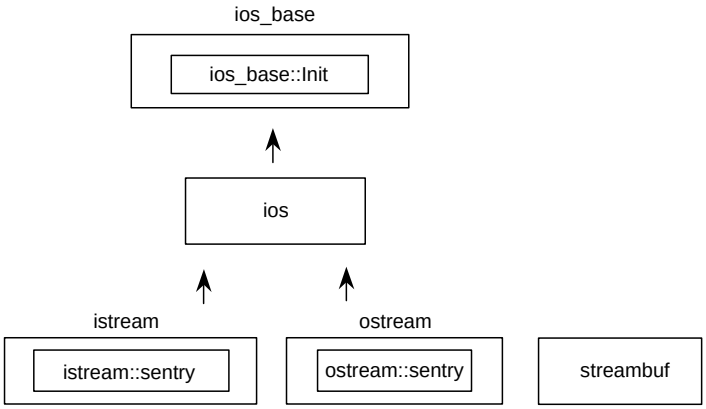
  入出力関数を定義します。

- <iomanip>

  引数を持つマニピュレータを定義します。

これらのクラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。なお、streambuf クラスには派生関係はありません。

これらのクラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。なお、streambuf クラスには派生関係はありません。



ストリーム入出力用クラスライブラリで共通に使用される型名を示します。

種別	定義名	説明
型	streamoff	long 型で定義された型です。
	streamsize	size_t 型で定義された型です。
	int_type	int 型で定義された型です。
	pos_type	long 型で定義された型です。
	off_type	long 型で定義された型です。

(a) ios_base::Init クラス

種別	定義名	説明
変数	init_cnt	ストリーム入出力オブジェクト数をカウントする静的データメンバです。 低水準インタフェースで 0 に初期化する必要があります。
関数	Init()	コンストラクタです
	~Init()	デストラクタです。

```
ios_base::Init::Init()
```

クラス Init のコンストラクタです。

init_cnt をインクリメントします。

```
ios_base::Init::~Init()
```

クラス Init のデストラクタです。

init_cnt をデクリメントします。

(b) ios_base クラス

種別	定義名	説明
型	fmtflags	フォーマット制御情報を表す型です。
	iostate	ストリームバッファの入出力状態を表す型です。
	openmode	ファイルのオープンモードを表す型です。
	seekdir	ストリームバッファのシーク状態を表す型です。
変数	fmtfl	書式フラグです。
	wide	フィールド幅です。
	prec	出力時の精度 (小数点以下の桁数) です。
	fillch	詰め文字です。
関数	void _ec2p_init_base()	初期化します。
	void _ec2p_copy_base(ios_base&ios_base_dt)	ios_base_dt をコピーします。
	ios_base()	コンストラクタです。
	~ios_base()	デストラクタです。
	fmtflags flags() const	書式フラグ (fmtfl) を参照します。
	fmtflags flags(fmtflags fmtflg)	fmtflg& 書式フラグ (fmtfl) を書式フラグ (fmtfl) に設定します。
	fmtflags setf(fmtflags fmtflg)	fmtflg を書式フラグ (fmtfl) に設定します。
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	mask&fmtflg を書式フラグ (fmtfl) に設定します。
	void unsetf(fmtflags mask)	~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
	char fill() const	詰め文字 (fillch) を参照します。
	char fill(char ch)	ch を詰め文字 (fillch) に設定します。
	int precision() const	精度 (prec) を参照します。
	streamsize precision(streamsize preci)	preci を精度 (prec) に設定します。
	streamsize width() const	フィールド幅 (wide) を参照します。
	streamsize width(streamsize wd)	wd をフィールド幅 (wide) に設定します。

ios_base::fmtflags

入出力に関するフォーマット制御情報を定義します。

fmtflags の各ビットマスクの定義は以下のようになります。

const ios_base::fmtflags ios_base::boolalpha=0x0000;

const ios_base::fmtflags ios_base::skipws=0x0001;

const ios_base::fmtflags ios_base::unitbuf=0x0002;

const ios_base::fmtflags ios_base::uppercase=0x0004;

const ios_base::fmtflags ios_base::showbase=0x0008;

const ios_base::fmtflags ios_base::showpoint=0x0010;

const ios_base::fmtflags ios_base::showpos=0x0020;

const ios_base::fmtflags ios_base::left=0x0040;

const ios_base::fmtflags ios_base::right=0x0080;

const ios_base::fmtflags ios_base::internal=0x0100;

const ios_base::fmtflags ios_base::adjustfield=0x01c0;

const ios_base::fmtflags ios_base::dec=0x0200;

const ios_base::fmtflags ios_base::oct=0x0400;

const ios_base::fmtflags ios_base::hex=0x0800;

const ios_base::fmtflags ios_base::basefield=0x0e00;

const ios_base::fmtflags ios_base::scientific=0x1000;

const ios_base::fmtflags ios_base::fixed=0x2000;

const ios_base::fmtflags ios_base::floatfield=0x3000;

const ios_base::fmtflags ios_base::_fmtmask=0x3fff;

ios_base::iostate

ストリームバッファの入出力状態を定義します。

iostate **の各ビットマスクの定義は以下のようになります。**

const ios_base::iostate ios_base::goodbit=0x0;

const ios_base::iostate ios_base::eofbit=0x1;

const ios_base::iostate ios_base::failbit=0x2;

const ios_base::iostate ios_base::badbit=0x4;

const ios_base::iostate ios_base::_statemask=0x7;

ios_base::openmode

ファイルのオープンモードを定義します。

openmode **の各ビットマスクの定義は以下のようになります。**

const ios_base::openmode ios_base::in=0x01;**入力用のファイルを** open **します。**

const ios_base::openmode ios_base::out=0x02;**出力用のファイルを** open **します。**

const ios_base::openmode ios_base::ate=0x04;**オープン後一度だけ** eof **に** seek **します。**

const ios_base::openmode ios_base::app=0x08;**書き込む度に** eof **に** seek **します。**

const ios_base::openmode ios_base::trunc=0x10;**ファイルを上書きモードで** open **します。**

const ios_base::openmode ios_base::binary=0x20;**ファイルをバイナリモードで** open **します。**

ios_base::seekdir

ストリームバッファのシーク状態を定義します。

引き続き入力または出力を行うためのストリーム内の位置を決定します。

seekdir **の各ビットマスクの定義は以下のようになります。**

const ios_base::seekdir ios_base::beg=0x0;

const ios_base::seekdir ios_base::cur=0x1;

const ios_base::seekdir ios_base::end=0x2;

void ios_base::_ec2p_init_base()

以下の値で初期設定します。

fmtfl=skipws | dec;

wide=0;

prec=6;

fillch=' ';

```
void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)
```

ios_base_dt をコピーします。

```
ios_base::ios_base()
```

クラス ios_base のコンストラクタです。

Init::Init() を呼び出します。

```
ios_base::~ios_base()
```

クラス ios_base のデストラクタです。

```
ios_base::fmtflags ios_base::flags() const
```

書式フラグ (fmtfl) を参照します。

リターン値は、書式フラグ (fmtfl) です。

```
ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
```

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

リターン値は、設定前の書式フラグ (fmtfl) です。

```
ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
```

fmtflg を書式フラグ (fmtfl) に設定します。

リターン値は、設定前の書式フラグ (fmtfl) です。

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

mask&fmtflg の値を書式フラグ (fmtfl) に設定します。

リターン値は、設定前の書式フラグ (fmtfl) です。

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

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

```
char ios_base::fill() const
```

詰め文字 (fillch) を参照します。

リターン値は、詰め文字 (fillch) です。

```
char ios_base::fill(char ch)
```

ch を詰め文字として設定します。

リターン値は、設定前の詰め文字 (fillch) です。

```
int ios_base::precision() const
```

精度 (prec) を参照します。

リターン値は、精度 (prec) **です。**

```
streamsize ios_base::precision(streamsize preci)
```

preci **を精度** (prec) **に設定します。**

リターン値は、設定前の精度 (prec) **です。**

```
streamsize ios_base::width() const
```

フィールド幅 (wide) **を参照します。**

リターン値は、フィールド幅 (wide) **です。**

```
streamsize ios_base::width(streamsize wd)
```

wd **をフィールド幅** (wide) **に設定します。**

リターン値は、設定前のフィールド幅 (wide) **です。**

(c) ios クラス

種別	定義名	説明
変数	sb	streambuf オブジェクトへのポインタです。
	tiestr	ostream オブジェクトへのポインタです。
	state	streambuf への状態フラグです。
関数	ios()	コンストラクタです。
	ios(streambuf* sbptr)	
	void init(streambuf* sbptr)	初期設定を行います。
	virtual ~ios()	デストラクタです。
	operator void*() const	エラー有無 (!state&(badbit | failbit)) を判定します。
	bool operator!() const	エラー有無 (state&(badbit | failbit)) を判定します。
	iostate rdstate() const	状態フラグ (state) を参照します。
	void clear(iostate st = goodbit)	指定された状態 (st) を除いて状態フラグ (state) をクリアします。
	void setstate(iostate st)	st を状態フラグ (state) に設定します。
	bool good() const	エラー有無 (state==goodbit) を判定します。
	bool eof() const	入力ストリームの最後かどうか (state&eofbit) を判定します。
	bool bad() const	エラー有無 (state&badbit) を判定します。
	bool fail() const	入力テキストが要求パターンと不一致であるかどうか (state&(badbit | failbit)) 判定します。
	ostream* tie() const	ostream オブジェクトへのポインタ (tiestr) を参照します。
	ostream* tie(ostream* tstrptr)	tstrptrをostreamオブジェクトへのポインタ(tiestr)に設定します。
	streambuf* rdbuf() const	streambuf オブジェクトへのポインタ (sb) を参照します。
	streambuf* rdbuf(streambuf* sbptr)	sbptr を streambuf オブジェクトへのポインタ (sb) に設定します。
	ios& copyfmt(const ios& rhs)	rhs の状態フラグ (state) をコピーします。

ios::ios()

クラス ios **のコンストラクタです。**

init(0) **を呼び出し、初期値をそのメンバオブジェクトに設定します。**

ios::ios(streambuf* sbptr)

クラス ios **のコンストラクタです。**

init(sbptr) **を呼び出し、初期値をそのメンバオブジェクトに設定します。**

void ios::init(streambuf* sbptr)

sbptr **を** sb **に設定します。**

state**、**tiestr **を** 0 **に設定します。**

virtual ios::~ios()

クラス ios **のデストラクタです。**

ios::operator void*() const

エラー有無 (!state&(badbit | failbit)) **を判定します。**

リターン値は次のとおりです。

エラー有の場合：false

エラー無の場合：true

bool ios::operator!() const

エラー有無 (state&(badbit | failbit)) **を判定します。**

リターン値は次のとおりです。

エラー有の場合：true

エラー無の場合：false

iostate ios::rdstate() const

状態フラグ (state) **を参照します。**

リターン値は、状態フラグ (state) **です。**

void ios::clear(iostate st = goodbit)

指定された状態 (st) **を除いて状態フラグ** (state) **をクリアします。**

streambuf **オブジェクトへのポインタ** (sb) **が** 0 **のときは、状態フラグ** (state) **に** badbit **を設定します。**

void ios::setstate(iostate st)

st **を状態フラグ** (state) **に設定します。**

bool ios::good() const

エラー有無 (state==goodbit) **を判定します。**

リターン値は次のとおりです。

エラー有の場合：false

エラー無の場合：true

```
bool ios::eof() const
```

入力ストリームの最後かどうか (state&eofbit) **を判定します。**

リターン値は次のとおりです。

入力ストリームの最後の場合 ：true

入力ストリームの最後以外の場合：false

```
bool ios::bad() const
```

エラー有無 (state&badbit) **を判定します。**

リターン値は次のとおりです。

エラー有の場合：true

エラー無の場合：false

```
bool ios::fail() const
```

入力テキストが要求パターンと不一致であるかどうか (state&(badbit | failbit)) **を判定します。**

リターン値は次のとおりです。

不一致の場合：true

一致の場合 ：false

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

ostream **オブジェクトへのポインタ** (tiestr) **を参照します。**

リターン値は、ostream **オブジェクトへのポインタ** (tiestr) **です。**

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

tstrptr **を** ostream **オブジェクトへのポインタ** (tiestr) **に設定します。**

リターン値は、設定前の ostream **オブジェクトへのポインタ** (tiestr) **です。**

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

streambuf **オブジェクトへのポインタ** (sb) **を参照します。**

リターン値は、streambuf **オブジェクトへのポインタ** (sb) **です。**

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

sbptr **を** streambuf **オブジェクトへのポインタ** (sb) **に設定します。**

リターン値は、設定前の streambuf **オブジェクトへのポインタ** (sb) **です。**

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

rhs **の状態フラグ** (state) **をコピーします。**

リターン値は *this **です。**

(d) ios クラスマニピュレータ

種別	定義名	説明
関数	ios_base& showbase(ios_base& str)	基数表示接頭辞モードに設定します。
	ios_base& noshowbase(ios_base& str)	基数表示接頭辞モードをクリアします。
	ios_base& showpoint (ios_base& str)	小数点生成モードに設定します。
	ios_base& noshowpoint (ios_base& str)	小数点生成モードをクリアします。
	ios_base& showpos(ios_base& str)	+ 記号生成モードに設定します。
	ios_base& noshowpos(ios_base& str)	+ 記号生成モードをクリアします。
	ios_base& skipws(ios_base& str)	空白読み飛ばしモードに設定します。
	ios_base& noskipws(ios_base& str)	空白読み飛ばしモードをクリアします。
	ios_base& uppercase(ios_base& str)	大文字変換モードに設定します。
	ios_base& nouppercase(ios_base& str)	大文字変換モードをクリアします。
	ios_base& internal(ios_base& str)	内部補充モードに設定します。
	ios_base& left(ios_base& str)	左側補充モードに設定します。
	ios_base& right(ios_base& str)	右側補充モードに設定します。
	ios_base& dec(ios_base& str)	10 進モードに設定します。
	ios_base& hex(ios_base& str)	16 進モードに設定します。
	ios_base& oct(ios_base& str)	8 進モードに設定します。
	ios_base& fixed(ios_base& str)	固定小数点モードに設定します。
	ios_base& scientific(ios_base& str)	科学表記法モードに設定します。

```
ios_base& showbase(ios_base& str)
```

データのはじめに基数を表示させるモードに設定します。

16 進数のときは、0x を行の先頭に付加します。10 進数のときは、そのまま出力します。

8 進数のときは、0 を行の先頭に付加します。

リターン値は str です。

```
ios_base& noshowbase(ios_base& str)
```

データのはじめに基数を表示させるモードをクリアします。

リターン値は str です。

```
ios_base& showpoint(ios_base& str)
```

小数点を出力するモードに設定します。

精度の指定がない場合、小数点以下 6 桁で表示します。

リターン値は str です。

```
ios_base& noshowpoint(ios_base& str)
```

小数点を出力するモードをクリアします。

リターン値は str です。

```
ios_base& showpos(ios_base& str)
```

+ 記号生成出力モード (正の数に対して + の符号を付加) に設定します。

リターン値は str です。

```
ios_base& noshowpos(ios_base& str)
```

+ 記号生成出力モードをクリアします。

リターン値は str です。

```
ios_base& skipws(ios_base& str)
```

空白読み飛ばし入力モード (連続する空白をスキップ) に設定します。

リターン値は str です。

```
ios_base& noskipws(ios_base& str)
```

空白読み飛ばし入力モードをクリアします。

リターン値は str です。

```
ios_base& uppercase(ios_base& str)
```

大文字変換出力モードに設定します。

16 進の基数表現が大文字の 0X になり、数値自体も大文字になります。

浮動小数点の指数表現も大文字の E になります。

リターン値は str です。

```
ios_base& nouppercase(ios_base& str)
```

大文字変換出力モードをクリアします。

リターン値は str です。

```
ios_base& internal(ios_base& str)
```

フィールド幅 (wide) の範囲で出力時に

符号、基数

詰め文字 (fill)

数値

の順で出力します。

リターン値は str です。

```
ios_base& left(ios_base& str)
```

フィールド幅 (wide) の範囲で出力時に左詰めします。

リターン値は str です。

```
ios_base& right(ios_base& str)
```

フィールド幅 (wide) の範囲で出力時に右詰めします。

リターン値は str **です。**

```
ios_base& dec(ios_base& str)
```

変換基数を 10 **進モードに設定します。**

リターン値は str **です。**

```
ios_base& hex(ios_base& str)
```

変換基数を 16 **進モードに設定します。**

リターン値は str **です。**

```
ios_base& oct(ios_base& str)
```

変換基数を 8 **進モードに設定します。**

リターン値は str **です。**

```
ios_base& fixed(ios_base& str)
```

固定小数点出力モードに設定します。

リターン値は str **です。**

```
ios_base& scientific(ios_base& str)
```

科学表記法出力モード (**指数表記** **) に設定します。**

リターン値は str **です。**

(e) streambuf クラス

種別	定義名	説明
定数	eof	ファイル終了を示します。
変数	_B_cnt_ptr	バッファの有効データ長へのポインタです。
	B_beg_ptr	バッファのベースポインタへのポインタです。
	_B_len_ptr	バッファの長さへのポインタです。
	B_next_ptr	バッファの次の読み出し位置へのポインタです。
	B_end_ptr	バッファの終端位置へのポインタです。
	B_beg_pptr	制御バッファの先頭位置へのポインタです。
	B_next_pptr	バッファの次の読み出し位置へのポインタです。
	C_flg_ptr	ファイルの入出力制御フラグへのポインタです。

種別	定義名	説明
関数	char* _ec2p_getflag() const	ファイル入出力制御フラグのポインタを参照します。
	char*& _ec2p_gnptr()	バッファの次の読み出し位置へのポインタを参照します。
	char*& _ec2p_pnptr()	バッファの次の書き込み位置へのポインタを参照します。
	void _ec2p_bcntplus()	バッファの有効データ長をインクリメントします。
	void _ec2p_bcntminus()	バッファの有効データ長をデクリメントします。
	void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)	streambuf のポインタを設定します。
	streambuf()	コンストラクタです。
	virtual ~streambuf()	デストラクタです。
	streambuf* pubsetbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。この関数では setbuf(s,n)*1 を呼び出します。
	pos_type pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	way で指定された方法で入出力ストリームの読み書き位置を移動させます。この関数では seekoff(off,way,which)*1 を呼び出します。
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	ストリームの先頭から現在の位置までのオフセットを求めます。この関数では seekpos(sp,which)*1 を呼び出します。
	int pubsync()	出力ストリームをフラッシュします。この関数では sync()*1 を呼び出します。
	streamsize in_avail()	入力ストリームの最後尾から現在位置までのオフセットを求めます。

種別	定義名	説明
関数	int_type snextc()	次の一文字を読み込みます。
	int_type sbumpc()	一文字読み込みポインタを次に設定します。
	int_type sgetc()	一文字読み込みます。
	int sgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	int_type sputbackc(char c)	読み込み位置をプットバックします。
	int sungetc()	読み込み位置をプットバックします。
	int sputc(char c)	文字 c を挿入します。
	int_type sputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	char* eback() const	入力ストリームの先頭ポインタを求めます。
	char* gptr() const	入力ストリームの次ポインタを求めます。
	char* egptr() const	入力ストリームの最後尾ポインタを求めます。
	void gbump(int n)	入力ストリームの次ポインタを n 進めます。
	void setg(char* gbeg, char* gnext, char* gend)	入力ストリームの各ポインタを代入します。
	char* pbase() const	出力ストリームの先頭ポインタを求めます。
	char* pptr() const	出力ストリームの次ポインタを求めます。
	char* epptr() const	出力ストリームの最後尾ポインタを求めます。
	void pbump(int n)	出力ストリームの次ポインタを n 進めます。
	void setp(char* pbeg, char* pend)	出力ストリームの各ポインタを設定します。
	virtual streambuf* setbuf(char* s, streamsize n)*1	派生する各クラスごとに、個別に定義する演算を実行します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*1	ストリーム位置を変更します。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	ストリーム位置を変更します。
	virtual int sync()*1	出力ストリームをフラッシュします。
	virtual int showmanyc()*1	入力ストリームの有効な文字数を求めます。
	virtual streamsize xsgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	virtual int_type underflow()*1	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type uflow()*1	次ポインタの一文字を読み込みます。
	virtual int_type pbackfail(int_type c = eof)*1	c によって示される文字をプットバックします。

種別	定義名	説明
関数	virtual streamsize xsputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	virtual int_type overflow(int_type c = eof)*1	c を出力ストリームに挿入します。

注1. このクラスでは処理を定義していません。

streambuf::streambuf()

コンストラクタです。

以下の値で初期化します。

_B_cnt_ptr=B_beg_ptr=B_next_ptr=B_end_ptr=C_flg_ptr=_B_len_ptr=0

B_beg_pptr=&B_beg_ptr

B_next_pptr=&B_next_ptr

virtual streambuf::~streambuf()

デストラクタです。

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

ストリーム入出力用のバッファを確保します。

この関数では setbuf(s,n) を呼び出します。

リターン値は、*this です。

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

way で指定された方法で入出力ストリームの読み書き位置を移動させます。

この関数では seekoff(off,way,which) を呼び出します。

リターン値は、新たに設定されたストリームの位置です。

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

ストリームの先頭から現在の位置までのオフセットを求めます。

現在のストリームポインタから sp だけ移動します。

この関数では seekpos(sp,which) を呼び出します。

リターン値は、先頭からのオフセットです。

int streambuf::pubsync()

出力ストリームをフラッシュします。

この関数では sync() を呼び出します。

リターン値は 0 です。

streamsize streambuf::in_avail()

入力ストリームの最後尾から現在位置までのオフセットを求めます。

リターン値は次のとおりです。

読み込み位置が有効の場合：最後尾から現在位置までのオフセット

読み込み位置が無効の場合：0(showmanyc() を呼び出します)

int_type streambuf::snextc()

一文字読み込みます。読み込んだ文字が eof でなければ、次の一文字を読み込みます。

リターン値は次のとおりです。

eof でない場合：読み込んだ文字

eof の場合：eof

int_type streambuf::sbumpc()

一文字読み込みポインタを次に設定します。

リターン値は次のとおりです。

読み込み位置が無効でない場合：読み込んだ文字

読み込み位置が無効の場合：eof

int_type streambuf::sgetc()

一文字読み込みます。

リターン値は次のとおりです。

読み込み位置が無効でない場合：読み込んだ文字

読み込み位置が無効の場合：eof

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

s の指す記憶領域に n 個の文字を設定します。

文字列中に eof を検出した場合、設定を終了します。

リターン値は、設定した文字数です。

int_type streambuf::sputbackc(char c)

読み込み位置が正常で読み込み位置のプットバックデータが c と同一の場合、読み込み位置をプットバックします。

リターン値は次のとおりです。

プットバックできた場合：c の値

プットバックできなかった場合：eof

int streambuf::sungetc()

読み込み位置が正常である場合、読み込み位置をプットバックします。

リターン値は次のとおりです。

プットバックできた場合：プットバックした値

プットバックできなかった場合：eof

int streambuf::sputc(char c)

文字 c を挿入します。

リターン値は次のとおりです。

書き込み位置が正しい場合：c **の値**

書き込み位置が不正な場合：eof

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

s **の指す** n **個の文字を挿入します。**

バッファが n **より小さい場合は、バッファサイズ分だけ挿入します。**

リターン値は、挿入された文字数です。

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

入力ストリームの先頭ポインタを求めます。

リターン値は、先頭ポインタです。

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

入力ストリームの次ポインタを求めます。

リターン値は、次ポインタです。

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

入力ストリームの最後尾ポインタを求めます。

リターン値は、最後尾ポインタです。

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

入力ストリームの次ポインタを n **進めます。**

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

入力ストリームの各ポインタに、以下の設定を行います。

*B_beg_pptr=gbeg;

*B_next_pptr=gnext;

B_end_ptr=gend;

*_B_cnt_ptr=gend-gnext;

*_B_len_ptr=gend-gbeg;

```
char* streambuf::pbase() const
```

出力ストリームの先頭ポインタを求めます。

リターン値は、先頭ポインタです。

```
char* streambuf::pptr() const
```

出力ストリームの次ポインタを求めます。

リターン値は、次ポインタです。

```
char* streambuf::epptr() const
```

出力ストリームの最後尾ポインタを求めます。

リターン値は、最後尾ポインタです。

```
void streambuf::pbump(int n)
```

出力ストリームの次ポインタを n **進めます。**

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

出力ストリームの各ポインタに、以下の設定を行います。

*B_beg_pptr=pbeg;

*B_next_pptr=pbeg;

B_end_ptr=pend;

*_B_cnt_ptr=pend-pbeg;

*_B_len_ptr=pend-pbeg;

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

streambuf **から派生する各クラスごとに、個別に定義する演算を実行します。**

リターン値は *this **です。このクラスでは処理を定義していません。**

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

ストリーム位置を変更します。

リターン値は -1 **です。このクラスでは処理を定義していません。**

```
virtual pos_type streambuf::seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode)(ios_base::in |
ios_base::out))
```

ストリーム位置を変更します。

リターン値は -1 **です。このクラスでは処理を定義していません。**

```
virtual int streambuf::sync()
```

出力ストリームをフラッシュします。

リターン値は 0 **です。このクラスでは処理を定義していません。**

```
virtual int streambuf::showmanyc()
```

入力ストリームの有効な文字数を求めます。

リターン値は 0 **です。このクラスでは処理を定義していません。**

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

s **の指す記憶領域に** n **個の文字を設定します。**

バッファが n **より小さい場合は、バッファサイズ分だけ設定します。**

リターン値は、入力された文字数です。

```
virtual int_type streambuf::underflow()
```

ストリーム位置を動かさずに一文字読み込みます。

リターン値は eof **です。このクラスでは処理を定義していません。**

```
virtual int_type streambuf::uflow()
```

次ポインタの一文字を読み込みます。

リターン値は eof です。このクラスでは処理を定義していません。

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

c によって示される文字をプットバックします。

リターン値は eof です。このクラスでは処理を定義していません。

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

s の指す n 個の文字を挿入します。

バッファが n より小さい場合は、バッファサイズ分だけ挿入します。

リターン値は、挿入された文字数です。

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

c を出力ストリームに挿入します。

リターン値は eof です。このクラスでは処理を定義していません。

（f）istream::sentry クラス

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

```
istream::sentry::sentry(istream& is, bool noskipws = _false)
```

内部クラス sentry のコンストラクタです。

good() が非 0 の場合、フォーマット付きまたはフォーマットなし入力を可能にします。

tie() が非 0 の場合、関連する出力ストリームをフラッシュします。

```
istream::sentry::~sentry()
```

内部クラス sentry のデストラクタです。

```
istream::sentry::operator bool()
```

ok_ を参照します。

リターン値は ok_ です。

（g）istream クラス

種別	定義名	説明
変数	chcount	最後にコールされた入力関数が抽出した文字数です。
関数	int _ec2p_getistr(char* str, unsigned int dig, int mode)	str を dig が示す基数で変換します。
	istream(streambuf* sb)	コンストラクタです。
	virtual ~istream()	デストラクタです。
	istream& operator>>(bool& n)	抽出した文字を n に格納します。
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(long long& n)	
	istream& operator>>(unsigned long long& n)	
	istream& operator>>(float& n)	
	istream& operator>>(double& n)	
	istream& operator>>(long double& n)	
	istream& operator>>(void*& p)	voidを指すポインタに変換してpに格納します。
	istream& operator>>(streambuf* sb)	文字を抽出し、sb の指す記憶領域へ格納します。
	streamsize gcount() const	chcount(抽出文字数) を求めます。
	int_type get()	文字を抽出します。
	istream& get(char& c)	文字を抽出し c に格納します。
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。文字列内に 'delim' を検出したら、入力を終了します。
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	

種別	定義名	説明
関数	istream& get(streambuf& sb)	文字列を抽出し、sb の指す記憶領域に格納します。
	istream& get(streambuf& sb, char delim)	文字列を抽出し、sb の指す記憶領域に格納します。途中で文字 'delim' を検出したら、入力を終了します。
	istream& getline(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	
	istream& getline(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。途中で文字 'delim' を検出したら、入力を終了します。
	istream& getline(signed char* s, streamsize n, char delim)	
	istream& getline(unsigned char* s, streamsize n, char delim)	
	istream& ignore(streamsize n = 1, int_type delim = streambuf::eof)	n 個の文字を読み飛ばします。途中で文字 'delim' を検出したら、読み飛ばし処理を中止します。
	int_type peek()	次の入手可能な入力文字を求めます。
	istream& read(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome(unsigned char* s, streamsize n)	
	istream& putback(char c)	文字を入力ストリームに戻します。
	istream& unget()	入力ストリームの位置を戻します。
	int sync()	入力ストリームがあるかどうかを調べます。この関数は streambuf::pubsync() を呼び出します。
	pos_type tellg()	入力ストリームの位置を調べます。この関数は streambuf::pubseekoff(0,cur,in) を呼び出します。

種別	定義名	説明
関数	istream& seekg(pos_type pos)	現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos) を呼び出します。
	istream& seekg(off_type off, ios_base::seekdir dir)	dir で指定された方法で入力ストリームの読み込み位置を移動します。この関数は streambuf::pubseekoff(off,dir) を呼び出します。

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

str を dig が示す基数で変換します。

リターン値は、変換した基数です。

```
istream::istream(streambuf* sb)
```

クラス istream のコンストラクタです。

ios::init(sb) を呼び出します。

chcount=0 の設定を行います。

```
virtual istream::~istream()
```

クラス istream のデストラクタです。

```
istream& istream::operator>>(bool& n)
```

抽出した文字を n に格納します。

リターン値は *this です。

```
istream& istream::operator>>(short& n)
```

抽出した文字を n に格納します。

リターン値は *this です。

```
istream& istream::operator>>(unsigned short& n)
```

抽出した文字を n に格納します。

リターン値は *this です。

```
istream& istream::operator>>(int& n)
```

抽出した文字を n に格納します。

リターン値は *this です。

```
istream& istream::operator>>(unsigned int& n)
```

抽出した文字を n に格納します。

リターン値は *this です。

```
istream& istream::operator>>(long& n)
```

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(unsigned long& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(long long& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(unsigned long long& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(float& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(double& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

istream& istream::operator>>(long double& n)

抽出した文字を n **に格納します。**

リターン値は *this **です。**

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

抽出した文字を void* **型に変換し、**p **の指す記憶領域に格納します。**

リターン値は *this **です。**

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

文字を抽出し、sb **の指す記憶領域に格納します。**

抽出文字がない場合は、setstate(failbit) **を呼び出します。**

リターン値は *this **です。**

streamsize istream::gcount() const

chcount(**抽出文字数**) **を参照します。**

リターン値は chcount **です。**

int_type istream::get()

文字を抽出します。

リターン値は次のとおりです。

抽出可能の場合：抽出した文字

抽出不可の場合：setstate(failbat) **を呼び出して、**streambuf::eof

istream& istream::get(char& c)

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

リターン値は *this **です。**

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

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

リターン値は *this **です。**

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

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

リターン値は *this **です。**

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

サイズ n-1 **の文字列を抽出し、**s **の指す記憶領域に格納します。**

ok_==false **または抽出した文字数が** 0 **の場合は、**failbit **を設定します。**

リターン値は *this **です。**

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

サイズ n-1 **の文字列を抽出し、**s **の指す記憶領域に格納します。**

ok_==false **または抽出した文字数が** 0 **の場合は、**failbit **を設定します。**

リターン値は *this **です。**

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

サイズ n-1 **の文字列を抽出し、**s **の指す記憶領域に格納します。**

ok_==false **または抽出した文字数が** 0 **の場合は、**failbit **を設定します。**

リターン値は *this **です。**

istream& istream::get(char* s, streamsize n, char delim)

サイズ n-1 **の文字列を抽出し、**s **の指す記憶領域に格納します。**

文字列内に 'delim' **を検出したら、終了します。**

ok_==false **または抽出した文字数が** 0 **の場合は、**failbit **を設定します。**

リターン値は *this **です。**

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

サイズ n-1 **の文字列を抽出し、**s **の指す記憶領域に格納します。**

文字列内に 'delim' **を検出したら、終了します。**

ok_==false **または抽出した文字数が** 0 **の場合は、**failbit **を設定します。**

リターン値は *this **です。**

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
文字列内に 'delim' を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

istream& istream::get(streambuf& sb)

文字列を抽出し、sb の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

文字列を抽出し、sb の指す記憶領域に格納します。
途中で文字 'delim' を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

istream& istream::getline(char* s, streamsize n, char delim)

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
途中で文字 'delim' を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
途中で文字 'delim' を検出したら、終了します。
ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
リターン値は *this です。

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

サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。

途中で文字 'delim' を検出したら、終了します。

ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。

リターン値は *this です。

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

n 個の文字を読み飛ばします。

途中で文字 'delim' を検出したら、読み飛ばし処理を中止します。

リターン値は *this です。

int_type istream::peek()

次の入力可能な入力文字を求めます。

リターン値は次のとおりです。

ok_==false の場合：streambuf::eof

ok_!=false の場合：rdbuf()->sgetc()

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

ok_!=false の場合、サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

抽出した文字数が n と異なる場合、eofbit を設定します。

リターン値は *this です。

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

ok_!=false の場合、サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

抽出した文字数が n と異なる場合、eofbit を設定します。

リターン値は *this です。

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

ok_!=false の場合、サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

抽出した文字数が n と異なる場合、eofbit を設定します。

リターン値は *this です。

streamsize istream::readsome(char* s, streamsize n)

サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

文字数がストリームサイズより大きければ、ストリームサイズ分格納します。

リターン値は、抽出した文字数です。

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

サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

文字数がストリームサイズより大きければ、ストリームサイズ分格納します。

リターン値は、抽出した文字数です。

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

サイズ n の文字列を抽出し、s の指す記憶領域に格納します。

文字数がストリームサイズより大きければ、ストリームサイズ分格納します。

リターン値は、抽出した文字数です。

istream& istream::putback(char c)

文字 c を入力ストリームに戻します。プットバックした文字が streambuf::eof の場合は、badbit を設定します。

リターン値は *this です。

istream& istream::unget()

入力ストリームのポインタをひとつ戻します。

抽出した文字が streambuf::eof の場合、badbit を設定します。

リターン値は *this です。

int istream::sync()

入力ストリームがあるかどうかを調べます。

この関数は streambuf::pubsync() を呼び出します。

リターン値は次のとおりです。

入力ストリームがない場合：streambuf::eof

入力ストリームがある場合：0

pos_type istream::tellg()

入力ストリームの位置を調べます。

この関数は streambuf::pubseekoff(0,cur,in) を呼び出します。

リターン値は次のとおりです。

ストリームの先頭からのオフセット

ただし、入力処理にエラーが発生した場合は -1

istream& istream::seekg(pos_type pos)

現在のストリームポインタから pos だけ移動します。

この関数は streambuf::pubseekpos(pos) を呼び出します。

リターン値は *this です。

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

dir で指定された方法で入力ストリームの読み込み位置を移動します。

この関数は streambuf::pubseekoff(off,dir) を呼び出します。

入力処理にエラーがある場合は処理は行いません。

リターン値は *this です。

(h) istream クラスマニピュレータ

種別	定義名	説明
関数	istream& ws(istream& is)	空白類を読み飛ばします。

istream& ws(istream& is)

空白類を読み飛ばします。

リターン値は is です。

(i) istream メンバ外関数

種別	定義名	説明
関数	istream& operator>>(istream& in, char* s)	文字列を抽出し、s の指す記憶領域に格納します。
	istream& operator>>(istream& in, signed char* s)	
	istream& operator>>(istream& in, unsigned char* s)	
	istream& operator>>(istream& in, char& c)	文字を抽出し、c に格納します。
	istream& operator>>(istream& in, singed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

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

文字列を抽出し、s の指す記憶領域に格納します。

(フィールド幅 -1) 個の文字を格納したか、または入力ストリームに streambuf::eof が現れたか、または次の入力可能な文字 c が isspace(c)==1 の場合、処理は終了します。格納文字数が 0 の場合は failbit を設定します。

リターン値は in です。

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

文字列を抽出し、s の指す記憶領域に格納します。

(フィールド幅 -1) 個の文字を格納したか、または入力ストリームに streambuf::eof が現れたか、または次の入力可能な文字 c が isspace(c)==1 の場合、処理は終了します。格納文字数が 0 の場合は failbit を設定します。

リターン値は in です。

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

文字列を抽出し、s の指す記憶領域に格納します。

(フィールド幅 -1) 個の文字を格納したか、または入力ストリームに streambuf::eof が現れたか、または次の入力可能な文字 c が isspace(c)==1 の場合、処理は終了します。格納文字数が 0 の場合は failbit を設定します。

リターン値は in です。

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

文字を抽出し、c に格納します。

抽出入力がない場合、failbit を設定します。

リターン値は in です。

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

文字を抽出し、c **に格納します。**

抽出入力がない場合、failbit **を設定します。**

リターン値は in **です。**

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

文字を抽出し、c **に格納します。**

抽出入力がない場合、failbit **を設定します。**

リターン値は in **です。**

(j) ostream::sentry クラス

種別	定義名	説明
変数	ok_	出力可能状態か否かを意味します。
	_ _ec2p_os	ostream オブジェクトへのポインタです。
関数	sentry(ostream& os)	コンストラクタです。
	~sentry()	デストラクタです。
	operator bool()	ok_ を参照します。

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

内部クラス sentry **のコンストラクタです。**

good() **が非** 0 **かつ** tie() **が非** 0 **なら** flush() **を呼び出します。**_ _ec2p_os **に** os **を設定します。**

ostream::sentry::~sentry()

内部クラス sentry **のデストラクタです。**

_ _ec2p_os->flags() & ios_base::unitbuf **が真なら、**flush() **を呼び出します。**

ostream::sentry::operator bool()

ok_ **を参照します。**

リターン値は ok_ **です。**

(k) ostream クラス

種別	定義名	説明
関数	ostream(streambuf* sbptr)	コンストラクタです。
	virtual ~ostream()	デストラクタです。
	ostream& operator<<(bool n)	n を出力ストリームに挿入します。
	ostream& operator<<(short n)	
	ostream& operator<<(unsigned short n)	
	ostream& operator<<(int n)	
	ostream& operator<<(unsigned int n)	
	ostream& operator<<(long n)	
	ostream& operator<<(unsigned long n)	
	ostream& operator<<(long long n)	
	ostream& operator<<(unsigned long long n)	
	ostream& operator<<(float n)	
	ostream& operator<<(double n)	
	ostream& operator<<(long double n)	
	ostream& operator<<(void* n)	
	ostream& operator<<(streambuf* sbptr)	sbptr の出力列を出力ストリームに挿入します。
	ostream& put(char c)	文字 c を出力ストリームに挿入します。
	ostream& write(const char* s, streamsize n)	s の n 個の文字を出力ストリームに挿入します。
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	出力ストリームをフラッシュします。この関数は streambuf::pubsync() を呼び出します。
	pos_type tellp()	現在の書き込み位置を求めます。この関数は streambuf::pubseekoff(0,cur,out) を呼び出します。
	ostream& seekp(pos_type pos)	ストリームの先頭から現在の位置までのオフセットを求めます。現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos) を呼び出します。
	ostream& seekp(off_type off, seekdir dir)	dir を基準として、ストリームの書き込み位置を off 分だけ移動します。この関数は streambuf::pubseekoff(off,dir) を呼び出します。

ostream::ostream(streambuf* sbptr)

コンストラクタです。

ios(sbptr) **を呼び出します。**

virtual ostream::~ostream()

デストラクタです。

ostream& ostream::operator<<(bool n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(short n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(unsigned short n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(int n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(unsigned int n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(long n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(unsigned long n)

sentry::ok_==true **のとき、**n **を出力ストリームに挿入します。**

sentry::ok_==false **のとき、**failbit **を設定します。**

リターン値は *this **です。**

ostream& ostream::operator<<(long long n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::operator<<(unsigned long long n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::operator<<(float n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::operator<<(double n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::operator<<(long double n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::operator<<(void* n)

sentry::ok_==true のとき、n を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

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

sentry::ok_==true のとき、sbptr の出力列を出力ストリームに挿入します。

sentry::ok_==false のとき、failbit を設定します。

リターン値は *this です。

ostream& ostream::put(char c)

sentry::ok_==true かつ rdbuf()->sputc(c)!=streambuf::eof のとき、c を出力ストリームに挿入します。

上記以外のとき、badbit を設定します。

リターン値は *this です。

ostream& ostream::write(const char* s, streamsize n)

sentry::ok_==true かつ rdbuf()->sputn(s, n)==n のとき、s の n 個の文字を出力ストリームに挿入します。

上記以外のとき、badbit を設定します。

リターン値は *this です。

```
ostream& ostream::write(const signed char* s, streamsize n)
```

sentry::ok_==true かつ rdbuf()->sputn(s, n)==n のとき、s の n 個の文字を出力ストリームに挿入します。

上記以外のとき、badbit を設定します。

リターン値は *this です。

```
ostream& ostream::write(const unsigned char* s, streamsize n)
```

sentry::ok_==true かつ rdbuf()->sputn(s, n)==n のとき、s の n 個の文字を出力ストリームに挿入します。

上記以外のとき、badbit を設定します。

リターン値は *this です。

```
ostream& ostream::flush()
```

出力ストリームをフラッシュします。

この関数は streambuf::pubsync() を呼び出します。

リターン値は *this です。

```
pos_type ostream::tellp()
```

現在の書き込み位置を求めます。

この関数は streambuf::pubseekoff(0,cur,out) を呼び出します。

リターン値は次のとおりです。

現在のストリームの位置

ただし、処理中にエラーが発生した場合は -1

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

エラーがないとき、ストリームの先頭から現在の位置までのオフセットを求めます。

また、現在のストリームポインタから pos だけ移動します。

この関数は streambuf::pubseekpos(pos) を呼び出します。

リターン値は *this です。

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

エラーがないとき、dir を基準として off 分ストリームの位置を移動します。

この関数は streambuf::pubseekoff(off,dir) を呼び出します。

リターン値は *this です。

(l) ostream クラスマニピュレータ

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

ostream& endl(ostream& os)

ストリームに改行文字を挿入します。

出力ストリームをフラッシュします。この関数は flush() を呼び出します。

リターン値は os です。

ostream& ends(ostream& os)

出力ストリームにヌルコードを挿入します。

リターン値は os です。

ostream& flush(ostream& os)

出力ストリームをフラッシュします。この関数は streambuf::sync() を呼び出します。

リターン値は os です。

(m) ostream メンバ外関数

種別	定義名	説明
関数	ostream& operator<<(ostream& os, char s)	s を出力ストリームに挿入します。
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const singed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

ostream& operator<<(ostream& os, char s)

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

ostream& operator<<(ostream& os, signed char s)

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

ostream& operator<<(ostream& os, unsigned char s)

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

```
ostream& operator<<(ostream& os, const char* s)
```

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

```
ostream& operator<<(ostream& os, const singed char* s)
```

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

```
ostream& operator<<(ostream& os, const unsigned char* s)
```

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。

上記以外のとき、failbit を設定します。

リターン値は os です。

（n）smanip クラスマニピュレータ

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

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

mask 値で指定されたフラグをクリアします。

リターン値は、入出力対象のオブジェクトです。

```
smanip setiosflags(ios_base::fmtflags mask)
```

書式フラグ (fmtfl) を設定します。

リターン値は、入出力対象のオブジェクトです。

```
smanip setbase(int base)
```

出力時に用いる基数を設定します。

リターン値は、入出力対象のオブジェクトです。

```
smanip setfill(char c)
```

詰め文字 (fillch) を設定します。

リターン値は、入出力対象のオブジェクトです。

smanip setprecision(int n)

精度 (prec) **を設定します。**

リターン値は、入出力対象のオブジェクトです。

smanip setw(int n)

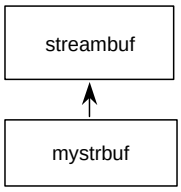
フィールド幅 (wide) **を設定します。**

リターン値は、入出力対象のオブジェクトです。

(o) EC++ 入出力ライブラリの使用例

istream, ostream のオブジェクトの初期化時に streambuf のかわりに mystrbuf クラスのオブジェクトへのポインタを使うことにより入出力ストリームが使用可能になります。

クラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。



種別	定義名	説明
変数	_file_Ptr	ファイルポインタです。
関数	mystrbuf)	コンストラクタです。streambuf バッファの初期化を行います。
	mystrbuvoid* ptr)	
	virtual ~mystrbuf()	デストラクタです。
	void* myfptr) const	FILE 型構造体へのポインタを返します。
	mystrbuf* openconst char* filename, int mode)	ファイル名とモードを指定して、ファイルをオープンします。
	mystrbuf* close()	ファイルのクローズを行います。
	virtual streambuf* setbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual int sync()	ストリームをフラッシュします。
	virtual int showmanyc()	入力ストリームの有効な文字数を返します。
	virtual int_type underflow()	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type pbackfail(int_type c = streambuf::eof)	c によって示される文字をプットバックします。

種別	定義名	説明
関数	virtual int_type overflow(int_type c = streambuf::eof)	c によって示される文字を挿入します。
	void _Init(_f_type* fp)	初期処理です。

例

```
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
#include <stdio.h>
void main(void)
{
    mystrbuf myfin(stdin);
    mystrbuf myfout(stdout);
    istream mycin(&myfin);
    ostream mycout(&myfout);

    int i;
    short s;
    long l;
    char c;
    string str;

    mycin >> i >> s >> l >> c >> str;
    mycout << "This is EC++ Library." << endl
           << i << s << l << c << str << endl;
    return;
}
```

6. 5. 2 メモリ管理用ライブラリ

メモリの管理用ライブラリに対応するヘッダファイルは以下の通りです。

- <new>

メモリの確保・解放を行う関数を定義します。

_ec2p_new_handler 変数に例外処理関数のアドレスを設定することにより、メモリ確保に失敗した場合、例外処理を実行することができます。

_ec2p_new_handler は static 変数で、初期値は NULL です。このハンドラを使用することにより、リエントラント性は失われます。

例外処理関数に要求される動作：

- 割り当て可能な領域を作成して返します。
- 作成できない場合の動作は規定されていません。

種別	定義名	説明
型	new_handler	void 型を返す関数へのポインタ型です。
変数	_ec2p_new_handler	例外処理関数へのポインタです。
関数	void* operator new(size_t size)	size 分の領域を確保します。
	void* operator new[ ](size_t size)	size 分の配列領域を確保します。
	void* operator new(size_t size, void* ptr)	ptr の指している領域を記憶領域として割り当てます。
	void* operator new[ ](size_t size, void* ptr)	ptr の指している領域を配列領域として割り当てます。
	void operator delete(void* ptr)	領域を解放します。
	void operator delete[ ](void* ptr)	配列領域を解放します。
	new_handler set_new_handler(new_handler new_P)	_ec2p_new_handler に例外処理関数アドレス(new_P)を設定します。

```
void* operator new(size_t size)
```

size バイト分の領域を割り当てます。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

```
void* operator new[ ](size_t size)
```

size 分の配列領域を確保します。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

```
void* operator new(size_t size, void* ptr)
```

ptr の指している領域を記憶領域として割り当てます。

リターン値は ptr です。

```
void* operator new[ ](size_t size, void* ptr)
```

ptr の指している領域を配列領域として割り当てます。

リターン値は ptr です。

```
void operator delete(void* ptr)
```

ptr が指す記憶領域を解放します。ptr が NULL のときは何もしません。

```
void operator delete[ ](void* ptr)
```

ptr が指す配列領域を解放します。ptr が NULL のときは何もしません。

new_handler set_new_handler(new_handler new_P)

_ec2p_new_handler に new_P を設定します。

リターン値は _ec2p_new_handler です。

6. 5. 3 複素数計算用クラスライブラリ

複素数計算用クラスライブラリに対応するヘッダファイルは以下のとおりです。

- <complex>

float_complex クラス、double_complex クラスを定義します。

これらのクラスには派生関係はありません。

(a) float_complex クラス

種別	定義名	説明
型	value_type	float 型です。
変数	_re	float 精度の実数部を定義します。
	_im	float 精度の虚数部を定義します。
関数	float_complex(float re = 0.0f, float im = 0.0f)	コンストラクタです。
	float_complex(const double_complex& rhs)	
	float real() const	実数部 (_re) を求めます。
	float imag() const	虚数部 (_im) を求めます。
	float_complex& operator=(float rhs)	rhs を実数部にコピーします。虚数部は 0.0f を設定します。
	float_complex& operator+=(float rhs)	rhs を実数部に加算し、和を *this に格納します。
	float_complex& operator-=(float rhs)	rhs を実数部から減算し、差を *this に格納します。
	float_complex& operator*=(float rhs)	rhs を乗算し、積を *this に格納します。
	float_complex& operator/=(float rhs)	rhs で除算し、商を *this に格納します。
	float_complex& operator=(const float_complex& rhs)	rhs をコピーします。
	float_complex& operator+=(const float_complex& rhs)	rhs を加算し、和を *this に格納します。
	float_complex& operator-=(const float_complex& rhs)	rhs を減算し、差を *this に格納します。
	float_complex& operator*=(const float_complex& rhs)	rhs を乗算し、積を *this に格納します。
	float_complex& operator/=(const float_complex& rhs)	rhs で除算し、商を *this に格納します。

float_complex::float_complex(float re = 0.0f, float im = 0.0f)

クラス float_complex のコンストラクタです。

以下の値で初期化します。

_re=re;

_im=im;

float_complex::float_complex(const double_complex& rhs)

クラス float_complex **のコンストラクタです。**

以下の値で初期化します。

_re=(float)rhs.real();

_im=(float)rhs.imag();

float float_complex::real() const

実数部を求めます。

リターン値は、this->_re **です。**

float float_complex::imag() const

虚数部を求めます。

リターン値は、this->_im **です。**

float_complex& float_complex::operator=(float rhs)

rhs **を実数部** (_re) **にコピーします。虚数部** (_im) **は** 0.0f **を設定します。**

リターン値は *this **です。**

float_complex& float_complex::operator+=(float rhs)

rhs **を実数部** (_re) **に加算し、結果を実数部** (_re) **に格納します。虚数部** (_im) **の値は変わりません。**

リターン値は *this **です。**

float_complex& float_complex::operator-=(float rhs)

rhs **を実数部** (_re) **から減算し、結果を実数部** (_re) **に格納します。虚数部** (_im) **の値は変わりません。**

リターン値は *this **です。**

float_complex& float_complex::operator*=(float rhs)

rhs **と乗算し、結果を** *this **に格納します。**

(_re=_re*rhs, _im=_im*rhs)

リターン値は *this **です。**

float_complex& float_complex::operator/=(float rhs)

rhs **で除算し、結果を** *this **に格納します。**

(_re=_re/rhs, _im=_im/rhs)

リターン値は *this **です。**

float_complex& float_complex::operator=(const float_complex& rhs)

rhs **をコピーします。**

リターン値は *this **です。**

float_complex& float_complex::operator+=(const float_complex& rhs)

rhs を加算し、結果を *this に格納します。

リターン値は *this です。

float_complex& float_complex::operator-=(const float_complex& rhs)

rhs を減算し、結果を *this に格納します。

リターン値は *this です。

float_complex& float_complex::operator*=(const float_complex& rhs)

rhs と乗算し、結果を *this に格納します。

リターン値は *this です。

float_complex& float_complex::operator/=(const float_complex& rhs)

rhs で除算し、結果を *this に格納します。

リターン値は *this です。

(b) float_complex メンバ外関数

種別	定義名	説明
関数	float_complex operator+(const float_complex& lhs)	lhs の単項＋演算を行います。
	float_complex operator+(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を加算した結果を返却します。
	float_complex operator+(const float_complex& lhs, const float& rhs)	
	float_complex operator+(const float& lhs, const float_complex& rhs)	
	float_complex operator-(const float_complex& lhs)	lhs の単項－演算を行います。
	float_complex operator-(const float_complex& lhs, const float_complex& rhs)	lhs から rhs を減算した結果を返却します。
	float_complex operator-(const float_complex& lhs, const float& rhs)	
	float_complex operator-(const float& lhs, const float_complex& rhs)	
	float_complex operator*(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を乗算した結果を返却します。
	float_complex operator*(const float_complex& lhs, const float& rhs)	
	float_complex operator*(const float& lhs, const float_complex& rhs)	
	float_complex operator/(const float_complex& lhs, const float_complex& rhs)	lhs を rhs で除算した結果を返却します。
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	

種別	定義名	説明
関数	bool operator==(const float_complex& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const float_complex& lhs, const float& rhs)	
	bool operator==(const float& lhs, const float_complex& rhs)	
	bool operator!=(const float_complex& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator!=(const float_complex& lhs, const float& rhs)	
	bool operator!=(const float& lhs, const float_complex& rhs)	
	istream& operator>>(istream& is, float_complex& x)	u,(u), または (u,v) (u: 実数部、v: 虚数部) 形式の x を入力します。
	ostream& operator<<(ostream& os, float_complex& x)	x を u,(u) または (u,v) (u: 実数部、v: 虚数部) 形式で出力します。
	float real(const float_complex& x)	実数部を求めます。
	float imag(const float_complex& x)	虚数部を求めます。
	float abs(const float_comlex& x)	絶対値を求めます。
	float arg(const float_complex& x)	位相角度を求めます。
	float norm(const float_complex& x)	２乗の絶対値を求めます。
	float_complex conj(const float_complex& x)	共役複素数を求めます。
	float_complex polar(const float& rho, const float& theta)	大きさが rho で位相角度が theta の複素数に対応する float_complex 値を求めます。
	float_complex cos(const float_complex& x)	複素余弦を求めます。
	float_complex cosh(const float_complex& x)	複素双曲余弦を求めます。
	float_complex exp(const float_complex& x)	指数関数を求めます。
	float_complex log(const float_complex& x)	自然対数を求めます。
	float_complex log10(const float_complex& x)	常用対数を求めます。

種別	定義名	説明
関数	float_complex pow(const float_complex& x, int y)	x の y 乗を求めます。
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	
	float_complex pow(const float& x, const float_complex& y)	
	float_complex sin(const float_complex& x)	複素正弦を求めます。
	float_complex sinh(const float_complex& x)	複素双曲正弦を求めます。
	float_complex sqrt(const float_complex& x)	右半空間における範囲での平方根を求めます。
	float_complex tan(const float_complex& x)	複素正接を求めます。
	float_complex tanh(const float_complex& x)	複素双曲正接を求めます。

```
float_complex operator+(const float_complex& lhs)
```

lhs の単項＋演算を行います。

リターン値は lhs です。

```
float_complex operator+(const float_complex& lhs, const float_complex& rhs)
```

lhs と rhs を加算した結果を返却します。

リターン値は、float_complex(lhs)+=rhs です。

```
float_complex operator+(const float_complex& lhs, const float& rhs)
```

lhs と rhs を加算した結果を返却します。

リターン値は、float_complex(lhs)+=rhs です。

```
float_complex operator+(const float& lhs, const float_complex& rhs)
```

lhs と rhs を加算した結果を返却します。

リターン値は、float_complex(lhs)+=rhs です。

```
float_complex operator-(const float_complex& lhs)
```

lhs の単項－演算を行います。

リターン値は、float_complex(-lhs.real(),-lhs.imag()) です。

```
float_complex operator-(const float_complex& lhs, const float_complex& rhs)
```

lhs から rhs を減算した結果を返却します。

リターン値は、float_complex(lhs)-=rhs **です。**

```
float_complex operator-(const float_complex& lhs, const float& rhs)
```

lhs **から** rhs **を減算した結果を返却します。**

リターン値は、float_complex(lhs)-=rhs **です。**

```
float_complex operator-(const float& lhs, const float_complex& rhs)
```

lhs **から** rhs **を減算した結果を返却します。**

リターン値は、float_complex(lhs)-=rhs **です。**

```
float_complex operator*(const float_complex& lhs, const float_complex& rhs)
```

lhs **と** rhs **を乗算した結果を返却します。**

リターン値は、float_complex(lhs)*=rhs **です。**

```
float_complex operator*(const float_complex& lhs, const float& rhs)
```

lhs **と** rhs **を乗算した結果を返却します。**

リターン値は、float_complex(lhs)*=rhs **です。**

```
float_complex operator*(const float& lhs, const float_complex& rhs)
```

lhs **と** rhs **を乗算した結果を返却します。**

リターン値は、float_complex(lhs)*=rhs **です。**

```
float_complex operator/(const float_complex& lhs, const float_complex& rhs)
```

lhs **を** rhs **で除算した結果を返却します。**

リターン値は、float_complex(lhs)/=rhs **です。**

```
float_complex operator/(const float_complex& lhs, const float& rhs)
```

lhs **を** rhs **で除算した結果を返却します。**

リターン値は、float_complex(lhs)/=rhs **です。**

```
float_complex operator/(const float& lhs, const float_complex& rhs)
```

lhs **を** rhs **で除算した結果を返却します。**

リターン値は、float_complex(lhs)/=rhs **です。**

```
bool operator==(const float_complex& lhs, const float_complex& rhs)
```

lhs **と** rhs **の実数部どうし、虚数部どうしを比較します。**float **型引数の場合、虚数部は** float **型の** 0.0f **と仮定されます。**

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() **です。**

```
bool operator==(const float_complex& lhs, const float& rhs)
```

lhs **と** rhs **の実数部どうし、虚数部どうしを比較します。**float **型引数の場合、虚数部は** float **型の** 0.0f **と仮定されます。**

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() **です。**

```
bool operator==(const float& lhs, const float_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() です。

```
bool operator!=(const float_complex& lhs, const float_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
bool operator!=(const float_complex& lhs, const float& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
bool operator!=(const float& lhs, const float_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
istream& operator>>(istream& is, float_complex& x)
```

u,(u), または (u,v) (u は実数部、v は虚数部) の形式の x を入力します。入力値は float_complex に変換されます。

u,(u),(u,v) 形式以外が入力された場合は、is.setstate(ios_base::failbit) を呼びます。

リターン値は is です。

```
ostream& operator<<(ostream& os, const float_complex& x)
```

x を os に出力します。

出力形式は u,(u) または (u,v) (u は実数部、v は虚数部) です。

リターン値は os です。

```
float real(const float_complex& x)
```

実数部を求めます。

リターン値は x.real() です。

```
float imag(const float_complex& x)
```

虚数部を求めます。

リターン値は x.imag() です。

```
float abs(const float_complex& x)
```

絶対値を求めます。

リターン値は、$(|x.real()|^2 + |x.imag()|^2)^{1/2}$ です。

```
float arg(const float_complex& x)
```

位相角度を求めます。

リターン値は、atan2f(x.imag() , x.real()) **です。**

```
float norm(const float_complex& x)
```

2 **乗の絶対値を求めます。**

リターン値は、$|x.real()|^2 + |x.imag()|^2$ **です。**

```
float_complex conj(const float_complex& x)
```

共役複素数を求めます。

リターン値は、float_complex(x.real(), (-1)*x.imag()) **です。**

```
float_complex polar(const float& rho, const float& theta)
```

大きさが rho **で位相角度 (偏角) が** theta **の複素数に対応する** float_complex **値を求めます。**

リターン値は、float_complex(rho*cosf(theta), rho*sinf(theta)) **です。**

```
float_complex cos(const float_complex& x)
```

複素余弦を求めます。

リターン値は、float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag())) **です。**

```
float_complex cosh(const float_complex& x)
```

複素双曲余弦を求めます。

リターン値は、cos(float_complex((-1)*x.imag(), x.real())) **です。**

```
float_complex exp(const float_complex& x)
```

指数関数を求めます。

リターン値は、expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag()) **です。**

```
float_complex log(const float_complex& x)
```

(e **を底とする**) **自然対数を求めます。**

リターン値は、float_complex(logf(abs(x)), arg(x)) **です。**

```
float_complex log10(const float_complex& x)
```

(10 **を底とする**) **常用対数を求めます。**

リターン値は、float_complex(log10f(abs(x)), arg(x)/logf(10)) **です。**

```
float_complex pow(const float_complex& x, int y)
```

x **の** y **乗を求めます。**

pow(0,0) **のとき、定義域エラーになります。**

リターン値は次のとおりです。

float_complex pow(const float_complex& x,const float_complex& y) **の場合:** exp(y*logf(x))

上記以外: exp(y*log(x))

float_complex pow(const float_complex& x, const float& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は次のとおりです。

float_complex pow(const float_complex& x,const float_complex& y) の場合：exp(y*logf(x))

上記以外：exp(y*log(x))

float_complex pow(const float_complex& x, const float_complex& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は次のとおりです。

float_complex pow(const float_complex& x,const float_complex& y) の場合：exp(y*logf(x))

上記以外：exp(y*log(x))

float_complex pow(const float& x, const float_complex& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は次のとおりです。

float_complex pow(const float_complex& x,const float_complex& y) の場合：exp(y*logf(x))

上記以外：exp(y*log(x))

float_complex sin(const float_complex& x)

複素正弦を求めます。

リターン値は、float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag())) です。

float_complex sinh(const float_complex& x)

複素双曲正弦を求めます。

リターン値は、float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real())) です。

float_complex sqrt(const float_complex& x)

右半空間における範囲での平方根を求めます。

リターン値は、float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2)) です。

float_complex tan(const float_complex& x)

複素正接を求めます。

リターン値は、sin(x)/cos(x) です。

float_complex tanh(const float_complex& x)

複素双曲正接を求めます。

リターン値は、sinh(x)/cosh(x) です。

(c) double_complex クラス

種別	定義名	説明
型	value_type	double 型です。
変数	_re	double 精度の実数部を定義します。
	_im	double 精度の虚数部を定義します。
関数	double_complex(double re = 0.0, double im = 0.0)	コンストラクタです。
	double_complex(const float_complex&)	
	double real() const	実数部を求めます。
	double imag() const	虚数部を求めます。
	double_complex& operator=(double rhs)	rhs を実数部にコピーします。虚数部は 0.0 を設定します。
	double_complex& operator+=(double rhs)	rhs を実数部に加算し、和を *this に格納します。
	double_complex& operator-=(double rhs)	rhs を実数部から減算し、差を *this に格納します。
	double_complex& operator*=(double rhs)	rhs を乗算し、積を *this に格納します。
	double_complex& operator/=(double rhs)	rhs で除算し、商を *this に格納します。
	double_complex& operator=(const double_complex& rhs)	rhs をコピーします。
	double_complex& operator+=(const double_complex& rhs)	rhs を加算し、和を *this に格納します。
	double_complex& operator-=(const double_complex& rhs)	rhs を減算し、差を *this に格納します。
	double_complex& operator*=(const double_complex& rhs)	rhs を乗算し、積を *this に格納します。
	double_complex& operator/=(const double_complex& rhs)	rhs で除算し、商を *this に格納します。

```
double_complex::double_complex(double re = 0.0, double im = 0.0)
```

クラス double_complex のコンストラクタです。

以下の値で初期化します。

_re=re;

_im=im;

```
double_complex::double_complex(const float_complex&)
```

クラス double_complex のコンストラクタです。

以下の値で初期化します。

_re=(double)rhs.real();

_im=(double)rhs.imag();

```
double double_complex::real() const
```

実数部を求めます。

リターン値は、this->_re **です。**

double double_complex::imag() const

虚数部を求めます。

リターン値は、this->_im **です。**

double_complex& double_complex::operator=(double rhs)

rhs **を実数部** (_re) **にコピーします。虚数部** (_im) **は** 0.0 **を設定します。**

リターン値は *this **です。**

double_complex& double_complex::operator+=(double rhs)

rhs **を実数部** (_re) **に加算し、結果を実数部** (_re) **に格納します。虚数部** (_im) **の値は変わりません。**

リターン値は *this **です。**

double_complex& double_complex::operator-=(double rhs)

rhs **を実数部** (_re) **から減算し、結果を実数部** (_re) **に格納します。虚数部** (_im) **の値は変わりません。**

リターン値は *this **です。**

double_complex& double_complex::operator*=(double rhs)

rhs **と乗算し、結果を** *this **に格納します。**

(_re=_re*rhs, _im=_im*rhs)

リターン値は *this **です。**

double_complex& double_complex::operator/=(double rhs)

rhs **で除算し、結果を** *this **に格納します。**

(_re=_re/rhs, _im=_im/rhs)

リターン値は *this **です。**

double_complex& double_complex::operator=(const double_complex& rhs)

rhs **をコピーします。**

リターン値は *this **です。**

double_complex& double_complex::operator+=(const double_complex& rhs)

rhs **を加算し、結果を** *this **に格納します。**

リターン値は *this **です。**

double_complex& double_complex::operator-=(const double_complex& rhs)

rhs **を減算し、結果を** *this **に格納します。**

リターン値は *this **です。**

double_complex& double_complex::operator*=(const double_complex& rhs)

rhs **と乗算し、結果を** *this **に格納します。**

リターン値は *this **です。**

double_complex& double_complex::operator/=(const double_complex& rhs)

rhs **で除算し、結果を** *this **に格納します。**

リターン値は *this **です。**

(d) double_complex メンバ外関数

種別	定義名	説明
関数	double_complex operator+(const double_complex& lhs)	lhs の単項＋演算を行います。
	double_complex operator+(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を加算し、和を lhs に格納します。
	double_complex operator+(const double_complex& lhs, const double& rhs)	
	double_complex operator+(const double& lhs, const double_complex& rhs)	
	double_complex operator-(const double_complex& lhs)	lhs の単項－演算を行います。
	double_complex operator-(const double_complex& lhs, const double_complex& rhs)	lhs から rhs を減算し、差を lhs に格納します。
	double_complex operator-(const double_complex& lhs, const double& rhs)	
	double_complex operator-(const double& lhs, const double_complex& rhs)	
	double_complex operator*(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	double_complex operator*(const double_complex& lhs, const double& rhs)	
	double_complex operator*(const double& lhs, const double_complex& rhs)	
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	lhs を rhs で除算し、商を lhs に格納します。
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	

種別	定義名	説明
関数	bool operator==(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	
	bool operator!=(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator!=(const double_complex& lhs, const double& rhs)	
	bool operator!=(const double& lhs, const double_complex& rhs)	
	istream& operator>>(istream& is, double_complex& x)	u,(u) または (u,v) (u: 実数部、v: 虚数部) 形式の x を入力します。
	ostream& operator<<(ostream& os, const double_complex& x)	x を u,(u) または (u,v) (u: 実数部、v: 虚数部) 形式で出力します。
	double real(const double_complex& x)	実数部を求めます。
	double imag(const double_complex& x)	虚数部を求めます。
	double abs(const double_comlex& x)	絶対値を求めます。
	double arg(const double_complex& x)	位相角度を求めます。
	double norm(const double_complex& x)	２乗の絶対値を求めます。
	double_complex conj(const double_complex& x)	共役複素数を求めます。
	double_complex polar(const double& rho, const double& theta)	大きさが rho で位相角度が theta の複素数に対応する double_complex 値を求めます。
	double_complex cos(const double_complex& x)	複素余弦を求めます。
	double_complex cosh(const double_complex& x)	複素双曲余弦を求めます。
	double_complex exp(const double_complex&)	指数関数を求めます。

種別	定義名	説明
関数	double_complex log(const double_complex& x)	自然対数を求めます。
	double_complex log10(const double_complex& x)	常用対数を求めます。
	double_complex pow(const double _complex& x, int y)	x の y 乗を求めます。
	double_complex pow(const double _complex& x, const double & y)	
	double_complex pow(const double _complex& x, const double _complex& y)	
	double_complex pow(const double & x, const double _complex& y)	
	double_complex sin(const double_complex& x)	複素正弦を求めます。
	double_complex sinh(const double_complex& x)	複素双曲正弦を求めます。
	double_complex sqrt(const double_complex& x)	右半空間における範囲での平方根を求めます。
	double_complex tan(const double_complex& x)	複素正接を求めます。
	double_complex tanh(const double_complex& x)	複素双曲正接を求めます。

```
double_complex operator+(const double_complex& lhs)
```

lhs の単項＋演算を行います。

リターン値は lhs です。

```
double_complex operator+(const double_complex& lhs, const double_complex& rhs)
```

lhs と rhs を加算し、結果を lhs に格納します。

リターン値は、double_complex(lhs)+=rhs です。

```
double_complex operator+(const double_complex& lhs, const double& rhs)
```

lhs と rhs を加算し、結果を lhs に格納します。

リターン値は、double_complex(lhs)+=rhs です。

```
double_complex operator+(const double& lhs, const double_complex& rhs)
```

lhs と rhs を加算し、結果を lhs に格納します。

リターン値は、double_complex(lhs)+=rhs **です。**

double_complex operator-(const double_complex& lhs)

lhs **の単項－演算を行います。**

リターン値は、double_complex(-lhs.real(), -lhs.imag()) **です。**

double_complex operator-(const double_complex& lhs, const double_complex& rhs)

lhs **から** rhs **を減算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)-=rhs **です。**

double_complex operator-(const double_complex& lhs, const double& rhs)

lhs **から** rhs **を減算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)-=rhs **です。**

double_complex operator-(const double& lhs, const double_complex& rhs)

lhs **から** rhs **を減算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)-=rhs **です。**

double_complex operator*(const double_complex& lhs, const double_complex& rhs)

lhs **と** rhs **を乗算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)*=rhs **です。**

double_complex operator*(const double_complex& lhs, const double& rhs)

lhs **と** rhs **を乗算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)*=rhs **です。**

double_complex operator*(const double& lhs, const double_complex& rhs)

lhs **と** rhs **を乗算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)*=rhs **です。**

double_complex operator/(const double_complex& lhs, const double_complex& rhs)

lhs **を** rhs **で除算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)/=rhs **です。**

double_complex operator/(const double_complex& lhs, const double& rhs)

lhs **を** rhs **で除算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)/=rhs **です。**

double_complex operator/(const double& lhs, const double_complex& rhs)

lhs **を** rhs **で除算し、結果を** lhs **に格納します。**

リターン値は、double_complex(lhs)/=rhs **です。**

bool operator==(const double_complex& lhs, const double_complex& rhs)

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() です。

```
bool operator==(const double_complex& lhs, const double& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() です。

```
bool operator==(const double& lhs, const double_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag() です。

```
bool operator!=(const double_complex& lhs, const double_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
bool operator!=(const double_complex& lhs, const double& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
bool operator!=(const double& lhs, const double_complex& rhs)
```

lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。

リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag() です。

```
istream& operator>>(istream& is, double_complex& x)
```

u,(u) または (u,v) (u は実数部、v は虚数部) の形式の複素数 x を入力します。入力値は double_complex に変換されます。

u,(u),(u,v) 形式以外が入力された場合は、is.setstate(ios_base::failbit) を呼びます。

リターン値は is です。

```
ostream& operator<<(ostream& os, const double_complex& x)
```

x を os に出力します。

出力形式は u,(u) または (u,v) (u は実数部、v は虚数部) です。

リターン値は os です。

```
double real(const double_complex& x)
```

実数部を求めます。

リターン値は x.real() です。

```
double imag(const double_complex& x)
```

虚数部を求めます。

リターン値は x.imag() **です。**

```
double abs(const double_complex& x)
```

絶対値を求めます。

リターン値は、$(|x.real()|^2 + |x.imag()|^2)^{1/2}$ **です。**

```
double arg(const double_complex& x)
```

位相角度を求めます。

リターン値は、atan2(x.imag() , x.real()) **です。**

```
double norm(const double_complex& x)
```

2 **乗の絶対値を求めます。**

リターン値は、$|x.real()|^2 + |x.imag()|^2$ **です。**

```
double_complex conj(const double_complex& x)
```

共役複素数を求めます。

リターン値は、double_complex(x.real(), (-1)*x.imag()) **です。**

```
double_complex polar(const double& rho, const double& theta)
```

大きさが rho **で位相角度 (偏角) が** theta **の複素数に対応する** double_complex **値を求めます。**

リターン値は、double_complex(rho*cos(theta), rho*sin(theta)) **です。**

```
double_complex cos(const double_complex& x)
```

複素余弦を求めます。

リターン値は、double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag())) **です。**

```
double_complex cosh(const double_complex& x)
```

複素双曲余弦を求めます。

リターン値は、cos(double_complex((-1)*x.imag(), x.real())) **です。**

```
double_complex exp(const double_complex& x)
```

指数関数を求めます。

リターン値は、exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag()) **です。**

```
double_complex log(const double_complex& x)
```

(e **を底とする**) **自然対数を求めます。**

リターン値は、double_complex(log(abs(x)), arg(x)) **です。**

```
double_complex log10(const double_complex& x)
```

(10 **を底とする**) **常用対数を求めます。**

リターン値は、double_complex(log10(abs(x)), arg(x)/log(10)) **です。**

double_complex pow(const double_complex& x, int y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は、exp(y*log(x)) です。

double_complex pow(const double_complex& x, const double& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は、exp(y*log(x)) です。

double_complex pow(const double_complex& x, const double_complex& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は、exp(y*log(x)) です。

double_complex pow(const double& x, const double_complex& y)

x の y 乗を求めます。

pow(0,0) のとき、定義域エラーになります。

リターン値は、exp(y*log(x)) です。

double_complex sin(const double_complex& x)

複素正弦を求めます。

リターン値は、double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag())) です。

double_complex sinh(const double_complex& x)

複素双曲正弦を求めます。

リターン値は、double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real())) です。

double_complex sqrt(const double_complex& x)

右半空間における範囲での平方根を求めます。

リターン値は、double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2)) です。

double_complex tan(const double_complex& x)

複素正接を求めます。

リターン値は、sin(x)/cos(x) です。

double_complex tanh(const double_complex& x)

複素双曲正接を求めます。

リターン値は、sinh(x)/cosh(x) です。

6. 5. 4 文字列操作用用クラスライブラリ

文字列操作用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <string>

  string クラスを定義します。

本クラスには派生関係はありません。

(a) string クラス

種別	定義名	説明
型	iterator	char* 型です。
	const_iterator	const char* 型です。
定数	npos	文字列の最大長 (UINT_MAX 文字) です。
変数	s_ptr	オブジェクトが文字列を格納している領域へのポインタです。
	s_len	オブジェクトが格納している文字列長です。
	s_res	オブジェクトが文字列を格納するために確保している領域のサイズです。
関数	string(void)	コンストラクタです。
	string::string(const string& str, size_t pos = 0, size_t n = npos)	
	string::string(const char* str, size_t n)	
	string::string(const char* str)	
	string::string(size_t n, char c)	
	~string()	デストラクタです。
	string& operator=(const string& str)	str を代入します。
	string& operator=(const char* str)	
	string& operator=(char c)	c を代入します。
	iterator begin()	文字列の先頭ポインタを求めます。
	const_iterator begin() const	
	iterator end()	文字列の最後尾ポインタを求めます。
	const_iterator end() const	
	size_t size() const	格納されている文字列の文字列長を求めます。
	size_t length() const	
	size_t max_size() const	確保している領域のサイズを求めます。
	void resize(size_t n, char c)	格納可能な文字列の文字数を n に変更します。
	void resize(size_t n)	格納可能な文字列の文字数を n に変更します。

種別	定義名	説明
関数	size_t capacity() const	確保している領域のサイズを求めます。
	void reserve(size_t res_arg = 0)	領域の再割り当てを行います。
	void clear()	格納されている文字列を clear します。
	bool empty() const	格納している文字列の文字数が 0 かチェックします。
	const char& operator[](size_t pos) const	s_ptr[pos] を参照します。
	char& operator[](size_t pos)	
	const char& at(size_t pos) const	
	char& at(size_t pos)	
	string& operator+=(const string& str)	str の文字列を追加します。
	string& operator+=(const char* str)	
	string& operator+=(char c)	c の文字を追加します。
	string& append(const string& str)	str の文字列を追加します。
	string& append(const char* str)	
	string& append(const string& str, size_t pos, size_t n)	オブジェクトの位置 pos に str の文字列を n 文字分追加します。
	string& append(const char* str, size_t n)	文字列 str の n 文字分を追加します。
	string& append(size_t n, char c)	n 個の文字 c を追加します。
	string& assign(const string& str)	str の文字列を代入します。
	string& assign(const char* str)	
	string& assign(const string& str, size_t pos, size_t n)	位置 pos に文字列 str の n 文字分を代入します。
	string& assign(const char* str, size_t n)	文字列 str の n 文字分を代入します。
	string& assign(size_t n, char c)	n 個の文字 c を代入します。
	string& insert(size_t pos1, const string& str)	位置 pos1 に str の文字列を挿入します。
	string& insert(size_t pos1, const string& str, size_t pos2, size_t n)	位置 pos1 に str の文字列の位置 pos2 から n 文字分を挿入します。
	string& insert(size_t pos, const char* str, size_t n)	pos の位置に文字列 str を n 文字分挿入します。
	string& insert(size_t pos, const char* str)	pos の位置に文字列 str を挿入します。

種別	定義名	説明
関数	string& insert(size_t pos, size_t n, char c)	位置 pos に n 個の文字 c の文字列を挿入します。
	iterator insert(iterator p, char c = char())	p が指す文字列の前に文字 c を挿入します。
	void insert(iterator p, size_t n, char c)	p が指す文字の前に、n 個の文字 c を挿入します。
	string& erase(size_t pos = 0, size_t n = npos)	位置 pos から n 個分取り除きます。
	iterator erase(iterator position)	position により参照された文字を取り除きます。
	iterator erase(iterator first, iterator last)	範囲 [first, last] において文字を取り除きます。
	string& replace(size_t pos1, size_t n1, const string& str)	位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。
	string& replace(size_t pos1, size_t n1, const char* str)	
	string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	位置 pos1 から n1 文字分の文字列を、str の位置 pos2 から n2 文字分の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, const char* str, size_t n2)	位置 pos から n1 文字分の文字列を、n2 個の str の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, size_t n2, char c)	位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。
	string& replace(iterator i1, iterator i2, const string& str)	位置 i1 から i2 までの文字列を str の文字列で置き換えます。
	string& replace(iterator i1, iterator i2, const char* str)	

種別	定義名	説明
関数	string& replace(iterator i1, iterator i2, const char* str, size_t n)	位置 i1 から i2 までの文字列を str の文字列の n 文字分で置き換えます。
	string& replace(iterator i1, iterator i2, size_t n, char c)	位置 i1 から i2 までの文字列を n 個の文字 c で置き換えます。
	size_t copy(char* str, size_t n, size_t pos = 0) const	位置 pos に文字列 str の n 文字分の文字列をコピーします。
	void swap(string& str)	str の文字列と交換します。
	const char* c_str() const	文字列を格納している領域へのポインタを参照します。
	const char* data() const	
	size_t find(const string& str, size_t pos = 0) const	位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。
	size_t find(const char* str, size_t pos = 0) const	
	size_t find(const char* str, size_t pos, size_t n) const	位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。
	size_t find(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t rfind(const string& str, size_t pos = npos) const	位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。
	size_t rfind(const char* str, size_t pos = npos) const	
	size_t rfind(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の n 文字分と同じ文字列が最後に現れる位置を検索します。
	size_t rfind(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。

種別	定義名	説明
関数	size_t find_first_of(const string& str, size_t pos = 0) const	位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(const char* str, size_t pos = 0) const	
	size_t find_first_of(const char* str, size_t pos, size_t n) const	位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t find_last_of(const string& str, size_t pos = npos) const	位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(const char* str, size_t pos = npos) const	
	size_t find_last_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。
	size_t find_first_not_of(const string& str, size_t pos = 0) const	位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(const char* str, size_t pos = 0) const	
	size_t find_first_not_of(const char* str, size_t pos, size_t n)	位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。
	size_t find_last_not_of(const string& str, size_t pos = npos) const	位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(const char* str, size_t pos = npos) const	

種別	定義名	説明
関数	size_t find_last_not_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。
	string substr(size_t pos = 0, size_t n = npos) const	格納された文字列に対し、範囲 [pos,n] の文字列を持つオブジェクトを生成します。
	int compare(const string& str) const	文字列と str の文字列を比較します。
	int compare(size_t pos1, size_t n1, const string& str) const	位置 pos1 から n1 文字分の文字列と str を比較します。
	int compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。
	int compare(const char* str) const	str と比較します。
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。

string::string(void)

以下のように設定します。

s_ptr=0;

s_len=0;

s_res=1;

string::string(const string& str, size_t pos = 0, size_t n = npos)

str **をコピーします。ただし、**s_len **は、**n **と** s_len **の小さい方の値になります。**

string::string(const char* str, size_t n)

以下に設定します。

s_ptr=str;

s_len=n;

s_res=n+1;

string::string(const char* str)

以下に設定します。

s_ptr=str;

s_len=str **の文字列長** ;

s_res=str **の文字列長** +1;

string::string(size_t n, char c)

以下に設定します。

s_ptr=**文字数** n **で文字** c **の文字列** ;

s_len=n;

s_res=n+1;

string::~string()

クラス string **のデストラクタです。**

文字列を格納している領域を解放します。

string& string::operator=(const string& str)

str **のデータを代入します。**

リターン値は *this **です。**

string& string::operator=(const char* str)

str **から** string **オブジェクトを生成し、そのデータを代入します。**

リターン値は *this **です。**

string& string::operator=(char c)

c **から** string **オブジェクトを生成し、そのデータを代入します。**

リターン値は *this **です。**

string::iterator string::begin()

文字列の先頭ポインタを求めます。

リターン値は、文字列の先頭ポインタです。

string::const_iterator string::begin() const

文字列の先頭ポインタを求めます。

リターン値は、文字列の先頭ポインタです。

string::iterator string::end()

文字列の最後尾ポインタを求めます。

リターン値は、文字列の最後尾ポインタです。

string::const_iterator string::end() const

文字列の最後尾ポインタを求めます。

リターン値は、文字列の最後尾ポインタです。

```
size_t string::size() const
```

格納されている文字列の文字列長を求めます。

リターン値は、格納されている文字列の文字列長です。

```
size_t string::length() const
```

格納されている文字列の文字列長を求めます。

リターン値は、格納されている文字列の文字列長です。

```
size_t string::max_size() const
```

確保している領域のサイズを求めます。

リターン値は、確保している領域のサイズです。

```
void string::resize(size_t n, char c)
```

オブジェクトが格納可能な文字列の文字数を n に変更します。

n<=size() のとき、長さを n にした元の文字列と置き換えます。

n>size() のとき、元の文字列の後ろに長さ n になるまで c をつめた文字列と置き換えます。

n<=max_size() である必要があります。

n>max_size() の場合 §n=max_size() として計算します。

```
void string::resize(size_t n)
```

オブジェクトが格納可能な文字列の文字数を n に変更します。

n<=size() のとき、長さを n にしたもとの文字列と置き換えます。

n<=max_size() である必要があります。

```
size_t string::capacity() const
```

確保している領域のサイズを求めます。

リターン値は、確保している領域のサイズです。

```
void string::reserve(size_t res_arg = 0)
```

記憶領域の再割り当てを行います。

reserve() 後、capacity() は reserve() の引数より大きいかまたは等しくなります。

再割り当てを行うと、すべての参照、ポインタ、この数列の中の要素の参照する iterator を無効にします。

```
void string::clear()
```

格納されている文字列をクリアします。

```
bool string::empty() const
```

格納している文字列の文字数が 0 かチェックします。

リターン値は次のとおりです。

格納している文字列長が 0 の場合：true

格納している文字列長が 0 以外の場合：false

```
const char& string::operator[](size_t pos) const
```

s_ptr[pos] を参照します。

リターン値は次のとおりです。

n<s_len **の場合：** s_ptr [pos]

n>=s_len **の場合：** '\0'

```
char& string::operator[](size_t pos)
```

s_ptr[pos] を参照します。

リターン値は次のとおりです。

n<s_len **の場合：** s_ptr [pos]

n>=s_len **の場合：** '\0'

```
const char& string::at(size_t pos) const
```

s_ptr[pos] を参照します。

リターン値は次のとおりです。

n<s_len **の場合：** s_ptr [pos]

n>=s_len **の場合：** '\0'

```
char& string::at(size_t pos)
```

s_ptr[pos] を参照します。

リターン値は次のとおりです。

n<s_len **の場合：** s_ptr [pos]

n>=s_len **の場合：** '\0'

```
string& string::operator+=(const string& str)
```

str **が格納している文字列を追加します。**

リターン値は *this **です。**

```
string& string::operator+=(const char* str)
```

str **から** string **オブジェクトを生成し、その文字列を追加します。**

リターン値は *this **です。**

```
string& string::operator+=(char c)
```

c **から** string **オブジェクトを生成し、その文字列を追加します。**

リターン値は *this **です。**

```
string& string::append(const string& str)
```

str **の文字列をオブジェクトに追加します。**

リターン値は *this **です。**

```
string& string::append(const char* str)
```

str **の文字列をオブジェクトに追加します。**

リターン値は *this **です。**

```
string& string::append(const string& str, size_t pos, size_t n)
```

オブジェクトの位置 pos **に** str **の文字列を** n **文字分追加します。**

リターン値は *this **です。**

```
string& string::append(const char* str, size_t n)
```

文字列 str **の** n **文字分を追加します。**

リターン値は *this **です。**

```
string& string::append(size_t n, char c)
```

n **個の文字** c **を追加します。**

リターン値は *this **です。**

```
string& string::assign(const string& str)
```

str **の文字列を代入します。**

リターン値は *this **です。**

```
string& string::assign(const char* str)
```

str **の文字列を代入します。**

リターン値は *this **です。**

```
string& string::assign(const string& str, size_t pos, size_t n)
```

位置 pos **に文字列** str **の** n **文字分を代入します。**

リターン値は *this **です。**

```
string& string::assign(const char* str, size_t n)
```

文字列 str **の** n **文字分を代入します。**

リターン値は *this **です。**

```
string& string::assign(size_t n, char c)
```

n **個の文字** c **を代入します。**

リターン値は *this **です。**

```
string& string::insert(size_t pos1, const string& str)
```

位置 pos1 **に** str **の文字列を挿入します。**

リターン値は *this **です。**

```
string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)
```

位置 pos1 **に** str **の文字列の位置** pos2 **から** n **文字分を挿入します。**

リターン値は *this **です。**

```
string& string::insert(size_t pos, const char* str, size_t n)
```

pos **の位置に文字列** str **を** n **文字分挿入します。**

リターン値は *this **です。**

```
string& string::insert(size_t pos, const char* str)
```

pos **の位置に文字列** str **を挿入します。**

リターン値は *this **です。**

```
string& string::insert(size_t pos, size_t n, char c)
```

位置 pos **に** n **個の文字** c **の文字列を挿入します。**

リターン値は *this **です。**

```
string::iterator string::insert(iterator p, char c = char())
```

p **が指す文字列の前に、文字** c **を挿入します。**

リターン値は、挿入された文字です。

```
void string::insert(iterator p, size_t n, char c)
```

p **が指す文字の前に、**n **個の文字** c **を挿入します。**

```
string& string::erase(size_t pos = 0, size_t n = npos)
```

位置 pos **から** n **個分取り除きます。**

リターン値は *this **です。**

```
iterator string::erase(iterator position)
```

position **により参照された文字を取り除きます。**

リターン値は次のとおりです。

削除要素の次の iterator **がある場合：削除要素の次の** iterator

削除要素の次の iterator **がない場合：**end()

```
iterator string::erase(iterator first, iterator last)
```

範囲 [first, last] **において文字を取り除きます。**

リターン値は次のとおりです。

last **の次の** iterator **がある場合：**last **の次の** iterator

last **の次の** iterator **がない場合：**end()

```
string& string::replace(size_t pos1, size_t n1, const string& str)
```

位置 pos1 **から** n1 **文字分の文字列を、**str **の文字列で置き換えます。**

リターン値は *this **です。**

```
string& string::replace(size_t pos1, size_t n1, const char* str)
```

位置 pos1 **から** n1 **文字分の文字列を、**str **の文字列で置き換えます。**

リターン値は *this **です。**

```
string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)
```

位置 pos1 **から** n1 **文字分の文字列を、**str **の位置** pos2 **から** n2 **文字分の文字列で置き換えます。**

リターン値は *this です。

```
string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)
```

位置 pos から n1 文字分の文字列を、n2 個の str の文字列で置き換えます。

リターン値は *this です。

```
string& string::replace(size_t pos, size_t n1, size_t n2, char c)
```

位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。

リターン値は *this です。

```
string& string::replace(iterator i1, iterator i2, const string& str)
```

位置 i1 から i2 までの文字列を str の文字列で置き換えます。

リターン値は *this です。

```
string& string::replace(iterator i1, iterator i2, const char* str)
```

位置 i1 から i2 までの文字列を str の文字列で置き換えます。

リターン値は *this です。

```
string& string::replace(iterator i1, iterator i2, const char* str, size_t n)
```

位置 i1 から i2 までの文字列を、str の n 文字分の文字列で置き換えます。

リターン値は *this です。

```
string& string::replace(iterator i1, iterator i2, size_t n, char c)
```

位置 i1 から i2 までの文字列を、n 個の文字 c で置き換えます。

リターン値は *this です。

```
size_t string::copy(char* str, size_t n, size_t pos = 0) const
```

位置 pos に文字列 str の n 文字分の文字列をコピーします。

リターン値は rlen です。

```
void string::swap(string& str)
```

str の文字列と交換します。

```
const char* string::c_str() const
```

文字列を格納している領域へのポインタを参照します。

リターン値は s_ptr です。

```
const char* string::data() const
```

文字列を格納している領域へのポインタを参照します。

リターン値は s_ptr です。

```
size_t string::find(const string& str, size_t pos = 0) const
```

位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find(const char* str, size_t pos = 0) const
```

位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find(const char* str, size_t pos, size_t n) const
```

位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find(char c, size_t pos = 0) const
```

位置 pos 以降で文字 c が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::rfind(const string& str, size_t pos = npos) const
```

位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::rfind(const char* str, size_t pos = npos) const
```

位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::rfind(const char* str, size_t pos, size_t n) const
```

位置 pos 以前で文字列 str の n 文字分と同じ文字列が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::rfind(char c, size_t pos = npos) const
```

位置 pos 以前で文字 c が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_of(const string& str, size_t pos = 0) const
```

位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_of(const char* str, size_t pos = 0) const
```

位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_of(const char* str, size_t pos, size_t n) const
```

位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_of(char c, size_t pos = 0) const
```

位置 pos 以降で文字 c が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_of(const string& str, size_t pos = npos) const
```

位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_of(const char* str, size_t pos = npos) const
```

位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_of(const char* str, size_t pos, size_t n) const
```

位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_of(char c, size_t pos = npos) const
```

位置 pos 以前で文字 c が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_not_of(const string& str, size_t pos = 0) const
```

位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_not_of(const char* str, size_t pos = 0) const
```

位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const
```

位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_first_not_of(char c, size_t pos = 0) const
```

位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_not_of(const string& str, size_t pos = npos) const
```

位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_not_of(const char* str, size_t pos = npos) const
```

位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

```
size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const
```

位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(char c, size_t pos = npos) const

位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。

リターン値は、文字列のオフセットです。

string string::substr(size_t pos = 0, size_t n = npos) const

格納された文字列に対し、範囲 [pos,n] の文字列を持つオブジェクトを生成します。

リターン値は、範囲 [pos,n] の文字列を持つオブジェクトです。

int string::compare(const string& str) const

文字列と str の文字列を比較します。

リターン値は次のとおりです。

文字列が同一の場合：0

文字列が異なる場合：this->s_len>str.s_len のとき 1

this->s_len < str.s_len のとき -1

int string::compare(size_t pos1, size_t n1, const string& str) const

位置 pos1 から n1 文字分の文字列と str を比較します。

リターン値は次のとおりです。

文字列が同一の場合：0

文字列が異なる場合：this->s_len>str.s_len のとき 1

this->s_len < str.s_len のとき -1

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。

リターン値は次のとおりです。

文字列が同一の場合：0

文字列が異なる場合：this->s_len>str.s_len のとき 1

this->s_len < str.s_len のとき -1

int string::compare(const char* str) const

str と比較します。

リターン値は次のとおりです。

文字列が同一の場合：0

文字列が異なる場合：this->s_len>str.s_len のとき 1

this->s_len < str.s_len のとき -1

int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const

位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。

リターン値は次のとおりです。

文字列が同一の場合：0

文字列が異なる場合：this->s_len>str.s_len のとき 1

this->s_len < str.s_len のとき -1

(b) string クラスマニピュレータ

種別	定義名	説明
関数	string operator+(const string& lhs, const string& rhs)	lhsの文字列(または文字)にrhsの文字列(または文字) を追加し、オブジェクトを生成してその文字列を格納します。
	string operator+(const char* lhs, const string& rhs)	
	string operator+(char lhs, const string& rhs)	
	string operator+(const string& lhs, const char* rhs)	
	string operator+(const string& lhs, char rhs)	
	bool operator==(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator==(const char* lhs, const string& rhs)	
	bool operator==(const string& lhs, const char* rhs)	
	bool operator!=(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator!=(const char* lhs, const string& rhs)	
	bool operator!=(const string& lhs, const char* rhs)	
	bool operator<(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<(const char* lhs, const string& rhs)	
	bool operator<(const string& lhs, const char* rhs)	
	bool operator>(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>(const char* lhs, const string& rhs)	
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	
	bool operator>=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>=(const char* lhs, const string& rhs)	
	bool operator>=(const string& lhs, const char* rhs)	
	void swap(string& lhs, string& rhs)	lhs の文字列と rhs の文字列を交換します。
	istream& operator>>(istream& is, string& str)	文字列を str に抽出します。
	ostream& operator<<(ostream& os, const string& str)	文字列を挿入します。

種別	定義名	説明
関数	istream& getline(istream& is, string& str, char delim)	is から文字列を抽出し，str に付加します。途中で文字 'delim' を検出したら、入力を終了します。
	istream& getline(istream& is, string& str)	is から文字列を抽出し，str に付加します。途中で改行文字を検出したら、入力を終了します。

```
string operator+(const string& lhs, const string& rhs)
```

lhs の文字列 (または文字) に rhs の文字列 (または文字) を追加し、オブジェクトを生成してその文字列を格納します。

リターン値は、結合した文字列を格納するオブジェクトです。

```
string operator+(const char* lhs, const string& rhs)
```

lhs の文字列 (または文字) に rhs の文字列 (または文字) を追加し、オブジェクトを生成してその文字列を格納します。

リターン値は、結合した文字列を格納するオブジェクトです。

```
string operator+(char lhs, const string& rhs)
```

lhs の文字列 (または文字) に rhs の文字列 (または文字) を追加し、オブジェクトを生成してその文字列を格納します。

リターン値は、結合した文字列を格納するオブジェクトです。

```
string operator+(const string& lhs, const char* rhs)
```

lhs の文字列 (または文字) に rhs の文字列 (または文字) を追加し、オブジェクトを生成してその文字列を格納します。

リターン値は、結合した文字列を格納するオブジェクトです。

```
string operator+(const string& lhs, char rhs)
```

lhs の文字列 (または文字) に rhs の文字列 (または文字) を追加し、オブジェクトを生成してその文字列を格納します。

リターン値は、結合した文字列を格納するオブジェクトです。

```
bool operator==(const string& lhs, const string& rhs)
```

lhs の文字列と rhs の文字列を比較します。

リターン値は次のとおりです。

文字列が同一の場合：true

文字列が異なる場合：false

```
bool operator==(const char* lhs, const string& rhs)
```

lhs の文字列と rhs の文字列を比較します。

リターン値は次のとおりです。

文字列が同一の場合： false

文字列が異なる場合： true

```
bool operator==(const string& lhs, const char* rhs)
```

lhs **の文字列と** rhs **の文字列を比較します。**

リターン値は次のとおりです。

文字列が同一の場合： false

文字列が異なる場合： true

```
bool operator<(const string& lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len < rhs.s_len **の場合：** true

lhs.s_len >= rhs.s_len **の場合：** false

```
bool operator<(const char* lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len < rhs.s_len **の場合：** true

lhs.s_len >= rhs.s_len **の場合：** false

```
bool operator<(const string& lhs, const char* rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len < rhs.s_len **の場合：** true

lhs.s_len >= rhs.s_len **の場合：** false

```
bool operator>(const string& lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len > rhs.s_len **の場合：** true

lhs.s_len <= rhs.s_len **の場合：** false

```
bool operator>(const char* lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len > rhs.s_len **の場合：** true

lhs.s_len <= rhs.s_len **の場合：** false

```
bool operator>(const string& lhs, const char* rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len > rhs.s_len **の場合：** true

lhs.s_len <= rhs.s_len **の場合：** false

```
bool operator<=(const string& lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len <= rhs.s_len **の場合：** true

lhs.s_len > rhs.s_len **の場合：** false

```
bool operator<=(const char* lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len <= rhs.s_len **の場合：** true

lhs.s_len > rhs.s_len **の場合：** false

```
bool operator<=(const string& lhs, const char* rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len <= rhs.s_len **の場合：** true

lhs.s_len > rhs.s_len **の場合：** false

```
bool operator>=(const string& lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len >= rhs.s_len **の場合：** true

lhs.s_len < rhs.s_len **の場合：** false

```
bool operator>=(const char* lhs, const string& rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len >= rhs.s_len **の場合：** true

lhs.s_len < rhs.s_len **の場合：** false

```
bool operator>=(const string& lhs, const char* rhs)
```

lhs **の文字列長と** rhs **の文字列長を比較します。**

リターン値は次のとおりです。

lhs.s_len >= rhs.s_len **の場合：** true

lhs.s_len < rhs.s_len **の場合：** false

```
void swap(string& lhs, string& rhs)
```

lhs **の文字列と** rhs **の文字列を交換します。**

```
istream& operator>>(istream& is, string& str)
```

文字列を str に抽出します。

リターン値は is です。

```
ostream& operator<<(ostream& os, const string& str)
```

文字列を挿入します。

リターン値は os です。

```
istream& getline(istream& is, string& str, char delim)
```

is から文字列を抽出し、str に付加します。

途中で文字 'delim' を検出したら、入力を終了します。

リターン値は is です。

```
istream& getline(istream& is, string& str)
```

is から文字列を抽出し、str に付加します。

途中で改行文字を検出したら、入力を終了します。

リターン値は is です。

6.6 未サポートライブラリ

C 言語仕様で定義しているライブラリのうち、本コンパイラでサポートしていないライブラリを表 6.16 に示します。

表 6―15 サポートしていないライブラリ

	ヘッダファイル	ライブラリ名
1	locale.h*1	setlocale、localeconv
2	signal.h*1	signal、raise
3	stdio.h	remove、rename、tmpfile、tmpnam、fgetpos、fsetpos
4	stdlib.h	abort、atexit、exit、_Exit、getenv、system、 mblen、mbtowc、wctomb、mbstowcs、wcstombs
5	string.h	strcoll、strxfrm
6	time.h*1	clock、difftime、mktime、time、asctime、ctime、gmtime、 localtime、strftime
7	wctype.h	iswalnum、iswalpha、iswblank、iswcntrl、iswdigit、 iswgraph、 iswlower、iswprintf、iswpunct、iswspace、iswupper、 iswxdigit、iswctype、wctype、towlower、towupper、 towctrans、wctrans
8	wchar.h	wcsftime、wcscoll、wcsxfrm、wctob、mbrtowc、wcrtomb、 mbsrtowcs, wcsrtombs

注 ヘッダファイルをサポートしません。

第 7 章　スタートアップ

この章では，スタートアップルーチンについて説明します。

7.1　概要

C 言語によるプログラムを実行させるには，システムへ組み込むための ROM 化処理，ユーザ・プログラム（main 関数）の起動などを行うプログラムが必要となります。このプログラムのことをスタートアップ・ルーチンと呼びます。

ユーザが作成したプログラムを実行させるには，そのプログラムに応じたスタートアップ・ルーチンを作成しなければなりません。CubeSuite+ では，プログラム実行前に必要な処理を含むスタートアップ・ルーチンのオブジェクト・モジュール・ファイルと，ユーザがシステムに合わせて変更できるようにスタートアップ・ルーチンのソース・ファイルを提供しています。

7.2　ファイルの構成

CubeSuite+ が提供しているスタートアップ・ルーチンは，以下のとおりです。

表 7―1　統合開発環境で生成されるプログラムの一覧

	ファイル名	内容
(a)	resetprg.c	初期設定ルーチン (リセットベクタ関数)
(b)	intprg.c	ベクタ関数の定義
(c)	vecttbl.c	固定ベクタテーブル
(d)	dbsct.c	セクションの初期化処理 (テーブル)
(e)	lowsrc.c	低水準インタフェースルーチン (C 言語部分)
(f)	lowlvl.src	低水準インタフェースルーチン (アセンブリ言語部分)
(g)	sbrk.c	低水準インタフェースルーチン (sbrk 関数)
(h)	typedefine.h	型定義ヘッダ
(i)	vect.h	ベクタ関数のヘッダ
(j)	stacksct.h	スタックサイズの設定
(k)	lowsrc.h	低水準インタフェースルーチン (C 言語ヘッダ)
(l)	sbrk.h	低水準インタフェースルーチン (sbrk 関数のヘッダ)

7.3　スタートアッププログラム

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

本節の内容は、標準となるスタートアップについて説明しています。PIC/PID 機能におけるアプリケーションに使用するスタートアップは特別な対応が必要ですので「7. 5. 7　アプリケーションのスタートアップ」も参照ください。

必要な手続きの概要は以下の通りです。

- 固定ベクタテーブルの設定

  パワーオンリセットで初期設定ルーチン (PowerON_Reset_PC) が起動されるように、固定ベクタテーブルを設定します。固定ベクタテーブルには、リセットベクタの他、特権命令例外、アクセス例外、未定義命令例外、浮動小数点例外、ノンマスカブル割り込みの各処理ルーチンを登録することができます。

- 初期設定

  main 関数に到達するまでに必要な手続きを行います。レジスタやセクションの初期化、各種初期設定ルーチンの呼び出しを行います。

- 低水準インタフェースルーチンの作成

  標準入出力 (stdio.h、ios、streambuf、istream、ostream)、メモリ管理ライブラリ (stdlib.h、new) を使用する場合に必要なライブラリ関数とユーザシステムとの間のインタフェースルーチンです。

- 終了処理ルーチン (exit、atexit、abort)*1 の作成

  プログラムの終了処理を行います。

注 1.　プログラムの終了処理を行う C ライブラリ関数 exit、atexit、abort 関数を使用する場合は、ユーザシステムにあわせてこれらの関数を作成する必要があります。
C++ プログラムを使用する場合、または C ライブラリ関数 assert マクロを使用する場合、abort 関数は必ず作成する必要があります。

7. 3. 1　固定ベクタテーブルの設定

パワーオンリセットで、初期設定ルーチン PowerON_Reset_PC が呼び出されるようにするためには、固定ベクタテーブルのリセットベクタに PowerON_Reset_PC のアドレスを設定します。以下にそのコーディング例を示します。

固定ベクタテーブルには、リセットベクタの他、特権命令例外、アクセス例外、未定義命令例外、浮動小数点例外、ノンマスカブル割り込みの各処理ルーチンを登録することができます。

なお、固定ベクタテーブルの詳細については、ハードウェアマニュアル等を参照してください。

例

```
extern void PoweON_Reset_PC(void);

#pragma section VECTTBL  /* #pragma section 宣言により RESET_Vectors を     */
                         /* CVECTTBL セクションに出力します。                 */
                         /* リンク時に start オプションで CVECTTBL セクションを */
                         /* リセットベクタに割り付けるよう指定します。          */
void (*const RESET_Vectors[])(void)={
    PowerON_Reset_PC,
};
```

7. 3. 2　初期設定

初期設定ルーチン PowerON_Reset_PC は、main 関数を実行する前、および実行した後に必要な手続きを記述する関数です。初期設定ルーチンの中で必要となる処理を、順番に記述します。

(1) 初期設定処理向けの PSW レジスタ初期化

初期設定処理を行うための、PSW レジスタ初期化を実施します。たとえば、初期設定処理中、割り込みを受け付けない設定にするために、PSW に対して割り込み禁止の設定をします。

リセット時の PSW は全 bit ゼロに初期化され、割り込み許可ビット (I ビット) も割り込み禁止状態 (ゼロ) として初期化されています。

(2) スタックポインタの初期化

スタックポインタ (USP レジスタおよび ISP レジスタ) を初期化します。PowerON_Reset_PC 関数に対して "#pragma entry" 宣言することにより、コンパイラが自動的に ISP/USP 初期化コードを関数先頭に生成します。

PowerON_Reset_PC 関数は、#pragma entry 宣言されているため、この手続きを記述する必要はありません。

(3) ベースレジスタに使用する汎用レジスタの初期化

コンパイラで base オプションを使用している場合、プログラム全体でベースアドレスとして使用する汎用レジスタを初期化する必要があります。PowerON_Reset_PC 関数に対して "#pragma entry" 宣言することにより、コンパイラが自動的に各レジスタへの初期化コードを関数先頭に生成します。

PowerON_Reset_PC 関数は、#pragma entry 宣言されているため、この手続きを記述する必要はありません。

(4) 各種制御レジスタの初期化

可変ベクタテーブルの配置アドレスを、INTB に書き込みます。その他、必要に応じて、FINTV、FPSW、BPC、BPSW の初期化を行います。これらの初期化は、コンパイラの組み込み関数を使って行うことができます。

ただし、PSW だけは、割り込みマスク設定を維持するため、初期化はまだ行いません。

(5) セクションの初期化処理

RAM 領域セクションの初期化用ルーチン (_INITSCT) を呼び出します。未初期化データセクションはゼロ初期化されます。初期化データセクションは、ROM 領域の初期値を RAM 領域へコピーします。_INITSCT は、標準ライブラリとして提供されます。

初期化対象のセクションは、ユーザがセクション初期化用テーブル (DTBL,BTBL) へ記述する必要があります。_INITSCT 関数が使用するセクションの先頭アドレスおよび最終アドレスを、セクションアドレス演算子を用いて設定します。

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

以下にコーディング例を示します。

例

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

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

(6) ライブラリの初期化処理

C/C++ 言語ライブラリ関数使用時に、必要な初期化を実施するルーチン (_INITLIB) を呼び出します。

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

- 作成した低水準インタフェースルーチンの中で初期設定が必要な場合、低水準インタフェースルーチンの仕様に合わせた初期設定 (_INIT_LOWLEVEL) が必要です。
- rand 関数、strtok 関数を使用する場合、標準入出力以外の初期設定 (_INIT_OTHERLIB) が必要です。

ライブラリの初期設定を行うプログラム例を以下に示します。

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;    // ヒープ領域確保サイズの最小単位を指定します
                                  // ( 省略時：1024)
extern char *_s1ptr;

#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
    _INIT_LOWLEVEL();             // 低水準インタフェースルーチンの初期設定をします
    _INIT_IOLIB();                // 入出力ライブラリの初期設定をします
    _INIT_OTHERLIB();             // rand 関数、strtok 関数の初期設定をします
}

void _INIT_LOWLEVEL (void)
{                                 // 低水準ライブラリで必要な初期設定をしてください
}

void _INIT_OTHERLIB(void)
{
    srand(1);                     // rand 関数を使用する場合の初期設定です
    _s1ptr=NULL;                  // strtok 関数を使用する場合の初期設定です
}
#ifdef __cplusplus
}
#endif
```

注 1. 標準入出力ファイルのファイル名を指定します。この名前は、低水準インタフェースルーチン「open」で使用します。

2. コンソール等対話的な装置の場合、バッファリングを行わないためのフラグを立てます。

(7) グローバルクラスオブジェクトの初期化

C++ 言語のプログラム開発の場合、グローバルに宣言されたクラスオブジェクトのコンストラクタを呼び出すためのルーチン (_CALL_INIT) を呼び出します。_CALL_INIT は、標準ライブラリとして提供されます。

(8) main 関数実行向けの PSW 初期化

PSW レジスタを初期化します。割り込みマスクの設定も、ここで解除します。

(9) PSW の PM ビットの変更

リセット以降は特権モード (PSW の PM ビットがゼロ) で動作していますが、ユーザモードに切り替えたい場合は、組み込み関数の chg_pmusr を実行します。

chg_pmusr 関数にはいくつか注意事項がありますので、使用する場合は、3. 2. 6　組み込み関数の chg_pmusr の項目を参照ください。

(10) ユーザプログラムの実行

main 関数を実行します。

(11) グローバルクラスオブジェクトの後処理

C++ 言語のプログラム開発の場合、グローバルに宣言されたクラスオブジェクトのデストラクタを呼び出すためのルーチン (_CALL_END) を呼び出します。_CALL_END は、標準ライブラリとして提供されます。

7. 3. 3　初期設定ルーチンの記述例

「初期設定」で説明した、PowerON_Reset_PC 関数のコーディング例を示します。

なお、統合開発環境で生成される実際の初期設定ルーチンは、「7. 4　コーディング例」を参照ください。

```
#include         <machine.h>
#include         <_h_c_lib.h>
#include         "typedefine.h"
#include         "stacksct.h"

#ifdef __cplusplus
extern "C" {
#endif
void PowerON_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus                          // Use SIM I/O
extern "C" {
#endif
extern void _INITLIB(void);
#ifdef __cplusplus
}
#endif

#define PSW_init  0x00010000
#define FPSW_init 0x00000100

#pragma section ResetPRG
#pragma entry PowerON_Reset_PC
void PowerON_Reset_PC(void)
{
set_intb(__sectop("C$VECT"));
set_fpsw(FPSW_init);

_INITSCT();
_INITLIB();
nop();
set_psw(PSW_init);
main();
brk();
}
```

7. 3. 4 低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリを C/C++ プログラムで使用する場合は、低水準インタフェースルーチンを作成しなければなりません。表 7.2 に C ライブラリ関数で使用している低水準インタフェースルーチンの一覧を示します。

表 7―2 低水準インタフェースルーチンの一覧

No.	名称	機能
1	open	ファイルのオープン
2	close	ファイルのクローズ
3	read	ファイルからの読み込み
4	write	ファイルへの書き出し
5	lseek	ファイルの読み込み／書き出しの位置の設定
6	sbrk	メモリ領域の確保
7	error_addr*	errno アドレスの取得
8	wait_sem*	セマフォの確保
9	signal_sem*	セマフォの解放

注 * リエントラントライブラリを使用する場合に必要です。

低水準インタフェースルーチンで必要な初期化は、プログラム起動時に行う必要があります。これは、ライブラリ初期設定処理 _INITLIB の中の「_INIT_LOWLEVEL」という関数の中で行ってください。

以下、低水準入出力の基本的な考え方を説明したあと、各インタフェースルーチンの仕様を説明します。

注 関数名 open、close、read、write、lseek、sbrk、error_addr、wait_sem、signal_sem は低水準インタフェースルーチンの予約済み識別子です。ユーザプログラム中では使用しないでください。

(1) 入出力の考え方

標準入出力ライブラリでは、ファイルを FILE 型のデータによって管理しますが、低水準インタフェースルーチンでは、実際のファイルと 1 対 1 に対応する正の整数を与え、これによって管理します。この整数をファイル番号といいます。

open ルーチンでは、与えられたファイル名に対してファイル番号を与えます。open ルーチンでは、この番号によってファイルの入出力ができるように、以下の情報を設定する必要があります。

- ファイルのデバイスの種類 (コンソール、プリンタ、ディスクファイル等)。
  コンソールやプリンタ等の特殊なデバイスに対しては、特別なファイル名をシステムで決めておいて open ルーチンで判定する必要があります。
- ファイルのバッファリングはバッファの位置、サイズ等の情報。
- ディスクファイルは、ファイルの先頭から次に読み込み / 書き出しを行う位置までのバイトオフセット。

open ルーチンで設定した情報に基づいて、以後、入出力 (read、write ルーチン)、読み込み / 書き出し位置の設定 (lseek ルーチン) を行います。

close ルーチンでは、出力ファイルのバッファリングを行っている場合はバッファの内容を実際のファイルに書き出し、open ルーチンで設定したデータの領域が再使用できるようにしてください。

(2) 低水準インタフェースルーチンの仕様

本項では低水準インタフェースルーチンを作成するための仕様を説明します。以下、各ルーチンごとに、ルーチンを呼び出す際のインタフェースとその動作および実現上の注意事項を示します。

各ルーチンのインタフェースは以下の形式で示します。なお、低水準インタフェースルーチンは必ず関数原型にしてください。また C++ プログラム内で宣言する場合は「extern "C"」を付加してください。

凡例：

（ルーチン名）

[説明]

- （ルーチンの機能概要を示します）

[リターン値]

正常：	（正常に終了した場合のリターン値を示します）
異常：	（エラーが生じた場合のリターン値を示します）

[引数]

（名前）	（意味）
（インタフェースに示す引数名です）	（引数として渡される値を示します）

long open(const char *name, long mode, long flg)

[説明]

- 第 1 引数のファイル名に対応するファイルを操作するための準備をします。
- open ルーチンでは、後で読み込み / 書き出しを行うために、ファイルの種類 (コンソール、プリンタ、ディスクファイル等) を決定しなければなりません。ファイルの種類は、以後 open ルーチンで返したファイル番号を用いて読み込み / 書き出しを行うたびに参照する必要があります。
- 第 2 引数の mode は、ファイルをオープンする時の処理の指定です。このデータの各ビットの意味について以下に示します。

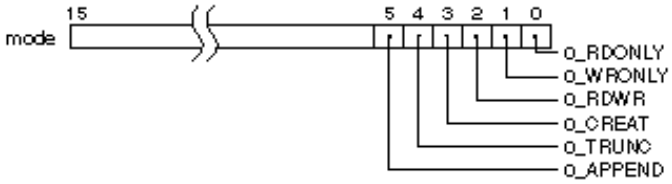


表 7―3　open ルーチン mode ビット説明

mode ビット	説明
O_RDONLY (0 ビット)	ビットが 1 のとき、ファイルを読み込み専用にオープン
O_WRONLY (1 ビット)	ビットが 1 のとき、ファイルを書き出し専用にオープン
O_RDWR (2 ビット)	ビットが 1 のとき、ファイルを読み込み、書き出し両用にオープン
O_CREAT (3 ビット)	ビットが 1 のとき、ファイル名で示すファイルが存在しない場合にファイルを新規に作成
O_TRUNC (4 ビット)	ビットが 1 のとき、ファイル名で示すファイルが存在する場合に ファイルの内容を捨て、ファイルのサイズを 0 に更新
O_APPEND (5 ビット)	次に読み込み / 書き出しを行うファイル内の位置を設定 ビットが 0 のとき：ファイルの先頭に設定 ビットが 1 のとき：ファイルの最後に設定

mode で示したファイルの処理の指定と実際のファイルの性質が矛盾する場合はエラーにしてください。正常にファイルがオープンできた場合は、以後の read、write、lseek、close ルーチンで使用されるファイル番号 (正の整数) を返してください。ファイル番号と実際のファイルの対応は低水準インタフェースルーチンで管理する必要があります。オープンに失敗した場合は -1 を返してください。

[リターン値]

正常：	正常オープンしたファイルのファイル番号
異常：	-1

[引数]

name	ファイルのファイル名を指す文字
mode	ファイルをオープンするときの処理の指定
flg	ファイルをオープンするときの処理の指定 (常に 0777)

long close(long fileno)

[説明]

- open ルーチンで得られたファイル番号が引数として渡されます。
- open ルーチンで設定したファイル管理情報の領域を、再び使用できるように解放してください。また、低水準インタフェースルーチン内で出力ファイルのバッファリングを行っている場合は、バッファの内容を実際のファイルに書き出してください。
- ファイルを正常にクローズできた場合は 0、失敗した場合は -1 を返してください。

[リターン値]

正常：	0
異常：	-1

[引数]

fileno	クローズするファイル番号

long read(long fileno, unsigned char *buf, long count)

[説明]

- 第 1 引数 (fileno) で示すファイルから、第 2 引数 (buf) の指す領域へデータを読み込みます。読み込むデータのバイト数は第 3 引数 (count) で示します。
- ファイルが終了した場合、count で示されたバイト数以下のバイト数しか読み込むことができません。
- ファイルの読み込み / 書き出しの位置は、読み込んだバイト数だけ先に進みます。
- 正常に読み込みができた場合は、実際に読み込んだバイト数を返してください。読み込みに失敗した場合は -1 を返してください。

[リターン値]

正常：	実際に読み込んだバイト数
異常：	-1

[引数]

fileno	読み込みの対象となるファイル番号
buf	読み込んだデータを格納する領域
count	読み込むバイト数

long write(long fileno, const unsigned char *buf, long count)

[説明]

- 第 2 引数 (buf) の指す領域から、第 1 引数 (fileno) の示すファイルにデータを書き出します。書き込むデータのバイト数は第 3 引数 (count) で示します。
- ファイルを書き出そうとしているデバイス (ディスク等) が満杯の時は、count で示されたバイト数以下のバイト数しか書き出すことができません。実際に書き出すことのできたバイト数が何度か連続して 0 バイトの場合、ディスクが満杯であると判断してエラー (-1) を返すように実現することをお勧めします。
- ファイルの読み込み / 書き出しの位置は、書き出したバイト数だけ先に進みます。
- 正常に書き出しができた場合は、実際に書き出したバイト数を返してください。書き出しに失敗した場合は -1 を返してください。

[リターン値]

正常：	実際に書き出されたバイト数
異常：	-1

[引数]

fileno	書き出しの対象となるファイル番号
buf	書き出すデータ領域
count	書き出すバイト数

long lseek(long fileno, long offset, long base)

[説明]

- ファイルの読み込み / 書き出しを行うファイル内の位置を、バイト単位で設定します。
- 新しいファイル内の位置は、第 3 引数 (base) によって、以下の方法で計算し設定してください。
  (1) base が 0 のときファイルの先頭から offset バイトの位置に設定します。
  (2) base が 1 のとき現在の位置に offset バイトを加えた位置に設定します。
  (3) base が 2 のときファイルのサイズに offset バイトを加えた位置に設定します。
- ファイルがコンソールやプリンタ等の対話的なデバイスの場合や、新しいオフセットの値が負になったり、(1)(2) のときファイルのサイズを超える場合はエラーにします。
- 正しくファイル位置を設定できた場合は、新しい読み込み / 書き出し位置のファイルの先頭からのオフセットを、そうでない場合は -1 を返してください。

[リターン値]

正常：	新しいファイルの読み込み / 書き出し位置のファイルの先頭からのオフセット (バイト単位)
異常：	-1

[引数]

fileno	対象となるファイル番号
offset	読み込み / 書き出しの位置を示すオフセット (バイト単位)
base	オフセットの起点

char *sbrk(size_t size)

[説明]

- メモリ領域を割り付けるサイズが引数として渡されます。
- 連続して sbrk ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。割り付けるメモリ領域が不足した場合はエラーにしてください。
- 正常に割り付けができた場合は、割り付けた領域の先頭のアドレスを、割り付けに失敗した場合は、「(char *)-1」を返してください。

[リターン値]

正常：	割り付けた領域の先頭アドレス
異常：	(char *)-1

[引数]

size	割り付けるデータのサイズ

long *errno_addr(void)

[説明]

- 現在のタスクが持つエラー番号のアドレスを返却します。
- 標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

[リターン値]

正常：	現在のタスクが持つエラー番号のアドレス

long wait_sem(long semnum)

[説明]

- semnum で示されたセマフォを確保します。
- 確保できた場合は 1、確保できなかった場合は 0 を返してください。
- 標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

[リターン値]

正常：	1
異常：	0

[引数]

semnum	セマフォ ID

long signal_sem(long semnum)

[説明]

- semnum で示されたセマフォを解放します。
- 解放できた場合は 1、解放できなかった場合は 0 を返してください。
- 標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

[リターン値]

正常：	1
異常：	0

[引数]

semnum	セマフォ ID

(3) 低水準インタフェースルーチンコーディング例

```
/************************************************************************/
/*                                    lowsrc.c:                         */
/*----------------------------------------------------------------------*/
/*      RX ファミリ　シミュレータ・デバッガ　インタフェースルーチン                */
/*            - 標準入出力 (stdin,stdout,stderr) だけをサポートしています -      */
/************************************************************************/
#include <string.h>

/*  ファイル番号  */
#define STDIN  0                        /*  標準入力 ( コンソール )*/
#define STDOUT 1                        /*  標準出力 ( コンソール )*/
#define STDERR 2                        /*  標準エラー出力 ( コンソール )*/

#define FLMIN 0                         /*  最小のファイル番号 */
#define FLMAX 3                         /*  ファイル数の最大値 */

/*  ファイルのフラグ  */
#define O_RDONLY 0x0001                 /*  読み込み専用 */
#define O_WRONLY 0x0002                 /*  書き出し専用 */
#define O_RDWR 0x0004                   /*  読み書き両用 */

/*  特殊文字コード  */
#define CR 0x0d                         /*  復帰 */
#define LF 0x0a                         /*  改行 */

/*  sbrk で管理する領域サイズ  */
#define HEAPSIZE 1024

/************************************************************************/
/*                              参照関数の宣言：                          */
/*     シミュレータ・デバッガでコンソールへの文字入出力を行うアセンブリプログラムの参照     */
/************************************************************************/
extern void charput(char);              /*  一文字入力処理 */
extern char charget(void);              /*  一文字出力処理 */

/************************************************************************/
/*                              静的変数の定義：                          */
/*                    低水準インタフェースルーチンで使用する静的変数の定義              */
/************************************************************************/
char flmod[FLMAX];                      /*  オープンしたファイルのモード設定場所  */

union HEAP_TYPE{
    long dummy;                         /*  4 バイトアライメントにするためのダミー */
    char heap[HEAPSIZE];                /*  sbrk で管理する領域の宣言 */
};

static union HEAP_TYPE heap_area;

static char *brk=(char*)&heap_area;     /*  sbrk で割り付けた領域の最終アドレス  */
```

```
/************************************************************************/
/*                              open: ファイルのオープン                */
/*                          リターン値：ファイル番号（成功）            */
/*                                      -1          （失敗）            */
/************************************************************************/
long open(const char *name,                         /*  ファイル名 */
          long mode,                                /*  ファイルのモード */
          long flg)                                 /*  処理の指定（未使用）*/
{
    /*  ファイル名に従ってモードをチェックし、ファイル番号を返す  */

    if (strcmp(name,"stdin")==0) {                  /*  標準入力ファイル  */
        if ((mode&O_RDONLY)==0) {
            return (-1);
        }
        flmod[STDIN]=mode;
        return (STDIN);
    }

    else if (strcmp(name,"stdout")==0) {            /*  標準出力ファイル  */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDOUT]=mode;
        return (STDOUT);
    }

    else if (strcmp(name,"stderr")==0){             /* 標準エラー出力ファイル  */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDERR]=mode;
        return (STDERR);
    }

    else {
        return (-1);                                /* エラー  */
    }
}

/************************************************************************/
/*                              close: ファイルのクローズ               */
/*                          リターン値：0           （成功）            */
/*                                      -1          （失敗）            */
/************************************************************************/
long close(long fileno)                             /*  ファイル番号  */
{
    if (fileno<FLMIN || FLMAX<fileno) {             /*  ファイル番号の範囲チェック  */
        return -1;
    }

    flmod[fileno]=0;                                /*  ファイルのモードリセット  */

    return 0;
}
```

```
/************************************************************************/
/*                              read:データの読み込み                   */
/*                        リターン値：実際に読み込んだ文字数（成功）    */
/*                                    -1                （失敗）        */
/************************************************************************/
long read(long fileno,                          /*  ファイル番号          */
          unsigned char *buf,                   /*  転送先バッファアドレス  */
          long  count)                          /*  読み込み文字数          */
{
    unsigned long i;

    /*  ファイル名に従ってモードをチェックし、一文字づつ入力してバッファに格納  */

    if (flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR) {
        for (i=count;i>0;i--) {
            *buf=charget();
            if (*buf==CR) {                     /*  改行文字の置き換え  */
                *buf=LF;
            }
        buf++;
        }
        return count;
    }

    else {
        return -1;
    }
}

/************************************************************************/
/*                              write:データの書き出し                  */
/*                        リターン値：実際に書き出した文字数（成功）    */
/*                                    -1                （失敗）        */
/************************************************************************/
long write(long fileno,                         /*  ファイル番号  */
           const unsigned char *buf,            /*  転送元バッファアドレス  */
           long count)                          /*  書き出し文字数  */
{
    unsigned long i;
    unsigned char c;

    /*  ファイル名に従ってモードをチェックし、一文字づつ出力  */

    if (flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR) {
        for (i=count; i>0; i--) {
            c=*buf++;
            charput(c);
        }
        return count;
    }

    else {
        return -1;
    }
}
```

```
/************************************************************************/
/*                      lseek: ファイルの読み込み／書き出し位置の設定                  */
/*          リターン値：読み込み／書き出し位置のファイル先頭からのオフセット　（成功）    */
/*                    -1                                          （失敗）    */
/*                  （コンソール入出力では、lseek はサポートしていません）          */
/************************************************************************/
long lseek(long fileno,                     /*  ファイル番号            */
           long offset,                     /*  読み込み／書き出し位置    */
           long  base)                      /*  オフセットの起点          */
{
    return -1;
}

/************************************************************************/
/*                          sbrk: メモリ領域の割り付け                      */
/*                  リターン値：割り付けた領域の先頭アドレス　（成功）              */
/*                             -1                          （失敗）    */
/************************************************************************/
char *sbrk(size_t size)                     /*  割り付ける領域のサイズ   */
{
    char *p;

    /*  空き領域のチェック   */

    if (brk+size>heap_area.heap+HEAPSIZE) {
        return (char *)-1;
    }

    p=brk;                                  /* 領域の割り付け        */
    brk+=size;                              /* 最終アドレスの更新     */
    return p;
}
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                          lowlvl.src                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  RX Family Simulator/Debugger Interface Routine                       ;
;             - Inputs and outputs one character -                      ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .GLB    _charput
        .GLB    _charget

SIM_IO  .EQU 0h

        .SECTION   P,CODE
;------------------------------------------------------------------------
;  _charput:
;------------------------------------------------------------------------
_charput:
        MOV.L      #IO_BUF,R2
        MOV.B      R1,[R2]
        MOV.L      #1220000h,R1
        MOV.L      #PARM,R3
        MOV.L      R2,[R3]
        MOV.L      R3,R2
        MOV.L      #SIM_IO,R3
        JSR        R3
        RTS

;------------------------------------------------------------------------
;  _charget:
;------------------------------------------------------------------------
_charget:
        MOV.L       #1210000h,R1
        MOV.L       #IO_BUF,R2
        MOV.L       #PARM,R3
        MOV.L       R2,[R3]
        MOV.L       R3,R2
        MOV.L       #SIM_IO,R3
        JSR         R3
        MOV.L       #IO_BUF,R2
        MOVU.B      [R2],R1
        RTS

;------------------------------------------------------------------------
;  I/O Buffer
;------------------------------------------------------------------------
        .SECTION  B,DATA,ALIGN=4
PARM:   .BLKL     1
        .SECTION  B_1,DATA
IO_BUF: .BLKB     1
        .END
```

(4) リエントラントライブラリ用低水準インタフェースルーチン例

リエントラントライブラリ用低水準インタフェース例を示します。標準ライブラリ構築ツールで reent オプションを指定して作成したライブラリを使用する場合に必要になります。

wait_sem 関数、signal_sem 関数で NG が返った場合、errno に以下を設定し、ライブラリ関数からリターンします。

wait_sem	EMALRESM	malloc 用セマフォ資源確保に失敗しました
	ETOKRESM	strtok 用セマフォ資源確保に失敗しました
	EIOBRESM	_iob 用セマフォ資源確保に失敗しました
signal_sem	EMALFRSM	malloc 用セマフォ資源解放に失敗しました
	ETOKFRSM	strtok 用セマフォ資源解放に失敗しました
	EIOBRESM	_iob 用セマフォ資源解放に失敗しました

割り込みに関しては、セマフォ確保後により優先度の高い割り込みが発生し、セマフォ確保を行うとデッドロックが発生するため、リソースを共有するような処理が割り込みでネストしないようにしてください。

```
#define MALLOC_SEM        1      /* malloc 用セマフォ No.          */
#define STRTOK_SEM        2      /* strtok 用セマフォ No.          */
#define FILE_TBL_SEM      3      /* _iob 用セマフォ No.            */
#define SEMSIZE           4
#define TRUE              1
#define FALSE             0
#define OK                1
#define NG                0

extern long *errno_addr(void);
extern long wait_sem(long);
extern long signal_sem(long);

long sem_errno;
int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];

/*****************************************************************************/
/*                          errno_addr:errno アドレスの取得                  */
/*                             リターン値:errno アドレス                     */
/*****************************************************************************/
long *errno_addr(void)
{
    /* 現在のタスクの errno アドレスを返してください */
    return (&sem_errno);
}


/*****************************************************************************/
/*                          wait_sem: 指定されたセマフォの確保               */
/*                             リターン値:OK(=1) ( 成功 )                    */
/*                                        NG(=0) ( 失敗 )                    */
/*****************************************************************************/
long wait_sem(long semnum)        /* セマフォ ID     */
{
    if((0 < semnum) && (semnum < SEMSIZE)) {
          if(semaphore[semnum] == FALSE) {
             semaphore[semnum] = TRUE;
             return(OK);
          }
    }
    return(NG);
}
```

```
/************************************************************************/
/*                        signal_sem: 指定されたセマフォの解放          */
/*                              リターン値：OK(=1) ( 成功 )             */
/*                                          NG(=0) ( 失敗 )             */
/************************************************************************/
long signal_sem(long semnum)    /* セマフォ ID    */
{
    if(!force_fail_signal_sem) {
         if((0 <= semnum) && (semnum < SEMSIZE)) {
            if( semaphore[semnum] == TRUE ) {
            semaphore[semnum] = FALSE;
            return(OK);
            }
         }
    }
    return(NG);
}
```

7. 3. 5 終了処理ルーチン

(1) 終了処理の登録と実行 (atexit) ルーチンの作成例

終了処理の登録を行うライブラリ atexit 関数の作成法を示します。

atexit 関数では、引数として渡された関数のアドレスを、終了処理のテーブルに登録します。登録された関数の個数が限界値 (ここでは、登録できる個数を 32 個とします) を超えた場合、あるいは同じ関数が二度以上登録された場合はリターン値として 0 以外 (ここでは 1) を返します。そうでなければ 0 を返します。

以下にプログラム例を示します。

例

```
#include <stdlib.h>

long _atexit_count=0 ;

void (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
long atexit(void (*f)(void))
{
    int i;

    for(i=0; i<_atexit_count ; i++)        // 既に登録されていないかチェックします
        if(_atexit_buf[i]==f)
            return 1;
    if (_atexit_count==32)                 // 登録数の限界値をチェックします
        return 1;
    else {
        _atexit_buf[_atexit_count++]=f;    // 関数のアドレスを登録します
        return 0;
    }
}
```

(2) プログラムの終了 (exit) ルーチンの作成例

プログラムの終了処理を行うライブラリ exit 関数の作成法を示します。プログラムの終了処理は、ユーザシステムによって異なりますので、以下のプログラム例を参考に、ユーザシステムの仕様に従った終了処理を作成してください。

exit 関数は、引数として渡されたプログラムの終了コードに従ってプログラムの終了処理を行い、プログラム起動時の環境に戻ります。ここでは、終了コードを外部変数に設定して、main 関数を呼び出す直前に setjmp 関数で退避した環境に戻ることによって実現します。プログラム実行前の環境に戻るためには、次の関

数「callmain」を作成し、初期設定関数「PowerON_Reset_PC」から関数「main」を呼び出す代わりに、関数「callmain」を呼び出してください。

以下にプログラム例を示します。

```
#include <setjmp.h>
#include <stddef.h>

extern long _atexit_count ;
extern void (*_atexit_buf[32])(void) ;
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
int main(void);
extern jmp_buf _init_env ;
int _exit_code ;

#ifdef __cplusplus
extern "C"
#endif
void exit(int code)
{
    int i;
    _exit_code=code ;                                   // _exit_code にリターンコードを設定します
    for(i=_atexit_count-1; i>=0; i--)                   // atexit 関数で登録した関数を順次実行します
        (*_atexit_buf[i])();
    _CLOSEALL();                                        // オープンした関数を全てクローズします
    longjmp(_init_env, 1) ;                             // setjmp で退避した環境にリターンします
}
#ifdef __cplusplus
extern "C"
#endif
void callmain(void)
{
     // setjmp を用いて現在の環境を退避し、main 関数を呼び出します
     if(!setjmp(_init_env))
         _exit_code=main();                             // exit 関数からのリターン時には処理を終了します
}
```

(3) 異常終了 (abort) ルーチンの作成例

異常終了の場合は、ご使用になっているユーザシステムの仕様に従って、プログラムを異常終了させる処理を行ってください。

C++ プログラムを使用する場合、以下のときにも abort 関数を呼び出します。

- 例外処理が正しく動作しなかった場合
- 純粋仮想関数自体をコールした場合
- dynamic_cast に失敗した場合
- typeid に失敗した場合
- クラス配列の delete 時に情報が取れなかった場合
- クラスオブジェクトのデストラクタコール情報登録時に矛盾が発生した場合

以下、標準出力装置にメッセージを出力したあと、ファイルをクローズしてから無限ループしてリセットを待つプログラム例を示します。

例

```
#include <stdio.h>
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef __cplusplus
extern "C"
#endif
void abort(void)
{
        printf("program is abort !!\n");     // メッセージを出力します
        _CLOSEALL();                         // ファイルをクローズします
        while(1)    ;                        // 無限ループします
}
```

7.4 コーディング例

統合開発環境で生成される実際のスタートアッププログラムの例として、CPU 種別として RX610 を選択したシミュレータ用の場合の例を示します。

(1) ソースファイル

スタートアッププログラムは、表 7.4 に示すファイルから構成されます。

表 7―4 統合開発環境で生成されるプログラムの一覧

	ファイル名	内容
(a)	resetprg.c	初期設定ルーチン (リセットベクタ関数)
(b)	intprg.c	ベクタ関数の定義
(c)	vecttbl.c	固定ベクタテーブル
(d)	dbsct.c	セクションの初期化処理 (テーブル)
(e)	lowsrc.c	低水準インタフェースルーチン (C 言語部分)
(f)	lowlvl.src	低水準インタフェースルーチン (アセンブリ言語部分)
(g)	sbrk.c	低水準インタフェースルーチン (sbrk 関数)
(h)	typedefine.h	型定義ヘッダ
(i)	vect.h	ベクタ関数のヘッダ
(j)	stacksct.h	スタックサイズの設定
(k)	lowsrc.h	低水準インタフェースルーチン (C 言語ヘッダ)
(l)	sbrk.h	低水準インタフェースルーチン (sbrk 関数のヘッダ)

ファイル内容を、以下、(a) ～ (l) に示します。

（a）resetprg.c -- 初期設定ルーチン (リセットベクタ関数)

```
#include  <machine.h>
#include  <_h_c_lib.h>
//#include   <stddef.h>           // Remove the comment when you use errno
//#include   <stdlib.h>           // Remove the comment when you use rand()
#include  "typedefine.h"         // Define Types
#include  "stacksct.h"           // Stack Sizes (Interrupt and User)

#ifdef __cplusplus               // For Use Reset vector
extern "C" {
#endif
void PowerON_Reset_PC(void);
void main(void);
#ifdef __cplusplus
}
#endif

#ifdef __cplusplus               // For Use SIM I/O
extern "C" {
#endif
extern void _INIT_IOLIB(void);
extern void _CLOSEALL(void);
#ifdef __cplusplus
}
#endif

#define PSW_init  0x00010000     // PSW bit pattern
#define FPSW_init 0x00000000     // FPSW bit base pattern

//extern void srand(_UINT);      // Remove the comment when you use rand()
//extern _SBYTE *_s1ptr;         // Remove the comment when you use strtok()

//#ifdef __cplusplus             // Use Hardware Setup
//extern "C" {
//#endif
//extern void HardwareSetup(void);
//#ifdef __cplusplus
//}
//#endif

//#ifdef __cplusplus             // Remove the comment when you use global class object
//extern "C" {                   // Sections C$INIT and C$END will be generated
//#endif
//extern void _CALL_INIT(void);
//extern void _CALL_END(void);
//#ifdef __cplusplus
//}
//#endif
#pragma section ResetPRG         // output PowerON_Reset_PC to PResetPRG section

#pragma entry PowerON_Reset_PC

void PowerON_Reset_PC(void)
{
     set_intb(__sectop("C$VECT"));
#ifdef __ROZ// Initialize FPSW
#define _ROUND 0x00000001        // Let FPSW RMbits=01 (round to zero)
#else
#define _ROUND 0x00000000        // Let FPSW RMbits=00 (round to nearest)
#endif
#ifdef __DOFF
#define _DENOM 0x00000100        // Let FPSW DNbit=1 (denormal as zero)
#else
#define _DENOM 0x00000000        // Let FPSW DNbit=0 (denormal as is)
#endif
     set_fpsw(FPSW_init | _ROUND | _DENOM);

     _INITSCT();                 // Initialize Sections

     _INIT_IOLIB();              // Use SIM I/O

//   errno=0;                    // Remove the comment when you use errno
//   srand((_UINT)1);            // Remove the comment when you use rand()
//   _s1ptr=NULL;                // Remove the comment when you use strtok()
```

```
//   HardwareSetup();             // Use Hardware Setup
   nop();

//   _CALL_INIT();                // Remove the comment when you use global class object

     set_psw(PSW_init);           // Set Ubit & Ibit for PSW
//   chg_pmusr();                 // Remove the comment when you need to change PSW PMbit
(SuperVisor->User)

     main();

     _CLOSEALL();                 // Use SIM I/O

//   _CALL_END();                 // Remove the comment when you use global class object

     brk();
}
```

（b）intprg.c -- ベクタ関数の定義

```
#include <machine.h>
#include "vect.h"
#pragma section IntPRG

// Exception(Supervisor Instruction)
void Excep_SuperVisorInst(void){/* brk(); */}

// Exception(Undefined Instruction)
void Excep_UndefinedInst(void){/* brk(); */}

// Exception(Floating Point)
void Excep_FloatingPoint(void){/* brk(); */}

// NMI
void NonMaskableInterrupt(void){/* brk(); */}

// Dummy
void Dummy(void){/* brk(); */}

// BRK
void Excep_BRK(void){ wait(); }
```

（c）vecttbl.c -- 固定ベクタテーブル

```
#include "vect.h"

#pragma section C FIXEDVECT

void (*const Fixed_Vectors[])(void) = {
//;0xffffffd0  Exception(Supervisor Instruction)
    Excep_SuperVisorInst,
//;0xffffffd4  Reserved
    Dummy,
//;0xffffffd8  Reserved
    Dummy,
//;0xffffffdc  Exception(Undefined Instruction)
    Excep_UndefinedInst,
//;0xffffffe0  Reserved
    Dummy,
//;0xffffffe4  Exception(Floating Point)
    Excep_FloatingPoint,
//;0xffffffe8  Reserved
    Dummy,
//;0xffffffec  Reserved
    Dummy,
//;0xfffffff0  Reserved
    Dummy,
//;0xfffffff4  Reserved
    Dummy,
//;0xfffffff8  NMI
    NonMaskableInterrupt,
//;0xfffffffc  RESET
//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
PowerON_Reset_PC
//;<<VECTOR DATA END (POWER ON RESET)>>
};
```

（d）dbsct.c -- セクションの初期化処理 (テーブル)

```
#include "typedefine.h"

#pragma unpack

#pragma section C C$DSEC
extern const struct {
    _UBYTE *rom_s;       /* Start address of the initialized data section in ROM */
    _UBYTE *rom_e;       /* End address of the initialized data section in ROM   */
    _UBYTE *ram_s;       /* Start address of the initialized data section in RAM */
}   _DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D_2"), __secend("D_2"), __sectop("R_2") },
    { __sectop("D_1"), __secend("D_1"), __sectop("R_1") }
};
#pragma section C C$BSEC
extern const struct {
    _UBYTE *b_s;         /* Start address of non-initialized data section */
    _UBYTE *b_e;         /* End address of non-initialized data section */
}   _BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B_2"), __secend("B_2") },
    { __sectop("B_1"), __secend("B_1") }
};

#pragma section

/*
** CTBL prevents excessive output of L1100 messages when linking.
** Even if CTBL is deleted, the operation of the program does not change.
*/
_UBYTE * const _CTBL[] = {
    __sectop("C_1"), __sectop("C_2"), __sectop("C"),
    __sectop("W_1"), __sectop("W_2"), __sectop("W")
};

#pragma packoption
```

（e）lowsrc.c -- 低水準インタフェースルーチン (C 言語部分)

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>
#include "lowsrc.h"

#define STDIN  0
#define STDOUT 1
#define STDERR 2

#define FLMIN  0
#define _MOPENR   0x1
#define _MOPENW   0x2
#define _MOPENA   0x4
#define _MTRUNC   0x8
#define _MCREAT   0x10
#define _MBIN     0x20
#define _MEXCL    0x40
#define _MALBUF   0x40
#define _MALFIL   0x80
#define _MEOF     0x100
#define _MERR     0x200
#define _MLBF     0x400
#define _MNBF     0x800
#define _MREAD    0x1000
#define _MWRITE   0x2000
#define _MBYTE    0x4000
#define _MWIDE    0x8000

#define O_RDONLY  0x0001
#define O_WRONLY  0x0002
#define O_RDWR    0x0004
#define O_CREAT   0x0008
#define O_TRUNC   0x0010
#define O_APPEND  0x0020

#define CR 0x0d
#define LF 0x0a

extern const long _nfiles;
char flmod[IOSTREAM];

unsigned char sml_buf[IOSTREAM];

#define FPATH_STDIN     "C:\\stdin"
#define FPATH_STDOUT    "C:\\stdout"
#define FPATH_STDERR    "C:\\stderr"

extern void charput(unsigned char);
extern unsigned char charget(void);

#include <stdio.h>
FILE *_Files[IOSTREAM];
char *env_list[] = {
    "ENV1=temp01",
    "ENV2=temp02",
    "ENV9=end",
    '\0'
};
```

```
char **environ = env_list;

void _INIT_IOLIB( void )
{
    if( freopen( FPATH_STDIN, "r", stdin ) == NULL )
        stdin->_Mode = 0xffff;
    stdin->_Mode  = _MOPENR;
    stdin->_Mode |= _MNBF;
    stdin->_Bend = stdin->_Buf + 1;

    if( freopen( FPATH_STDOUT, "w", stdout ) == NULL )
        stdout->_Mode = 0xffff;
    stdout->_Mode |= _MNBF;
    stdout->_Bend = stdout->_Buf + 1;

    if( freopen( FPATH_STDERR, "w", stderr ) == NULL )
        stderr->_Mode = 0xffff;
    stderr->_Mode |= _MNBF;
    stderr->_Bend = stderr->_Buf + 1;
}

void _CLOSEALL( void )
{
    long i;
    for( i=0; i < _nfiles; i++ )
    {
        if( _Files[i]->_Mode & (_MOPENR | _MOPENW | _MOPENA ) )
        fclose( _Files[i] );
    }
}

long open(const char *name,
     long  mode,
     long  flg)
{

    if( strcmp( name, FPATH_STDIN ) == 0 )
    {
        if( ( mode & O_RDONLY ) == 0 ) return -1;
        flmod[STDIN] = mode;
        return STDIN;
    }
    else if( strcmp( name, FPATH_STDOUT ) == 0 )
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDOUT] = mode;
        return STDOUT;
    }
    else if(strcmp(name, FPATH_STDERR ) == 0 )
    {
        if( ( mode & O_WRONLY ) == 0 ) return -1;
        flmod[STDERR] = mode;
        return STDERR;
    }
    else return -1;
}

long close( long fileno )
{
    return 1;
}
```

```
long write(long  fileno,
      const unsigned char *buf,
      long  count)
{
    long    i;
    unsigned char    c;

    if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR)
    {
        if( fileno == STDIN ) return -1;
        else if( (fileno == STDOUT) || (fileno == STDERR) )
        {
            for( i = count; i > 0; --i )
            {
                c = *buf++;
                charput(c);
            }
            return count;
        }
        else return -1;
    }
    else return -1;
}

long read( long fileno, unsigned char *buf, long count )
{
   long i;
       if((flmod[fileno]&_MOPENR) || (flmod[fileno]&O_RDWR)){
             for(i = count; i > 0; i--){
                   *buf = charget();
                   if(*buf==CR){
                         *buf = LF;
                   }
                   buf++;
             }
             return count;
       }
       else {
             return -1;
       }
}

long lseek( long fileno, long offset, long base )
{
    return -1L;
}
```

（f）lowlvl.src -- 低水準インタフェースルーチン (アセンブリ言語部分)

```
          .GLB    _charput
          .GLB    _charget

SIM_IO    .EQU 0h

          .SECTION   P,CODE
;------------------------------------------------------------------------
;  _charput:
;------------------------------------------------------------------------
_charput:
          MOV.L      #IO_BUF,R2
          MOV.B      R1,[R2]
          MOV.L      #1220000h,R1
          MOV.L      #PARM,R3
          MOV.L      R2,[R3]
          MOV.L      R3,R2
          MOV.L      #SIM_IO,R3
          JSR        R3
          RTS

;------------------------------------------------------------------------
;  _charget:
;------------------------------------------------------------------------
_charget:
         MOV.L       #1210000h,R1
         MOV.L       #IO_BUF,R2
         MOV.L       #PARM,R3
         MOV.L       R2,[R3]
         MOV.L       R3,R2
         MOV.L       #SIM_IO,R3
         JSR         R3
         MOV.L       #IO_BUF,R2
         MOVU.B      [R2],R1
         RTS

;------------------------------------------------------------------------
;  I/O Buffer
;------------------------------------------------------------------------
          .SECTION  B,DATA,ALIGN=4
PARM:     .BLKL     1
          .SECTION  B_1,DATA
IO_BUF:   .BLKB     1
          .END
```

（g）sbrk.c -- 低水準インタフェースルーチン (sbrk 関数)

```
#include <stddef.h>
#include <stdio.h>
#include "typedefine.h"
#include "sbrk.h"

_SBYTE  *sbrk(size_t size);

//const size_t _sbrk_size=              /* Specifies the minimum unit of*/
                                        /* the defined heap area*/

extern _SBYTE *_s1ptr;

union HEAP_TYPE {
    _SDWORD  dummy ;                    /* Dummy for 4-byte boundary*/
    _SBYTE heap[HEAPSIZE];              /* Declaration of the area managed by sbrk*/
};

static union HEAP_TYPE heap_area ;

/* End address allocated by sbrk*/
static _SBYTE *brk=(_SBYTE *)&heap_area;

/************************************************************************/
/*     sbrk:Memory area allocation                                      */
/*          Return value:Start address of allocated area (Pass)         */
/*                       -1                          (Failure)          */
/************************************************************************/
_SBYTE  *sbrk(size_t size)                  /* Assigned area size   */
{
    _SBYTE  *p;

    if(brk+size > heap_area.heap+HEAPSIZE){     /* Empty area size      */
        p = (_SBYTE *)-1;
    }
    else {
        p = brk;                            /* Area assignment      */
        brk += size;                        /* End address update   */
    }
    return p;
}
```

（h）typedefine.h -- 型定義ヘッダ

```
typedef signed char _SBYTE;
typedef unsigned char _UBYTE;
typedef signed short _SWORD;
typedef unsigned short _UWORD;
typedef signed int _SINT;
typedef unsigned int _UINT;
typedef signed long _SDWORD;
typedef unsigned long _UDWORD;
typedef signed long long _SQWORD;
typedef unsigned long long _UQWORD;
```

(i) vect.h -- ベクタ関数のヘッダ

```
// Exception(Supervisor Instruction)
#pragma interrupt (Excep_SuperVisorInst)
void Excep_SuperVisorInst(void);

// Exception(Undefined Instruction)
#pragma interrupt (Excep_UndefinedInst)
void Excep_UndefinedInst(void);

// Exception(Floating Point)
#pragma interrupt (Excep_FloatingPoint)
void Excep_FloatingPoint(void);

// NMI
#pragma interrupt (NonMaskableInterrupt)
void NonMaskableInterrupt(void);

// Dummy
#pragma interrupt (Dummy)
void Dummy(void);

// BRK
#pragma interrupt (Excep_BRK(vect=0))
void Excep_BRK(void);

//;<<VECTOR DATA START (POWER ON RESET)>>
//;Power On Reset PC
extern void PowerON_Reset_PC(void);
//;<<VECTOR DATA END (POWER ON RESET)>>
```

(j) stacksct.h -- スタックサイズの設定

```
// #pragma stacksize su=0x100        // Remove the comment when you use user stack
#pragma stacksize si=0x300
```

(k) lowsrc.h -- 低水準インタフェースルーチン (C 言語ヘッダ)

```
/*Number of I/O Stream*/
#define IOSTREAM 20
```

(l) sbrk.h -- 低水準インタフェースルーチン (sbrk 関数のヘッダ)

```
/* size of area managed by sbrk */
#define HEAPSIZE 0x400
```

(2) 実行コマンド

これらのファイルをビルドするのに必要なコマンド列の例を示します。

この例では、ユーザプログラム (main 関数を含む) は UserProgram.c、生成するロードモジュールやライブラリなどのファイル名を LoadModule(拡張子を除いた部分) とします。

```
lbgrx -cpu=rx600 -output=LoadModule.lib
ccrx -cpu=rx600 -output=obj UserProgram.c
ccrx -cpu=rx600 -output=obj resetprg.c
ccrx -cpu=rx600 -output=obj intprg.c
ccrx -cpu=rx600 -output=obj vecttbl.c
ccrx -cpu=rx600 -output=obj dbsct.c
ccrx -cpu=rx600 -output=obj lowsrc.c
asrx -cpu=rx600 lowlvl.src
ccrx -cpu=rx600 -output=obj sbrk.c
rlink -rom=D=R,D_1=R_1,D_2=R_2 -list=LoadModule.map
-start=B_1,R_1,B_2,R_2,B,R,SI/01000,PResetPRG/0FFFF8000,C_1,C_2,C,C$*,D_1,D_2,D,P,PIntPRG,
W*,L/0FFFF8100,FIXEDVECT/0FFFFFFD0 -library=LoadModule.lib -output=LoadModule.abs
UserProgram.obj resetprg.obj intprg.obj vecttbl.obj dbsct.obj lowsrc.obj lowlvl.obj sbrk.obj
rlink -output=LoadModule.sty -form=stype -output=LoadModule.mot LoadModule.abs
```

7.5 PIC/PID 機能の利用

本章では、PIC/PID 機能の概要と、PIC/PID 機能を利用する場合のスタートアップの作成方法について説明します。

PIC/PID 機能は、一度リンクが完了して配置アドレスが確定した ROM 上のコードやデータを、リンクをやり直すことなく、任意のアドレスに配置して利用できるようにする機能です。

PIC は位置独立コード (Position Independent Code)、PID は位置独立データ (Position Independent Data) をそれぞれ意味します。PIC を生成する機能が PIC 機能、PID を生成する機能が PID 機能で、ここでは、これらの機能を総称して PIC/PID 機能と呼びます。

7.5.1 用語の定義

(1) マスタとアプリケーション

PIC/PID 機能では、ROM 上のコードやデータを PIC や PID にしたプログラムをアプリケーション、アプリケーションを実行させるのに必要なプログラムをマスタと呼びます。

マスタは、アプリケーションの起動処理のほか、アプリケーションから呼び出される共有ライブラリ、およびアプリケーションの RAM 領域を持ちます。PIC および PID はアプリケーションにのみ含まれ、マスタには含まれていません。

(2) 共有ライブラリ

マスタ内にあり、複数のアプリケーションから呼び出すことのできる関数群です。

(3) ジャンプテーブル

アプリケーションから共有ライブラリ関数への呼び出しを中継するプログラムです。

7.5.2 各オプションの機能

PIC/PID 機能と関連するオプションを説明します。

各オプションの機能詳細については、ビルド編の各オプションの説明を参照ください。

(1) アプリケーションのコード生成 (pic,pid オプション)

pic オプションを有効にしてコンパイルすると、PIC 機能が有効になり、コード領域 (P セクション) が PIC になります。PIC は分岐先アドレスや関数アドレスの取得を全て PC 相対で行うため、リンク後も任意のアドレスに配置することができます。

pid オプションを有効にしてコンパイルすると、PID 機能が有効になり、ROM データ領域 (C, C_2, C_1, W, W_2, W_1 および L セクション) が PID になります。プログラムは PID に対しその先頭アドレスを示すレジスタ (PID レジスタ) から相対のアクセスでアクセスします。ユーザはマスタで PID レジスタの設定値を変化させて、リンク後も PID を任意のアドレスに移動することができます。

なお、PIC 機能 (pic オプション) と PID 機能 (pid オプション) は、それぞれ独立した機能として動作できるように設計しておりますが、同時に有効にしたうえで、PIC と PID を隣接させてご利用いただくことを推奨します。PIC 機能と PID 機能を個別に利用したり、PIC と PID の相対距離を変更したアプリケーションのデバッグは、デバッガのバージョンによりサポートされない場合があります。本書でも PIC 機能と PID 機能を同時に有効にした例で説明しています。

(2) 共有ライブラリ対応 (jump_entries_for_pic, nouse_pid_register オプション)

アプリケーションからマスタにあるライブラリを呼び出すための機能です。

nouse_pid_register オプションは、マスタのコンパイル時に使用し、PID レジスタを使用しないコードを生成します。

jump_entries_for_pic オプションを、マスタのリンク時に最適化リンケージエディタに指定すると、アプリケーションから固定アドレスにあるライブラリ関数を呼び出すためのジャンプテーブルを生成します。

(3) RAM 領域の共有 (Fsymbol オプション)

マスタ上の変数を、リンク単位の違うアプリケーションからでも読み書きできるようにするための機能です。

マスタのリンク時に、Fsymbol オプションを最適化リンケージエディタに指定すると、アプリケーションから固定アドレスで変数を参照するためのシンボルテーブルを生成します。

7.5.3　アプリケーションに関する制限事項

(1) RAM 領域

RAM 領域には PID 機能を適用できません。

(2) アプリケーションの同時実行

PIC/PID 機能を使うと、同じアプリケーションのコピーを複数個 ROM に置き、それぞれを実行できますが、RAM 領域が重なっているため、同じアプリケーションのコピーを同時に複数個実行することはできません。

(3) スタートアップ

アプリケーションに使用するスタートアップとしては、標準のスタートアップ（統合開発環境が生成。詳しくは「7.3　スタートアッププログラム」を参照）はそのままでは使用できません。「7.5.7　アプリケーションのスタートアップ」を参考に、スタートアップを作成してください。

7.5.4　PIC/PID 機能で必要なシステム依存処理

次の処理は、システム仕様に合わせてお客様にてご用意いただく必要があります。

(1) マスタの初期化

PIC/PID 機能を使用しない通常のプログラムと同様の処理を行います。

(2) マスタからアプリケーションを起動

アプリケーションの PID の先頭アドレスを PID レジスタに設定し、PIC の起動アドレスへ分岐することで、アプリケーションを起動します。

(3) アプリケーションの初期化

セクションの初期化を行い、アプリケーションの main 関数を実行します。

(4) アプリケーションの終了

main 関数が終了したら、マスタに処理を返します。

7.5.5 コード生成オプションの組み合わせ

マスタとアプリケーションをビルドするときは、構成するオブジェクト間で PIC/PID 機能に関係するオプション指定を合わせておく必要があります。

以下に、オブジェクトごとのコンパイル時のオプションの指定規則と、組み合わせて利用できるオブジェクトのオプション指定の制限について示します。

(1) マスタ

マスタをビルドするときは、PIC/PID 機能オプションを表 7.5 のように指定してください。

表 7―5 マスタ内の PIC/PID 機能オプションの指定規則

	オプション名	コンパイル時	リンク可能オブジェクトのオプション指定の条件
1	pic	× 指定不可	pic の指定なし
2	pid	× 指定不可	pid の指定なし
3	nouse_pid_register	△ 標準ライブラリ、スタートアップ内の PID レジスタ設定箇所以外は指定可	制限なし
4	fint_register	○ 指定可	同一パラメータの fint_register の指定が必須
5	base	○ 指定可	同一パラメータの base の指定が必須

(2) アプリケーション

アプリケーションをビルドするときは、PIC/PID 機能オプションを表 7.6 のように指定してください。

表 7―6 アプリケーション内の PIC/PID 機能オプションの指定規則

	オプション名	コンパイル時	リンク可能オブジェクトのオプション指定の条件
1	pic	○ 指定可	pic の指定が必須
2	pid	○ 指定可	pid の指定が必須
3	nouse_pid_register	× 指定不可	nouse_pid_register の指定なし
4	fint_register	○ 指定可	同一パラメータの fint_register の指定が必須
5	base	○ 指定可	同一パラメータの base*1 の指定が必須

注 1. pid 指定時は base=rom=< レジスタ > の指定はできません。

(3) マスタとアプリケーション間

マスタとアプリケーションはそれぞれ PIC/PID 機能オプションを表 7.7 のように指定する必要があります。

表 7―7 マスタとアプリケーション間の PIC/PID 機能オプションの組み合わせ規則

	アプリケーションのオプション	マスタのオプション
1	pic	制限なし
2	pid	アプリケーションからマスター上の関数を呼び出す場合は、nouse_pid_register が必須
3	fint_register	同一パラメータの fint_register が必須
4	base	同一パラメータの base*1 が必須

注 1. pid 指定時は base=rom=< レジスタ > の指定はできません。

7. 5. 6 マスタのスタートアップ

次の 2 点を除き、必要な処理は PIC/PID 機能を用いない通常のプログラムと同じです。「7.3 スタートアッププログラムの作成」に従ったスタートアップに、次の 2 つの内容を追加してください。

(1) アプリケーションの起動と復帰

main 関数で、PID レジスタを設定し、PIC のエントリアドレスに分岐してアプリケーションを起動します。また、アプリケーションからマスタに戻れる手段を用意しておく必要があります。

(2) 使用する共有ライブラリ関数の参照

アプリケーションが利用する共有ライブラリは、あらかじめマスタでも参照しておく必要があります。

以下に、main 関数から PIC/PID アプリケーションを呼び出す例を示します。

なお、この例は次の条件に基づきます。

- アプリケーションの終了時、RTS 命令でマスタに復帰できるものとします。
- アプリケーションは戻り値を持たないものとします。
- アプリケーションに対する PIC の起動アドレス (PIC_entry)、および PID の先頭アドレス (PID_address) は、マスタをビルドする時点で既知および固定であるとします。
- PID レジスタは R13 であるものとします。
- アプリケーション側のセクション領域の初期化は、マスタ側では行わないこととします。
- 共有ライブラリとして、アプリケーションは printf 関数のみ使用するものとします。

例

```
/* マスタ側プログラム */
/* PIC/PID アプリケーションを起動する */
/* ( アプリケーションが、戻り値を返さず、RTS で復帰するシステム仕様の場合 ) */
#include <stdio.h>
#pragma inline_asm Launch_PICPID_Application
void Launch_PICPID_Application(void *pic_entry, void *pid_address)
{
    MOV.L    R2,R13
    JSR      R1
}
int main()
{
    void *PIC_entry   = (void*)0x500000;  /* PIC の起動アドレス */
    void *PID_address = (void*)0x120000;  /* PID の先頭アドレス */

    /* (1) アプリケーションの起動と復帰 */
    Launch_PICPID_Application(PIC_entry, PID_address);

    return 0;
}

/* (2) アプリケーションで使用する共有ライブラリの参照 */
void *_dummy_ptr = (void*)printf;  /* printf 関数 */
```

7.5.7 アプリケーションのスタートアップ

アプリケーションでは、次の項目を設定してください。

【オプション】と書かれている項目は、不要な場合があります。

(1) エントリポイント (PIC の起動アドレス) の用意

アプリケーションの起動アドレスです。

(2) スタックポインタの初期化【オプション】

マスタとスタックを共有する場合は、不要です。

必要な場合は、7.3.2(2) を参考に、設定を追加してください。

(3) ベースレジスタに使用する汎用レジスタの初期化【オプション】

ベースレジスタを使用しない場合は、不要です。

必要な場合は、7.3.2(3) を参考に、設定を追加してください。

(4) セクションの初期化処理【オプション】

マスタ側で初期化する場合は、不要です。

必要な場合は、後述の例を参考に、設定を追加してください。

なお、7.3.2(5) の方法はそのままでは使用できません。

(5) ライブラリの初期化処理【オプション】

標準ライブラリを使用しない場合は、不要です。

必要な場合は、7.3.2(6) を参考に、設定を追加してください。

(6) main 関数向け PSW 初期化【オプション】

必要に応じて、割り込みマスクやユーザモードへの移行を行います。

7.3.2(8) と (9) を参考に、設定を追加してください。

(7) ユーザプログラムの実行

main 関数を実行します。

7.3.2(10) を参考に、設定してください。

以下、アプリケーション側のスタートアップ例を示します。

3 つのファイルに分かれています。

- startup_picpid.c … スタートアップ本体。
- initsct_pid.src … セクション初期化の PID 版である _INITSCT_PID です。

  7.3.2(5) で述べている _INITSCT 関数を PID 対応にしたものです。

  なお、PID レジスタを R13 に固定化しているので、PID レジスタが R13 ではない場合は、R13 を PID レジスタに書き換える必要があります。
- initiolib.c … 標準ライブラリの初期化を行う、_INITLIB を収録しています。

  7.3.2(6) をアプリケーション用に変更したものです。

```
[startup_picpid.c]
// マニュアル 7.3.2(5) セクションの初期化処理
#pragma section C C$DSEC // セクション名を C$DSEC にします
const struct {
    void *rom_s; // 初期化データセクションの ROM 上の先頭アドレスメンバ
    void *rom_e; // 初期化データセクションの ROM 上の最終アドレスメンバ
    void *ram_s; // 初期化データセクションの RAM 上の先頭アドレスメンバ
} DTBL[] = {__sectop("D"), __secend("D"), __sectop("R")};
#pragma section C C$BSEC // セクション名を C$BSEC にします
const struct {
    void *b_s; // 未初期化データセクションの先頭アドレスメンバ
    void *b_e; // 未初期化データセクションの最終アドレスメンバ
} BTBL[] = {__sectop("B"), __secend("B")};

extern void main(void);
extern void _INITLIB(void); // マニュアル 7.3.2(6) ライブラリの初期化処理
#pragma entry application_pic_entry
void application_pic_entry(void)
{
    _INITSCT_PICPID();
    _INITLIB();
     main();
}

[initsct_pid.src]
; PID 対応 セクション初期化ルーチン
; ** 注意 ** PID レジスタのチェックが必要です
; このコードは PID レジスタが R13 であると想定しています . もし、PID レジスタが
; R13 でなければ、以下の R13 の記述を PID レジスタに変更してください。
    .glb    __INITSCT_PICPID
    .glb    __PID_TOP
    .section  C$BSEC,ROMDATA,ALIGN=4
    .section  C$DSEC,ROMDATA,ALIGN=4
    .section  P,CODE

__INITSCT_PICPID:               ; function: _INITSCT
    .STACK    __INITSCT_PICPID=28
    PUSHM     R1-R6
    ADD       #-__PID_TOP,R13,R6   ;   How long distance PID moves
;;;
;;; clear BBS(B)
;;;
    ADD       #TOPOF C$BSEC,  R6, R4
    ADD       #SIZEOF C$BSEC, R4, R5
    MOV.L     #0, R2
    BRA       next_loop1
```

```
loop1:
    MOV.L     [R4+], R1
    MOV.L     [R4+], R3
    CMP       R1, R3
    BLEU      next_loop1
    SUB       R1, R3
    SSTR.B
next_loop1:
    CMP       R4,R5
    BGTU      loop1

;;;
;;; copy DATA from ROM(D) to RAM(R)
;;;
    ADD       #TOPOF C$DSEC,  R6, R4
    ADD       #SIZEOF C$DSEC, R4, R5
    BRA       next_loop3

loop3:
    MOV.L     [R4+], R2
    MOV.L     [R4+], R3
    MOV.L     [R4+], R1
    CMP       R2, R3
    BLEU      next_loop3
    SUB       R2, R3
    ADD       R6, R2       ; Adjust for real address of PID
    SMOVF
next_loop3:
    CMP       R4, R5
    BGTU      loop3
    POPM      R1-R6
    RTS

    .end

[initiolib.c]
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520; // ヒープ領域確保サイズの最小単位を指定します
// ( 省略時：1024)

void _INIT_LOWLEVEL(void);
void _INIT_OTHERLIB(void);

void _INITLIB (void)
{
    _INIT_LOWLEVEL(); // 低水準インタフェースルーチンの初期設定をします
    _INIT_IOLIB(); // 入出力ライブラリの初期設定をします
    _INIT_OTHERLIB(); // rand 関数、strtok 関数の初期設定をします
}
void _INIT_LOWLEVEL(void)
{ // 低水準ライブラリで必要な初期設定をしてください
}
void _INIT_OTHERLIB(void)
{
     srand(1); // rand 関数を使用する場合の初期設定です
}
```

第 8 章　コンパイラとアセンブラの相互参照

この章では，CCRX におけるプログラム呼び出し時の引数などの扱い方について説明します。

8. 1　スタックに関する規則

(1) スタックポインタ

スタックポインタの指すアドレスよりも下位 (0 番地の方向) のスタック領域に有効なデータを格納してはいけません。スタックポインタより下位アドレスに格納されたデータは、割り込み処理で破壊される可能性があります。

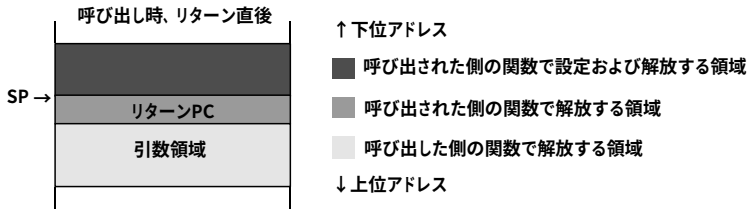
(2) スタックフレームの割り付け、解放

関数呼び出しが行われた時点 (JSR または BSR 命令の実行直後) では、スタックポインタは呼び出した関数側で使用したスタックの最下位アドレスを指しています。このスタックポインタの指している領域より上位アドレスのデータの割り付け、設定は呼び出す側の関数の役目です。

関数のリターン時は、呼び出された関数で確保した領域を解放してから、通常 RTS 命令を用いて呼び出した関数へ返ります。これより上位アドレスの領域 (引数の領域) は、呼び出した側の関数で解放します。

図 8.1 は、関数呼び出し直後のスタックフレームの状態を説明したものです。

図 8―1　スタックフレームの割り付け、解放に関する規則



8. 2　レジスタに関する規則

関数呼び出し前後において、レジスタの値が同一であることを保証するかどうかは、レジスタにより異なります。また、オプションにより特定の用途向けに使用するレジスタがあります。レジスタの使用規則を表 8.1 に示します。

表 8―1　レジスタ使用規則

レジスタ	関数呼び出し前後で値を保証	関数入口	関数出口	高速割り込み用レジスタ *1	ベースレジスタ *2	PIDレジスタ *3
R0	保証する	スタックポインタ	スタックポインタ	-	-	-
R1	保証しない	引数 1	戻り値 1	-	-	-
R2	保証しない	引数 2	戻り値 2	-	-	-
R3	保証しない	引数 3	戻り値 3	-	-	-
R4	保証しない	引数 4	戻り値 4	-	-	-
R5	保証しない	-	(不定)	-	-	-
R6	保証する	-	(入口の値を保持)	-	-	-
R7	保証する	-	(入口の値を保持)	-	-	-
R8	保証する	-	(入口の値を保持)	-	○	-
R9	保証する	-	(入口の値を保持)	-	○	○
R10	保証する	-	(入口の値を保持)	○	○	○
R11	保証する	-	(入口の値を保持)	○	○	○
R12	保証する	-	(入口の値を保持)	○	○	○
R13	保証する	-	(入口の値を保持)	○	○	○
R14	保証しない	-	(不定)	-	-	-
R15	保証しない	構造体戻り値へのポインタ	(不定)	-	-	-
ISP USP	スタックポインタの場合は R0 と同じ。 そうでない場合は変化しません。*4			-	-	-
PC	-	プログラムカウンタ *5		-	-	-
PSW	保証しない	-	(不定)	-	-	-
FPSW	保証しない	-	(不定)	-	-	-
ACC	保証しない *6	-	(不定) *6	-	-	-
INTB BPC BPSW FINTV CPEN	-	変化しません *4	-	-	-	-

注 1.　R10 ～ R13 の 4 本は、fint_register オプションにより、一部または全部が「高速割り込み機能」に使われることがあります。「高速割り込み機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。

2.　R8 ～ R13 の 6 本は、base オプションにより、一部または全部が「ベースレジスタ機能」に使われることがあります。「ベースレジスタ機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。

3. R9 ～ R13 のうちの 1 本は、pid オプションにより「PID 機能」に使われることがあります。「PID 機能」に割り当てられたレジスタは、他の用途に使用することはできません。機能の詳細はオプションの説明を参照してください。
4. 組み込み関数または #pragma inline_asm で、これらのレジスタを設定したり更新したりする場合を除きます。
5. 関数の呼び出しに使用する命令の仕様に従います。関数の呼び出しには、BSR, JSR, BRA および JMP のいずれかの命令を用います。
6. ACC(アキュムレータ) を更新する命令は、RX のソフトウェアマニュアルを参照してください。

8. 3　引数の設定、参照に関する規則

引数に対する一般的な規則と、引数の割り付け方について述べます。
引数が実際どのように割り付けられるかは、「8. 5　引数割り付けの具体例」を参照ください。

(1) 引数の渡し方

引数の値を、必ずレジスタまたはスタック上の引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出した側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出した側の処理は直接には影響を受けません。

(2) 型変換の規則

(a) 関数原型によって型が宣言されている引数は、宣言された型に変換します。

(b) 関数原型によって型が宣言されていない引数の型変換は、以下の規則に従って変換します。

- 2 バイト以下の整数型は、4 バイト整数型に変換されます。
- float 型の引数は、double 型に変換します。
- 上記以外の引数は、変換しません。

例

```
void p(int,... );
void f( )
{
  char c;
  p(1.0, c);
}
```

cは、対応する引数の型宣言がないので、4バイト整数型に変換されます。

1.0は、対応する引数の型がint型なので、4バイト整数型に変換されます。

(3) 引数の割り付け領域

引数は、レジスタに割り付ける場合とスタック上の引数領域に割り付ける場合があります。引数の割り付け領域を図 8.2 に示します。

通常、ソースプログラムにおける引数の宣言順に、番号の小さいレジスタから順に割り付けを行い、レジスタが全て割り付いたらスタックに割り付けます。但し、可変個の引数を持つ関数など、レジスタが余っていて

もスタックに割り付ける場合もあります。また、C++ プログラムの非静的関数メンバの this ポインタは、常に R1 に割り付けられます。

引数割り付け領域の一般規則を表 8―2 にそれぞれ示します。

図 8―2　引数の割り付け領域

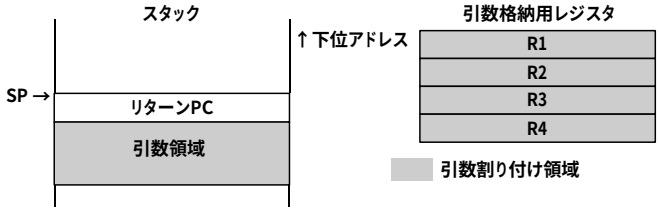


表 8―2　引数割り付け領域の一般規則

レジスタで渡される引数			スタック渡しになる引数
対象の型	引数格納用レジスタ	割り付け方	
signed char, (unsigned)char, bool, _Bool, (signed)short, unsigned short, (signed)int, unsigned int, (signed)long, unsigned long, float、double*1, long double*1, ポインタ , データメンバへのポインタ , リファレンス	R1 ～ R4 のうち 1 つ	signed char, (signed)short は符号拡張、(unsigned)char, unsigned short はゼロ拡張を行った結果を割り付ける その他の型はそのままレジスタに割り付ける	(1) 引数の型がレジスタ渡しの対象の型以外のもの (2) 関数原型により可変個の引数を持つ関数として宣言しているもの *3 (3) R1 ～ R4 のうち、まだ他の引数に割り当てられていないものの本数が、割り当てに必要な本数より少ない場合
(signed)long long, unsigned long long, double*2, long double*2	R1 ～ R4 のうち 2 つ	下位 4 バイトを番号の少ない方に、上位 4 バイトを番号の大きい方に割り付ける	
16 バイト以内でサイズが 4 の倍数の構造体型、共用体型、クラス型	R1 ～ R4 のうち、サイズを 4 で割った数	メモリイメージの先頭から 4 バイトずつ、レジスタ番号が増える方向に割り付ける	

注 1. dbl_size=8 を指定しなかった場合です。

2. dbl_size=8 を指定した場合です。

3. 関数原型により可変個の引数をもつ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタック渡しになります。型のない引数は、2 バイト以下の整数は long 型に、float 型は double 型にそれぞれ変換して、全て境界調整数が 4 の引数として取り扱います。

例

```
int f2(int,int,int,int,...);
:
f2(a,b,c,x,y,z); → x、y、z はスタック渡しになります。
```

(4) スタック渡しとなる引数の割り付け方

表 8.2 で、スタック渡しとなる引数の、配置アドレス、およびスタックへの配置の仕方は以下となります。

- 各引数は、その境界調整数に応じたアドレスに配置します。
- 引数並びの左から右の順に、スタックが深くなる方向に配置されるように、スタックの引数用領域に格納します。すなわち、引数 A とその右隣の引数 B がともにスタック渡しとなる場合、引数 B のアドレスは、引数 A の配置アドレスに引数 A の占有サイズを加えたアドレスを、引数 B の境界調整数に整合させたアドレス、となります。

8.4　リターン値の設定、参照に関する規則

リターン値に対する一般的な規則と、リターン値の設定場所について述べます。

(1) リターン値の型変換

リターン値は、その関数の返す型に変換します。

例

```
long f( );
long f( )
{
    float x;
    return x;  <-- 関数原型にしたがってリターン値はlong型に変換されます。
}
```

(2) リターン値の設定場所

関数のリターン値の型によっては、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は表 8―3 を参照してください。

表 8―3　リターン値の型と設定場所

No.	リターン値の型	リターン値の設定場所
1	singed char, (unsigned)char, (singed)short, unsigned short, (singed)int, unsigned int, (signed)long, unsigned long, float, double*2, long double*2, ポインタ , bool, _Bool, リファレンス , データメンバへのポインタ	R1 但し、signed char, (signed)short は符号拡張、 (unsigned)char, unsigned short はゼロ拡張を行った結果を設定
2	double*3, long double*3, (signed)long long, unsigned long long	R1, R2 下位 4 バイトを R1 に、上位 4 バイトを R2 に設定
3	16バイト以内かつ4の倍数であるサイズの構造体、共用体、クラス型	メモリイメージの先頭から 4 バイトずつ R1,R2,R3,R4 の順に設定
4	3. 以外の構造体、共用体、クラス型	リターン値設定領域 (メモリ)*1

注 1. 関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスを R15 に設定してから関数を呼び出します。

2. dbl_size=8 を指定しなかった場合です。

3. dbl_size=8 を指定した場合です。

8.5　引数割り付けの具体例

引数割り付けの具体例を示します。なお、アドレスは全ての図で右から左に向かって増加します (左側が上位アドレス)。

例 1. レジスタ渡しの対象の型である引数は、宣言順にレジスタ R1 ～ R4 に割り付けます。
途中でレジスタ渡しとならない引数があった場合、それ以降の引数はレジスタ渡しの対象となります。スタック上では、その引数の境界調整数に補正したアドレスに配置されます。

```
int f(
    unsigned char,
    long long,
    long long,
    short,
    int,
    char,
    short,
    char,
    char,
    short);
        :
f(1,2,3,4,5,6,7,8,9,10);
 /*
** 1, 2, 4 がレジスタ渡しとなる
*/
```

＜レジスタ＞

R1	0x000000 (ゼロ拡張)	0x01
R2	0x00000002	
R3	0x00000000	
R4	0x0000 (符号拡張)	0x0004

＜スタック＞

*(R0+0)	0x00000003		
*(R0+4)	0x00000000		
*(R0+8)	0x00000005		
*(R0+12)	0x0007	空領域	0x06
*(R0+16)	0x000A	0x09	0x08

2. サイズが 16 バイト以下、かつ 4 の倍数である構造体および共用体型の引数は、レジスタ渡しの対象となります。それ以外の構造体および共用体型の引数は、スタック渡しとなります。

```
union  U { int a[2]; int b; } u;
struct S { short d; char c[4]; } s;
struct T { char g;  char f[2]; char e; } t;
int f(union U, struct S, struct T);
...
f(u, s, t);

 /*
** uは8バイトなのでレジスタ渡し
** sは6バイトなのでスタック渡し
** tは4バイトなのでレジスタ渡し
*/
```

＜レジスタ＞

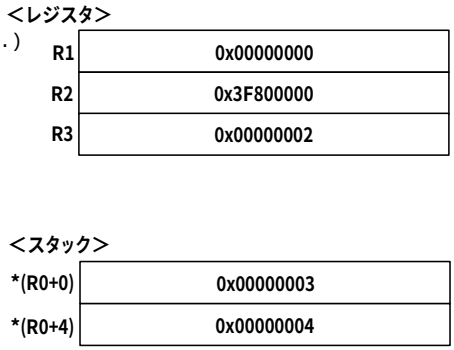
R1	u.a[0] (=u.b)			
R2	u.a[1]			
R3	t.e	t.f[1]	t.f[0]	t.g

＜スタック＞

*(R0+0)	s.c[1]	s.c[0]	s.d	
*(R0+4)	空領域		s.c[3]	s.c[2]

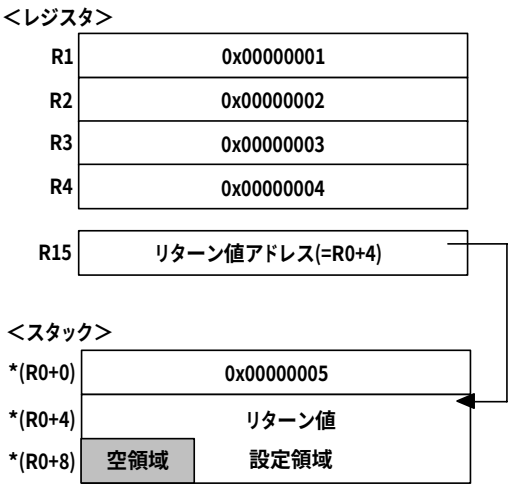
3. 関数原型により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタック渡しになります。

```
int f(int, float, int, int, ...)
    :
f(0, 1.0, 2, 3, 4)
```



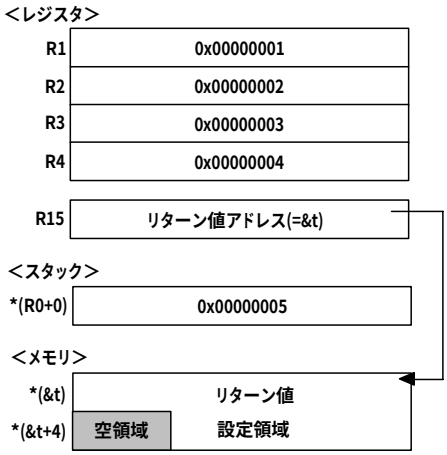
4. 関数の返す型が 16 バイトを超える、または 4 の倍数でないサイズの構造体または共用体型の場合、R15 にリターン値アドレスを設定します。

```
struct S{char a[7];};
struct S f(
   int a1,
   int a2,
   int a3,
   int a4,
   int a5);
   :
f(1,2,3,4,5);
```



5. リターン値をメモリに設定する場合、通常例 4 のようにスタックを確保して設定しますが、リターン値を変数に設定する場合、スタックを確保せず、その変数のメモリ領域に直接設定します。この場合、R15 にはその変数のアドレスを設定します。

```
struct S{char a[7];}t;
struct S f(
   int a1,
   int a2,
   int a3,
   int a4,
   int a5);
   :
t=f(1,2,3,4,5);
```



8. 6　外部名の相互参照方法

C/C++ プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。コンパイラは、次のものを外部名として扱います。

- 大域変数であって、かつ static 記憶クラスでないもの (C/C++ プログラム)
- extern 記憶クラスで宣言されている変数名 (C/C++ プログラム)
- static 記憶クラスを指定されていない関数名 (C プログラム)
- static 記憶クラスを指定されてない非メンバ非インライン関数名 (C++ プログラム)
- 非インラインメンバ関数名 (C++ プログラム)
- 静的データメンバ名 (C++ プログラム)

(1) アセンブリプログラムの外部名を C/C++ プログラムで参照する方法

アセンブリプログラムでは、.glb を用いてシンボル名 (先頭に下線 "_" を付与) を外部定義宣言します。

C/C++ プログラムでは、シンボル名 (先頭に下線 "_" がない) を「extern」宣言します。

アセンブリプログラム (定義する側)

```
        .glb    _a, _b
        .SECTION D,ROMDATA,ALIGN=4
_a:     .LWORD 1
_b:     .LWORD 1
        .END
```

C/C++ プログラム (参照する側)

```
extern int a,b;

void f()
{
a+=b;
}
```

(2) C/C++ プログラムの外部 (変数および C 関数) 名をアセンブリプログラムから参照する方法

C/C++ プログラムでは、変数名 (先頭に下線 "_" がない) を外部定義します。

アセンブリプログラムでは、.IMPORT を用いて外部名 (先頭に下線 "_" を付与) を外部参照宣言します。

C/C++ プログラム (定義する側)

```
int a;
```

アセンブリプログラム (参照する側)

```
        .GLB      _a
        .SECTION  P,CODE
        MOV.L     #A_a,R1
        MOV.L     [R1],R2
        ADD       #1,R2
        MOV.L     R2,[R1]
        RTS
        .SECTION  D,ROMDATA,ALIGN=4
A_a:    .LWORD    _a
        .END
```

(3) C++ プログラムの外部 (関数) 名をアセンブリプログラムから参照する方法

アセンブリプログラムで参照する関数を「extern "C"」を用いて宣言することにより、(2) と同じ規則で参照できます。ただし、「extern "C"」を用いて宣言した関数は多重定義できません。

C++ プログラム (呼び出される側)

```
 extern "C"
void sub ( )
{
    :
}
```

アセンブリプログラム (呼び出す側)

```
.GLB     _sub
.SECTION P,CODE
          :



PUSH.L   R13
MOV.L    4[R0],R1
MOV.L    R3,R12
MOV.L    #_sub,R14
JSR      R14
POP      R13
RTS
          :
.END
```

第 9 章　注意事項

この章では，CCRX を用いる際に注意すべき点について説明します。

9.1　コーディング上の注意事項

(1) 関数原型について

関数を呼び出す際には、呼び出される関数の関数原型を行ってください。関数原型を行わない場合、パラメータの受け渡しが正しく行えない場合があります。

例 1.

float 型パラメータをもつ関数（dbl_size=8 を指定した場合）

```
void g()
{
            float a;
            ...
            f(a);                       // a は double 型に変換されます。
}
void f(float x)
{...}
```

2.

スタック渡しとなる signed char、(unsigned)char、(signed)short、および unsigned short 型のパラメータをもつ関数

```
void h();
void g()
{
            char a,b;
            ...
            h(1,2,3,4,a,b);             // a,b は int 型に変換されます。
}
void h(int a1, int a2, int a3, int a4, char a5, char a6)
{...}
```

(2) 引数に型情報のない関数宣言

同じ関数に対して関数宣言 (関数定義を含む) を複数行うとき、引数並びに型を記述しない形式と、型を記述する形式を両方使用しないでください。

この場合、呼び出す関数と呼び出される関数とで引数の解釈に違いが生じるため、生成コードが型を正しく処理できない場合があります。

コンパイル時に C5147 のエラーメッセージが表示された場合、この問題に該当している可能性がありますので、引数並びに型を記述する形式に変更するか、生成コードを確認して引数の受け渡しに問題がないかを確認してください。

例

old_style を異なる形式で記述しているために、引数 d と e の型の意味が呼び出す関数と呼び出される関数で異なるため、引数の受け渡しが正しく行われません。

```
extern int old_style(int,int,int,short,short);  /* 関数宣言 : 引数並びに型を記述する形式 */
int old_style(a,b,c,d,e)    /* 関数定義 : 引数並びに型を記述しない形式 */
 int a,b,c;
 short d,e;
{
    return a + b + c + d + e;
}
int result;
func()
{
    result = old_style(1,2,3,4,5);
}
```

(3) C/C++ 言語で評価順序を規定していない式

C/C++ 言語規格で評価順序が規定されていない式を用いる際、評価順序で結果が変わるようなコーディングをした場合は動作保証しません。

例

```
a[i]=a[++i];   代入式の右辺を先に評価するか後に評価するかで左辺の値が変わります。
sub(++i, i);   関数の第 1 引数を先に評価するか後に評価するかで第 2 引数の値が変わります。
```

(4) オーバーフロー演算、ゼロ除算

オーバーフロー演算や浮動小数点のゼロ除算があっても、エラーメッセージを出力しません。ただし、一つの定数または定数どうしの演算でのオーバーフロー演算があれば、コンパイル時にエラーメッセージを出力します。

例

```
void main()
{
  int ia;
  int ib;
  float fa;
  float fb;

  ib=32767;
  fb=3.4e+38f;

  /* 定数または定数どうしの演算時はオーバーフローに対する                    */
  /* コンパイルエラーメッセージを出力します                                  */

  ia=99999999999;          /* (W) 定数のオーバーフローを検出します          */
  fa=3.5e+40f;             /* (E) 浮動小数点演算のオーバーフローを検出します */

  /* 実行時のオーバーフローに対するエラーメッセージは出力しません           */

  ib=ib+32767;             /* 演算結果のオーバーフローを無視します          */
  fb=fb+3.4e+38f;          /* 浮動小数点演算結果のオーバーフローを無視します */
}
```

(5) const 型変数への書き込み

const 型の変数を宣言していても、型変換で const 型でない型に変換して代入した場合や、分割コンパイルしたプログラムの間で、型を統一して扱っていない場合は、const 型変数への書き込みをコンパイラでチェックできませんので、注意が必要です。

例

```
const char *p;                  /* ライブラリ関数 strcat の第１引数は char 型への     */
  :                             /* ポインタ型なので、引数の指す領域が書き換わる       */
strcat(p, "abc");               /* ことがあります                                     */

  ファイル１
const int i;

  ファイル２
extern int i;                   /* 変数 i は、ファイル２では const 型で宣言していま   */
  :                             /* せんのでファイル２の中で書き込んでもエラーに       */
i=10;                           /* なりません                                         */
```

(6) 数学関数ライブラリの精度について

acos(x)、asin(x) 関数では x　１で誤差が大きくなりますので注意が必要です。

誤差範囲は以下のとおりです。

acos(1.0 - ε) における絶対誤差	倍精度 2^{-39} (ε = 2^{-33})
	単精度 2^{-21} (ε = 2^{-19})
asin(1.0 - ε) における絶対誤差	倍精度 2^{-39} (ε = 2^{-28})
	単精度 2^{-21} (ε = 2^{-16})

(7) 最適化により削除される可能性のあるコーディング

連続した同一変数の参照や、結果を使用しない式を記述した場合、コンパイラの最適化により冗長コードとして削除される場合があります。常にアクセスを保証する場合は、宣言時に volatile を指定してください。

例

```
[1] b=a;                /* １行目の式は冗長コードとして削除されることがあります     */
    b=a;
[2] while(1)a;          /* 変数 a の参照およびループ文は冗長コードとして削除される   */
                        /* ことがあります                                           */
```

(8) C89 と C99 の動作の差異

C99 では、選択文および反復文は、{} で囲まれます。そのため、C89 と C99 で動作が異なることがあります。

例

```
enum {a,b};
int g(void)
{
     if(!sizeof(enum{b,a}))
     return a;
     return b;
}
```

上記を -lang=c99 を指定してコンパイルすると、以下の解釈となります。

```
enum {a,b};
int g(void)
{
     {
         if(!sizeof(enum{b,a}))
         return a;
     }
     return b;
}
```

-lang=c では、g()=0 となりますが、-lang=c99 では、g()=1 となります。

(9) オーバーフローを伴う演算および型変換に関する注意事項

数値演算や型変換を行う場合、その型で取り扱える値の範囲外 (オーバーフロー) とならないようにご注意ください。オーバーフローが発生すると、得られる結果がコンパイルオプションなどの条件によって変化する場合があります。

標準の C 言語では、オーバーフローを伴う演算処理の結果は未定義となっており、コンパイル条件などにより得られる結果が異なる場合があります。

演算処理を行うプログラムでは、オーバーフローを発生させないようにご注意ください。

実際に結果が異なる例を次に示します。

例 float 型→ unsigned short 型への変換

```
float f = 2147483648.0f;
unsigned short ui2;
void ex1func(void)
{
       ui2 = f; /* float → unsigned short への変換 */
}
```

ex1func 関数を実行して得られる ui2 の値は、FPU あり (-fpu) と FPU なし (-nofpu) とで次のように異なります。

FPU あり : ui2 = 65535

FPU なし : ui2 = 0

これは、float 型から unsigned short 型への変換の方法が FPU ありと FPU なしで異なるためです。

(10) アンダーバー (_) を 2 つ並べたシンボルの記述

アンダーバーを 2 つ以上並べたシンボルは記述しないようにしてください。

生成されるコードは問題なく動作しますが、リンクマップ表示などでは関数名がそのまま表示されず、異なる名前の C++ の関数名として表示される場合があります。

例

```
int sample__Fc(void) { return 0; }
```

この場合、リンクマップ出力には、_sample__Fc ではなく、sample(char) と表示されます。

9.2 C プログラムを C++ コンパイラでコンパイルするときの注意事項

(1) 関数原型

関数を使用する前に関数原型が必要です。そのときには、仮引数の型も必ず宣言してください。

```
extern void func1();
void g()
{
    func1(1); // エラー
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

(2) const オブジェクトのリンケージ

const オブジェクトのリンケージは、C プログラムでは外部結合であるのに対し、C++ プログラムでは内部結合になります。また、const オブジェクトは初期値を必要とします。

```
const cvalue1;      // エラー

const cvalue2 = 1; // 内部結合
```

```
const cvalue1=0;
   // 初期値を与えます

extern const cvalue2 = 1;
   // Cプログラムと同様に外部結合に
   // なります
```

(3) void* からの代入

C++ プログラムでは、明示的なキャストを用いないと他のオブジェクト型へのポインタ (関数へのポインタ、メンバへのポインタを除く) 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
}
```

9.3 オプションに関する注意事項

(1) 指定の統一が必要なオプションについて

指定の統一が必要なオプションを以下 (a) (b) に示します。これらのオプション指定の異なるリロケータブルファイルおよびライブラリファイルをリンクした場合、実行時の動作は保証しません。

[1] cpu、endian、base、および fint_register の 4 つのオプションは、コンパイラ、アセンブラ、およびライブラリジェネレータで統一してください。

[2] 「2.5 マイコンオプション」に該当する (a) 以外のオプションについては、コンパイラおよびライブラリジェネレータで統一してください。

9.4 旧バージョン・旧リビジョンとの互換性

バージョンもしくはリビジョン変更に伴う互換性に関する影響について説明します。

9.4.1 V.1.00 との互換性

(1) 組み込み関数の仕様変更について

アドレスを表す引数や戻り値を持つ組み込み関数については、その型を従来の unsigned long から void * に変更しました。変更になった関数を表 9.1 に示します。

表 9―1 型が変更された組み込み関数の一覧

	項目	仕様	機能	変更内容	
				箇所	内容
1	ユーザスタックポインタ (USP)	void set_usp(void *data)	USP の設定	引数	unsigned long → void *
2		void *get_usp(void)	USP の参照	戻り値	unsigned long → void *
3	割り込みスタックポインタ (ISP)	void set_isp(void *data)	ISP の設定	引数	unsigned long → void *
4		void *get_isp(void)	ISP の参照	戻り値	unsigned long → void *
5	割り込みテーブルレジスタ (INTB)	void set_intb (void *data)	INTB の設定	引数	unsigned long → void *
6		void *get_intb(void)	INTB の参照	戻り値	unsigned long → void *
7	バックアップ PC(BPC)	void set_bpc(void *data)	BPC の設定	引数	unsigned long → void *
8		void *get_bpc(void)	BPC の参照	戻り値	unsigned long → void *
9	高速割り込みベクタレジスタ (FINTV)	void set_fintv(void *data)	FINTV の設定	引数	unsigned long → void *
10		void *get_fintv(void)	FINTV の参照	戻り値	unsigned long → void *

この変更により、V.1.00 でこれらの関数を利用されていたプログラムでは、V.1.01 では型が合わない等の警告やエラーになる場合があります。この場合は、キャストを追加または削除して型を合わせてください。

例として、V.1.00 で標準的に使用されていたスタートアッププログラム例を示します。この例は V.1.01 では W0520167 の警告メッセージが表示されますが、キャストをはずし型を合わせることで警告を回避できます。

例

set_intb 関数の利用例

```
#include <machine.h>
#pragma entry Reset_Program
void PowerON_Reset_PC(void)
{
        ...
        set_intb((unsigned long)__sectop("C$VECT")); // 警告 W0520167 になる
        ...
}
```

V.1.01 用の記述に変更した例

```
#include <machine.h>
#pragma entry Reset_Program
void PowerON_Reset_PC(void)
{
       ...
       set_intb(__sectop("C$VECT")); // キャスト (unsignd long) を削除
       ...
}
```

(2) L セクション追加について（section オプション、Start オプション）

V.1.01 では、文字列リテラルなどのリテラル領域を収録する L セクションを導入しました。

セクションが増えたことで、リンク時に L セクションが末尾に並ぶため、最適化リンケージエディタからアドレスエラー F0563100 が発生する場合があります。

これを回避するためには、次のいずれかの方法を採ってください。

(a) リンク時の最適化リンケージエディタの Start オプションに指定するセクション列に L を追加する

例

V.1.00 での指定例

```
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/01000,PResetPRG/0FFFF8000,C_1,C_2,C,C$*,D*,P,PIntPRG,
W*/0FFFF8100,FIXEDVECT/0FFFFFFD0
```

変更例 (C の後に L を追加する)

```
-start=B_1,R_1,B_2,R_2,B,R,SU,SI/01000,PResetPRG/0FFFF8000,C_1,C_2,C,L,C$*,D*,P,
PIntPRG,W*/0FFFF8100,FIXEDVECT/0FFFFFFD0
```

(b) コンパイル時に -section=L=C を選択する

コンパイル時に -section=L=C を指定することで、リテラル領域の出力先が C セクションに変更され、V.1.00 互換のセクション構成にすることができます。

なお、上記のリンク時の Start オプションを変更する方法に比べ、コード効率に影響が出る場合があります。

9. 4. 2 V.1.01、V.1.02 との互換性

(1) コンパイラで -merge_files オプションを指定した場合のリンカの制限事項

コンパイラで -merge_files オプションを指定して生成したオブジェクト・モジュール・ファイルをリンクする場合、リンカオプション -delete、-rename または -replace を同時に指定した場合の動作は保証しません。

(2) optimize=0 指定時の if 文に対する生成コードに関する注意事項

本バージョンでは、コンパイル時に、optimize=0 の指定の有無にかかわらず、定数だけの式からなる条件式を持つ if 文は、条件判定および実行されることのない側の文を、コード生成時に削除します。

以下に例を示します。

コメントに [削除] と書かれている行が、コード生成時に削除されます。

例 1. 定数のみの式の場合

```
int a,b,c;
void func01(void)
{
    if (1+2) {   /* [ 削除 ] */
        /* 実行される */
        a = b;
    } else {
        /* 実行されることがない */
        a++;     /* [ 削除 ] */
        b = c;   /* [ 削除 ] */
    }
}
```

2. シンボルアドレスを含む定数式も、定数式に含みます。

```
void f1(void),f2(void);
void func02(void)
{
    if (f1==0) {  /* [ 削除 ] */
        /* 実行されることがない */
        f2();     /* [ 削除 ] */
    } else {
        /* 実行される */
        f1();
    }
}
```

(3) -show=source の出力内容の変更

本バージョンでは、-show=source 指定時にアセンブリソースに出力される内容のうち、次の項目が変更されています。

- -debug オプションがない場合、.LINE は表示されません。
- #include の内容が展開されません。
- #line 指定に続くソース行は、命令との対応が正しくない場合があります。

付録 A　ウインドウ・リファレンス

ここでは，コーディングに関するウインドウ／パネル／ダイアログについて説明します。

A. 1　説　　明

以下に，コーディングに関するウインドウ／パネル／ダイアログの一覧を示します。

表 A―1　ウインドウ／パネル／ダイアログ一覧

ウインドウ／パネル／ダイアログ名	機能概要
エディタ パネル	ファイルの表示／編集
ファイル・エンコードの選択 ダイアログ	ファイル・エンコードの選択
ブックマーク ダイアログ	ブックマークの表示／削除
指定行へのジャンプ ダイアログ	指定した行にキャレットを移動
Print Preview ウインドウ	印刷する前のソース・ファイルのプレビュー
ファイルを開く ダイアログ	オープンするファイルの選択

エディタ パネル

ファイルの表示／編集を行います。

自動的にファイルのエンコードと改行コードを判別してオープンし，保存の際は元のフォーマットで保存します。ただし，ファイル・エンコードの選択 ダイアログによりエンコードを指定してオープンすることができます。また，ファイルの保存設定 ダイアログでエンコードと改行コードを指定した場合は，それに従って保存します。

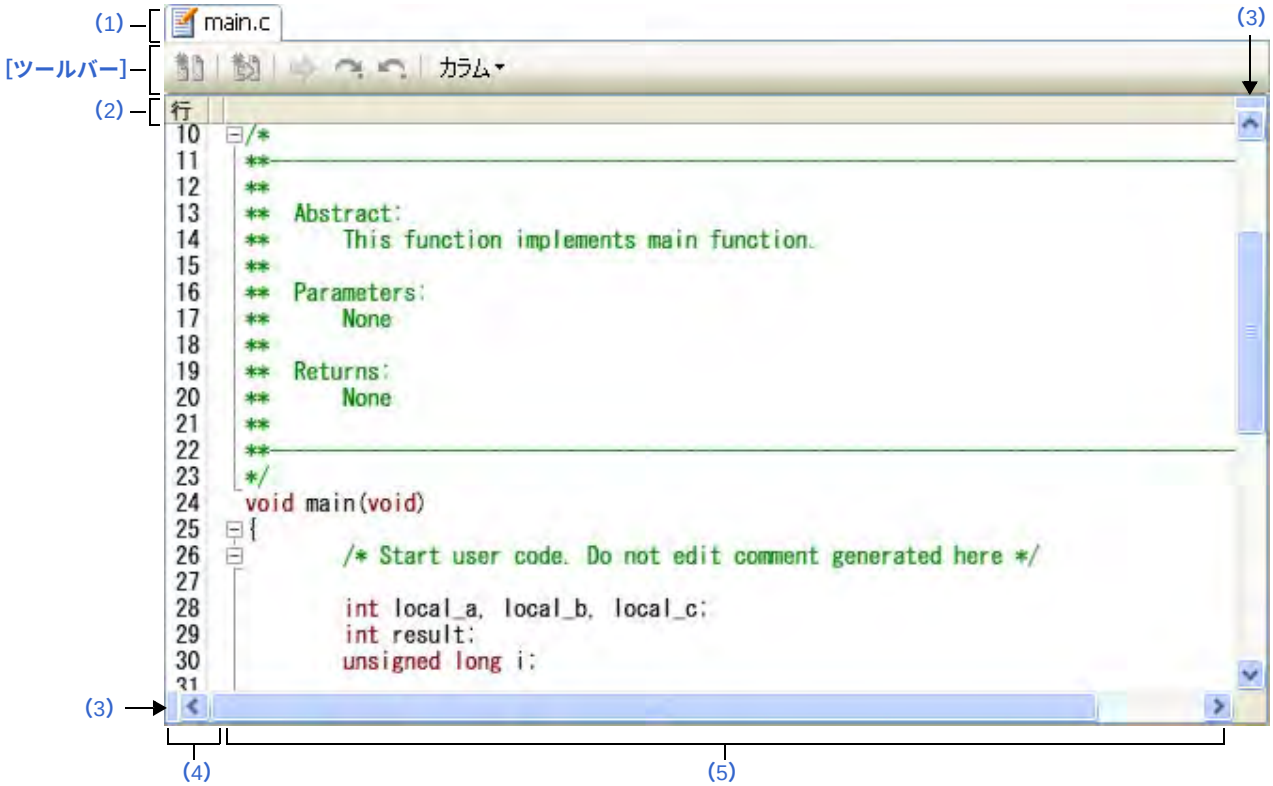
本パネルは複数オープンすることができます（最大個数：100 個）。

備考 1. プロジェクトをクローズすると，該当プロジェクト内で登録されているファイルをオープンしているすべてのエディタ パネルがクローズします。

2. プロジェクトからファイルの登録を外すと，該当ファイルをオープンしているエディタ パネルがクローズします。

3. ソース・ファイルをオープンする際，ダウンロードしているロード・モジュールの更新日時よりオープンするソース・ファイルの更新日時が新しい場合，メッセージを表示します。参照されているソースコードとデバッグ情報が一致していないことが原因です。

図 A―1　エディタ パネル



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［ツールバー］
- ［［ファイル］メニュー（エディタ パネル専用部分）］
- ［［編集］メニュー（エディタ パネル専用部分）］
- ［［ウインドウ］メニュー（エディタ パネル専用部分）］
- ［コンテキスト・メニュー］

［オープン方法］

- プロジェクト・ツリー パネルにおいて，ファイルをダブルクリック
- プロジェクト・ツリー パネルにおいて，ファイルを選択したのち，コンテキスト・メニューの［開く］を選択
- プロジェクト・ツリー パネルにおいて，コンテキスト・メニューの［追加］→［新しいファイルを追加］を選択したのち，テキスト・ファイル／ソース・ファイルを作成

［各エリアの説明］

(1) タイトルバー

オープンしているテキスト・ファイル／ソース・ファイルのファイル名を表示します。
なお，ファイル名の末尾に表示するマークの意味は次のとおりです。

マーク	意味
*	テキスト・ファイルをオープンしたのち，編集している場合に表示します。
(編集不可)	書き込み禁止状態のテキスト・ファイルをオープンしている場合に表示します。

(2) カラム・ヘッダ

エディタ パネルの各列のタイトルを表示します（マウス・カーソルを重ねることによりタイトル名をホップ・アップ表示します）。

表示	タイトル名	説明
行	行	行番号を表示します（「(4) 行番号エリア」参照）。
（表示なし）	選択	編集状況に応じた色表示を行います（「(4) 行番号エリア」参照）。

備考　カラム・ヘッダは，ツールバーの設定により，表示／非表示を切り替えることができます。

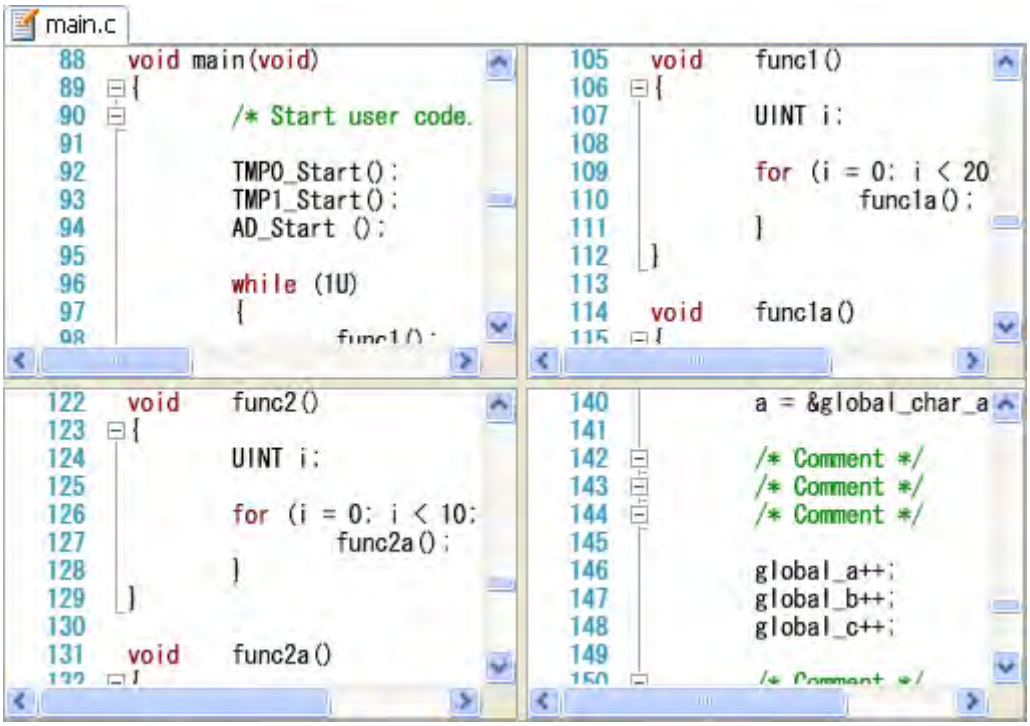
(3) 分割バー

縦と横の分割バーを使うことにより，エディタ パネルを分割して表示することができます。分割の上限は，縦 2 分割，横 2 分割までです。

- 分割表示するには，分割バーを下方向／右方向の目的の位置までドラッグします。または，分割バーをダブルクリックします。

- 分割表示を解除するには，分割バーをダブルクリックします。

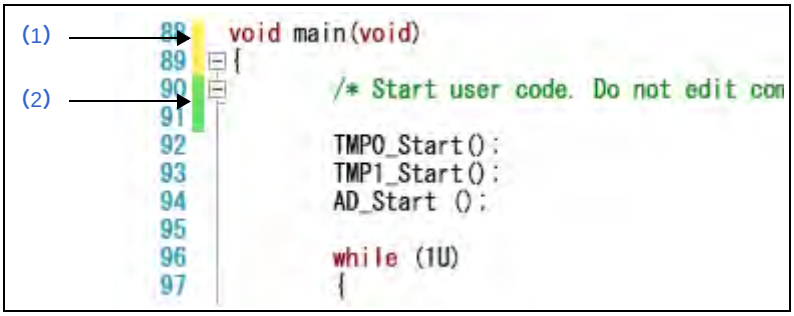
図 A―2　エディタ パネル（縦横 2 分割表示した場合）



(4) 行番号エリア

表示しているテキスト・ファイル／ソース・ファイルの行番号を表示します。

各行に付いた色が，その行の編集状態を示します。



(1)		新規または変更したが保存されていない。
(2)		新規または変更後，保存済み。 パネルをクローズしたのち，再度該当ソース・ファイルを表示するとこのマークは消失します。

(5) 文字列エリア

テキスト・ファイル／ソース・ファイルの文字列の表示／編集を行います。

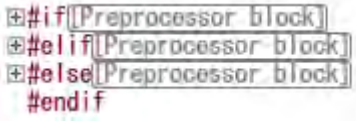
本エリアは，次の機能を備えています。

(a) コードのアウトライン表示

ソース・コード・ブロックの展開／折りたたみ表示を行い，現在編集中，またはデバッグ中のコード領域に集中して作業することができます。使用できるファイルの種類は，C ソース・ファイル／ C++ ソース・ファイルです。

展開／折りたたみを行うには，それぞれ文字列エリアの左にあるプラス／マイナス記号をクリックします。

展開／折りたたみ可能なソース・コード・ブロックの種類を次に示します。

左中かっこと右中かっこ（“{”と“}”）	
複数行のコメント（“/*”と“*/”）	
プリプロセッサ文（“if”, “elif”, “else”, “endif”）	

(b) 文字列の編集

キーボードより，IME などの日本語入力システムを使用した文字列を入力することができます。

また，編集機能を充実させるための様々なショートカットキーを使用することができます。

(c) タグ・ジャンプ

現在キャレットのある行にファイル名／行／桁の情報がある場合，［表示］メニュー→［タグ・ジャンプ］，またはコンテキスト・メニューの［タグ・ジャンプ］を選択することにより，該当ファイルを新たなエディタ パネルにオープンし，該当行／該当桁へジャンプすることができます（該当ファイルがすでにオープンしている場合は，そのエディタ パネルにジャンプ）。

(d) ファイルの監視機能

ソース・ファイルを管理するために，次の監視機能を備えています。

- CubeSuite+ 以外によって，現在表示しているファイルの内容が変更（リネーム／削除を含む）された場合，ファイルを更新するか否かのメッセージを表示し，どちらかを選択することができます。
- ソース・ファイルを保存する際（［ファイル］メニュー→［*ファイル名*を保存］の選択），CubeSuite+ 以外によって，現在表示しているファイルの内容が変更されていた場合，ファイルを保存するか否かのメッセージを表示し，どちらかを選択することができます。

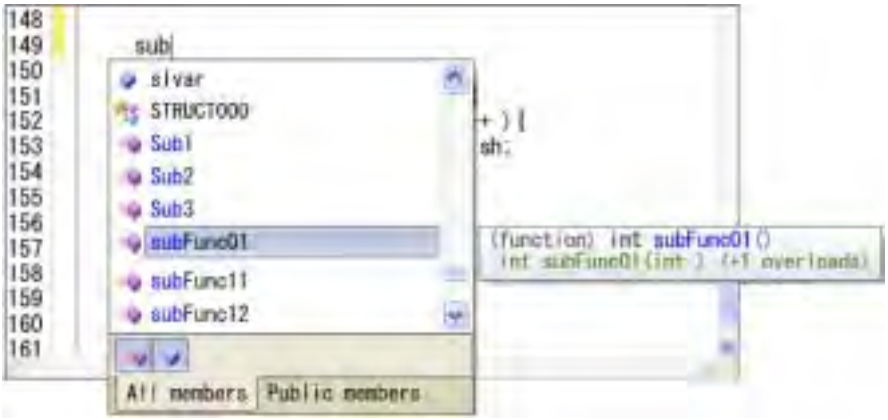
(e) スマート・エディット機能を使用する

スマート・エディット機能とは，コーディング中に関数情報や変数情報，および関数の引数情報などをリスト表示することにより，ユーザの入力を補完する機能です。

スマート・エディット機能では，次の情報の入力補完を行います。

- C 言語におけるグローバル関数
- C 言語におけるグローバル変数

図 A―3　スマート・エディット機能の表示例（関数／変数の候補表示）



なお，スマート・エディット機能を使用するためには，次の設定が必要となります。

- オプション ダイアログの［全般 - テキスト・エディタ］カテゴリ内の［スマート・エディット］チェック・ボックスを選択してくだい（デフォルト）。
- スマート・エディット機能は，ビルド・ツールが出力するクロス・リファレンス情報を使用して入力候補表示を行います。したがって，使用するビルド・ツールのプロパティ パネルにおいて，クロス・リファレンス情報を出力する設定[注]にしたのち，ビルドの実行を完了させてください。
  なお，ビルドの際にエラーが発生した場合，エラー発生前のクロス・リファレンスが存在する場合はそれを使用します。

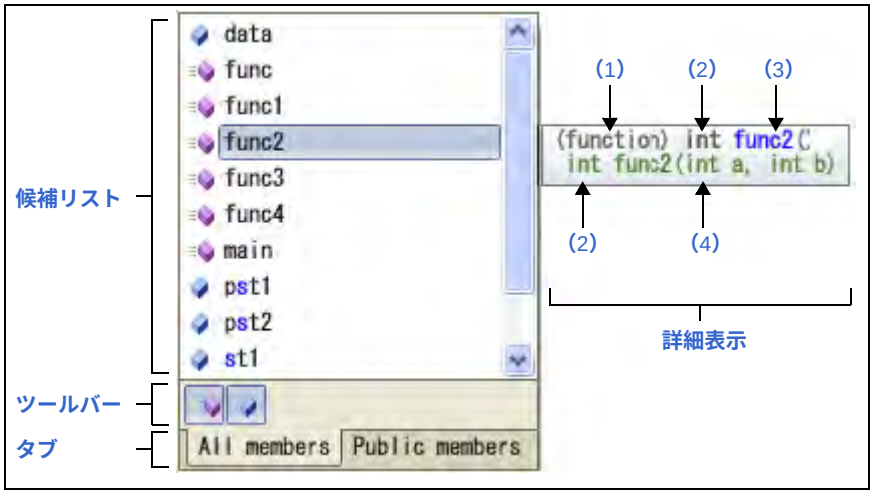
  注　［共通オプション］タブ→［出力ファイルの種類と場所］カテゴリ→［クロス・リファレンス情報を出力する］プロパティ→［はい (-Xcref)］
  この設定が無効な場合，クロスリファレンス情報がクリアされるため，スマート・エディット機能は使用不可となります。

- 関数／変数の候補表示
  - 表示方法
    関数／変数の候補は，次のいずれかの操作により表示されます。
    - C 言語において，“ . ”，または “ -> ” を入力した時点で左辺に対して該当するメンバがある場合
    - キーボードより，［Ctrl］ + ［Space］ キーを入力した場合（すべての候補を表示）
      ただし，候補が 1 つのみの場合は候補表示を行わず，該当文字列がこの時点で挿入されます。

- 文字列の挿入方法
  - 候補表示のリストより文字列を［↑］／［↓］キー，またはマウスにより選択したのち，[Enter]，または［TAB］キーを押下します。

- 各エリアの説明

図 A―4　関数／変数の候補表示



- 候補リスト

  関数／変数の候補をアルファベット順に表示します。

  キャレット位置の文字列と一致する文字列がある場合は，その文字列が強調表示されます（大文字／小文字不問）。

  なお，各候補の先頭には，次のアイコンを表示します。

アイコン	説明
	候補が typedef であることを示します。
	候補が関数であることを示します。
	候補が変数であることを示します。
	候補が構造体，または共用体であることを示します。

- ツールバー

  関数／変数の候補の表示／非表示を切り替えます。

ボタン	説明
	選択することにより，関数の候補を表示します。
	選択することにより，変数の候補を表示します。

- タブ

表示するメンバを切り替えます。

タブ名	説明
All members	すべての候補を表示します。
Public members	Public 属性の候補のみ表示します。

- 詳細表示

現在選択されている関数／変数の詳細情報を表示します。

項目	説明
(1) 種別	次の種別を表示します。 - (function)：関数であることを示します。 - (variable)：変数であることを示します。
(2) 型	関数／変数の型を表示します。
(3) 名称	関数／変数の名称を表示します。
(4) 名称と引数	関数／変数の名称を表示します。関数の場合は引数も表示します。

- 引数の候補表示

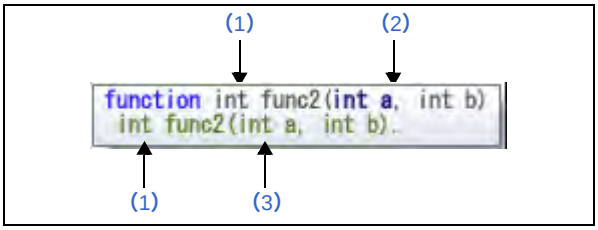
- 表示方法

引数の候補は，次のいずれかの操作により表示されます。

- 関数名において，“ (” を入力した時点で左辺に対して該当する関数の引数がある場合
- 関数の引数位置にキャレットがある状態で，キーボードより，[Ctrl] + [Shift] + [Space] キーを入力した場合

- 各エリアの説明

図 A―5　引数の候補表示



項目	説明
(1) 型	関数の型を表示します。
(2) 名称と引数	関数の名称と引数表示します。現在のキャレット位置の引数が強調表示されます。
(3) 名称と引数	関数の名称と引数表示します。

- 候補表示の消去

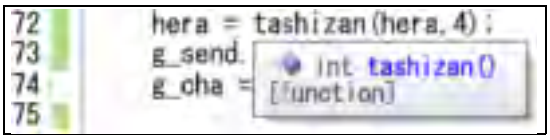
候補表示は，次のいずれかの操作により消去されます。

- ［ESC］キーの押下
- 英数字以外のキーの入力
  - 候補リストで何も選択していない場合：　何もしません。
  - 候補リストで選択している場合：　　選択している候補の文字列を挿入します。

- 候補表示の際の注意
  - 次の項目は候補表示に対応していません。
    - マクロ定義
    - ローカル変数
    - Typedef
  - 関数中に構造体宣言／共用体宣言を行った場合，宣言以降関数内で候補表示は行いません。
  - 変数のサイズに影響するコンパイル・オプションを設定した場合，表示される変数の型が実際の宣言と異なる場合があります。

備考　ソース・テキスト上の関数名／変数名にマウス・カーソルを重ねると，その関数／変数の情報をポップアップ表示します。
ただし，次の注意が必要です。
- 共用体の宣言内における共用体タグについてはポップアップ表示を行いません。
- const，static，および volatile 属性は表示されません。

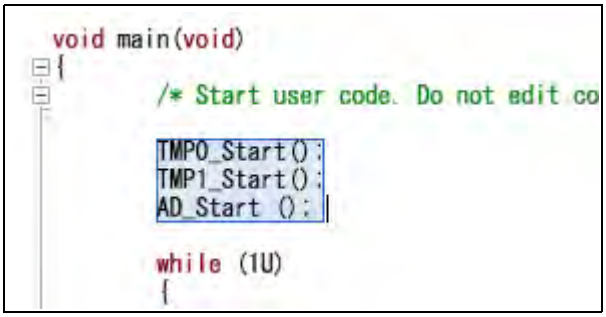
図 A―6　スマート・エディット機能によるポップアップ表示



(f) ブロックの選択

次の操作のいずれかより，複数行にわたるブロックを選択することができます。

- ［Alt］キーを押下しながら，左マウス・ボタンでドラッグ
- ［Alt］＋［Shift］キーを押下しながら，［→］／［←］／［↑］／［↓］ボタンを押下



備考　選択したブロックは，［編集］メニューの［切り取り］／［コピー］／［貼り付け］／［削除］による編集が可能です。

(g) 表示の拡大／縮小

［Ctrl］キーとマウス・ホイールを組み合わせることにより，エディタ パネルの表示を拡大／縮小することができます。

- ［Ctrl］キーを押下しながらマウス・ホイールを前方に動かすと，エディタ パネルの表示を拡大し，表示が大きく見やすくなります（最大 300%）。
- ［Ctrl］キーを押下しながらマウス・ホイールを後方に動かすと，エディタ パネルの表示を縮小し，表示が小さくなります（最小 50%）。

備考　オプション ダイアログの設定により，次の項目をカスタマイズすることができます。

- 表示フォント
- タブ幅
- 空白記号の表示／非表示
- 予約語／コメントの色分け

[ツールバー]

	このパネルの表示モードとして，通常表示モード（デフォルト）と混合表示モードを切り替えます。 ただし，デバッグ・ツールと接続中で，かつダウンロードしたソース・ファイルを表示している場合のみ有効となります。
	ステップ実行を行う際の単位として，ソース・レベル（デフォルト）と命令レベルを切り替えます。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合のみ有効となります。
	現在の PC 位置を表示します。 ただし，デバッグ・ツールと接続中の場合のみ有効となります。
	［コンテキスト・メニュー］→［ジャンプ前の位置へ戻る］を実行する前の位置へ進みます。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合は無効となります。
	［コンテキスト・メニュー］→［関数へジャンプ］を実行する前の位置へ戻ります。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合は無効となります。
カラム	このパネルで表示するカラム，またはマークの表示／非表示を切り替える次の項目を表示します。 チェックを外すことにより非表示となります（デフォルトではすべての項目がチェックされています）。
行	行番号エリアにおいて，行番号を表示します。
選択	行番号エリアにおいて，行の編集状態を示すマークを表示します。

[[ファイル] メニュー（エディタ パネル専用部分)]

エディタ パネル専用の［ファイル］メニューは次のとおりです（その他の項目は共通)。

*ファイル名*を閉じる	現在編集しているエディタ パネルをクローズします。 なお，パネルの内容を保存していない場合は，確認メッセージを表示します。
*ファイル名*を保存	現在編集しているエディタ パネルの内容を上書き保存します。 なお，ファイルを一度も保存していない，またはファイルが書き込み禁止の場合は，［名前を付けて *ファイル名*を保存 ...］の選択と同等の動作となります。
名前を付けて *ファイル名*を保存 ...	現在編集しているエディタ パネルの内容を新規保存するために，名前を付けて保存 ダイアログをオープンします。
*ファイル名*の保存設定 ...	現在編集しているエディタ パネルでオープンしているファイルのエンコードと改行コードを変更するために，ファイルの保存設定 ダイアログをオープンします。
印刷 ...	現在編集しているエディタ パネルの内容を印刷するために，Windows の印刷用 ダイアログをオープンします。
印刷プレビュー	印刷するファイル内容のプレビューを行うために，Print Preview ウインドウをオープンします。

[[編集] メニュー（エディタ パネル専用部分)]

エディタ パネル専用の［編集］メニューは次のとおりです（その他の項目はすべて無効)。

元に戻す	前回行った操作をキャンセルし，文字とキャレット位置を元に戻します（最大 100 回まで)。
やり直し	前回行った［元に戻す］の操作をキャンセルし，文字とキャレット位置を元に戻します。
切り取り	選択範囲の文字列を切り取り，クリップ・ボードにコピーします。 何も選択されていない場合は，その行を切り取ります。
コピー	選択範囲の文字列をクリップ・ボードにコピーします。 何も選択されていない場合は，その行をコピーします。
貼り付け	クリップ・ボードにコピーしている文字列を キャレット位置に，挿入モードの場合は挿入し，上書きモードの場合は上書きします。 ただし，クリップ・ボードの内容を文字列として認識できない場合は無効となります。 また，現在のソース・ファイルのモードはステータスバーに表示されます。
削除	キャレット位置の文字を 1 文字削除します。 ただし，範囲選択している場合は，選択しているの文字列を削除します。
すべて選択	現在編集中のテキストの先頭から最終までを選択状態にします。
検索 ...	検索・置換 ダイアログを ［クイック検索］ タブが選択状態でオープンします。
置換 ...	検索・置換 ダイアログを ［クイック置換］ タブが選択状態でオープンします。
移動 ...	指定した行へキャレットを移動するため，指定行へのジャンプ ダイアログ をオープンします。
コードのアウトライン	ソース・ファイルのコードの展開／折りたたみ表示を行うためのカスケード・メニューを表示します（「(a) コードのアウトライン表示」参照)。
定義を折りたたむ	関数定義など，実装ブロックとして登録されているすべてのノードを折りたたみます。

アウトラインを切り替える	折りたたまれた部分で，カーソルが置かれている最も内側のアウトライン部分の現在の状態を切り替えます。
すべてのアウトラインを切り替える	すべてのノードの状態を切り替え，すべて同じ状態（展開または折りたたみ）に設定します。折りたたまれているノードと展開されたノードが混在している場合，すべてを展開します。
アウトラインを中止する	コードのアウトラインを中止します。現在のソース・ファイルからすべてのアウトライン情報を削除します。
自動アウトラインを開始する	コードの自動アウトラインを開始します。サポートしているソース・ファイルのアウトライン情報を自動的に表示します。
高度な設定	エディタ パネルに関する高度な操作を行うためのカスケード・メニューを表示します。
行インデントを増やす	現在カーソルのある行のインデントをタブ 1 個分増やします。
行インデントを減らす	現在カーソルのある行のインデントをタブ 1 個分減らします。
行コメントを削除する	現在カーソルのある行の先頭から，言語（C++ など）に応じた行コメントの区切り記号の最初のセットを削除します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
行コメントを付ける	現在カーソルのある行の先頭に，言語（C++ など）に応じた行コメントの区切り記号を設定します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
タブをスペースに変換する	現在カーソルのある行のすべてのタブをスペースに変換します。
スペースをタブに変換する	現在カーソルのある行の連続したスペースの一組をタブに変換します。ただし，そのスペースの各組がタブ 1 個以上の幅に等しい場合に限ります。
選択行をタブ化する	現在の行をタブ化します。行の先頭にある（テキストの前の）すべてのスペースを可能な限りタブに変換します。
選択行を非タブ化する	現在の行を非タブ化します。行の先頭にある（テキストの前の）すべてのタブをスペースに変換します。
大文字にする	選択しているすべての文字を大文字に変換します。
小文字にする	選択しているすべての文字を小文字に変換します。
大文字／小文字を切り替える	選択しているすべての文字を，大文字または小文字に切り替えます。
先頭を大文字にする	選択しているすべての単語の先頭文字を大文字に変換します。
前後の空白を削除する	カーソル位置の前後にある余分な空白を削除し，空白文字を 1 個だけ残します。カーソルが単語内にある場合，または前後に空白文字がない場合，何も行いません。
末尾の空白を削除する	カーソルのある行で，最後の非空白文字の後にある空白を削除します。
行を削除する	現在カーソルのある行を完全に削除します。
行をコピーする	現在カーソルのある行をコピーして，その直後に挿入します。
空白行を削除する	カーソルのある行が空である場合，または空白文字しかない場合，その行を削除します。

[[ウインドウ] メニュー（エディタ パネル専用部分)]

エディタ パネル専用の［ウインドウ］メニューは次のとおりです（その他の項目は共通)。

分割	アクティブのエディタ パネルを水平方向に分割します。 分割の対象は，アクティブのエディタ パネルのみで，他のパネルは分割されません。分割の上限は，2 分割までです。
分割の解除	エディタ パネルの分割表示を解除します。

[コンテキスト・メニュー]

【タイトルバー・エリア】

閉じる	選択しているパネルを閉じます。
このタブ以外すべて閉じる	選択しているパネルと同じパネル表示エリアに表示されているパネルを，選択しているパネルのみ残し，すべて閉じます。
ファイル名 の保存	ファイルの内容を保存します。
完全パスのコピー	ファイルの絶対パスをクリップ・ボードにコピーします。
含んでいるフォルダを開く	テキスト・ファイルが保存されているフォルダをエクスプローラで開きます。
新しい水平タブグループ	アクティブなパネルの表示領域を水平方向に均等に 2 分割して，新たなタブ・グループを表示します。新たなタブ・グループには，アクティブなパネルが 1 つだけ入ります。 分割の上限は，4 分割までです。 以下の場合は，この項目は表示されません。 - タブ・グループにパネルが 1 つしか開いていない - 垂直方向にタブ・グループが分割されている - タブ・グループが 4 分割されている
新しい垂直タブグループ	アクティブなパネルの表示領域を垂直方向に均等に 2 分割して，新たなタブ・グループを表示します。新たなタブ・グループには，アクティブなパネルが 1 つだけ入ります。 分割の上限は，4 分割までです。 以下の場合は，この項目は表示されません。 - タブ・グループにパネルが 1 つしか開いていない - 水平方向にタブ・グループが分割されている - タブ・グループが 4 分割されている
次のタブグループへ移動	表示領域を水平方向に分割している場合，選択しているパネルを表示しているタブ・グループの下側のタブ・グループに移動します。 表示領域を垂直方向に分割している場合，選択しているパネルを表示しているタブ・グループの右側のタブ・グループに移動します。 移動する側にタブ・グループがない場合は，この項目は表示されません。
前のタブグループへ移動	表示領域を水平方向に分割している場合，選択しているパネルを表示しているタブ・グループの上側のタブ・グループに移動します。 表示領域を垂直方向に分割している場合，選択しているパネルを表示しているタブ・グループの左側のタブ・グループに移動します。 移動する側にタブ・グループがない場合は，この項目は表示されません。

【文字列エリア】

切り取り	選択範囲の文字列を切り取り，クリップ・ボードにコピーします。 何も選択されていない場合は，その行を切り取ります。
コピー	選択範囲の文字列をクリップ・ボードにコピーします。 何も選択されていない場合は，その行をコピーします。
貼り付け	クリップ・ボードにコピーされている文字列を キャレット位置に，挿入モードの場合は挿入し，上書きモードの場合は上書きします。 ただし，クリップ・ボードの内容を文字列として認識できない場合は無効となります。 また，現在のソース・ファイルのモードはステータスバーに表示されます。
検索 ...	検索・置換 ダイアログを［クイック検索］タブが選択状態でオープンします。
移動 ...	指定した行へキャレットを移動するため，指定行へのジャンプ ダイアログをオープンします。
ジャンプ先の位置へ進む	［ジャンプ前の位置へ戻る］を実行する前の位置へ進みます。
ジャンプ前の位置へ戻る	［関数へジャンプ］を実行する前の位置へ戻ります。
関数へジャンプ	選択している文字列，またはキャレット位置の単語を関数と判断し，該当する関数へジャンプします。
タグ・ジャンプ	キャレットのある行にファイル名／行／桁の情報がある場合，該当するファイルの該当行／該当桁へジャンプします（「(c) タグ・ジャンプ」参照）。
高度な設定	エディタ パネルに関する高度な操作を行うためのカスケード・メニューを表示します。
行インデントを増やす	現在カーソルのある行のインデントをタブ 1 個分増やします。
行インデントを減らす	現在カーソルのある行のインデントをタブ 1 個分減らします。
行コメントを削除する	現在カーソルのある行の先頭から，言語（C++ など）に応じた行コメントの区切り記号の最初のセットを削除します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
行コメントを付ける	現在カーソルのある行の先頭に，言語（C++ など）に応じた行コメントの区切り記号を設定します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
タブをスペースに変換する	現在カーソルのある行のすべてのタブをスペースに変換します。
スペースをタブに変換する	現在カーソルのある行の連続したスペースの一組をタブに変換します。ただし，そのスペースの各組がタブ 1 個以上の幅に等しい場合に限ります。
選択行をタブ化する	現在の行をタブ化します。行の先頭にある（テキストの前の）すべてのスペースを可能な限りタブに変換します。
選択行を非タブ化する	現在の行を非タブ化します。行の先頭にある（テキストの前の）すべてのタブをスペースに変換します。
大文字にする	選択しているすべての文字を大文字に変換します。
小文字にする	選択しているすべての文字を小文字に変換します。
大文字／小文字を切り替える	選択しているすべての文字を，大文字または小文字に切り替えます。
先頭を大文字にする	選択しているすべての単語の先頭文字を大文字に変換します。
前後の空白を削除する	カーソル位置の前後にある余分な空白を削除し，空白文字を 1 個だけ残します。カーソルが単語内にある場合，または前後に空白文字がない場合，何も行いません。
末尾の空白を削除する	カーソルのある行で，最後の非空白文字の後にある空白を削除します。

行を削除する	現在カーソルのある行を完全に削除します。
行をコピーする	現在カーソルのある行をコピーして，その直後に挿入します。
空白行を削除する	カーソルのある行が空である場合，または空白文字しかない場合，その行を削除します。

ファイル・エンコードの選択 ダイアログ

ファイル・エンコードの選択を行います。

備考　タイトルバーには，設定対象ファイルの名前を表示します。

図 A―7　ファイル・エンコードの選択 ダイアログ

ここでは，次の項目について説明します。

- [オープン方法]
- [各エリアの説明]
- [機能ボタン]

[オープン方法]

- ［ファイル］メニュー→［エンコードを指定して開く ...］を選択してファイルを開く ダイアログをオープン→ダイアログ上で［開く］ボタンをクリック

[各エリアの説明]

(1) [利用可能なエンコード :]

設定するエンコードをドロップダウン・リストにより選択します。

現在の OS が対応するコード・ページ／エンコード名を，アルファベット順で表示します。

ただし，同じエンコード名，および現在の OS が対応していないエンコード名は表示されません。

デフォルトでは，オプション ダイアログにおける［全般 - テキスト・エディタ］カテゴリのデフォルトのエンコードを選択します。

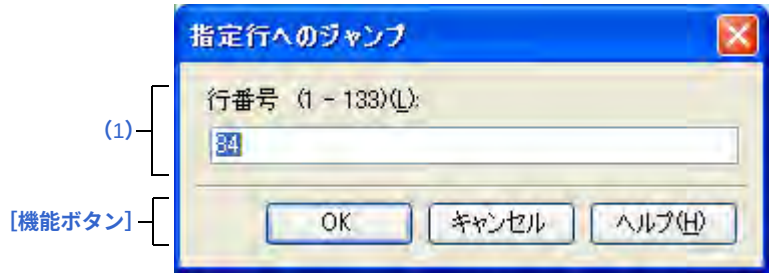
[機能ボタン]

ボタン	機能
OK	指定したファイル・エンコードを使用し，ファイルを開く ダイアログで選択したファイルをオープンします。
キャンセル	ファイルを開く ダイアログで選択したファイルをオープンせずに，このダイアログをクローズします。
ヘルプ	このダイアログのヘルプを表示します。

指定行へのジャンプ ダイアログ

指定したソース行にキャレットを移動します。

図 A―8　指定行へのジャンプ ダイアログ



ここでは，次の項目について説明します。

- [オープン方法]
- [各エリアの説明]
- [機能ボタン]

[オープン方法]

- ［編集］メニュー→［移動 ...］を選択
- エディタ パネルにおいて，コンテキスト・メニューの［移動 ...］を選択

[各エリアの説明]

(1)［行番号（*有効な行の範囲*）］

“（*有効な行の範囲*）”には，現在のファイルの有効な行の範囲が表示されます。

キャレットを移動したい行番号を 10 進数で直接入力により指定します。

デフォルトでは，エディタ パネル上の現在のキャレット位置の行番号が表示されます。

[機能ボタン]

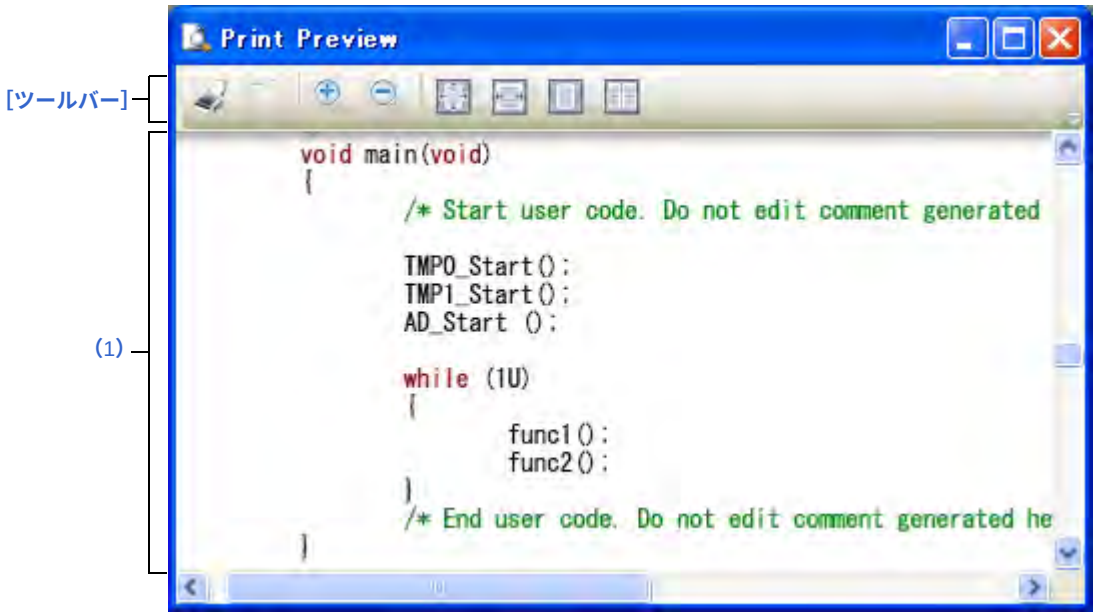
ボタン	機能
OK	指定したソース行の先頭にキャレットを移動します。
キャンセル	移動を無効とし，このダイアログをクローズします。
ヘルプ	このダイアログのヘルプを表示します。

Print Preview ウインドウ

印刷をする前に，現在エディタ パネルで表示されているファイルのプレビューを行います。

備考　［Ctrl］キーを押下しながらマウス・ホイールを前後方に動かすことにより，本ウインドウの表示を拡大／縮小することができます。

図 A―9　Print Preview ウインドウ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［ツールバー］
- ［コンテキスト・メニュー］

［オープン方法］

- エディタ パネルにフォーカスがある状態で，［ファイル］メニュー→［印刷プレビュー］を選択

［各エリアの説明］

(1) プレビュー エリア

印刷イメージをプレビュー表示します。

ヘッダ／フッダ部に，ファイル名（絶対パス）／ページ番号を表示します。

デバッグ・ツールと切断時，デバッグ・ツールと接続時（通常モード），またはデバッグ・ツールと接続時（混合表示モード）により表示するカラムが異なります。

ただし，エディタ パネルにおいて，非表示に設定しているカラムは表示されません（印刷されません）。

なお，アウトライン設定をしている場合，折りたたんだ状態を表示するマーク（「(a) コードのアウトライン表示」参照）とともに，たたまれている行の内容も表示します。

[ツールバー]

	印刷プレビュー表示しているアクティブなエディタ パネルの内容を印刷するために，Windows で用意されている，印刷 ダイアログをオープンします。
	選択範囲をクリップボードにコピーします。
	表示サイズを拡大します。
	表示サイズを縮小します。
	100% の倍率で表示します（デフォルト）。
	ページ幅で表示します。
	1 ページ全体を表示します。
	見開き 2 ページを表示します。

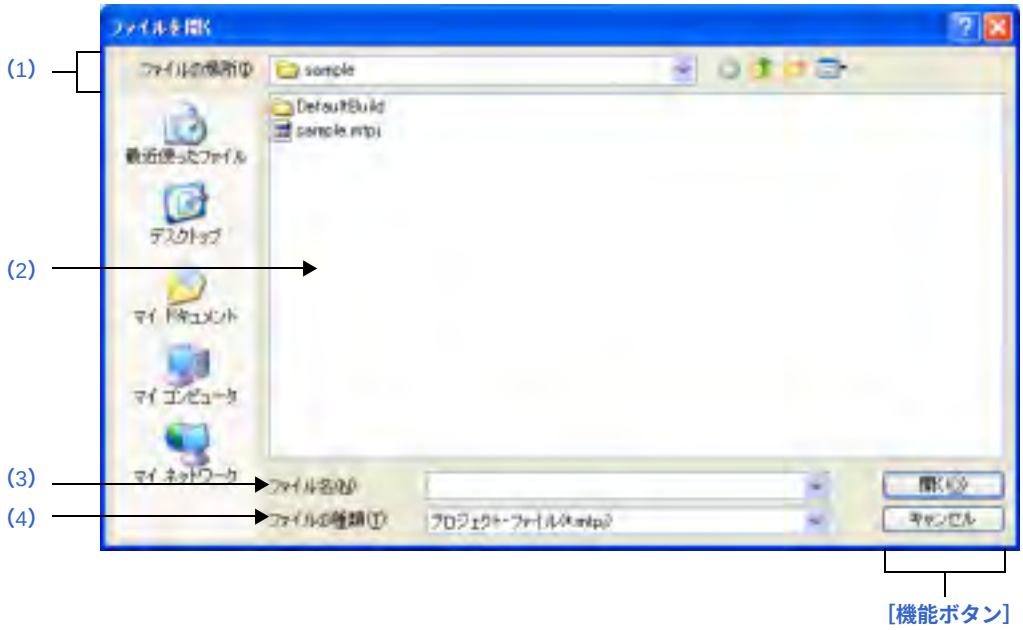
[コンテキスト・メニュー]

ズームの拡大	表示サイズを拡大します。
ズームの縮小	表示サイズを縮小します。

ファイルを開く ダイアログ

オープンするファイルの選択を行います。

図 A―10　ファイルを開く ダイアログ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［機能ボタン］

［オープン方法］

- ［ファイル］メニュー→［ファイルを開く ...］，または［エンコードを指定して開く ...］を選択

［各エリアの説明］

(1)［ファイルの場所］エリア

オープンするファイルが存在するフォルダを選択します。

初回は“ C: \ Documents and Settings \ユーザ名\ My Documents ”，2 回目以降は前回選択したフォルダが，デフォルトで選択されます。

(2) ファイルの一覧エリア

［ファイルの場所］，および［ファイルの種類］で選択した条件に合致するファイルの一覧を表示します。

(3) [ファイル名] エリア

オープンするファイルの名前を指定します。

(4) [ファイルの種類] エリア

オープンするファイルの種類（ファイル・タイプ）を選択します。

すべてのファイル (*.*)	すべての形式
プロジェクト・ファイル (*.mtpj)	プロジェクト・ファイル
CubeSuite 用プロジェクト・ファイル (*.cspj)	CubeSuite 用プロジェクト・ファイル
HEW 用ワークスペース・ファイル (*.hws)	HEW 用ワークスペース・ファイル
HEW 用プロジェクト・ファイル (*.hwp)	HEW 用プロジェクト・ファイル
PM+ 用ワークスペース・ファイル (*.prw)	PM+ 用ワークスペース・ファイル
PM+ 用プロジェクト・ファイル (*.prj)	PM+ 用プロジェクト・ファイル
C ソース・ファイル (*.c)	C ソース・ファイル
C++ ソース・ファイル (*.cpp; *.cc; *.cp)	C++ ソース・ファイル
ヘッダ・ファイル (*.h; *.hpp; *.inc)	ヘッダ・ファイル
アセンブル・ファイル（*.src）	アセンブラ・ソース・ファイル
マップ・ファイル (*.map; *.lbp)	マップ・ファイル
ヘキサ・ファイル (*.hex)	ヘキサ・ファイル
アセンブル・リスト・ファイル（*.lst）	アセンブル・リスト・ファイル
S レコード・ファイル（*.mot）	S レコード・ファイル
テキスト・ファイル (*.txt)	テキスト形式

[機能ボタン]

ボタン	機能
開く	- [ファイル] メニュー→[ファイルを開く ...] からオープンした場合 指定したファイルをオープンします。 - [ファイル] メニュー→[エンコードを指定して開く ...] からオープンした場合 ファイル・エンコードの選択 ダイアログをオープンします。
キャンセル	本ダイアログをクローズします。

付録B 索　引

【B】

【C】

【D】

【E】

【F】

【G】

【H】

【I】

【N】

【O】

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
Rev.1.00	2013.03.25	–	初版発行

CubeSuite+ V2.00.00　**ユーザーズマニュアル**
RX **コーディング編**

発行年月日　2013.03.25　　Rev.1.00

発行　　**ルネサス エレクトロニクス株式会社**
〒211-8668　**神奈川県川崎市中原区下沼部** 1753

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）　(03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/contact/

Colophon 1.1

CubeSuite+ V2.00.00

RENESAS

ルネサス エレクトロニクス株式会社

R20UT2470JJ0100