User's Manual

CubeSuite Ver.1.40

統合開発環境
ユーザーズマニュアル　コーディング編（CX コンパイラ）

対象デバイス
V850 マイクロコントローラ

本資料に記載の全ての情報は本資料発行時点のものであり、ルネサス エレクトロニクスは、
予告なしに、本資料に記載した製品または仕様を変更することがあります。
ルネサス エレクトロニクスのホームページなどにより公開される最新情報をご確認ください。

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2010.10

ご注意書き

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

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

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

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

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

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

このマニュアルの使い方

このマニュアルは，V850 マイクロコントローラ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，V850 マイクロコントローラの統合開発環境（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的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	：$\overline{\times\times\times}$（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　…　××××
	16 進数　…　0x××××

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	R20UT0256J	R20UT0256E
	解析編	R20UT0265J	R20UT0265E
	プログラミング編	R20UT0266J	R20UT0266E
	メッセージ編	R20UT0267J	R20UT0267E
	コーディング編（CX コンパイラ）	このマニュアル	R20UT0259E
	ビルド編（CX コンパイラ）	R20UT0261J	R20UT0261E
	78K0　コーディング編	R20UT0004J	R20UT0004E
	78K0　ビルド編	R20UT0005J	R20UT0005E
	78K0　デバッグ編	R20UT0262J	R20UT0262E
	78K0　設計編	R20UT0006J	R20UT0006E
	78K0R　コーディング編	U19382J	U19382E
	78K0R　ビルド編	U19385J	U19385E
	78K0R　デバッグ編	R20UT0263J	R20UT0263E
	78K0R　設計編	R20UT0007J	R20UT0007E
	V850　コーディング編	U19383J	U19383E
	V850　ビルド編	U19386J	U19386E
	V850　デバッグ編	R20UT0264J	R20UT0264E
	V850　設計編	R20UT0257J	R20UT0257E

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。

〔メ　モ〕

〔メ　モ〕

〔メ　モ〕

目　　次

第 1 章 概 説

この章では，V850 マイクロコントローラ用 C コンパイラ・パッケージ（CX）の全体概要について説明します。

1.1 概 要

V850 マイクロコントローラ用 C コンパイラ・パッケージ（CX）は，C 言語，またはアセンブリ言語で記述されたプログラムを機械語に変換するプログラムです。

1.2 特 長

V850 マイクロコントローラ用 C コンパイラ・パッケージ（CX）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 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

C 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 C 言語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

CX 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 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) コンパイラの最大値

コンパイラの最大値については，「(9) 翻訳限界」を参照してください。

(2) アセンブラの最大値

表 1―1 アセンブラの最大値

項目	最大値
シンボルの長さ（トークンの長さ）	4,294,967,294 注
ラベルの長さ（トークンの長さ）	4,294,967,294 注
シンボルの最大個数	4,294,967,294 注
LOCAL 疑似命令の仮パラメータ数	4,294,967,294 注
LOCAL 疑似命令の自動生成シンボル数	4,294,967,294 注
INCLUDE 疑似命令のネスト	4,294,967,294 注
TIDATA.BYTE，TIBSS.BYTE 再配置属性セクションのサイズの合計	128 バイト
TIDATA.WORD，TIBSS.WORD 再配置属性セクションのサイズの合計	256 バイト
ALIGN 疑似命令	2 以上 2e31 未満の偶数
IRP 疑似命令の実パラメータ	4,294,967,294 注
LOCAL 疑似命令の仮パラメータ	4,294,967,294 注

注 動作するホスト・マシンのメモリに依存します。

第 2 章 機 能

この章では，CX 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 短い命令長でアクセスできる領域へ配置する

V850 には，命令長が 2 バイトのロード／ストア命令があります。変数をこの命令でアクセスできるセクションに配置することによりコード・サイズを削減することができます。

変数の定義／参照時は，#pragma section を使用し，セクション種別に“tidata”を指定します。

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

【記述例】

```
#pragma section tidata
int a = 1;              /*tidata.word セクション配置 */
int b;                  /*tibss.word セクション配置 */
#pragma section default
```

備考 「#pragma section 指令」を参照してください。

2. 1. 2 配置領域を変更する

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：.sbss セクション
- 初期値あり変数：.sdata セクション
- const 変数：.const 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 でセクション種別を指定します。

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

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。

セクション種別	初期値あり・なし	デフォルト・セクション名	セクション名変更	ベースレジスタ	アクセス命令
data	あり	.data	可	gp	ld/st 2 命令
	なし	.bss	可	gp	ld/st 2 命令
sdata	あり	.sdata	可	gp	ld/st 1 命令
	なし	.sbss	可	gp	ld/st 1 命令
sedata	あり	.sedata	不可	ep	ld/st 1 命令
	なし	.sebss	不可	ep	ld/st 1 命令
sidata	あり	.sidata	不可	ep	ld/st 1 命令
	なし	.sibss	不可	ep	ld/st 1 命令
tidata_byte	あり	.tidata.byte	不可	ep	sld/sst 1 命令
	なし	.tibss.byte	不可	ep	sld/sst 1 命令
tidata_byte	あり	.tidata.word	不可	ep	sld/sst 1 命令
	なし	.tibss.word	不可	ep	sld/sst 1 命令
sconst	あり	.sconst	不可	r0	ld/st 1 命令
const	あり	.const	可	r0	ld/st 2 命令
default	これ以降，#pragma section 指定がなかったものとみなし，デフォルトの配置を行います。				

【記述例】

```
#pragma section sdata "mysdata"
int a = 1;              /*mysdata.sdata セクション配置 */
int b;                  /*mysdata.sbss セクション配置 */
#pragma section default
```

なお，#pragma section 命令を使った変数に別ファイル（参照ファイル）の関数から参照する場合には，参照ファイルにも #pragma section 命令を記述し，該当変数を extern 宣言する必要があります。

【記述例：変数を定義しているファイル】

```
#pragma section sconst
const unsigned char table_data[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
#pragma section default
```

【記述例：変数を参照するファイル】

```
#pragma section sconst
extern const unsigned char  table_data[];
#pragma section default
```

備考 「#pragma section 指令」を参照してください。

2. 1. 3 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の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外され，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。また，volatile 指定された変数のアクセス幅も変更されません。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。

【volatile 指定した場合のソースと出力コードの例】

“変数 a”，“変数 b”，および“変数 c”を volatile 指定した場合，これらの変数値を必ずメモリから読み込み，操作後にメモリへ書き込むコードが出力されます。たとえば，この間に割り込みが入り，割り込み内で変数値が変更されても，その変更が反映された結果を取得することができます（このような例の場合，割り込みのタイミングによっては，変数の操作区間内を割り込み禁止にするなどの処置が必要となります）。

volatile 指定をすると，メモリの読み込み／書き込み処理が入るため，volatile 指定しなかった場合よりもコード・サイズは大きくなります。

```
volatile int a;
volatile int b;
volatile int c;

void func(void) {
        if(a <= 0) {
            b++;
        } else {
            c++;
        }
        b++;
        c++;
}
```

```
_func:
.BB.LABEL.0:
        callt   0
        ld.w    $_a, r12
        cmp     r0, r12
        ble     .BB.LABEL.2
.BB.LABEL.1:
        ld.w    $_c, r12
        add     1, r12
        st.w    r12, $_c
        br      .BB.LABEL.3
.BB.LABEL.2:
        ld.w    $_b, r12
        add     1, r12
        st.w    r12, $_b
.BB.LABEL.3:
        ld.w    $_b, r12
        add     1, r12
        st.w    r12, $_b
        ld.w    $_c, r13
        add     1, r13
        st.w    r13, $_c
.BB.LABEL.4:
        callt   30
```

2.1.4 ユーザ・ポートを定義する

ユーザ・ポートについては，次の例のように，volatile 指定して最適化を避けてください。

【ポート記述の手順例】

```
/* 1. ポート・マクロ（型）の定義 */
#define DEFPORTB(addr)  (*((volatile unsigned char *)addr))   /* 8 ビット・ポート */
#define DEFPORTH(addr)  (*((volatile unsigned short *)addr))  /* 16 ビット・ポート */
#define DEFPORTW(addr)  (*((volatile unsigned int *)addr))    /* 32 ビット・ポート */

/* 2. ポートの定義（例：PORT1 0x00100000 8bit）*/
#define PORT1   DEFPORTB(0x00100000)    /* 0x00100000  8 ビット・ポート */

/* 3. ポートの使用 */
{
  PORT1 = 0xFF;   /* PORT1 への書き込み */
  a = PORT1;      /* PORT1 からの読み出し */
}

/* 4.C コンパイラの出力コードイメージ */
  :
mov     1048576, r10
st.b    r20, [r10]

mov     1048576, r11
ld.bu   [r11], r12
  :
```

備考 1. 構造体を宣言し，その構造体変数を特定のセクションに割り当て，リンク・ディレクティブで対応するポート・アドレスに割り当てることにより，CX の内蔵周辺 I/O レジスタと同様に“X.X”形式のビット・アクセスができます。
ただし，1 ビット /8 バイト・アクセスがある場合，ビット・フィールドとバイトの共用体にする必要があるので，“X.X.X”形式になります。

2. 変数のセクション割り当ては，#pragma section やシンボル情報ファイルで行ってください。

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。

なお，.const セクションを .sconst セクションに割り当てるときは #pragma section sconst 指定をしてください。

- 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（.sdata/.data）に配置されます。

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

- 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（.sconst/.const）に配置されます。

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

- 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（.sconst/.const）に配置されます。

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

2.2 関 数

この節では，関数について説明します。

2.2.1 配置領域を変更する

関数のセクション名を変更する場合は，以下のように #pragma text 指令を使用して関数を指定します。

```
#pragma text  [" セクション名 "]  [ 関数名 [, 関数名 ]…]
```

また，セクション名を変更した text 属性のセクションは，リンク・ディレクティブで入力セクションを作成したときのセクション名を指定してください。

例 C ソース内で「#pragma text "sec1" func1」と記述した場合のリンク・ディレクティブの記述方法（セグメント名：FUNC1）

```
FUNC1:  !LOAD ?RX {
            sec1.text = $PROGBITS ?AX sec1.text;
};
```

#pragma text 指令で，特定の関数を独自に指定した text 属性のセクションに配置する場合，実際に生成されるセクション名は“指定した文字列＋ .text”となり，このセクション名をリンク・ディレクティブに記述する必要があります。

上記の例であれば“sec1.text セクション”になります。

備考 「#pragma text 指令」を参照してください。

2. 2. 2 離れた関数をコールする

CX は，関数呼び出しに jarl 命令を使用します。

しかし，jarl 命令は 22 ビット・ディスプレースメントであるため，プログラム配置によってはアドレス解決ができず，リンク時にエラーとなります。

このような場合，CX の -Xfar_jump オプションで，関数呼び出しをディスプレースメント幅に依存しない関数呼び出しにすることができます。

far jump を指定した関数の呼び出しに対しては，jarl 命令ではなく，jarl32，および jr32 命令が出力されます。

-Xfar_jump オプションで指定するファイルには，1 行に 1 関数を記述していきます。記述する名前は，C 言語関数名の先頭に“_（アンダースコア）”を付けた名前になります。

【記述例】

```
_func_led
_func_beep
_func_motor
    :
_func_switch
```

“_関数名”のかわりに次のように記述すると，すべての関数を far jump 呼び出しの対象にします。

```
{all_function}
```

また，次のように記述すると，すべての割り込み関数呼び出しを far jump 呼び出しの対象にします。

```
{all_interrupt}
```

備考 「far jump 機能」を参照してください。

2. 2. 3 アセンブラ命令の埋め込み

CX では，次に示す形式において，C ソース・プログラム中にアセンブラ命令を記述できます。

- asm 宣言

```
__asm( 文字列定数 );
```

- #pragma 指令

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

挿入するアセンブラ命令でレジスタを使用する場合，必要な退避／復帰はプログラム内で行ってください。CX では行いません。

【記述例】

```
__asm("nop");
__asm(".str \"string\\0\"");

#pragma asm
        mov     r0, r10
        st.w    r10, $_i
#pragma endasm
```

なお，asm 宣言や #pragma asm ～ #pragma endasm 指令内に書かれたアセンブラ命令は，C 言語による #define 定義されたものがアセンブラ・ソース内にあっても展開されることはありません。

また，asm 宣言や #pragma asm ～ #pragma endasm 指令内のアセンブラ命令中のマクロ疑似命令などは，そのままアセンブラに渡されるため，CX で -P オプションをつけても展開されません。

備考 「アセンブラ命令の記述」を参照してください。

2. 2. 4 RAM で実行する

リンク時とコピー後で，各セクションと各シンボル（r0，TP，EP，GP）の相対値を壊さなければ，ROM 内に配置されているプログラムを RAM にコピーして，RAM にてプログラムを実行することができます。

コピーされるものと，されないものが存在するので，注意してください。

リセット後に，内蔵 RAM にコピーして，そのプログラムが変更されないのであれば，ROM 化の機能を使用して text セクションをパッキング対象にすることで簡単に実現できます。CX ではデフォルトで ROM 化処理を実行します。

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 言語で周辺 I/O レジスタへアクセスする

C 言語でデバイス内部の周辺 I/O レジスタを読み書きする場合，C ソースに #pragma 指令を追加することにより，周辺 I/O レジスタ名やビット名を用いて読み書きすることが可能となります。

周辺 I/O レジスタ名は，通常の符号なし（unsigned）外部変数のように扱うことができます。また，& 演算子で，周辺 I/O レジスタのアドレスを取得することも可能です。

```
#pragma ioreg
    レジスタ名 = …
    ビット名 = …
    … = & レジスタ名
```

上記 #pragma 指令を記述した以降，周辺 I/O レジスタ名が使用可能となります。

【記述例】

```
#pragma ioreg

void func(void) {
        int i;
        unsigned long adr;
        P0 = 1;         /* P0 に 1 を書き込む */
        i = RXB0;       /* RXB0 から読み込み */
        adr = &P1;      /* P1 のアドレスを取得する */
}
```

なお，周辺 I/O レジスタのビット名称は，該当ビット名称が CX によって定義されているものに限定されます。したがって，ビット名称が未定義の場合はエラーになります。

備考 「周辺 I/O レジスタへのアクセス」を参照してください。

2. 3. 2 C 言語での割り込み処理を記述する

CX では，割り込みハンドラの指定を“#pragma interrupt 指令”で行います。

以下に，割り込みハンドラの記述例を示します。

【記述例：ノンマスカブル割り込みの場合】

```
#pragma interrupt NMI func1   /* ノンマスカブル割り込み */

void func1(void) {
  :
}
```

【記述例：多重割り込みの場合】

```
#pragma interrupt INTP0 func2 multi     /* 多重割り込み */

void func2(void) {
  :
}
```

備考 「割り込み／例外処理ハンドラ」を参照してください。

2. 3. 3 C 言語での CPU 命令を使用する

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，CX で用意した関数の形式で記述します。

次に，関数として記述できる命令を示します。

アセンブラ命令	機能	組み込み関数
di	割り込み制御	__DI();
ei		__EI();
nop	ノー・オペレーション	__nop();
halt	プロセッサの停止	__halt();
satadd	飽和加算	long a, b; long __satadd(a, b);
satsub	飽和減算	long a, b; long __satsub(a, b);
bsh	ハーフワード・データのバイト・スワップ	long a; long __bsh(a);
bsw	ワード・データのバイト・スワップ	long a; long __bsw(a);
hsw	ワード・データのハーフワード・スワップ	long a; long __hsw(a);
sxb	バイト・データの符号拡張	char a; long __sxb(a);
sxh	ハーフワード・データの符号拡張	short a; long __sxh(a);
mul	mul 命令を用いて 32 ビット× 32 ビットの符号つき乗算結果の 64 ビットを変数に代入する命令	long a, b; long long __mul(a, b);
mulu	mulu 命令を用いて 32 ビット× 32 ビットの符号なし乗算結果の 64 ビットを変数に代入する命令	unsigned long a, b; Unsigned long long __mulu(a,b);
mul32	mul32 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令	long a, b; long __mul32(a, b);
mul32u	mulu32 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令	unsigned long a, b; unsigned long __mul32u(a, b);
sasf	論理左シフト付きフラグ条件の設定	long a; unsigned int b; long __sasf(a, b);
sch0l	MSB 側からのビット（0）検索【V850E2V3】	long a; long __sch0l(a);
sch0r	LSB 側からのビット（0）検索【V850E2V3】	long a; long __sch0r(a);
sch1l	MSB 側からのビット（1）検索【V850E2V3】	long a; long __sch1l(a);

アセンブラ命令	機能	組み込み関数
sch1r	LSB 側からのビット（1）検索【V850E2V3】	long a; long __sch1r(a);
ldsr	システム・レジスタへのロード【V850E2V3】	long a; void __ldsr(regID 注, a);
stsr	システム・レジスタの内容のストア【V850E2V3】	unsigned long __stsr(regID 注);
ldgr	汎用レジスタへのロード【V850E2V3】	long a; void __ldgr(regID 注, a);
stgr	汎用レジスタの内容のストア【V850E2V3】	unsigned long __stgr(regID 注);
caxi	比較と交換【V850E2V3】	long *a; long b, c; void __caxi(a, b, c);

注 regID にはシステム・レジスタ番号（0 ～ 31）を指定してください。
ただし，ldsr では regID として 0 を指定しないでください。

【記述例】

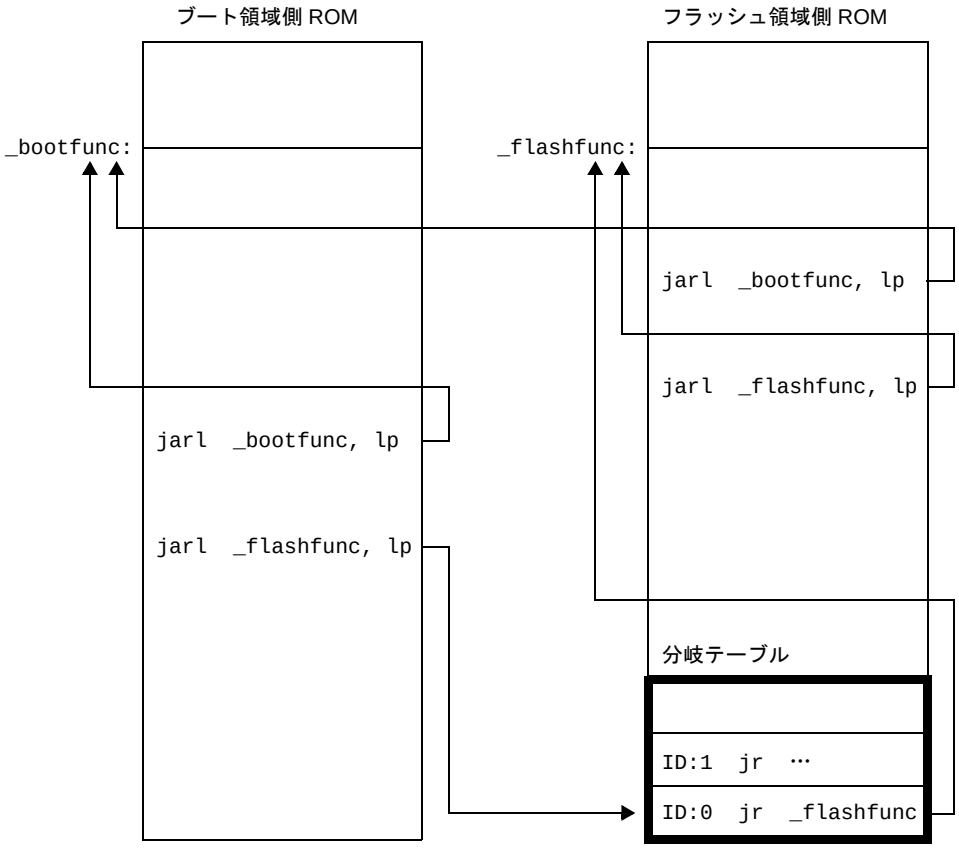
```
long a, b, c;

void func(void) {
        :
    c = __satsub(a, b);      /* a と b の飽和演算結果を c に格納する (c = a - b) */
        :
    __nop();
        :
}
```

2. 3. 4 セルフプログラミングのブート領域を作成する

フラッシュ領域 - ブート領域の変数／関数は，以下の操作により参照することができます。

- フラッシュ領域からは，ブート領域の関数を直接呼び出すことができます。
- ブート領域からフラッシュ領域への関数呼び出しは，分岐テーブルを介して行います。
- フラッシュ領域からは，ブート領域の外部変数を参照できます。
- ブート領域からは，フラッシュ領域の外部変数を参照できません。
- ブート領域のプログラムとフラッシュ領域のプログラムで，同じ外部変数，および外部関数を定義できます。
  この場合，定義と同じ領域側にある変数，または関数が参照されます。

図 2―1 フラッシュ領域／ブート領域のイメージ



ブート領域から呼び出すフラッシュ領域の関数を .ext_func 疑似命令で指定します。

```
.ext_func 関数名 , ID 番号
```

【記述例（C 言語プログラム中）】

```
#pragma asm
        .ext_func _func_flash0, 0
        .ext_func _func_flash1, 1
        .ext_func _func_flash2, 2
#pragma endasm
```

その他にオプション等で設定する必要があります。

備考 詳細は「CubeSuite ビルド編（CX コンパイラ）」の“ブートーフラッシュ再リンク機能”を参照してください。

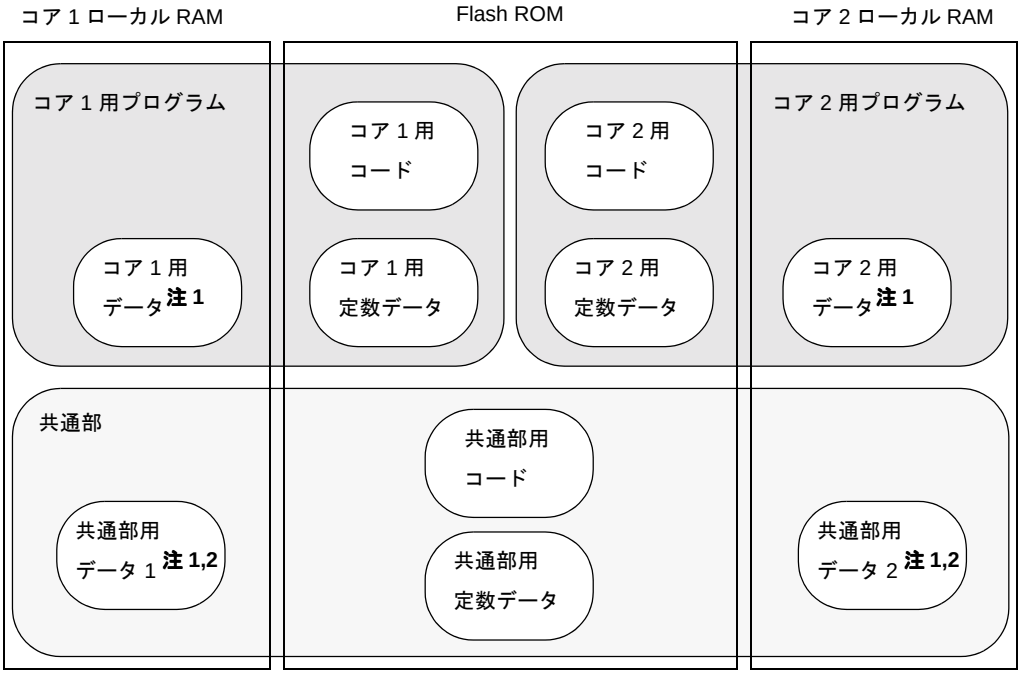
2. 3. 5 マルチコア用プログラムを作成する

ここでは，CX を用いてマルチコア用プログラムを作成する方法について説明します。以下では，ターゲット CPU が μPD70F3515（コアの数が 2 個）である場合を例に説明します。

(1) マルチコア用プログラム

CX が出力するマルチコア用プログラムは，複数のコア上で動作するプログラムを，1 個のロード・モジュール・ファイルに結合したものです。マルチコア用プログラムは，各コア用プログラム（コード／データ）と，複数のコアから共通で参照するコードおよびデータを含む共通部とで構成されます（以下で，各コア用プログラムおよび共通部の各々をサブプログラムとします）。次図にマルチコア用プログラムの構成例を示します。

図 2―2 マルチコア用プログラムの構成例



注 1. コア 1 用データ／コア 2 用データ／共通部データを外部 RAM に置くこともできます。

2. 共通部用データは，分けずにコア 1，またはコア 2 どちらか一方のローカル RAM に置くこともできます。

CX のマルチコア用プログラムには次のような特徴があります。

- 実行開始アドレス（0 番地）は共通ですが、その後でそれぞれのコア用のプログラムへ分岐します。
- 各コア用プログラムでは，シングルコア用プログラムと同様にすべての属性のセクションにデータを配置することが可能です。
- 共通部のデータ（const および sconst 属性のものを除く）はすべて data 属性セクションに配置します。また，共通部内ではデータおよびコードは gp/ep/tp 相対ではなく r0 相対の命令でアクセスします。
- 各サブプログラムから他のサブプログラム内で定義されたデータおよびコードに対しては，r0 相対命令でアクセスします。

- 各サブプログラムで定義したデータおよびコードは，自サブプログラム内のみならず，他のサブプログラムからアクセスすることも可能です。ただし，各コア用プログラムの独立性，データ・アクセスの安全性を考慮すると，各コア用データおよびコードは原則として定義されたサブプログラム内での使用を推奨します。複数のコアからアクセスされる可能性のあるデータについては，あるコアで参照中に他コアにより書き換えが起こらないように，プログラム側で注意する必要があります。
- コードおよびデータはソース・ファイル単位で各サブプログラムに割り当てられます（たとえば 1 つのソース・ファイル中にコア 1 用データとコア 2 用データを定義することはできません）。

(2) コーディングの注意点

マルチコア用プログラムを記述する場合，次の点に注意してください。

(a) C ソース・プログラム

マルチコア用プログラムを C 言語で記述する場合，次の点に注意してください。

- 異なるコア用のプログラムで同名の関数を定義することはできないため，メイン関数名として main を使用している場合は名前を変えてください（デフォルトのスタートアップ・ルーチンでは，コア 1 用のメイン関数名を main，コア 2 用のメイン関数名を main_pe2 と仮定しています）。
- 各コア用プログラムにおいて他のサブプログラムで定義した変数または関数を参照する場合は，その変数または関数の extern 宣言の前に #pragma nopic を記述してください（共通部ではデフォルトで #pragma nopic が記述されているとみなします）。デフォルトに戻す場合は #pragma pic を記述してください。

  ただし，全サブプログラムで共通に使用されるインクルード・ファイル内の extern 宣言を #pragma nopic ～ #pragma pic で囲む際には注意が必要です。単純に extern 宣言を #pragma nopic ～ #pragma pic で囲っただけでは，共通部でコンパイル時エラーになったり，同一サブプログラム内での変数参照に対しても r0 相対命令が生成されたりします。この場合は，-Xmulti 指定時に自動的に定義されるプリプロセッサ・マクロを利用してソースの記述を切り換えるようにしてください。
- 共通部で定義した変数に対して #pragma section 指令で data 以外の再配置属性を指定することはできません。シンボル・ファイルおよび -Xsdata オプションで別の属性を指定した場合は無視されます。

(b) アセンブラ・ソース・プログラム

マルチコア用プログラムをアセンブリ言語で記述する場合，次の点に注意してください。

- 共通部のデータ（const および sconst 属性のものを除く）はすべて data 属性セクションに配置されます。また，共通部内ではデータおよびコードは gp/ep/tp 相対ではなく r0 相対でアクセスする必要があります。

(3) マルチコア対応プログラムのビルドの手順

以下では，コアの数が 2 個の場合のビルドの例を示します。コアの数が 2 個の場合はここに示すように 4 回 CX を起動します。N 個の場合は N+2 回となります。

(a) コア 1 用プログラムのビルド

まずコア 1 用プログラムをコンパイル（アセンブル），リンクします。このときリンクまでを一度に行う必要はありませんが，必ず -Xmulti=pe1 を指定してください。この段階でのリンク時には，コア 1 内で定義されたシンボルの参照は解決しますが，コア 2 および共通部で定義されたシンボルの参照は解決されないまま残ります。

なお，コア 1 専用ライブラリがある場合はこの時点でリンクします。ただし，-Xmulti オプション指定時には -l オプションは無視されるため，ライブラリ・ファイル名を直接指定する必要があります。

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe1 file_pe1_1.c file_pe1_2.c -ope1.lmf
```

(b) コア 2 用プログラムのビルド

次に，コア 2 用プログラムをコンパイル（アセンブル），リンクします。これはコア 1 用プログラムと同様ですが，オプションは -Xmulti=pe2 を指定してください。

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe2 file_pe2_1.c file_pe2_2.c -ope2.lmf
```

(c) 共通部のビルド

さらに，共通部をビルドします。コア 1 用プログラム，コア 2 用プログラムと同様に，リンクまでを一度に行う必要はありませんが，必ず -Xmulti=cmn を指定してください。

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=cmn file_cmn_1.c file_cmn_2.c -ocmn.lmf
```

(d) 各サブプログラムのビルド（最終リンク）

最後に，各サブプログラムをリンクして 1 つのロード・モジュール・ファイルを作成します。(a) ～ (c) の段階で未解決であったシンボル参照はこの時点で解決します。スタートアップ・ルーチンおよびライブラリもこの時点でリンクします。この時点で ROM 化処理，ヘキサ・ファイル生成も行われます。

```
> cx -Cf3515 -Xlink_directive=multi.dir -Xstartup=cstartM.obj -Xmulti_link pe1.lmf pe2.lmf cmn.lmf
-otarget.lmf -lmulti_lib
```

備考 オプションの詳細は「CubeSuite ビルド編（CX コンパイラ）」を参照してください。

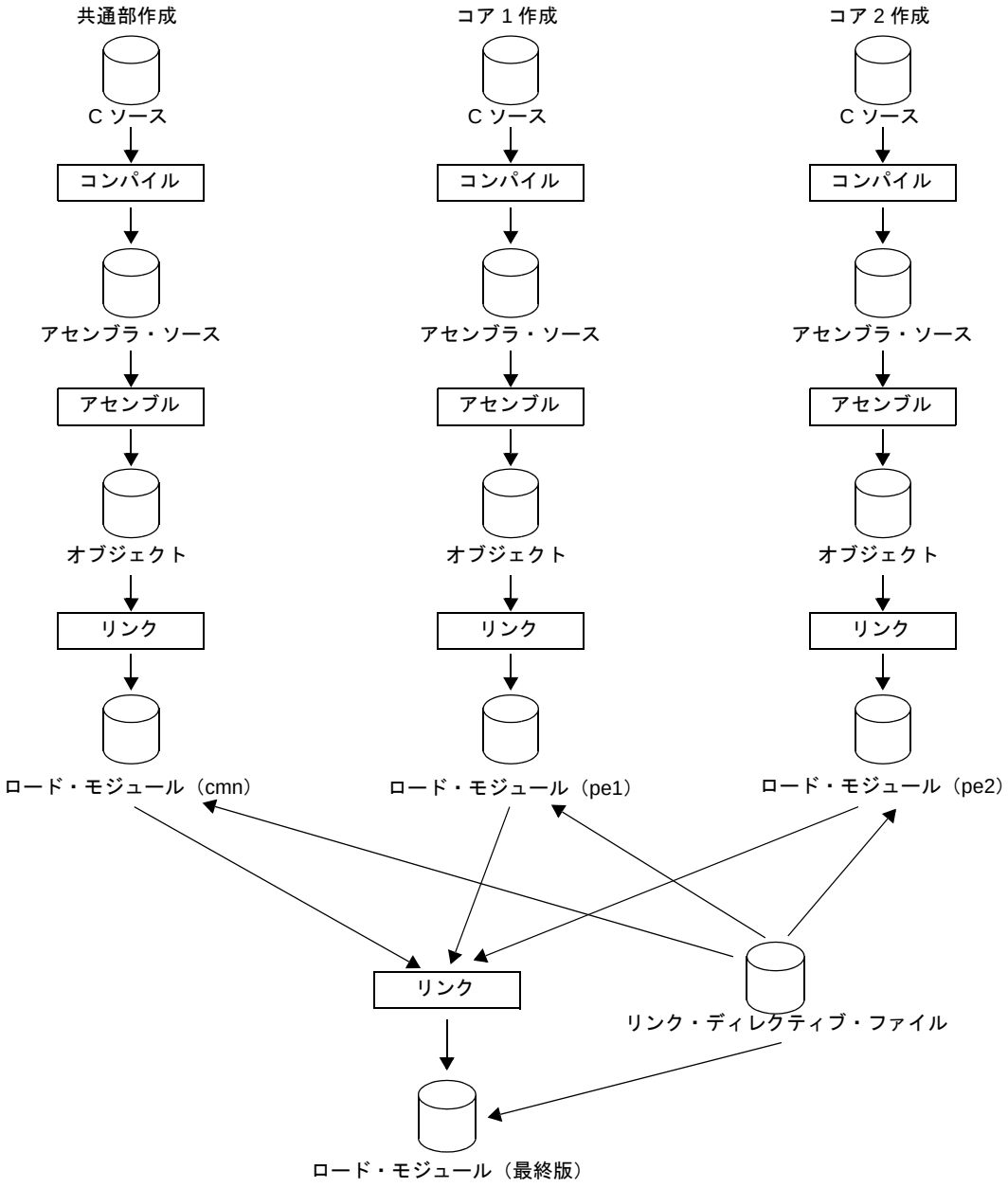
(4) マルチコア用アプリケーション開発フロー

マルチコア用アプリケーション開発フローを示します。

説明する開発フローは，共通 部，コア 1 部，コア 2 部の 3 部構成での例です。

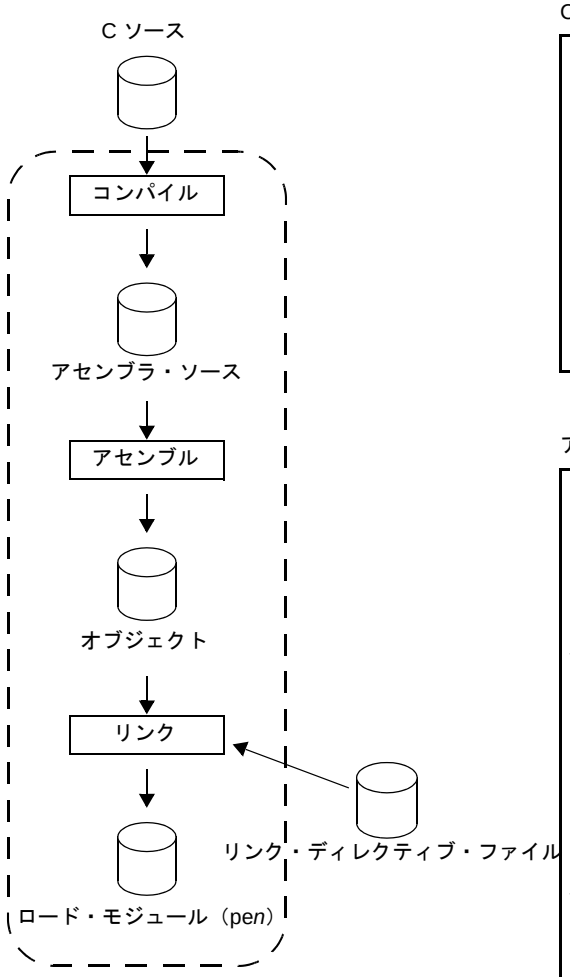
備考 3 部構成がリンク処理の必須条件ではありません。たとえばコア 1 部のロード・モジュール・ファイルを複数作成したり，共通部部やコア 部のみのロード・モジュール・ファイルを作成しアプリケーションを作成することも可能です。ただし，その場合でも“-Xmulti_link”オプションを指定した最終ロード・モジュール・ファイル 作成過程は省略できません。

(a) 全体開発フロー



(b) コア *n* 用プログラム作成開発フロー

> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=pe*n* file_pe*n*_1.c file_pe*n*_2.c -ope*n*.lmf
>
> (-Xmulti=pe*n* 指定時は、ドライバ側で -Xno_startup -Xno_romize -Xrelinkable_object も同時に指定されたと解釈します。)



C ソース例

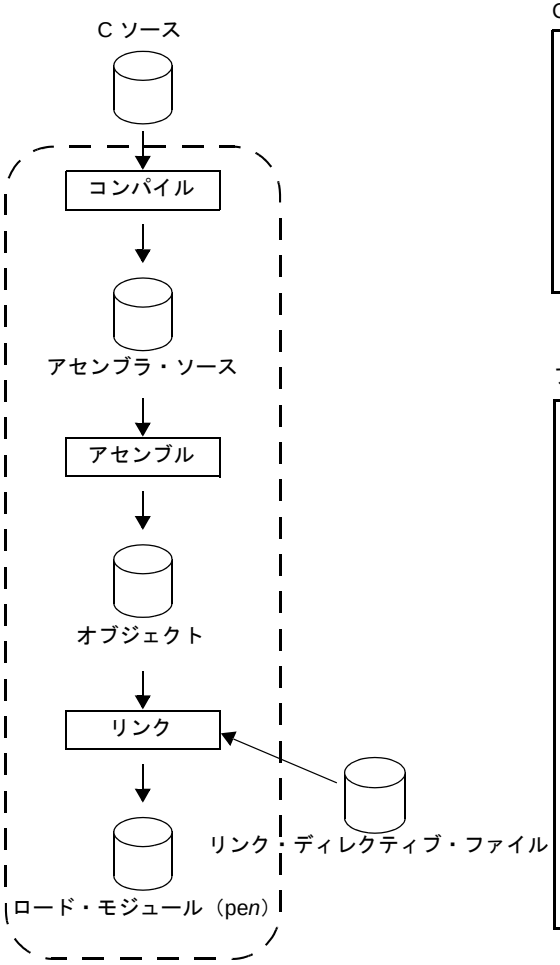
```
extern void func();

void main()
{
        func();
}

int var1 = 0;
```

アセンブラ・ソース例

```
        .extern _func
        .dseg   sdata
        .public _var1, 4
        .align  4
_var1:
        .dw     0
        .cseg   text
        .func   _main, _main.end-_main, 4
        .public _main
        .align  2
_main:
        callt   0
        jarl    _func, lp
        callt   30
_main.end:
```

（c）共通部プログラム作成開発フロー

> cx -Cf3515 -Xlink_directive=multi.dir -Xmulti=cmn file_cmn_1.c file_cmn_2.c -ocmn.lmf
>
> （-Xmulti=cmn 指定時は、ドライバ側で -Xno_startup -Xno_romize -Xrelinkable_object も同時に指定されたと解釈します。）



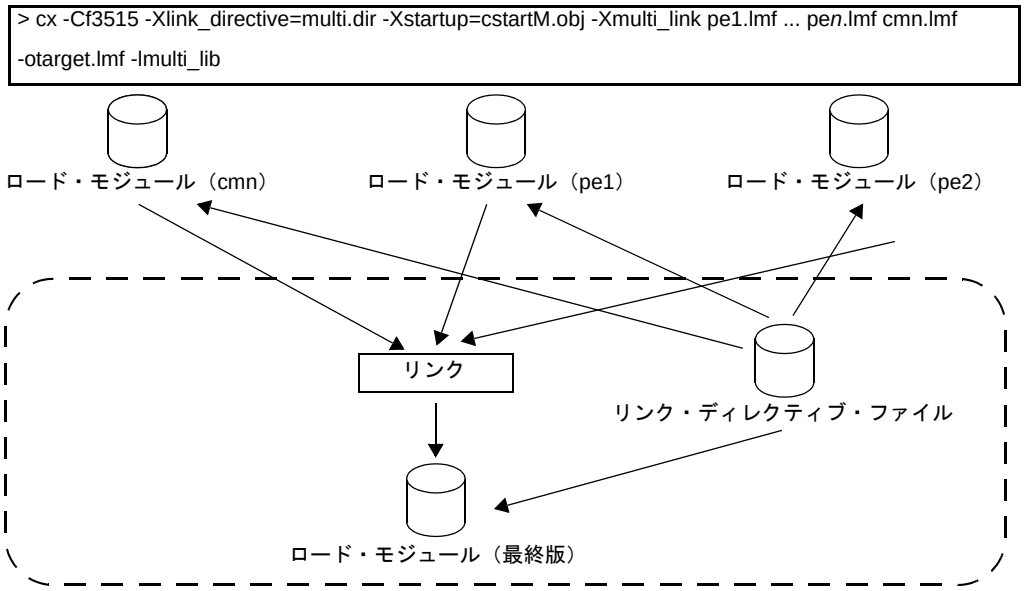
C ソース例

```
int var_cmn = 0;


void func()
{
        ...     // func の処理
}
```

アセンブラ・ソース例

```
        .dseg   data
        .public _var_cmn, 4
        .align  4
_var_cmn:
        .dw     0
        .cseg   text
        .func   _func, _func.end-_func, 0
        .public _func
        .align  2
_func:
        ...      ; func の処理
        jmp     [lp]
_func.end:
```

(d) 最終ロード・モジュール・ファイル部作成開発フロー



(e) リンク・ディレクティブ・ファイル例

```
SCONST_CMN: !LOAD ?R {
        .sconst             = $PROGBITS ?A      .sconst;
        .sconst.cmn         = $PROGBITS ?A      .sconst.cmn;
};
SCONST_PE1: !LOAD ?R {
        .sconst.pe1         = $PROGBITS ?A      .sconst.pe1;
};
SCONST_PE2: !LOAD ?R {
        .sconst.pe2         = $PROGBITS ?A      .sconst.pe2;
};
CONST_CMN: !LOAD ?R {
        .const.cmn          = $PROGBITS ?A      .const.cmn;
        .const              = $PROGBITS ?A      .const;
};
CONST_PE1: !LOAD ?R {
        .const.pe1          = $PROGBITS ?A      .const.pe1;
};
CONST_PE2: !LOAD ?R {
        .const.pe2          = $PROGBITS ?A      .const.pe2;
};
TEXT_CMN: !LOAD ?RX {
        .pro_epi_runtime    = $PROGBITS ?AX     .pro_epi_runtime;
        .text.cmn           = $PROGBITS ?AX     .text.cmn;
        .text               = $PROGBITS ?AX     .text;
};
```

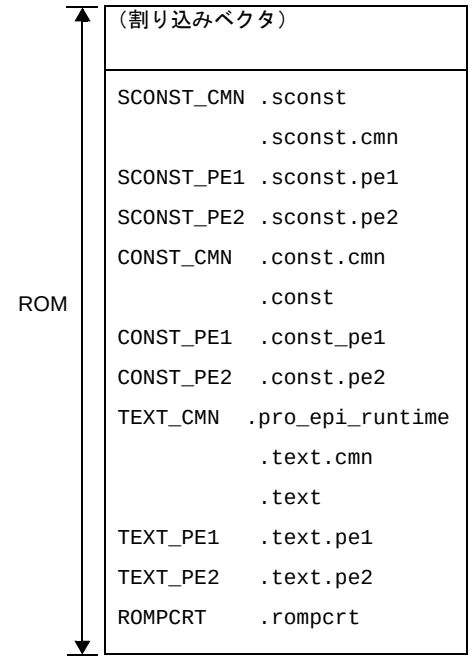
```
TEXT_PE1: !LOAD ?RX {
        .text.pe1           = $PROGBITS ?AX     .text.pe1;
};
TEXT_PE2: !LOAD ?RX {
        .text.pe2           = $PROGBITS ?AX     .text.pe2;
};
ROMPCRT: !LOAD ?RX {
        .rompcrt            = $PROGBITS ?AX     .text {rompcrt.obj};
};
DATA_PE2: !LOAD ?RW {
        .data.pe2           = $PROGBITS ?AW     .data.pe2;
        .sdata.pe2          = $PROGBITS ?AWG    .sdata.pe2;
        .sbss.pe2           = $NOBITS   ?AWG    .sbss.pe2;
        .bss.pe2            = $NOBITS   ?AW     .bss.pe2;
};
SEDATA_PE2: !LOAD ?RW {
        .sedata.pe2         = $PROGBITS ?AW     .sedata.pe2;
        .sebss.pe2          = $NOBITS   ?AW     .sebss.pe2;
};
SIDATA_PE2: !LOAD ?RW {
        .tidata.byte.pe2    = $PROGBITS ?AW     .tidata.byte.pe2;
        .tibss.byte.pe2     = $NOBITS   ?AW     .tibss.byte.pe2;
        .tidata.word.pe2    = $PROGBITS ?AW     .tidata.word.pe2;
        .tibss.word.pe2     = $NOBITS   ?AW     .tibss.word.pe2;
        .tidata.pe2         = $PROGBITS ?AW     .tidata.pe2;
        .tibss.pe2          = $NOBITS   ?AW     .tibss.pe2;
        .sidata.pe2         = $PROGBITS ?AW     .sidata.pe2;
        .sibss.pe2          = $NOBITS   ?AW     .sibss.pe2;
};
DATA_CMN: !LOAD ?RW {
        .data.cmn           = $PROGBITS ?AW     .data.cmn;
        .bss.cmn            = $NOBITS   ?AW     .bss.cmn;
};
DATA_PE1: !LOAD ?RW {
        .data.pe1           = $PROGBITS ?AW     .data.pe1;
        .sdata.pe1          = $PROGBITS ?AWG    .sdata.pe1;
        .sbss.pe1           = $NOBITS   ?AWG    .sbss.pe1;
        .bss.pe1            = $NOBITS   ?AW     .bss.pe1;
        .data               = $PROGBITS ?AW     .data;
        .sdata              = $PROGBITS ?AWG    .sdata;
        .sbss               = $NOBITS   ?AWG    .sbss;
        .bss                = $NOBITS   ?AW     .bss;
};
SEDATA_PE1: !LOAD ?RW {
        .sedata.pe1         = $PROGBITS ?AW     .sedata.pe1;
```

```
        .sebss.pe1          = $NOBITS   ?AW     .sebss.pe1;
};
SIDATA_PE1: !LOAD ?RW {
        .tidata.byte.pe1    = $PROGBITS ?AW     .tidata.byte.pe1;
        .tibss.byte.pe1     = $NOBITS   ?AW     .tibss.byte.pe1;
        .tidata.word.pe1    = $PROGBITS ?AW     .tidata.word.pe1;
        .tibss.word.pe1     = $NOBITS   ?AW     .tibss.word.pe1;
        .tidata.pe1         = $PROGBITS ?AW     .tidata.pe1;
        .tibss.pe1          = $NOBITS   ?AW     .tibss.pe1;
        .sidata.pe1         = $PROGBITS ?AW     .sidata.pe1;
        .sibss.pe1          = $NOBITS   ?AW     .sibss.pe1;
};
__tp_TEXT_PE1@%TP_SYMBOL {TEXT_PE1};
__tp_TEXT_PE2@%TP_SYMBOL {TEXT_PE2};
__gp_DATA_PE1@%GP_SYMBOL &__tp_TEXT_PE1 {DATA_PE1};
__gp_DATA_PE2@%GP_SYMBOL &__tp_TEXT_PE2 {DATA_PE2};
__ep_DATA_PE1@%EP_SYMBOL;
__ep_DATA_PE2@%EP_SYMBOL;
```

(f) マルチコア用プログラムの配置イメージ

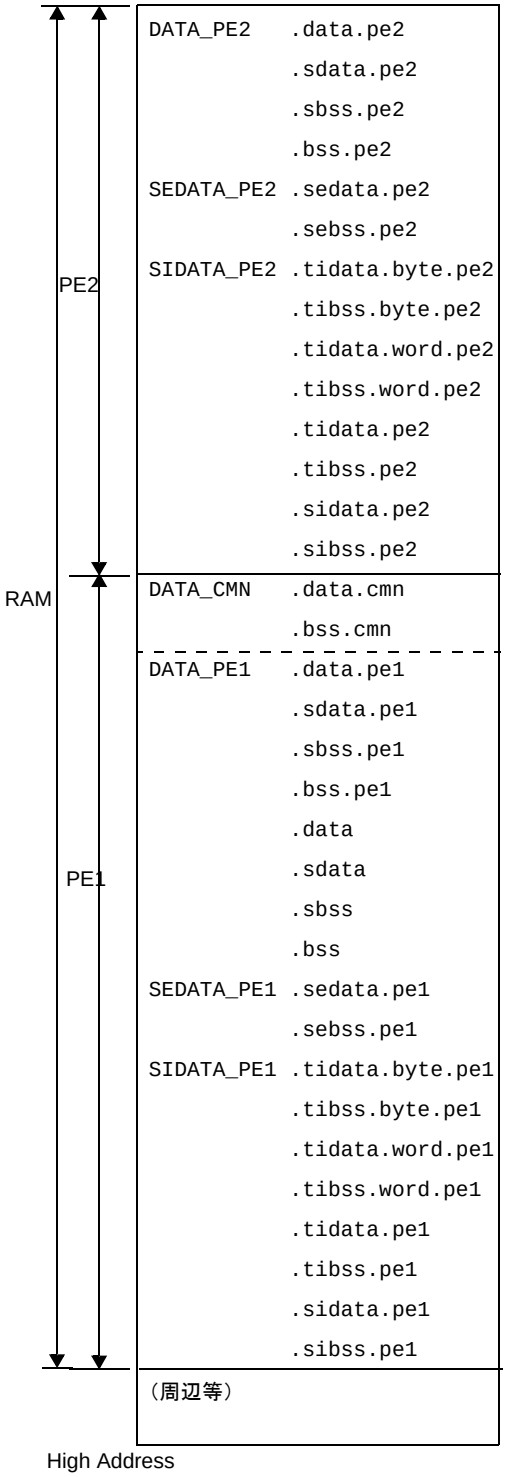
(e) リンク・ディレクティブ・ファイル例の配置イメージを示します（例は，μPD70F3515 です）。

セグメント／セクションのメモリ配置イメージ



リンク・ディレクティブ情報

```
SCONST_CMN: !LOAD ?R {
  .sconst          = $PROGBITS ?A   .sconst;
  .sconst.cmn      = $PROGBITS ?A   .sconst.cmn;
};
SCONST_PE1: !LOAD ?R {
  .sconst.pe1      = $PROGBITS ?A   .sconst.pe1;
};
SCONST_PE2: !LOAD ?R {
  .sconst.pe2      = $PROGBITS ?A   .sconst.pe2;
};
CONST_CMN: !LOAD ?R {
  .const.cmn       = $PROGBITS ?A   .const.cmn;
  .const           = $PROGBITS ?A   .const;
};
CONST_PE1: !LOAD ?R {
  .const.pe1       = $PROGBITS ?A   .const.pe1;
};
CONST_PE2: !LOAD ?R {
  .const.pe2       = $PROGBITS ?A   .const.pe2;
};
TEXT_CMN: !LOAD ?RX {
  .pro_epi_runtime = $PROGBITS ?AX  .pro_epi_runtime;
  .text.cmn        = $PROGBITS ?AX  .text.cmn;
  .text            = $PROGBITS ?AX  .text;
};
TEXT_PE1: !LOAD ?RX {
  .text.pe1        = $PROGBITS ?AX  .text.pe1;
};
TEXT_PE2: !LOAD ?RX {
   text.pe2        = $PROGBITS ?AX  .text.pe2;
};
ROMPCRT: !LOAD ?RX {
  .rompcrt        = $PROGBITS ?AX .text {rompcrt.obj};
};
```




```
DATA_PE2: !LOAD ?RW {
  .data.pe2       = $PROGBITS ?AW  .data.pe2;
  .sdata.pe2      = $PROGBITS ?AWG .sdata.pe2;
  .sbss.pe2       = $NOBITS   ?AWG .sbss.pe2;
  .bss.pe2        = $NOBITS   ?AW  .bss.pe2;
};
SEDATA_PE2: !LOAD ?RW {
  .sedata.pe2     = $PROGBITS ?AW  .sedata.pe2;
  .sebss.pe2      = $NOBITS   ?AW  .sebss.pe2;
};
SIDATA_PE2: !LOAD ?RW {
  .tidata.byte.pe2 = $PROGBITS ?AW  .tidata.byte.pe2;
    :
  .sibss.pe2      = $NOBITS   ?AW  .sibss.pe2;
};
DATA_CMN: !LOAD ?RW {
  .data.cmn       = $PROGBITS ?AW  .data.cmn;
  .bss.cmn        = $NOBITS   ?AW  .bss.cmn;
};
DATA_PE1: !LOAD ?RW {
  .data.pe1       = $PROGBITS ?AW  .data.pe1;
    :
  .bss            = $NOBITS   ?AW  .bss;
};
SEDATA_PE1: !LOAD ?RW {
  .sedata.pe1     = $PROGBITS ?AW  .sedata.pe1;
  .sebss.pe1      = $NOBITS   ?AW  .sebss.pe1;
};
SIDATA_PE1: !LOAD ?RW {
  .tidata.byte.pe1 = $PROGBITS ?AW  .tidata.byte.pe1;
    :
  .sibss.pe1       = $NOBITS   ?AW  .sibss.pe1;
};
__tp_TEXT_PE1@%TP_SYMBOL {TEXT_PE1};
__tp_TEXT_PE2@%TP_SYMBOL {TEXT_PE2};
__gp_DATA_PE1@%GP_SYMBOL &__tp_TEXT_PE1 {DATA_PE1};
__gp_DATA_PE2@%GP_SYMBOL &__tp_TEXT_PE2 {DATA_PE2};
__ep_DATA_PE1@%EP_SYMBOL;
__ep_DATA_PE2@%EP_SYMBOL;
```

（5）注意事項

CX のマルチコア用プログラムにおいては，以下の点に注意する必要があります。

- 各コア用および共通部用のロード・モジュール・ファイルにおいて，同名シンボルを定義することはできません。同名シンボルを定義した場合，最終リンク時にエラーとなります。
- 独自のリンク・ディレクティブ・ファイルを作成する場合は，すべてのリンク時に同一のディレクティブ・ファイルを使用することを推奨します。
- デフォルトのマルチコア用スタートアップ・ルーチンを使う場合には，コア 1，コア 2 用のスタック領域として __stack.pe1，__stack.pe2 というラベルで始まる領域を別途確保（定義）する必要があります。

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

初期値なし変数領域を確保するには，初期値なしセクション中で，.ds 疑似命令を使用します。

```
[ ラベル :]  .ds     ( 絶対式 )
```

他のファイルからも参照できるようにするには，そのラベルを .public 疑似命令で宣言する必要があります。

```
[ ラベル :]  .public  ラベル名 [, サイズ ]
```

【記述例】

```
        .dseg   sbss
        .public _val0, 4        -- _val0 を他のファイルから参照できるようにします
        .public _val1, 2        -- _val1 を他のファイルから参照できるようにします
        .public _val2, 1        -- _val2 を他のファイルから参照できるようにします
        .align  4
_val0:  .ds  (4)                -- val0 は 4 バイトの領域を確保します
_val1:  .ds  (2)                -- val1 は 2 バイトの領域を確保します
_val2:  .ds  (1)                -- val2 は 1 バイトの領域を確保します
```

2. 4. 2 初期値あり const 定数を定義する

初期値あり const 定数を定義するには，.const/.sconst セクション中で，.db 疑似命令 /.db2/.dhw 疑似命令 /.db4/.dw 疑似命令を使用します。

- 1 バイトの値の場合

```
[ ラベル :]  .db     値
```

- 2 バイトの場合

```
[ ラベル :]  .db2    値
```

```
[ ラベル :]  .dhw    値
```

- 4 バイトの場合

```
[ ラベル :]  .db4    値
```

```
[ ラベル :]  .dw     値
```

【記述例：1 ハーフワード分確保し，100 を格納】

```
        .cseg   const
        .public _p, 2
        .align  4
_p:     .db2    100
```

2. 4. 3 セクションのアドレスを参照する

.data や .sdata などのセクションの先頭と末尾を指すシンボル（予約シンボル）が用意されています。したがって，アセンブラ・ソースから特定セクションのアドレス値を使用する際には，該当シンボル名を利用します。

先頭シンボル： __s セクション名

末尾シンボル： __e セクション名

たとえば，.sbss セクションの先頭シンボルは __ssbss，末尾シンボルは __esbss という名前になります。

これらのシンボルを使用することにより，セクションの先頭アドレスと末尾アドレスを取得することができますが，C 言語レベルでは，これらのシンボル名を使用して直接参照することはできません。

これらのシンボル値を取得するには，この値を格納する外部変数を作成し，スタートアップ・ルーチンなどのアセンブラ・ソースにて，変数にシンボル値を格納します。

その変数を C ソース上で参照することによって実現することができます。

これは __gp_DATA などのシンボルについても同じです。

たとえば，.data セクションの先頭アドレスと末尾アドレスを取得する方法は，次のようになります。

【アセンブラ・ソース内】

```
        .extern __sdata, 4
        .extern __edata, 4
        .dseg   sdata
        .public _data_top, 4
        .public _data_end, 4
        .align  4
_data_top:
        .ds     (4)
_data_end:
        .ds     (4)
        .cseg   text
        mov     #__sdata, r12
        st.w    r12, $_data_top
        mov     #__edata, r13
        st.w    r13, $_data_end
```

【C ソース内】

```
extern  int data_top;   /* data_top を extern 宣言する */
extern  int data_end;   /* data_end を extern 宣言する */

void func1(void) {
      int top, end;
      top = data_top;
      end = data_end;
}
```

特定のセクションのみを，C 言語レベルで初期化するような場合，この方法を用いてください。

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

スタック・ポインタ（sp）に値を設定する際には，以下の点に注意する必要があります。

- スタック・フレームは，sp に設定した値から下位方向に生成されます。
- sp は，必ず 4 バイト境界の位置を指すように設定してください。
- コンパイラはスタック相対でメモリを参照する場合，スタック・ポインタが 4 バイト境界の位置を指していることを想定して，コードを生成しています。
  なるべく gp と離れたデータ・セクション（bss 属性セクション）に配置してください。
  gp と近いと，プログラムのデータ領域を破壊するおそれがあります。

【sp の設定例】

```
STACKSIZE       .set    0x3F0
        .dseg   bss
        .align  4
__stack:
        .ds     (STACKSIZE)
        .cseg   text
        mov     #__stack + STACKSIZE, sp
```

なお，上記の例では，アプリケーションで使用するスタック・フレームのサイズを，0x3F0 バイトと指定して領域を確保します。

“__stack”は，スタック・フレームの最下位（先頭）を指すラベルとなります。

デフォルトのスタートアップ・ルーチンでは，__stack を外部変数定義（.public 宣言）をしていないため，他ファイルから __stack の参照ができません。

そこで，__stack に対して .public 宣言を実施すれば，他ファイルからの参照が可能になります。

また，スタック領域は，最下位アドレスに __stack というシンボルを定義して，スタック・ポインタに __stack のアドレスとサイズを加算したものを設定しています。

そのため，エンドのアドレスにシンボルはありません。

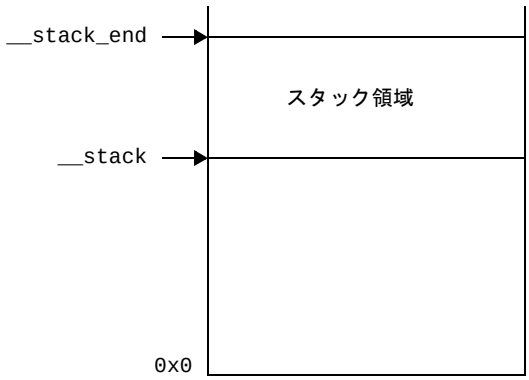
次のようにすることにより，スタック領域のエンドのアドレスの次のアドレスを定義することは可能になります。

スタック領域の最後のアドレスではないので注意してください。

```
STACKSIZE       .set    0x3F0
        .dseg   bss
        .public __stack         -- 追加
        .public __stack_end     -- 追加
        .align  4
__stack:
        .ds     (STACKSIZE)
__stack_end:
```

上記の定義により，C ソース上で，_stack，_stack_end のシンボルを参照することが可能です。
マッピングのイメージは，次のようになります。

図 2―3 スタック領域のマッピング・イメージ



なお，__stack シンボルはスタートアップ・ルーチンでサイズを指定してあるので，C ソースでは次のように配列で定義してください。
スタック領域の最後のアドレスではないので注意してください。

```
extern unsigned long  _stack[];
```

備考 アセンブラで定義したラベルを C 言語で使用する場合には，先頭のアンダーバー“_”を 1 つ削除した名前になります。

アセンブリ言語定義： __stack
C 言語での参照： _stack

C ソース・プログラムのスタック領域については，スタック見積りツールを使うことにより計測することができます。

2.5.2 スタック領域を確保し配置を指定する

この項では，スタック領域を確保し配置を指定する方法について説明します。

(1) スタック領域の確保

スタートアップ・ルーチンで，セクション名を指定した初期値なし変数のセクションにスタック領域を確保します。

【領域確保の例】

```
STACKSIZE       .set    0x3F0
        .stack  .dseg   bss
        .align  4
__stack:
        .ds     (STACKSIZE)
```

なお，上記の例では，アプリケーションで使用するスタック・フレームのセクションを .stack，サイズを 0x3F0 バイトと指定して領域を確保します。

“__stack”は，スタック・フレームの最下位（先頭）を指すラベルとなります。

(2) スタック領域の配置指定

リンク・ディレクティブ・ファイルで，(1) で作成したセクションの配置を指定します。

【配置指定例】

```
STACK:  !LOAD ?RW V0x3FFEE00 {
        .stack  = $NOBITS ?AW .stack;
};
```

なお，上記の例では，スタック領域用のセグメントを STACK とし，0x3FFEE00 番地に配置しています。

2. 5. 3 RAM を初期化する

この項では，RAM の初期化について説明します。

(1) 初期値なし変数

デフォルトのスタートアップ・ルーチンでは，.sbss セクション，.bss セクションを 0 でクリアする処理が組み込まれています。

上記以外のセクションを使用した場合に，ゼロクリアしたい場合には，スタートアップ・ルーチンに処理を追加してください。クリアする際には，セクションの先頭と終端を示すシンボルを使用します。

【記述例：.tibss.byte セクションのゼロクリア】

```
        .extern __stibss.byte, 4        --  .tibss.byte 領域の先頭シンボル
        .extern __etibss.byte, 4        --  .tibss.byte 領域の終端シンボル
        mov     #__stibss.byte, r13
        mov     #__etibss.byte, r12
        cmp     r12, r13
        jnl     .L20
.L21:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L21
.L20:
```

(2) RAM の初期化

ROM 化を行わず，インサーキット・エミュレータにロード・モジュールをダウンロードした場合，data 領域や sdata 領域に置かれる初期値ありデータは，ダウンロードした時点でセットされます。

ROM 化を行っていないロード・モジュールを使用してデバッグするときは，初期値あり変数の RAM 領域を初期化するような処理は外す必要があります。初期値なし変数の RAM 領域の初期化だけであれば，外す必要はありません。

ROM 化用ロード・モジュールの場合は，コピー関数 _rcopy で初期値ありデータ・コピー等を行う必要があります。

この処理は，スタートアップ・ルーチンではなく，main 関数の初期値あり変数をアクセスする前でも可能ですので，周辺の設定が完了した後に行ってください。

2. 5. 4 関数／変数アクセスを準備する

関数をアクセスする際にはテキスト・ポインタを，変数をアクセスする際には，グローバル・ポインタ，またはエレメント・ポインタを使用します。

(1) 関数アクセスの準備

アプリケーションのテキスト領域であるプログラム・コードを参照する際に，配置される位置に依存しない参照（PIC：Position Independent Code）を実現するために用意されているポインタが“テキスト・ポインタ（tp）”です。たとえば，プログラム実行中に，コード内のある箇所を参照する必要がある場合，CX は tp 相対でアクセスするコードを出力します。

したがって，tp が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で tp を正しく設定する必要があります。

テキスト・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，テキスト・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__tp_TEXT@%TP_SYMBOL {TEXT};
```

このとき，テキスト・ポインタの値は“TEXT セグメント”の先頭になり，その値は“__tp_TEXT”に入ります。

スタートアップ・ルーチン内で tp をセットするには，次のように記述してください。

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

(2) 変数アクセスの準備（グローバル・ポインタの設定）

アプリケーション内で定義した外部変数／データはメモリ上に配置されます（.sdata/.sbss セクション）。そのメモリに配置されている変数／データを参照する際，配置位置に依存することのない参照（PID：Position Independent Data）を実現するために用意されているポインタが“グローバル・ポインタ（gp）”です。gp 相対でアクセスするセクションが存在する場合，CX は gp 相対でアクセスするコードを出力します。

したがって，gp が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で gp を正しく設定する必要があります。

グローバル・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，グローバル・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__gp_DATA@%GP_SYMBOL {DATA};
```

また，gp シンボル値は，上記のように“DATA セグメントなどの「データ用セグメント」の先頭を gp シンボル値とする方法”のほかに，“テキスト・シンボルからのオフセットを gp シンボル値とする”方法もあります。

この方法の場合，gp シンボルを「tp の値と，tp からのオフセット値を加える」ことによって決定できます。つまり，配置に依存しないコードの生成が可能になります。たとえば，“プログラム・コード”と“そのコードが使用するデータ”を同時 RAM 領域にコピーしてから実行させたい場合，コードの先頭（コピー先の先頭アドレス）さえ分かれば，gp の値もすぐ導き出せるというメリットがあります。この場合のシンボル・ディレクティブ記述は次のようになります。

```
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT {DATA};
```

グローバル・ポインタの値は，“__tp_TEXT に __gp_DATA の値を加えた値”となり，加える値（オフセット値）が“__gp_DATA”に入ります。したがって，スタートアップ・ルーチン内で gp をセットするには，次のように記述します。

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

これにより，正しいグローバル・ポインタの値が gp に設定されます。

(3) 変数アクセスの準備（エレメント・ポインタの設定）

アプリケーション内でグローバル宣言したデータ（変数）を，V850 コアに内蔵されている RAM 領域へ配置し，より高速な参照を実現するために用意されているポインタが“エレメント・ポインタ（ep）”です。

アプリケーション内で定義した外部変数／データのうち，次に割り当てられているものは，エレメント・ポインタ（ep）からの相対でアクセスされます。

- sedata/sebss セクション
- sidata/sibss セクション
- tidata/tibss セクション
- tidata.byte/tibss.byte セクション
- tidata.word/tibss.word セクション

これらのセクションが存在する場合，CX は ep 相対でアクセスするコードを出力します。

したがって，ep が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で ep を正しく設定する必要があります。

エレメント・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，エレメント・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__ep_DATA@%EP_SYMBOL;
```

エレメント・ポインタの値は，デフォルトで“SIDATA セグメントの先頭”になり，その値は“__ep_DATA”に入ります。

したがって，スタートアップ・ルーチン内で ep をセットするには，次のように記述します。

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

__ep_DATA の絶対アドレス参照を行い，その値を ep に設定します。

2. 5. 5 コード・サイズ削減機能を使用する準備をする

V850Ex コアを使用している場合で，コード・サイズを削減するために，関数前後処理ランタイム関数を使用する場合（実行速度優先の最適化（-Ospeed オプション）を指定していない場合，または -Xpro_epi_runtime=on を指定している場合）に，この設定が必要になります。

V850Ex コアで関数の関数前後処理ランタイム関数を呼び出すとき，CALLT 命令を使用するため，その CALLT 命令に必要な CTBP の値を，関数前後処理ランタイム関数の関数テーブルの先頭に設定しておく必要があります。

関数前後処理ランタイム関数を使用する設定になるのは，次の場合です。

- コンパイラ・オプション“-Xpro_epi_runtime=on”を設定している

コンパイラの最適化オプション“-Ospeed”「以外」を指定していると，自動的に“-Xpro_epi_runtime=on”になります。

関数前後処理ランタイム関数の関数テーブルの先頭シンボルは，次のとおりです。

- ___PROLOG_TABLE

このシンボルを用いて，次のコードを記述します。

```
mov     #___PROLOG_TABLE, r12
ldsr    r12, 20
```

備考 CTBP はシステム・レジスタ 20 番なので，ldsr 命令を使用して値を設定します。

2. 5. 6 スタートアップ・ルーチンを終了する

スタートアップ・ルーチンの最後の処理は，リアルタイム OS を使用するかしないにより違いがあります。

(1) リアルタイム OS を使用しない場合

スタートアップ・ルーチンで行う必要のある処理がすべて終わったとき，main 関数への分岐命令を実行します。

main 関数への分岐には，次のコードを記述します。

```
jarl    _main, lp
```

また，main 関数の実行がすべて終わった後，この分岐命令の 4 バイト先に戻ってくることになります。

戻ってこないことが分かっている場合は，次の命令も使用できます。

```
jr      _main
```

```
mov     #_main, lp
jmp     [lp]
```

jmp 命令を使用すると，32 ビット全空間をアクセスすることができます。もし，“jarl _main, lp”を使用する場合は，main 関数実行後に戻ってくるので，戻ってきた後デッドロックしないように，対策を施しておくと安全です。

(2) リアルタイム OS（RX850V4）を使用する場合

リアルタイム OS を使用したアプリケーションで，スタートアップ・ルーチンで行う必要のある処理がすべて終わったとき，リアルタイム OS の初期化ルーチンへ分岐します。

```
        .extern __kernel_sit
        .extern __kernel_start
        mov     #__kernel_sit, r6
        jarl    __kernel_start, lp

__boot_error:
        jbr     __boot_error
```

2.6 リンク・ディレクティブ

この節では，リンク・ディレクティブについて説明します。
リンク・ディレクティブ・ファイルは，CubeSuite 上で自動生成することも可能です。

備考 リンク・ディレクティブ・ファイルの自動作成方法については，「CubeSuite ビルド編（CX コンパイラ）」のユーザーズ・マニュアルを参照してください。

2.6.1 関数のセクション配置を追加する

関数のセクション配置をするには，.text セクションの指定部分を流用し，セグメント名，セクション名を変更してください。

```
TEXT:       !LOAD ?RX {
            .pro_epi_runtime    = $PROGBITS ?AX .pro_epi_runtime;
            .text               = $PROGBITS ?AX .text;
};
```

【記述例：USRTEXT セグメント，usr.text セクションの配置を指定】

```
USRTEXT:    !LOAD ?RX {
            usr.text            = $PROGBITS ?AX usr.text;
};
```

2.6.2 変数のセクション配置を追加する

変数のセクションの配置指定を追加するには，同じ属性のセクション指定部分を流用して，セグメント名，セクション名を変更してください。

セクション属性は，#pragma section で変数にセクションを指定する際にセクション種別を指定しています。

セクション種別	流用するセクション
data	.data/.bss
sdata	.sdata/.sbss
sconst	.sconst
const	.const

【記述例：USRCONST セグメント，usr.const セクションの配置を指定】

```
USRCONST:   !LOAD ?R {
            usr.const           = $PROGBITS ?A usr.const;
};
```

2. 6. 3 セクション配置を振り分ける

セクション配置を振り分ける方法として，次の 3 通りの方法があります。

(1) セクション名で振り分ける

C ソースやアセンブラ・ソースで配置するセクション名で別々の名前を指定します。

リンク・ディレクティブ中では，入力セクション名をそれぞれ指定することにより，その名前のセクションが，指定した部分に配置されます。

【記述例】

```
TEXT:   !LOAD ?RX {
    .text           = $PROGBITS ?AX .text;          ← .text セクションを配置
};

FUNC1:  !LOAD ?RX {
    funcsec1.text   = $PROGBITS ?AX funcsec1.text;  ← funcsec1.text セクションを配置
};
```

(2) オブジェクト・モジュール・ファイル名で振り分ける

リンク・ディレクティブ中では，入力オブジェクト名をそれぞれ指定することにより，そのオブジェクト中の該当する属性のセクションが，指定した部分に配置されます。

【記述例】

```
TEXT1:  !LOAD ?RX {
    .text1 = $PROGBITS ?AX .text {filel.obj file2.obj};
                                        ← file1.obj, file2.obj の .text セクションを配置
};

TEXT2:  !LOAD ?RX {
    .text2 = $PROGBITS ?AX .text {file3.obj};
                                        ← file3.obj の .text セクションを配置
};
```

なお，ライブラリ（.lib ファイル）内のオブジェクト・モジュール・ファイル名を指定する場合，.lib ファイル名を ( ) で囲んで指定します。

【記述例】

```
.text3 = $PROGBITS ?AX .text {strcmp.obj(libc.lib)};
```

(3) セクション属性で振り分ける

入力セクション，入力オブジェクトを指定せずに，属性のみで配置指定します。この指定は，セクション名やオブジェクト名を指定する部分よりも優先度が低いので，セクション名やオブジェクト名を指定していない部分全部の配置を指定するときに使用できます。

【記述例】

```
TEXT4:  !LOAD ?RX {
        .text4 = $PROGBITS ?AX  {file1.obj file2.obj};
                ← file1.obj, file2.obj の TEXT 属性のセクションを配置
};

TEXT5:  !LOAD ?RX {
        .text5 = $PROGBITS ?AX;
                ← file1.obj, file2.obj 以外のオブジェクト中の TEXT 属性のセクションを配置
};
```

(4) 配置指定の優先度

入力セクションの指定，入力オブジェクトの指定の有無により，優先度があります。リンカは，セクション配置時に優先度が高い指定から配置していきます。

優先度と指定の関係を次に示します（優先度は数値が小さいものが優先度が高くなります）。

優先度	指定	出力
1	入力セクション名 ＋オブジェクト・モジュール・ファイル名	指定した入力セクションを，指定したオブジェクトから抽出して出力
2	入力セクション名のみ	指定した入力セクションを，すべてのオブジェクトから抽出して出力
3	オブジェクト・モジュール・ファイル名のみ	作成する出力セクションと同じ属性を持つセクションを，指定されたオブジェクトから抽出して出力
4	両方とも記述しなかった場合	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出して出力

2.7 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。

2.7.1 コード・サイズの削減（C 言語）

この項では，C 言語でコード・サイズを削減する方法について説明します。

(1) 変数へのアクセス

外部変数アクセスではロード／ストアに各 4 バイト必要となるため，代入以外の場合にも，外部変数の値をいったんテンポラリ変数に代入してそのテンポラリ変数を使い回すと，メモリ・アクセスがレジスタ・アクセスに変わりコード・サイズが減る可能性があります。

以下の例で s は外部変数であるとします。

変更前

```
if(x != 0) {
        if((s & 0x00F00F00) != MASK1) {
                return;
        }
        s >>= 12;
        s &= 0xFF;
} else {
        if((s & 0x00FF0000) != MASK2) {
                return;
        }
        s >>= 24;
}
```

変更後

```
unsigned int  tmp = s;

if(x != 0) {
        if((tmp & 0x00F00F00) != MASK1) {
                return;
        }
        tmp >>= 12;
        tmp &= 0xFF;
} else {
        if((tmp & 0x00FF0000) != MASK2) {
                return;
        }
        tmp >>= 24;
}
s = tmp;
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(2) ループ処理のループ回数

以下の例のように，実行回数が少なく，かつ，ループ・ボディが小さい場合，展開した方がサイズが小さくなることがあります。

この場合，実行速度も向上します。

変更前	変更後

for(i = 0; i < 4; i++) {
　　array[i] = 0;
}
	long　*p;
	p = array;
	*p = 0;
	*(p + 1) = 0;
	*(p + 2) = 0;
	*(p + 3) = 0;

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(3) auto 変数の初期化

関数内で auto 変数が初期化されずに使用されている場合，その変数がレジスタに割り付けられずにメモリのまま残るため，コード・サイズが大きくなることがあります。

以下の例では，switch のいずれのケースにも該当しない場合に変数 a が初期化されず，return 文で参照されることになります。

実際には必ずいずれかのケースに該当するとしても，CX がレジスタ割り付けにおいてプログラムを解析する際には分からないため，初期化されない場合があるとみなされます。

このような場合に CX のレジスタ割り付けでは割り付けられません。

そこで，初期化を追加することにより，レジスタに割り付けられるようになり，コード・サイズが削減されます。

変更前

```
int func(int x) {
  int a;

  switch(x) {
    case  0:
      a = VAL0;
      break;
    case  1:
      a = VAL1;
  }
  return(a);
}
```

変更後

```
int func(int x) {
  int a = 0;

  switch(x) {
    case  0:
      a = VAL0;
      break;
    case  1:
      a = VAL1;
  }
  return(a);
}
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（4）switch 文

CX は switch 文に対して，case ラベルの個数が 4 個以上で，かつラベルの値の下限と上限の差が case の数の 3 倍までであれば，テーブル分岐形式のコードを生成します。

この場合，case の数がおよそ 16 個以下（ただし，この数は switch の式の形式やラベルの値の分布によって異なります）ならば，等価な if-else 文に変更し，比較命令と分岐命令の並びにした方が，コード・サイズが小さくなります。

なお，switch の式が外部変数参照や複雑な式の場合は，いったんテンポラリ変数に値を代入して，if の式ではそのテンポラリを参照するように変更する必要があります。

以下の例では x は auto 変数であるものとします。

変更前

```
switch(x) {
        case  VAL0:
                return(RETVAL0);
        case  VAL1:
                return(RETVAL1);
        case  VAL2:
                return(RETVAL2);
        case  VAL3:
                return(RETVAL3);
        case  VAL4:
                return(RETVAL4);
        case  VAL5:
                return(RETVAL5);
}
```

変更後

```
if(x == VAL0)
        return(RETVAL0);
else if(x == VAL1)
        return(RETVAL1);
else if(x == VAL2)
        return(RETVAL2);
else if(x == VAL3)
        return(RETVAL3);
else if(x == VAL4)
        return(RETVAL4);
else if(x == VAL5)
        return(RETVAL5);
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

4. CX では，-Xswitch オプションにより，switch 文の展開コードを指定することが可能です。

- -Xswitch=ifelse

  case 文の並びに沿った if-else 文と同じ形で出力します。

- -Xswitch=binary

  バイナリ・サーチ形式で出力します。

- -Xswitch=table

  テーブル・ジャンプ方式で出力します。

(5) if 文

if-else の組み合わせにおいて，複数のケースで同じ処理を実行する場合，別の条件を用いてその「複数のケース」を 1 つにまとめられるのであればまとめます。

これにより，冗長な部分を削除します。

以下の例で，「x の初期値が 0 で，s，および t の値は 0，または 1 のいずれかである」という条件が揃っている場合，次のように変形することが可能です。

変更前	変更後

`if(!s) {`
`  if(t) {`
`  x = 1;`
`  }`
`} else {`
`  if(!t) {`
`  x = 1;`
`  }`
`}`
`if(x) {`
`  if((++u) >= v) {`
`  u = 0;`
`  } else {`
`  x = 0;`
`  }`
`}`
	`if((s^t)) {`
	`  if((++u) >= v) {`
	`  u = 0;`
	`  x = 1;`
	`  }`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

代入文の直後で，その代入された値が参照されている場合，参照されている箇所を代入文で置換して 1 つにまとめます。

これにより，余分なレジスタ転送が削除されてコード・サイズが削減される可能性があります。

ただし，多くの場合では C コンパイラの最適化によって冗長なレジスタ転送が削除されているため，コード・サイズは変わりません。

変更前	変更後

--s;
if(s == 0) {
  :
}
	if(--s == 0)) {
	  :
	}

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

（6）if-else 文

以下の例のように，if-else 文の分岐先のおのおのが同じ変数へ異なる値を代入する文のみを含む場合，一方を if 文の前に移動することにより，else のブロックが削除されて if のブロックから else のブロックの後へのジャンプ命令が減るため，コード・サイズが削減される可能性があります。

変更前	変更後

`if(x == 10) {`
`  s = 1;`
`} else {`
`  s = 0;`
`}`
	`s = 0;`
	`if(x == 10) {`
	`  s = 1;`
	`}`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

以下の例のように，if-else 文の分岐先のおのおのが return 文のみを含み，その戻り値が分岐条件の結果そのものである場合，分岐条件の式の値を返すようして，if-else 文を削除します。

変更前	変更後

`if(s1 == s2) {`
`  return(1);`
`}`
`return(0);`
	`return(s1 == s2);`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおので，同一関数に対し異なる引数を用いて呼び出しを行っている場合，関数呼び出しを合流後に移動可能であるならば移動します。

このとき，元々の呼び出しの箇所では，その異なる引数をテンポラリ変数へ代入し，呼び出しにおいては，そのテンポラリ変数を引数として用います。

変更前	変更後

if(s) {
  :
  func(0, 1, 2);
} else {
  :
  func(0, 1, 3);
}
	int tmp;
	if(s) {
	  :
	  tmp = 2;
	} else {
	  :
	  tmp = 3;
	}
	func(0, 1, tmp);

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおのにおいて，同一の代入文や関数呼び出しが存在する場合には，分岐前に移動可能であれば移動します。

その文の評価結果が参照されている場合，いったんテンポラリ変数へ代入し，そのテンポラリを参照するようにします。

以下の例は関数呼び出しの場合です。

変更前	変更後

if(x >= 0) {
  if(x > func(0, 1, 2)) {
  :
  }
} else {
  if(x < -func(0, 1, 2)) {
  :
  }
}
	long tmp;
	
	tmp = func(0, 1, 2);
	if(x >= 0) {
	  if(x > tmp) {
	  :
	  }
	} else {
	  if(x < -tmp) {
	  :
	  }
	}

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

分岐後のおのおのにおいて，同一の代入文や関数呼び出しが存在する場合，分岐前に移動することが不可能であり，かつ，合流後に移動することは可能であるならば，合流後に移動します。

以下の例は代入文の場合です。

変更前	変更後

`if(tmp & MASK) {`
`  :`
`  j++;`
`} else {`
`  :`
`  j++;`
`}`
	`if(tmp & MASK) {`
	`  :`
	`} else {`
	`  :`
	`}`
	`j++;`

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(7) switch/if-else 文

以下の例のように，switch 文や if-else 文の分岐先のおのおので，同じ外部変数へ異なる値を代入する場合，おのおのの箇所ではいったんテンポラリ変数へ代入し，再び合流したあとにテンポラリ変数から元の外部変数へ代入を行うことにより，コード・サイズが削減される可能性があります。

これは，外部変数はレジスタに割り付けられることが少ないため，外部変数への代入はメモリへのストア命令（4 バイト）となる一方，テンポラリ変数への代入は，多くの場合，レジスタ転送（2 バイト）になるためです。

以下の例で s は外部変数であるとします。

変更前

```
switch(x) {
  case  0:
    s = 0;
    break;
  case  1:
    s = 0x5555;
    break;
  case  2:
    s = 0xAAAA;
    break;
  case  3:
    s = 0xFFFF;
}
```

変更後

```
int tmp;
if(x == 0) {
  tmp = 0;
} else if(x == 1) {
  tmp = 0x5555;
} else if(x == 2) {
  tmp = 0xAAAA;
} else if(x == 3) {
  tmp = 0xFFFF;
} else {
  goto  label;
}
s = tmp;
label:
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。

(8) 戻り値のない関数

戻り値のない関数は，“void”と宣言します。

2. 7. 2 変数の定義方法で変数領域を削減する

この項では，変数の定義方法で変数領域を削減する方法について説明します。

（1）変数の型

ANSI-C の仕様により，短整数（(unsigned) short,（unsigned）char）型の変数は，演算時に int 型，または unsigned int 型に拡張されるため，これらの変数を使用したプログラムに対しては（特にこれらの変数がレジスタに割り付けられた場合には），型変換命令が多く生成されます。

（unsigned）int 型にすればこの型変換が不要となるため，コード・サイズが削減されます。

特に，比較的レジスタに割り付けられやすいスタック変数は，できるだけ（unsigned）int 型を用いてください。

変更前	変更後

unsigned char i;
 	
for(i = 0; i < 4; i++) {
  array[2 + i] = *(p + i);
}
	int i;
	
	for(i = 0; i < 4; i++) {
	  array[2 + i] = *(p + i);
	}

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

(2) 自動変数への代入と参照

以下の例のように，スタック変数に値が代入位置と参照位置が離れている場合，その間，レジスタが占有されて他の変数がレジスタに割り付けられる機会が減ることになります。

このような場合には，実際に参照する直前に代入を行うように変更することにより，他の変数のレジスタ割り付けの機会が増えてメモリ・アクセスが減り，コード・サイズが小さくなります。

変更前

```
int i = 0, j = 0, k = 0, m = 0;

/* この間，関数呼び出しあり */
/* これらの変数の使用はなし */

while((k & 0xFF) != 0xFF) {
  k = s1;
  if(k & MASK) {
    if(m != 1) {
      s2 += 2;
      m = 1;
      array[15+i+j] = 0xFF;
      j++;
    }
  }
}
  :
```

変更後

```
int i, j, k, m;
  :
i = 0;
j = 0;
k = 0;
m = 0;
while((k & 0xFF) != 0xFF) {
  k = s1;
  if(k & MASK) {
    if(m != 1) {
      s2 += 2;
      m = 1;
      array[15+i+j] = 0xFF;
      j++;
    }
  }
}
  :
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

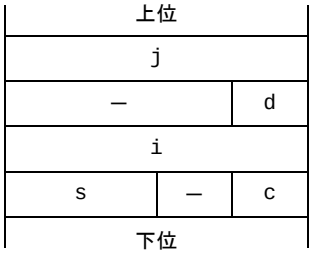
(3) 変数の型と定義順

定義は，データ長の長いものからまとめて行ってください。

V850 マイクロコントローラでは，int 型等のワード・データはワード境界，short 型等のハーフワード・データはハーフワード境界に整列している必要があります。

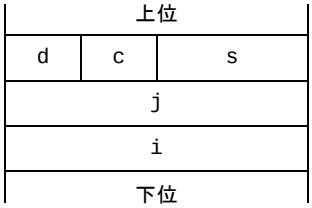
このため，次のようなソースに対しては整列のためのパディング領域が発生します。

```
char  c = 'c';
short s = 0;
int   i = 1;
char  d = 'b';
int   j = 2;
```



このようなパディング領域の発生を防ぐため，変数や構造体メンバの定義では，データ長の長いものからまとめて宣言してください。

```
int   i = 1;
int   j = 2;
short s = 0;
char  c = 'a';
char  d = 'b';
```



2.8 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。

2.8.1 記述方法で処理を高速化する

この項では，記述方法で処理を高速化する方法について説明します。

(1) ループ処理のポインタ

以下の例のように，ループの制御を行う変数を帰納変数（誘導変数）と呼びます。

「帰納変数の削除」とは，ループの制御を他の変数を用いるように変更することで帰納変数を削除する最適化です。

この最適化は CX でも実装されていますが，適用される条件が限られるため，すべての場合を最適化することはできません。

以下のようにプログラムを修正することにより，この最適化を“手動”で行うことができます。

以下の例では，テンポラリ変数（ポインタ）p を用いることによって，i を削除します。

変更前

```
int i;

for(i = 0; *(table + i) != NULL; ++i) {
  if((*(table + i) & 0xFF) == x) {
    return(*(table + i) & 0xFF00);
  }
}
```

変更後

```
const unsigned short  *p;

for(p = table; *p != NULL; ++p) {
  if((*p & 0xFF) == x) {
    return(*p & 0xFF00);
  }
}
```

備考 1. 削減量は，個々のケースにより異なります。

2. ソースを変更した結果，出力命令が減少し，高速化もされる場合があります。

3. ソース変更を行う際には，次のような点に注意してください。

- ソース変更によりレジスタの使用状況が変わるため，意図しない箇所において，それまで最適化されずに残っていたレジスタ転送が削除されたり，逆に，最適化が効かなくなって冗長なレジスタ転送が残ったりする可能性があります。
- テンポラリ変数を追加することにより，新たなレジスタ変数用レジスタが使用されるようになり，それに伴い，関数の入口／出口でそのレジスタの退避／復帰が追加されることがあります。この場合，退避／復帰のコード分（8 バイト）だけコード・サイズが増加します。

(2) auto 変数の宣言

auto 変数は，10 個以下（なるべく 6 ～ 7 個くらい）にしてください。

auto 変数はレジスタに割り当てられます。

CX では，1 つの関数内で，作業用レジスタとして 10 本，レジスタ変数用レジスタとして 10 本，合計 20 本のレジスタを変数用として使用可能です（32 レジスタ・モードの場合）。

1 つの関数内の処理が，時間のかかる処理であれば，auto 変数を多く使用することを推奨します。

あまり時間のかからない処理であれば，できるだけ作業用レジスタの 10 本のみを使用するようにしてください。

レジスタ変数用レジスタは，退避／復帰のオーバヘッドがかかります。

レジスタ変数を使用するかしないかは，C コンパイラが自動的に判断します。

したがって，auto 変数は 6 ～ 7 個とし，C コンパイラが作業用として使用可能なレジスタを 3 ～ 4 本残してください。

(3) 関数の引数

引数用のレジスタは，r6 ～ r9 の 4 個です。

引数が 5 個以上の場合は，5 個目以降はスタック領域を使用します。

したがって，引数はなるべく 4 個以内にしてください。

どうしても 5 個以上になる場合，構造体のポインタで引数を渡すようにします。

2.9 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。

2.9.1 変数を相互参照する

この項では，変数を相互参照する方法について説明します。

(1) C 言語で定義した変数を参照する

C 言語プログラム中で定義した外部変数をアセンブリ言語ルーチン中で参照する場合，extern 宣言します。

アセンブリ言語ルーチン中では，定義した変数の先頭に“_”（アンダースコア）を付けます。

例 C ソース

```
extern  void subf(void);
char    c = 0;
int     i = 0;

void main(void) {
    subf();
}
```

例 アセンブラ・ソース

```
        .public _subf
        .extern _c, 1
        .extern _i, 4
        .cseg   text
        .align  4
_subf:
        mov     4, r10
        st.b    r10, $_c
        mov     7, r10
        st.w    r10, $_i
        jmp     [lp]
```

(2) アセンブリ言語で定義した変数を参照する

アセンブリ言語プログラム中で定義した外部変数を C 言語ルーチン中で参照する場合，extern 宣言します。

アセンブリ言語ルーチン中で定義する変数の先頭に“_”（アンダースコア）を付けます。

例 C ソース

```
extern char c;
extern int i;

void subf(void) {
    c = 'A';
    i = 4;
}
```

例 アセンブラ・ソース

```
        .public _i, 4
        .public _c, 1
        .dseg   sbss
        .align  4
_i:
        .ds     (4)
_c:
        .ds     (1)
```

2. 9. 2 関数を相互参照する

この項では，関数を相互参照する方法について説明します。

(1) C 言語で定義した関数を参照する

アセンブリ言語ルーチンから C 言語により記述された関数を呼び出すときの注意点について説明します。

- スタック・フレーム

「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，アセンブラ関数から C 言語関数へ分岐する前に，スタック領域中の未使用領域の上位アドレスを指すように SP を設定してください。

- 作業用レジスタ

C 言語関数呼び出しの前後において，レジスタ変数用レジスタの値は保持しますが，作業用レジスタの値は保持しません。そのため，保持しなくてはならない値を作業用レジスタに割り当てたままにしないでください。

- アセンブラ関数への戻り先アドレス

「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。C 言語関数へ分岐するとき，lp に関数の戻り先アドレスを格納する必要あります。

(2) アセンブリ言語で定義した関数を参照する

C 言語により記述された関数からアセンブリ言語ルーチンを呼び出すときの注意点について説明します。

- 識別子

先頭に“_（アンダースコア）”を付けた名前になります。

- スタック・フレーム

「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，C ソースからアセンブラ関数へ分岐後は，アセンブラ関数内では，SP の指すアドレスよりも下位のアドレス領域は自由に使用することができます。逆に上位のアドレス領域の内容を変更した場合，C 言語関数で使用していた領域を破壊することにつながり，以降の動作は保証されません。上記を回避するためには，アセンブラ関数の先頭で SP を変更してからスタックを使用してください。

ただし，その際は呼び出しの前後で SP の値が保持されるようにしてください。

- レジスタ変数用レジスタ

アセンブラ関数内でレジスタ変数用レジスタを使用する場合は，アセンブラ関数の呼び出し前後でレジスタ値が保持されるようにしてください（使用前にレジスタ変数用レジスタの値を退避し，使用後は復帰してください）。

- C 言語関数への戻り先アドレス

「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。アセンブラ関数へ分岐するとき，lp に関数の戻り先アドレスが格納されているので，C 言語関数へ戻るときは“jmp [lp]”を実行してください。

第 3 章　コンパイラ言語仕様

この章では，CX がサポートする言語仕様について説明します。

3. 1　基本言語仕様

CX 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，V850 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。

また，厳密な ANSI 準拠処理のオプションを指定した場合と指定しない場合の差分についても説明します。

なお，CX 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

この節では，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. 3. 1　C 言語関数間の呼び出し」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

この節では，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) 関数呼び出しー適合しない型

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

const 修飾型で定義されたオブジェクトを非 const 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。キャストすることも禁止です。

(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

この節では，ANSI 規格における処理系依存項目について説明します。

(1) データ型とサイズ

ワード（4 バイト）中のバイト順序は，“下位から上位”です。また，符号付き整数は，2 の補数で表現します。最上位ビットには，符号（正，または 0 の場合 0，負の場合 1）が入ります。

- 1 バイト中のビット数は，8 ビットとします。
- オブジェクト中のバイト数，バイト順序，符号化は，次のように規定します。

表 3―1 データ型とサイズ

データ型	サイズ
char 型	1 バイト
short 型	2 バイト
int，long，float 型	4 バイト
double，long double，long long 型	8 バイト
ポインタ	unsigned int と同じ

(2) 翻訳段階

ANSI 規格では，翻訳における構文規則間の優先順位を 8 つの翻訳段階（翻訳フェーズ）に規定しています。3 段階目の“前処理字句と空白類文字の並びへの分割”で処理系定義となっている，“改行文字以外の空白類文字の空でない並び”は 1 つに置き換えられずそのまま保持されます。

(3) 診断メッセージ

何らかの構文規則違反，および制約違反を含む翻訳単位に対して，ソース・ファイル名，行番号（特定可能な場合のみ）を含むエラー・メッセージを出力します。なお，エラー・メッセージの書式は“ワーニング”，“アボート・エラー”，“フェイタル・エラー”などに区別されます。出力形式については，「CubeSuite メッセージ編」のユーザーズ・マニュアルを参照してくださいい。

(4) フリー・スタンディング環境

(a) フリー・スタンディング環境注においては，プログラム開始処理時に呼び出される関数の名前，および型は特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

注 オペレーティング・システムの機能を使用せずに C ソース・プログラムを実行する環境のことです。
ANSI 規格では，実行環境にはフリー・スタンディング環境とホスト環境の 2 つが規定されていますが，CX では，ホスト環境は現在提供されていません。

(b) フリー・スタンディング環境におけるプログラム終了処理の効果は，特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(5) プログラムの実行

対話型装置の構成については，特に規定しません。
したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(6) 文字集合

実行環境文字集合の要素の値は，ASCII コードです。

(7) 多バイト文字

利用できる多バイト文字は，EUC，SJIS，UTF-8 です。
コメントと文字列における日本語記述をサポートしています。

(8) 文字表示の意味

拡張表記の値は，次のように規定します。

表 3―2 拡張表記と意味

拡張表記	値（ASCII）	意味
\a	07	アラート（警告音）
\b	08	バックスペース
\f	0C	フォーム・フィード（改ページ）
\n	0A	ニュー・ライン（改行）
\r	0D	キャリッジ・リターン（復帰）
\t	09	水平タブ
\v	0B	垂直タブ

(9) 翻訳限界

リンク可能なファイル数は 2000 個となります。リンク時に 2000 個以上指定した場合，E0511138 エラーとなります。それ以外に翻訳に際しての限界値はありません。翻訳可能な最大値は動作するホスト・マシンのメモリに依存します。

(10) 数量的限界

(a) 汎整数型の限界値（limits.h ファイル）

汎整数型（char 型，符号付き／符号なし整数型，および列挙型）で表現できる値の各種限界値を limits.h ファイルに定義しています。

なお，多バイト文字はサポートしていないため，MB_LEN_MAX は該当する限界値を持ちません。そこで，MB_LEN_MAX には 1 として，定義のみ行っています。

また，CX の -Xchar=unsigned オプション（単なる char 型の符号を指定）が指定された場合，CHAR_MIN は 0，CHAR_MAX は UCHAR_MAX と同値となります。

次に，limits.h ファイルで定義されている各種限界値を示します。

表 3―3 汎整数型の各種限界値（limits.h ファイル）

名前	値	意味
CHAR_BIT	+8	ビット・フィールドではない最小のオブジェクトのビット数（= 1 バイト）
SCHAR_MIN	-128	signed char 型の最小値
SCHAR_MAX	+127	signed char 型の最大値
UCHAR_MAX	+255	unsigned char 型の最大値
CHAR_MIN	-128	char 型の最小値
CHAR_MAX	+127	char 型の最大値
SHRT_MIN	-32768	short int 型の最小値

名前	値	意味
SHRT_MAX	+32767	short int 型の最大値
USHRT_MAX	+65535	unsigned short int 型の最大値
INT_MIN	-2147483648	int 型の最小値
INT_MAX	+2147483647	int 型の最大値
UINT_MAX	+4294967295	unsigned int 型の最大値
LONG_MIN	-2147483648	long int 型の最小値
LONG_MAX	+2147483647	long int 型の最大値
ULONG_MAX	+4294967295	unsigned long int 型の最大値
LLONG_MIN	-9223372036854775807	long long int 型の最小値
LLONG_MAX	+9223372036854775807	long long int 型の最大値
ULLONG_MAX	18446744073709551615	unsigned long long int 型の最小値

(b) 浮動小数点型の各種限界値（float.h ファイル）

浮動小数点型の特性に関する各種限界値を float.h ファイルに定義しています。

次に，float.h ファイルで定義されている各種限界値を示します。

表 3―4 浮動小数点型の各種限界値の定義（float.h ファイル）

名前	値	意味
FLT_ROUNDS	+1	浮動小数点加算に対する丸めのモード 1（もっとも近い方向へ丸める）とします。
FLT_RADIX	+2	指数表現の基数（b）
FLT_MANT_DIG	+24	浮動小数点仮数部における FLT_RADIX を底とする数字の数（p）
DBL_MANT_DIG	+53	
LDBL_MANT_DIG	+53	
FLT_DIG	+6	q 桁の 10 進数の浮動小数点数を基数 b の p 桁をもつ浮動小数点数に丸めることができ，再び変更なしに q 桁の 10 進数値に戻すことができるような 10 進数の桁数（q）
DBL_DIG	+15	
LDBL_DIG	+15	
FLT_MIN_EXP	-125	FLT_RADIX をその値から 1 引いた値でべき乗したとき，正規化された浮動小数点数となるような最小の負の整数（e_{min}）
DBL_MIN_EXP	-1021	
LDBL_MIN_EXP	-1021	
FLT_MIN_10_EXP	-37	10 をその値でべき乗したとき，正規化された浮動小数点数の範囲内になるような最小の負の整数 $\log_{10}b^{e_{min-1}}$
DBL_MIN_10_EXP	-307	
LDBL_MIN_10_EXP	-307	

名前	値	意味
FLT_MAX_EXP	+128	FLT_RADIX をその値から 1 引いた値でべき乗したとき，表現可能な有限浮動小数点数となるような最大の整数 （e_{max}）
DBL_MAX_EXP	+1024	
LDBL_MAX_EXP	+1024	
FLT_MAX_10_EXP	+38	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP	+308	
LDBL_MAX_10_EXP	+308	
FLT_MAX	3.40282347E + 38F	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX	1.7976931348623158E+308	
LDBL_MAX	1.7976931348623158E+308	
FLT_EPSILON	1.19209290E - 07F	指定された浮動小数点型で表現できる 1.0 と，1.0 より大きい最も小さい値との差異 $b^{1 - p}$
DBL_EPSILON	2.2204460492503131E-016	
LDBL_EPSILON	2.2204460492503131E-016	
FLT_MIN	1.17549435E - 38F	正規化された正の浮動小数点数の最小値 $b^{e_{min - 1}}$
DBL_MIN	2.2250738585072014E-308	
LDBL_MIN	2.2250738585072014E-308	

（11）識別子

識別子すべてが意味のあるものとして扱います。また，識別子の長さは無制限です。

なお，英字の大文字と小文字は区別されます。

（12）char 型

型指定子（signed，unsigned）の付かない単なる char 型は，符号付き整数として扱います。

ただし，CX の -Xchar=unsigned オプションを指定することにより，符号なし整数として扱うこともできます。

ANSI 規格において要求されるソース・プログラムの文字集合に含まれないもの（エスケープ・シーケンス）は，char 型以外を char 型へ代入する場合と同様に，型変換して格納されます。

```
char    c = '\777';     /*c の値は -1 となる */
```

（13）浮動小数点定数

浮動小数点定数は，IEEE754 **注**に準拠しています。

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。

（14）文字定数

（a）ソース・プログラムの文字集合と実行環境における文字集合は，基本的に両者とも ASCII コードで，同一の値をもつメンバと対応します。
ただし，ソース・プログラムにおける文字列には，日本語文字コードが使用できます（「（8）文字表示の意味」を参照）。

（b）2 つ以上の文字を含む整数文字定数の値は，最後の 1 文字が有効値となります。

（c）基本的な実行環境文字集合で表現されない文字やエスケープ・シーケンスを含む場合，次のようになります。

- 8 進数エスケープ・シーケンス，および 16 進数エスケープ・シーケンスは，その 8 進数表記，および 16 進数表記で示される値となります。

\777	511

- 単純エスケープ・シーケンスは，次のようになります。

\'	'
\"	"
\?	?
\\	\

- \a，\b，\f，\n，\r，\t，\v については，「（8）文字表示の意味」で示されている値と同値になります。

（d）多バイト文字の文字定数はサポートしていません。

（15）文字列

文字列中に日本語が記述できます。
デフォルトの文字コードは，シフト JIS となります。
入力ソース・ファイルの中の文字コードは，CX の -Xcharacter_set オプションで選択できます。

【オプション指定】

-Xcharacter_set=[none | euc_jp | sjis | utf8]

（16）ヘッダ・ファイル名

ヘッダ・ファイル名の 2 つの形式（< >，" "）内の列を，ヘッダ・ファイル，または外部ソース・ファイル名に反映する方法は，「（33）ヘッダ・ファイル取り込み」で規定します。

(17) コメント

コメント中に日本語が記述できます。文字コード，「(15) 文字列」の場合と同じです。

(18) 符号付き定数と符号なし定数

汎整数型の値がよりサイズの小さい符号付き整数に変換される場合，上位ビットを切り捨てて，ビット列イメージをコピーします。

また，符号なし整数が，対応する符号付き整数に変換される場合，内部表現は変化しません。

(19) 浮動小数点と汎整数

汎整数型の値が浮動小数点型に型変換される際，型変換される値が，表現しうる値の範囲内にはあるが正確に表現することができない場合，その結果は，表現しうる最も近い値へ丸められます。

なお，結果がちょうど中央の値である場合には，偶数（仮数の最下位ビットが 0 のもの）に丸められます。

(20) double 型と float 型

CX コンパイラでは，double 型は 64 ビット・データ（倍精度）として，float 型は 32 ビット・データ（単精度）として扱われます。

(21) ビット単位の演算子における符号付き型

ビット単位の演算子における符号付き型に対する特性は，シフト演算子については，「(27) ビット単位のシフト演算子」の規定に準じます。

また，その他の演算子については，符号なしの値として（ビット・イメージのままで）計算するものとします。

(22) 構造体と共用体のメンバ

共用体のメンバの値がそれと異なるメンバに格納される場合，整列条件に従って格納されるため，その共用体のあるメンバへのアクセスは，整列条件に従って行われます（「(6) 構造体型」，および「(7) 共用体型」を参照）。

ただし，共通の先頭メンバの並びを共有している構造体だけをメンバとして含んでいる共用体の場合，内部表現は同じであるため，どの構造体の共通の先頭メンバを参照しても同じになります。

(23) sizeof 演算子

“sizeof”演算子の結果は，「(1) データ型とサイズ」におけるオブジェクト中のバイトに関する規定に準じます。

なお，構造体と共用体については，パディング領域を含んだバイト数とします。

(24) キャスト演算子

ポインタを汎整数型に変換する場合，要求される変数のサイズは，unsigned long 型と同じサイズです。変換結果は，ビット列がそのまま保存されます。

また，任意の整数はポインタに型変換できますが，int 型よりも小さい整数の場合，結果はその型に従って拡張されます。

（25）乗除／剰余演算子

整数同士の除算で割り切れず，オペランドが負の値をもつ場合，“/”演算子の結果は，除数，または被除数のいずれか一方が負の場合は，代数的な商よりも大きい最小の整数となります。

ただし，どちらも負の場合は，代数的な商よりも小さい最大の整数となります。

また，オペランドが負の値をもつ場合，“%”演算子の結果の符号は第 1 オペランドの符号とします。

（26）加減演算子

同一配列の要素を指す 2 つのポインタが減算される場合，結果の型は unsigned long 型となります。

（27）ビット単位のシフト演算子

“E1 >> E2”において，E1 が符号付きの型で負の値をもつ場合，算術シフトを行います。

（28）記憶域クラス指定子

記憶域クラス指定子“register”の宣言の有無にかかわらず，可能なかぎり高速にアクセスするように最適化を行います。

（29）構造体と共用体指定子

（a）signed，unsigned の付かない単なる int 型ビット・フィールドは，符号付きとして扱い，最上位ビットは符号ビットとして扱います。ただし，CX の -Xbitfield オプション（単なる int 型ビット・フィールドの符号を指定）を指定することにより，符号なしとして扱うこともできます。

（b）ビット・フィールドを保持するために，十分な大きさの任意のアドレス付け可能な記憶域単位を割り付けることができますが，十分な領域がなかった場合，合わなかったビット・フィールドはフィールドの型の整列条件に合わせて次の単位に詰め込まれます。

（c）単位内のビット・フィールドの割り付け順序は下位から上位です。

（d）1 つの構造体，または共用体の非ビット・フィールドの各メンバは，次のように境界整列されます。

char，unsignd char 型，およびその配列	バイト境界
short，unsigned short 型，およびその配列	2 バイト境界
その他（ポインタを含む）	4 バイト境界

（30）列挙型指定子

列挙型の型は，signed int 型とします。

ただし，-Xenum_type=auto オプション指定時は，各列挙型について，その型のすべての列挙子の値を表現可能な最小の整数型として扱います。

（31）型修飾子

“volatile”修飾された型をもつデータへのアクセスは，データがマッピングされているアドレス（I/O ポートなど）に依存します。

（32）条件組み込み

（a）条件組み込みで指定される文字定数に対する値と，その他の式中に現れる文字定数の値とは等しくなります。

（b）単一文字の文字定数は，負の値を持たないようにしてください。

（33）ヘッダ・ファイル取り込み

（a）“#include < 文字列 >”という形式の前処理指示

“#include < 文字列 >”という形式の前処理指示は，“文字列”がフルパスのファイル名でない場合**注**，指定されたフォルダ（-I オプション）からヘッダ・ファイルを検索し，次に標準インクルード・ファイル・フォルダ（cx が置かれた bin フォルダからの相対パスでの .. \ inc フォルダ）を検索します。

なお，“<”と“>”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “\”と“/”の両者がフォルダの区切りとしてみなされます。

例

```
#include    <header.h>
```

検索順は，次のとおりです。

- -I で指定したフォルダ
- 標準インクルード・ファイル・フォルダ

（b）“#include " 文字列 "”という形式の前処理指示

“#include " 文字列 "”という形式の前処理指示は，“文字列”がフルパスのファイル名でない場合**注**，ソース・ファイルがあるフォルダからヘッダ・ファイルを検索し，次に，指定したフォルダ（-I オプション），最後に標準インクルード・ファイル・フォルダ（cx が置かれた bin フォルダからの相対パスでの .. \ inc フォルダ）を検索します。

なお，“ " ”“ " ”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “\”と“/”の両者がフォルダの区切りとしてみなされます。

例

```
#include    "header.h"
```

検索順は，次のとおりです。

- ソース・ファイルがあるフォルダ
- -I で指定したフォルダ
- 標準インクルード・ファイル・フォルダ

(c) “#include 前処理字句列” という形式

“#include 前処理字句列” という形式において，前処理字句列が単一で < 文字列 >，または " 文字列 " の形式に置換されるマクロである場合にのみ，単一のヘッダ・ファイル名の前処理字句として扱われます。

(d) “#include < 文字列 >” という形式の前処理指示

（最終的に）区切られた列とヘッダ・ファイル名との間においては，列中の英文字の長さを判別し，

```
コンパイラ動作環境において有効なファイル名長までが有効
```

となります。ファイルを探すフォルダについては，上記の規定に準じます。

(34) #pragma 指令

CX では，次の #pragma 指令が指定できます。

(a) アセンブラ命令の記述

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

C 言語中に，アセンブラ命令を記述することができます。

なお，記述方法についての詳細は「(5) アセンブラ命令の記述」を参照してください。

(b) インライン展開指定

```
#pragma inline  関数名 [, 関数名 , …]
```

インライン展開する関数を指定することができます。

なお，インライン展開についての詳細は「(9) インライン展開」を参照してください。

(c) データ／プログラムのメモリ割り当て

```
#pragma section セクション種別    [" セクション名 "]
#pragma text    [" セクション名 " ][ 関数名 [, 関数名 ]…]
```

- section

変数を任意のセクションに割り当てます。

なお，配置方法についての詳細は「(2) データのセクション割り当て」を参照してください。

- text

任意の名称のテキスト・セクションに関数を指定できます。

なお，配置指定についての詳細は「(3) 関数のセクション割り当て」を参照してください。

(d) 周辺 I/O レジスタ名有効化指定

```
#pragma ioreg
```

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。周辺 I/O レジスタ名をそのまま用いてプログラミングする場合はこの #pragma 指令を指定する必要があります。

(e) 割り込み／例外ハンドラ指定

```
#pragma interrupt    割り込み要求名    関数名    [ 配置方法 ]  [ オプション [ オプション ]…]
```

割り込み／例外処理ハンドラを C 言語で記述します。

なお，記述方法については「(c) 割り込み／例外ハンドラの記述方法」を参照してください。

(f) 割り込み禁止関数指定

```
#pragma block_interrupt 関数名
```

関数全体を割り込み禁止にします。

(g) タスク指定

```
#pragma rtos_task    [ 関数名 ]
```

リアルタイム OS 上で動作するタスクを C 言語で記述します。

なお，記述方法についての詳細は「(a) タスクの記述」を参照してください。

(h) 構造体パッキング指定

```
#pragma pack([1248])
```

構造体パッキングを指定します。数値はパッキング値，すなわちメンバのアライメント値を指定します。数値には 1，2，4，8 が指定できます。数値を指定しない場合，デフォルト（8）[注]となります。

注　本バージョンではアライメント値“4”と“8”は同じになります。

(i) スマート・コレクション指定

```
#pragma smart_correct 関数名 関数名
```

スマート・コレクション機能を指定します。

なお，記述方法についての詳細は「(13) スマート・コレクション機能」を参照してください。

(35) あらかじめ定義されたマクロ名

以下に，サポートしているマクロ名を示します。

なお，“__”で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に“__”のある形式のマクロを利用するようにしてください。

表 3―5　サポートしているマクロ

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）。
__FILE__	仮定されたソース・ファイルの名前（文字列定数）。
__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）。
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような“hh : mm : ss”の型式をもつ文字列定数）。
__STDC__	10 進定数 1（-Xansi オプション指定時に定義）。[注]
__v850 __v850__	10 進定数 1。
__v850e __v850e__	10 進定数 1（CX で，ターゲット・デバイスに V850E*x* を指定した場合に定義）。
__v850e2 __v850e2__	10 進定数 1（CX で，ターゲット・デバイスに V850E2 を指定した場合に定義）。
__v850e2v3 __v850e2v3__	10 進定数 1（CX で，ターゲット・デバイスに V850E2V3 のインストラクション・セットをもつデバイスを指定した場合に定義）。

マクロ名	定義
__K0R __K0R__	10 進定数 1（CX で，ターゲット・デバイスに 78K0R を指定した場合に定義）。
__CX __CX__	10 進定数 1。
__CHAR_SIGNED__	10 進定数 1（-Xchar オプションで，符号つきを指定した場合，および -Xchar オプションを指定しない場合に定義）。
__CHAR_UNSIGNED__	10 進定数 1（-Xchar オプションで，符号なしを指定した場合に定義）。
__DOUBLE_IS_64BITS__	10 進定数 1。
__CPU マクロ _ __CPU マクロ __	ターゲット CPU を示すマクロで 10 進定数 1。 デバイス・ファイル中の「品種指定名」で示される文字列の先頭に“__”，末尾に“_”，または“__”を付けたものが定義されます。
レジスタ・モード・マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 レジスタ・モードと定義されるマクロは次のとおりです。 32 レジスタ・モード ： __reg32__ 26 レジスタ・モード ： __reg26__ 22 レジスタ・モード ： __reg22__ 汎用レジスタ・モード ： __reg_common__
__MULTI_CORE__	10 進定数 1（-Xmulti オプションを指定した場合に定義）。
__MULTI_CMN__ __MULTI_PE*n*__	10 進定数 1（-Xmulti オプションによるコア種別を指定した場合に定義（*n* は数値））。

注 -Xansi オプション指定時の処理については，「3. 1. 5 ANSI オプション」を参照してください。

3. 1. 4 C99 言語機能

この節では，CX がサポートする C99 言語機能について説明します。

(1) 可変個数引数マクロ

C プリプロセッサの可変個数引数マクロをサポートします。

```
#define pr(fmt, ...)          printf(fmt, __VA_ARGS__)
```

上記のような記述ができ，任意の個数の引数を書くことができます。

```
pr("%s%d\n", "aa", 1)   ->    printf("%s%d\n", "aa", 1)
pr("%d\n", 2)           ->    printf("%d\n", 2)
```

(2) _Bool 型

_Bool 型をサポートします。

(3) // によるコメント

// （スラッシュ 2 つ）より始まり改行までをコメントとします。改行の直前の文字が￥の場合，次の行も続いた 1 つのコメントとします。

(4) inline キーワード（inline 関数）

inline 関数をサポートします。

また，以下の書式による pragma 指令でも指定できます。

```
#pragma inline　関数名 [, 関数名 , …]
```

なお，インライン展開についての詳細は「(9) インライン展開」を参照してください。

(5) long long int 型

long long int 型をサポートします。long long int 型は 8 バイトの整数型です。

定数値の末尾 LL もサポートします。ビットフィールドの型にも指定可能です。

(6) enum 定義の最後の列挙子の後のカンマ許可

enum 型を定義する際，列挙子の列挙の最後の”，（カンマ）”を許可します。

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

3. 1. 5　ANSI オプション

CX で -Xansi オプションを指定した場合，ANSI 規格に厳密な処理が行われます。

次に，-Xansi オプションを指定した場合と，指定しない場合の処理の違いを示します。

表 3―6　言語仕様に厳密な -Xansi オプション指定時の処理

項目	-Xansi 指定あり	-Xansi 指定なし
ビット・フィールド	ビット・フィールドに int 型以外の型を指定した場合，エラー[注 1] とします。	許可します。
# 行番号	エラーとします。	“#line”行番号と同様に扱います。[注 2]
行の途中の # 文字	“#”文字が行の途中で現れた場合，エラーとします。	警告メッセージを出力し，許可します。
__STDC__	値が 1 のマクロとして定義します。	定義しません。
2 進定数	“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びをエラーとします。	“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びを 2 進定数とします。

注 1. “E”で始まる通常のエラー。以下同じです。

2. ANSI 規格を参照してください。

3. 1. 6 データの内部表現と領域

この項では，CX が扱うデータのそれぞれの型における，内部表現と値域について説明します。

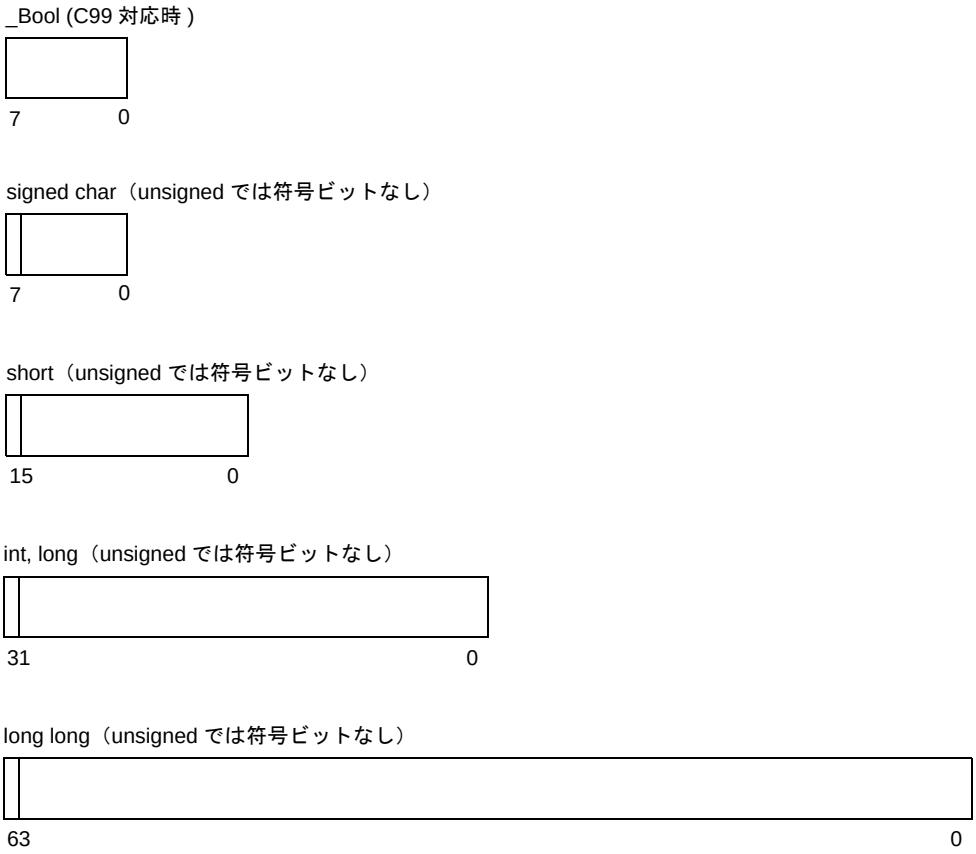
(1) 整数型

(a) 内部表現

領域の左端ビットは，符号付きの型（“unsigned”を伴わずに宣言された型）では，符号ビットとなります。符号付きの型において，値は 2 の補数表現で表されます。

ただし，-Xchar=unsigned が指定された場合，“signed”も“unsigned”も伴わずに宣言された char 型は，符号なし（unsigned）となります。

図 3―1 整数型の内部表現



(b) 値域

表 3―7 整数型の値域

型	値域
char 注	-128 ～ +127
short	-32768 ～ +32767
int	-2147483648 ～ +2147483647
long	-2147483648 ～ +2147483647
long long	-9223372036854775807 ～ +9223372036854775807
unsigned char	0 ～ 255
unsigned short	0 ～ 65535
unsigned int	0 ～ 4294967295
unsigned long	0 ～ 4294967295
unsigned long long	0 ～ 18446744073709551615

注 CX で“-Xchar=unsigned”が指定された場合，0 ～ 255 の値域です。

(2) 浮動小数点型

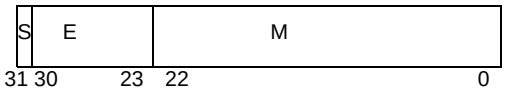
(a) 内部表現

浮動小数点型データの内部表現は，IEEE754 注に準拠しています。領域の左端のビットは，符号ビットとなります。この符号ビットの値が 0 であれば正の値に，1 であれば負の値になります。

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。

図 3―2　浮動小数点型の内部表現

float

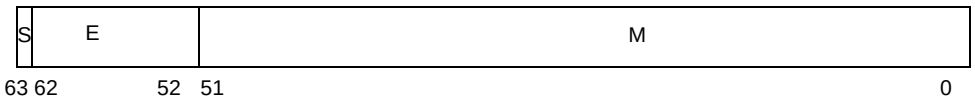


S： 仮想部の符号ビット

E： 指数部（8 ビット）

M： 仮想部（23 ビット）

double



S： 仮想部の符号ビット

E： 指数部（11 ビット）

M： 仮想部（51 ビット）

（b）値域

表 3―8　浮動小数点型の値域

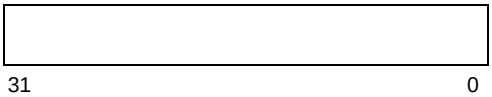
型	値域
float	$1.18 \times 10^{-38} \sim 3.40 \times 10^{38}$
double	$2.23 \times 10^{-308} \sim 1.80 \times 10^{308}$

（3）ポインタ型

（a）内部表現

ポインタ型の内部表現は，unsigned int 型の内部表現と同じです。

図 3―3　ポインタ型の内部表現

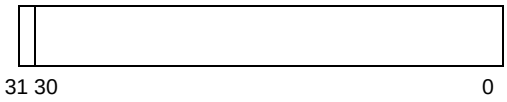


（4）列挙型

（a）内部表現

列挙型の内部表現は，singed int 型の内部表現と同じです。領域の左端のビットは，符号ビットとなります。

図 3―4 列挙型の内部表現



-Xenum_type=*string* オプション指定時には，「（30）列挙型指定子」を参照してください。

（5）配列型

（a）内部表現

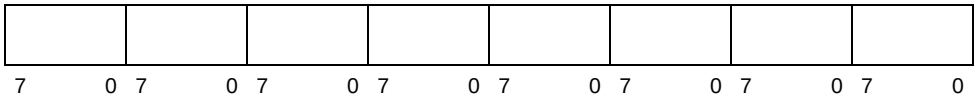
配列型の内部表現は，配列の要素を，その要素の整列条件（alignment）を満たす形で並べたものとなります。

例

```
char    a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
```

上記の例に示した配列に対する内部表現は，次のようになります。

図 3―5 配列型の内部表現



（6）構造体型

（a）内部表現

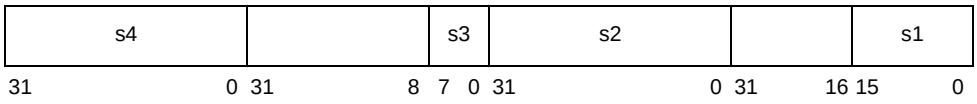
構造体型の内部表現は，構造体の要素をその要素の整列条件を満たす形で並べたものとなります。

例

```
struct {
    short   s1;
    int     s2;
    char    s3;
    int     s4;
} tag;
```

この例に示した構造体に対する内部表現は，次のようになります。

図 3―6 構造体型の内部表現



なお，構造体パッキング機能利用時の内部表現は，「（12）構造体パッキング」を参照してください。

（7）共用体型

（a）内部表現

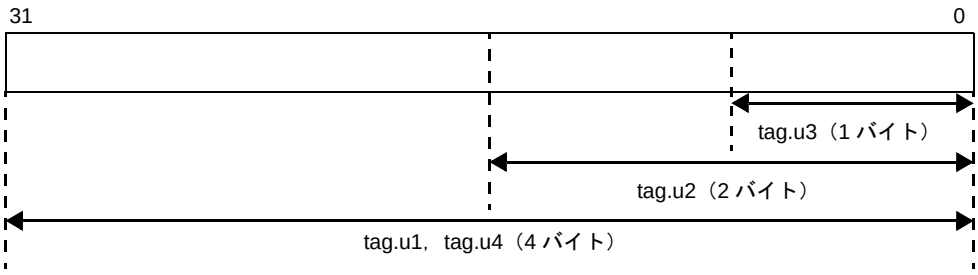
共用体はそのメンバがすべてオフセット 0 から始まり，そのメンバの任意のものを収容するのに十分なサイズを持つ構造体と考えられます。つまり，共用体型の内部表現は，同じアドレスに共用体の要素それぞれが単体で置かれているのと同様です。

例

```
union {
    int     u1;
    short   u2;
    char    u3;
    long    u4;
} tag;
```

この例に示した共用体に対する内部表現は，次のようになります。

図 3―7 共用体型の内部表現



（8）ビット・フィールド

（a）内部表現

ビット・フィールドに対しては，宣言された数のビットを含む領域が取られます。符号付きの型として宣言されたビット・フィールドに対しては，最上位ビットは符号ビットとなります。

最初に宣言されたビット・フィールドは，4 バイト領域の最下位ビットから割り当てられます。ビット・フィールドに対し，その前のビット・フィールドに続けて領域を割り当てると，その領域がそのビッ

ト・フィールドの宣言において指定された型の整列条件を満たす境界を越えてしまう場合，そのビット・フィールドに対する領域はその整列条件を満たしている境界から割り当てられます。

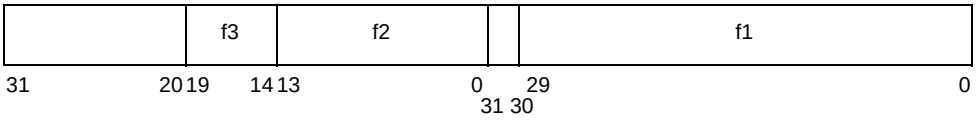
ただし，long long 型のビット・フィールドの場合，整列条件である 4 バイトではなく，long long 型の領域幅である 64 ビットを越えてしまう場合，次の境界から割り当てられます。

例

```
struct {
    unsigned long   f1:30;
    int             f2:14;
    unsigned int    f3:6;
} flag;
```

この例に示したビット・フィールドに対する内部表現は，次のようになります

図 3―8 ビット・フィールドの内部表現



なお，ANSI 規格ではビット・フィールドに char 型，および short 型は指定できませんが，CX では，ビット・フィールドに char 型，short 型，long 型，long long 型，およびそれらの unsigned 型を指定することができます。

なお，構造体パッキング機能利用時のビットフィールドの内部表現は，「(12) 構造体パッキング」を参照してください。

(9) 整列条件

(a) 基本型に対する整列条件

次に，基本型に対する整列条件を示します。

ただし，CX の -Xinline_strcpy を指定した場合，配列型はすべて 4 バイト境界となります。

表 3―9 基本型に対する整列条件

基本型	整列条件
(unsigned) char とその配列型	バイト境界
(unsigned) short とその配列型	2 バイト境界
(ポインタを含む) その他の基本型	4 バイト境界

(b) 共用体型に対する整列条件

共用体型に対する整列条件は，最大メンバ・サイズにより，次のようになります。

表 3―10 基本型に対する整列条件

最大メンバ・サイズ	整列条件
2 バイト＜サイズ	4 バイト境界
サイズ≦ 2 バイト	最大メンバ・サイズ境界

それぞれの場合における例を示します。

例 1.

```
union   tug1 {
    unsigned short  i;  /*2 バイト・メンバ */
    unsigned char   c;  /*1 バイト・メンバ */
};  /* 共用体は 2 バイトで整列 */
```

2.

```
union   tug2 {
    unsigned int    i;  /*4 バイト・メンバ */
    unsigned char   c;  /*1 バイト・メンバ */
};  /* 共用体は 4 バイトで整列 */
```

(c) 構造体型に対する整列条件

構造体型に対する整列条件は，構造体を構成するメンバのうち，最大の整列条件をもつ型の整列条件と同じになります。

ただし，CX の -Xinline_strcpy を指定した場合，配列型はすべて 4 バイト境界となります。

それぞれの場合における例を示します。

例 1.

```
struct  ST {
    char    c;      /*1 バイト・メンバ */
};  /* 構造体は 1 バイトで整列 */
```

2.

```
struct  ST {
    char    c;      /*1バイト・メンバ*/
    short   s;      /*2バイト・メンバ*/
};  /*構造体は2バイトで整列*/
```

3.

```
struct  ST {
    char    c;      /*1バイト・メンバ*/
    short   s;      /*2バイト・メンバ*/
    short   s2;     /*2バイト・メンバ*/
};  /*構造体は2バイトで整列*/
```

4.

```
struct  ST {
    char    c;      /*1バイト・メンバ*/
    short   s;      /*2バイト・メンバ*/
    int     i;      /*4バイト・メンバ*/
};  /*構造体は4バイトで整列*/
```

5.

```
struct  ST {
    char        c;      /*1バイト・メンバ*/
    short       s;      /*2バイト・メンバ*/
    int         i;      /*4バイト・メンバ*/
    long long   ll;     /*4バイト・メンバ*/
};  /*構造体は4バイトで整列*/
```

(d) 関数引数に対する整列条件

関数引数に対する整列条件は，4バイト境界となります。

(e) 実行プログラムに対する整列条件

リロケータブルなオブジェクト・モジュール・ファイルをリンクして実行可能なオブジェクト・モジュール・ファイルを生成する際の整列条件は，2バイト境界となります。

3. 1. 7　汎用レジスタ

以下に，CX における汎用レジスタの用い方を示します。

なお，汎用レジスタには，次の機能があります。

(1) ソフトウエア・レジスタ・バンク

作業用レジスタ（r10 ～ r19），およびレジスタ変数用レジスタ（r20 ～ r29）は，CX の -Xreg_mode オプションにより使用本数を抑制できます（「3. 1. 9　ソフトウエア・レジスタ・バンク」を参照）。

表 3―11　汎用レジスタの用い方

レジスタ		使用方法
r0	ゼロ・レジスタ	0 の値として演算時に使用 また，const セクション（ROM 領域に置く読み出し専用セクション）注などに配置されたデータの参照にも使用
r1	アセンブラ予約レジスタ	アセンブラにおける命令展開時に使用
r2（hp）	ハンドラ・スタック・ポインタ	システム予約
r3（sp）	スタック・ポインタ	スタック・フレームの先頭を指すものとして使用
r4（gp）	グローバル・ポインタ	外部変数の参照時に使用
r5（tp）	テキスト・ポインタ	テキスト・セクション（.text セクション）の先頭を指すものとして使用
r6 ～ r9	引数用レジスタ	引数の受け渡しに使用
r10 ～ r19	作業用レジスタ	演算時の作業用の領域として使用（r10 は関数の戻り値の受け渡しにも使用）
r20 ～ r29	レジスタ変数用レジスタ	レジスタ変数用の領域として使用
r30（ep）	エレメント・ポインタ	内蔵 RAM や外部 RAM のセクション注に配置指定された外部変数の参照に使用
r31（lp）	リンク・ポインタ	関数の戻り先アドレスの受け渡しに使用

注　データのセクションへの配置については，「(2) データのセクション割り当て」を参照してください。

3. 1. 8 データの参照方法

次に，CX におけるデータの参照方法を示します。

表 3―12 データの参照方法

種類	参照方法
数値定数	イミーディエト
文字列定数	グローバル・ポインタ（gp）からのオフセット エレメント・ポインタ（ep）からのオフセット r0 からのオフセット
自動変数，引数	スタック・ポインタ（sp）からのオフセット
外部変数，関数内静的変数	グローバル・ポインタ（gp）からのオフセット エレメント・ポインタ（ep）からのオフセット r0 からのオフセット
関数のアドレス	テキスト・ポインタ（tp）からのオフセットを用いて実行時に演算

3. 1. 9 ソフトウエア・レジスタ・バンク

CX では，ソフトウエアによるレジスタ・バンク機能を実現するため，3 つのレジスタ・モードが提供されています。レジスタ・モードを効率的に指定することにより，割り込み処理時やタスク切り替え時に，一部のレジスタの退避，復帰処理が不要となり，処理速度が高められます。レジスタ・モードの指定は，CX のレジスタ・モード指定オプション（-Xreg_mode）によって行います。この機能は，CX が内部で使用するレジスタの本数を段階的に抑制し，次の効果が期待できます。

- 余ったレジスタをアプリケーション・プログラム（アセンブラ・ソース・プログラム）で自由に使うことができる。
- 退避，復帰で生じるオーバーヘッドが減少する。

注意 CX によるレジスタ割り付けの対象となる変数の多いアプリケーション・プログラムでは，レジスタ・モードの指定によって，それまでレジスタに割り付けられていた変数がメモリ・アクセスとなり，その分，処理速度が低下することがあります。

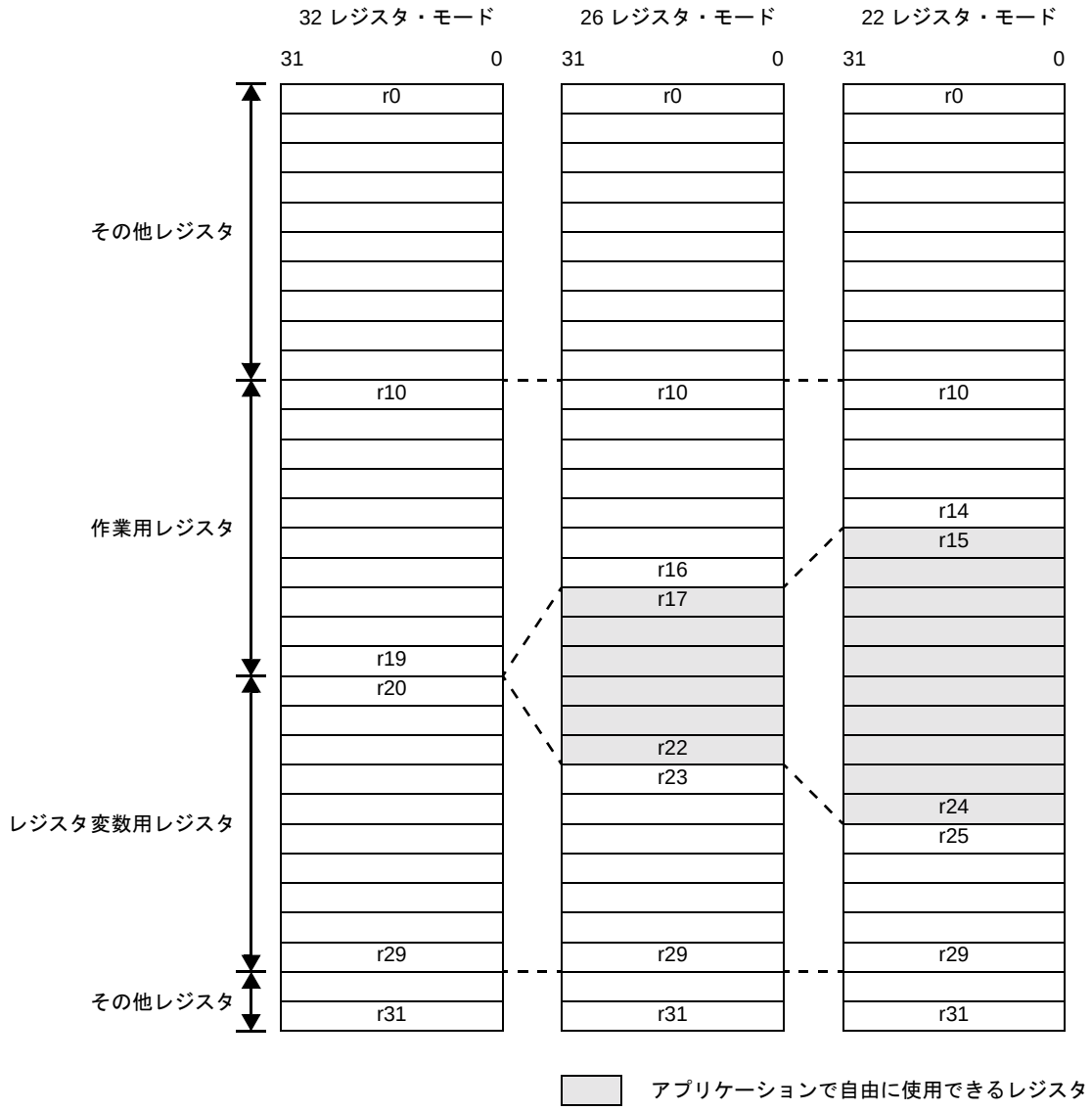
(1) レジスタ・モード

次表，および次図に，CX のレジスタ・モードとして提供されている 3 つのモードを示します。

表 3―13 CX が提供するレジスタ・モード

レジスタ・モード	作業用レジスタ	レジスタ変数用レジスタ
32 レジスタ・モード（デフォルト）	r10 ～ r19	r20 ～ r29
26 レジスタ・モード	r10 ～ r16	r23 ～ r29
22 レジスタ・モード	r10 ～ r14	r25 ～ r29

図 3―9 レジスタ・モードと使用可能レジスタ



コマンド・ラインにおける指定例

```
> cx -Cf3507 -Xreg_mode=26 file.c      ← 26 レジスタ・モードでコンパイル
```

3. 1. 10 デバイス・ファイル

デバイス・ファイルとは，ターゲット・デバイスの各品種ごと，または各グループごとに１つずつ，パッケージとして用意された，機種依存情報を持つバイナリ・ファイルです。コンパイラでは，アプリケーション・システムで使用するターゲット・システムに対応したオブジェクト・コードを生成するために，デバイス・ファイルを参照します。このため，使用するデバイス・ファイルは，デバイス・ファイルの標準フォルダに置くか，デバイス・ファイルの置かれているフォルダをコンパイラのオプションで指定するようにしてください。デバイス・ファイルが見つからないと，コンパイル時にエラーとなります。

(1) デバイス・ファイルの指定方法

C 言語のプログラムで参照するデバイス・ファイルの指定方法には，次の方法があります。

(a) コンパイラのオプション（-C *デバイス名*）によるデバイス名指定

例

```
> cx -Cf3507 file.c
```

CubeSuite でビルドする場合，デバイス指定することでこのオプションと同等になります。

上記の例では，“f3507”がデバイス名（V850E2/PJ4）です。“デバイス名”として指定できる文字列は，大文字，小文字は区別しません。

なお，デバイス名として指定できる文字列については，各デバイスのユーザーズ・マニュアルを参照してください。

(2) デバイス・ファイル指定時の注意

(a) デバイス名を指定しない場合

-C オプションによる指定がなく，-Xcommon=v850e オプション，-Xcommon=v850e2 オプション，または -Xcommon=v850e2v3 オプション[注]の指定がない場合，コンパイラはエラー・メッセージを出力し，コンパイルを中止します。ただし，-V/-h/-P オプションが指定されている場合はエラーとなりません。

注 -Xcommon=v850e オプション，-Xcommon=v850e2 オプション，または -Xcommon=v850e2v3 オプションを指定した場合でも，リンク時にはデバイス・ファイルが必要となります。

(b) アセンブラ命令で記述したプログラムの場合

この場合も，リンク可能なオブジェクト・モジュール・ファイル作成時に，アセンブラのオプション，または PROCESSOR 制御命令によるデバイス指定が必要です。

3. 2 拡張言語仕様

この節では，CX 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，データ／テキストのセクション配置指定や，デバイス内蔵の周辺 I/O レジスタのアクセスを C 言語レベルで行う方法，C 言語にアセンブラ命令の記述を挿入する方法，関数ごとにインライン展開を指定する方法，割り込みや例外発生時のハンドラの定義，割り込み禁止指定を C 言語レベルで行う方法，ターゲット環境にリアルタイム OS を使用した場合に有効なリアルタイム OS 用機能，C 言語への命令の組み込みなどがあります。

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。

なお，“__”で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に“__”のある形式のマクロを利用するようにしてください。

表 3―14 サポートしているマクロ

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）。
__FILE__	仮定されたソース・ファイルの名前（文字列定数）。
__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）。
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような“hh:mm:ss”の型式をもつ文字列定数）。
__STDC__	10 進定数 1（-Xansi オプション指定時に定義[注]）。
__v850 __v850__	10 進定数 1。
__v850e __v850e__	10 進定数 1（CX で，ターゲット・デバイスに V850Ex を指定した場合に定義）。
__v850e2 __v850e2__	10 進定数 1（CX で，ターゲット・デバイスに V850E2/xxx を指定した場合に定義）。
__v850e2v3 __v850e2v3__	10 進定数 1（CX で，ターゲット・デバイスに V850E2V3 のインストラクション・セットをもつデバイスを指定した場合に定義）。
__K0R __K0R__	10 進定数 1（CX で，ターゲット・デバイスに 78K0R を指定した場合に定義）。
__CX __CX__	10 進定数 1。
__CHAR_SIGNED__	10 進定数 1（-Xchar オプションで，符号つきを指定した場合，および -Xchar オプションを指定しない場合に定義）。
__CHAR_UNSIGNED__	10 進定数 1（-Xchar オプションで，符号なしを指定した場合に定義）。
__DOUBLE_IS_64BITS__	10 進定数 1。

マクロ名	定義
CPU マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 デバイス・ファイル中の「品種指定名」で示される文字列の先頭と末尾に“__”を付けたものが定義されます。
レジスタ・モード・マクロ	ターゲット CPU を示すマクロで 10 進定数 1。 レジスタ・モードと定義されるマクロは次のとおりです。 32 レジスタ・モード ： __reg32__ 26 レジスタ・モード ： __reg26__ 22 レジスタ・モード ： __reg22__ 汎用レジスタ・モード：__reg_common__
_MULTI_CORE__	10 進定数 1（-Xmulti オプションを指定した場合に定義）。
__MULTI_CMN__	10 進定数 1（-Xmulti=cmn オプションを指定した場合に定義）。
__MULTI_PE*n*__	10 進定数 1（-Xmulti=pe*n* オプションを指定した場合に定義）。

注 -Xansi オプション指定時の処理については，「3. 1. 5 ANSI オプション」を参照してください。

3. 2. 2 キーワード

CX 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，CX で追加されているキーワード一覧を示します。

_bsh，_bsw，__caxi，data，__DI，__EI，_halt，_hsw，__ldgr，__ldsr，__mul，__mulu，_mul32，_mul32ut，_nop，_sasf，_satadd，_satsub，__sch0l，__sch0r，__sch1l，__sch1r，__set_il，__stgr，__stsr，_sxb，_sxh

3. 2. 3 #pragma 指令

CX では，次の #pragma 指令が指定できます。

(1) アセンブラ命令の記述

C 言語中に，アセンブラ命令を記述することができます。

なお，記述方法についての詳細は「(5) アセンブラ命令の記述」を参照してください。

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

(2) インライン展開指定

インライン展開する関数を指定することができます。

なお，インライン展開についての詳細は「(9) インライン展開」を参照してください。

```
#pragma inline  関数名 [, 関数名 , …]
```

(3) データ／プログラムのメモリ割り当て

(a) section

変数を任意のセクションに割り当てます。

なお，配置方法についての詳細は「(2) データのセクション割り当て」を参照してください。

(b) text

任意の名称のテキスト・セクションに関数を指定できます。

なお，配置指定についての詳細は「(3) 関数のセクション割り当て」を参照してください。

```
#pragma section セクション種別　　[" セクション名 "]
#pragma text　　[" セクション名 "] [ 関数名 [, 関数名 ]…]
```

(4) 周辺 I/O レジスタ名有効化指定

周辺 I/O レジスタ名を用いて，デバイスの持つ周辺 I/O レジスタにアクセスします。周辺 I/O レジスタ名をそのまま用いてプログラミングする場合はこの #pragma 指令を指定する必要があります。

```
#pragma ioreg
```

(5) 割り込み／例外ハンドラ指定

割り込み／例外処理ハンドラを C 言語で記述します。

なお，記述方法については「(c) 割り込み／例外ハンドラの記述方法」を参照してください。

```
#pragma interrupt　割り込み要求名　　関数名　[ 配置方法 ]　[ オプション [ オプション ]…]
```

(6) 割り込み禁止関数指定

関数全体を割り込み禁止にします。

```
#pragma block_interrupt 関数名
```

(7) タスク指定

リアルタイム OS 上で動作するタスクを C 言語で記述します。

なお，記述方法についての詳細は「(a) タスクの記述」を参照してください。

```
#pragma rtos_task　[ 関数名 ]
```

(8) 構造体パッキング指定

構造体パッキングを指定します。数値はパッキング値，すなわちメンバのアライメント値を指定します。数値には 1，2，4，8 が指定できます。数値を指定しない場合，デフォルト 8 [注]となります。

```
#pragma pack([1248])
```

注　本バージョンではアライメント値“4”と“8”は同じになります。

(9) スマート・コレクション指定

スマート・コレクション機能を指定します。

なお，記述方法についての詳細は「(13) スマート・コレクション機能」を参照してください。

```
#pragma smart_correct 関数名 関数名
```

(10) ポジション・インディペンデント・アクセス

ポジション・インディペンデント・アクセスを指定します。指定されると，それ以降に宣言，定義された変数／関数へのアクセスは相対アドレスになります。

なお，記述方法についての詳細は「(14) ポジション・インディペンデント操作」を参照してください。

```
#pragma pic
```

(11) 固定アドレス・アクセス

固定アドレス・アクセスを指定します。指定されると，それ以降に宣言，定義された変数／関数へのアクセスは絶対アドレスになります。

なお，記述方法についての詳細は「(14) ポジション・インディペンデント操作」を参照してください。

```
#pragma nopic
```

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。

- 定数値記述
- データのセクション割り当て
- 関数のセクション割り当て
- 周辺 I/O レジスタへのアクセス
- アセンブラ命令の記述
- 割り込みレベルの制御
- 割り込み禁止
- 割り込み／例外処理ハンドラ
- インライン展開
- リアルタイム OS 対応機能
- 組み込み関数
- 構造体パッキング
- スマート・コレクション機能
- ポジション・インディペンデント操作

(1) 定数値記述

CX は，2 進定数，8 進定数を書くことが可能です。2 進定数は，“0b”，または“0B”と，その後ろに続く 1 個以上の“0”，または“1”の数字の並びで構成されます。8 進定数は“0o”と，その後ろに続く 1 個以上の“0”～“7”の数字の並びで構成されます。

例

```
0b00010110111110101011111010010111
0o001726354
```

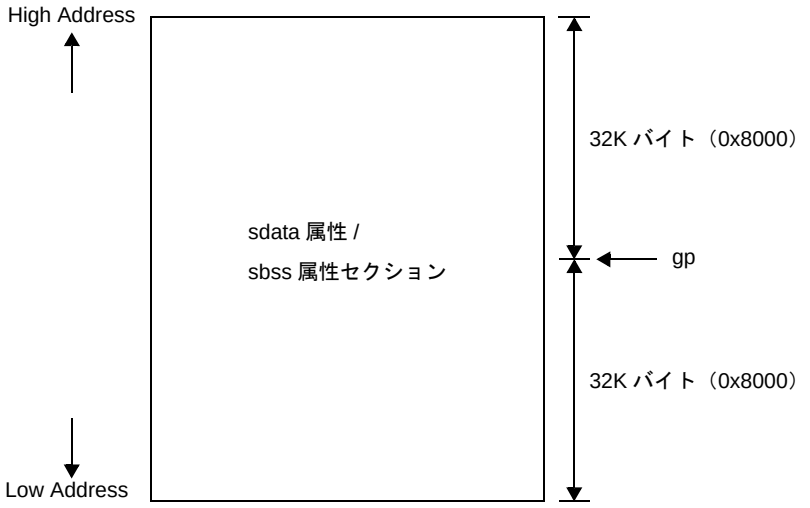
2 進定数，8 進定数を使用した場合に，-Xansi オプション指定すると，エラー・メッセージが出力されます。

(2) データのセクション割り当て

CX は，C ソース上で外部変数やデータが定義されると，それらをメモリ上に配置します。配置されるメモリ領域は，基本的に“グローバル・ポインタ（gp）”の指すアドレスからのオフセットによって参照できる領域です。つまり，プログラム中で変数やデータにアクセスする場合，デフォルトで gp を使ってアクセスするコードを出力しようとします。

さらに CX は，できるだけオブジェクト効率や実行効率を上げるため，gp から 1 命令で参照可能な領域に配置するコードを出力しようとします。しかし，gp から 1 命令で参照可能な範囲は，V850 アーキテクチャ上，gp から± 32K バイト内（合計で 64K バイト内）である必要があります。CX では，gp から± 32K バイト内の領域に専用のセクションを設けており，“sdata 属性 /sbss 属性セクション”と呼びます。

図 3―10 sdata 属性 /sbss 属性セクション



しかし，変数の数が多いアプリケーションでは，この範囲内に収まりきらない場合があります。その場合は，それ以外のセクションに割り当てなくてはなりません。CX では sdata 属性 /sbss 属性セクション以外にも，変数やデータを配置するためのさまざまなセクションを用意しています。それぞれに特徴があり，より高速にアクセスしたい変数を配置できるセクションなどもあるので，用途によって使い分けることができます。

次に，sdata 属性 /sbss 属性セクションを含め，CX で使用できるセクションを示します。

- sdata 属性 /sbss 属性セクション

  gp から 1 命令で参照可能なセクションで，gp から± 32K バイト内に配置される必要があります。初期値ありデータは sdata 属性セクションへ，初期値なしデータは sbss 属性セクションへ配置されます。

  CX では，まずこのセクションに配置するコードを生成しようとします。

ただし，この属性のセクションに収まりきらないような場合は，data 属性 /bss 属性セクションへ配置するコードを生成します。
なお，sdata 属性 /sbss 属性セクションへの配置データを少しでも多くする方法として，CX のオプション“-Xsdata”で，配置されるデータのサイズの上限を指定し，それ以上のサイズのデータは sdata 属性 /sbss 属性セクションに配置しないという指定ができます（オプションの詳細は「CubeSuite ビルド編（CX コンパイラ）」を参照）。
なお，プログラム中で sdata 属性 /sbss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「（a）#pragma section 指令」を参照）。

```
#pragma section sdata
int a = 1;  /*sdata セクション配置 */
int b;      /*sbss セクション配置 */
#pragma section default
```

- data 属性 /bss 属性セクション

gp から 2 命令で参照可能なセクションです。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。
したがって，RAM 上であれば，どこにでも配置が可能なセクションです。
なお，C 言語プログラム中で data 属性 /bss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「（a）#pragma section 指令」を参照）。

```
#pragma section data
int a = 1;  /*data セクション配置 */
int b;      /*bss セクション配置 */
#pragma section default
```

- sconst 属性セクション

r0，つまり，0 番地から 1 命令で参照可能なセクションで，0 番地から＋ 32K バイト内に配置される必要があります。基本的に“ROM に固定してもよいデータ”を配置するセクションです。V850 で内蔵 ROM を持つデバイスの場合，0 番地からプラス方向が内蔵 ROM である場合が多く，そこに 1 命令で参照したい，かつ ROM 固定してもよいデータを，sconst 属性セクションとして配置します。sconst 属性セクションに配置するデータは，const 修飾子をつけて宣言された変数／データが対象となります。この属性のセクションに収まりきらないような場合は，const 属性セクションへ配置することになります。
なお，sconst 属性セクションへの配置データを少しでも多くする方法として，CX のオプション“-Xsconst”で，配置されるデータのサイズの上限を指定し，それ以上のサイズは sconst 属性セクションに配置しないという指定ができます（オプションの詳細は「CubeSuite ビルド編（CX コンパイラ）」を参照）。
なお，プログラム中で sconst 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「（a）#pragma section 指令」を参照）。

```
#pragma section sconst
const int   a = 1;  /*sconst セクション配置 */
#pragma section default
```

- const 属性セクション

r0，つまり，0 番地から 2 命令で参照可能なセクションです。sconst 属性セクションに入りきらなかった“ROM 固定してもよいデータ”や，V850 の ROM レス品で，外部 ROM にデータを配置したい場合に，const 属性セクションに配置します。const 属性セクションに配置するデータは const 修飾子をつけて宣言された変数／データが対象になります。

また，const 修飾子をつけて宣言された変数，文字列定数は #pragma section 指令で .const セクションに配置する指令がない場合でも const 属性セクションに割り付けられます。アドレス生成を行ってからアクセスするため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。したがって，32 ビット空間内であれば，どこにでも配置が可能なセクションです。

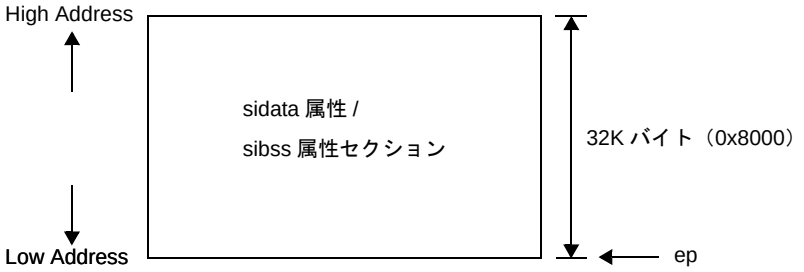
なお，プログラム中で const 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section const
const int   a = 1;  /*const セクション配置 */
#pragma section default
```

- sidata 属性 /sibss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。つまり，ep からプラス方向 32K バイト内に配置されるセクションです

図 3―11 sidata 属性 /sibss 属性セクション



初期値ありデータは sidata 属性セクションへ，初期値なしデータは sibss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性 /sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。

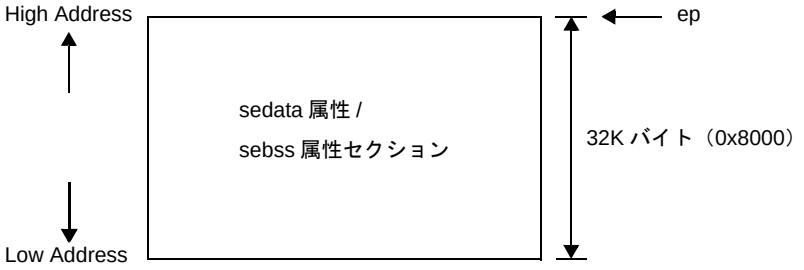
なお，プログラム中で sidata 属性 /sibss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section sidata
int a = 1;  /*sidata セクション配置 */
int b;      /*sibss セクション配置 */
#pragma section default
```

- sedata 属性 /sebss 属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からマイナス方向へアクセスするセクションです。つまり，ep からマイナス方向 32K バイト内に配置されるセクションです。

図 3―12 sedata 属性 /sibss 属性セクション



初期値ありデータは sedata 属性セクションへ，初期値なしデータは sebss 属性セクションへ配置されます。gp から 1 命令でアクセスできる sdata 属性 /sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。

なお，プログラム中で sedata 属性 /sebss 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section sedata
int a = 1;  /*sedata セクション配置 */
int b;      /*sebss セクション配置 */
#pragma section default
```

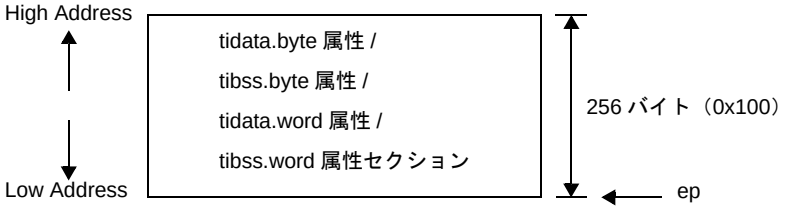
- tidata（tidata.byte，tidata.word）属性 /tibss（tibss.byte，tibss.word）属性セクション

ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。sidata 属性 /sibss 属性セクションと違うところは，同じ 1 命令アクセスでも，使用するアセンブル命令が違うことです。

sidata 属性 /sibss 属性セクション，および sedata 属性 /sebss 属性セクションは，格納／参照に 4 バイト長命令の“st/ld 命令”を使用しますが，tidata 属性 /tibss 属性セクションは，2 バイト長命令の“sst/sld 命令”を使用してアクセスします。つまり，sidata 属性 /sibss 属性セクション，および sedata 属性 /sebss 属性セクションよりもコード効率がよくなります。

ただし，sst/sld 命令が適用できる範囲は小さいので，多くの変数を配置することはできません。

図 3―13 tidata 属性 /tibss 属性セクション



初期値ありデータは tidata（tidata.byte，tidata.word）属性セクションへ，初期値なしデータは tibss（tibss.byte，tibss.word）属性セクションへ配置されます。バイト・データを配置する場合は tidata.byte/tibss.byte 属性を，ワード・データを配置する場合は tidata.word/tibss.word 属性を指定しますが，CX に自動判別させたい場合は tidata/tibss 属性を指定します。
システムの中でも，より高速にアクセスしたいデータを配置するために使用します。
ただし，配置できる量が少ないため，厳選する必要があります。プログラム中で tidata.byte/tibss.byte 属性，tidata.word/tibss.word 属性セクションに配置したい変数を指定する場合は，#pragma section 指令を使用します（詳細は「(a) #pragma section 指令」を参照）。

```
#pragma section tidata_byte
char            a = 1;  /*tidata.byte セクション配置 */
unsigned char   b;      /*tibss.byte セクション配置 */
#pragma section default
```

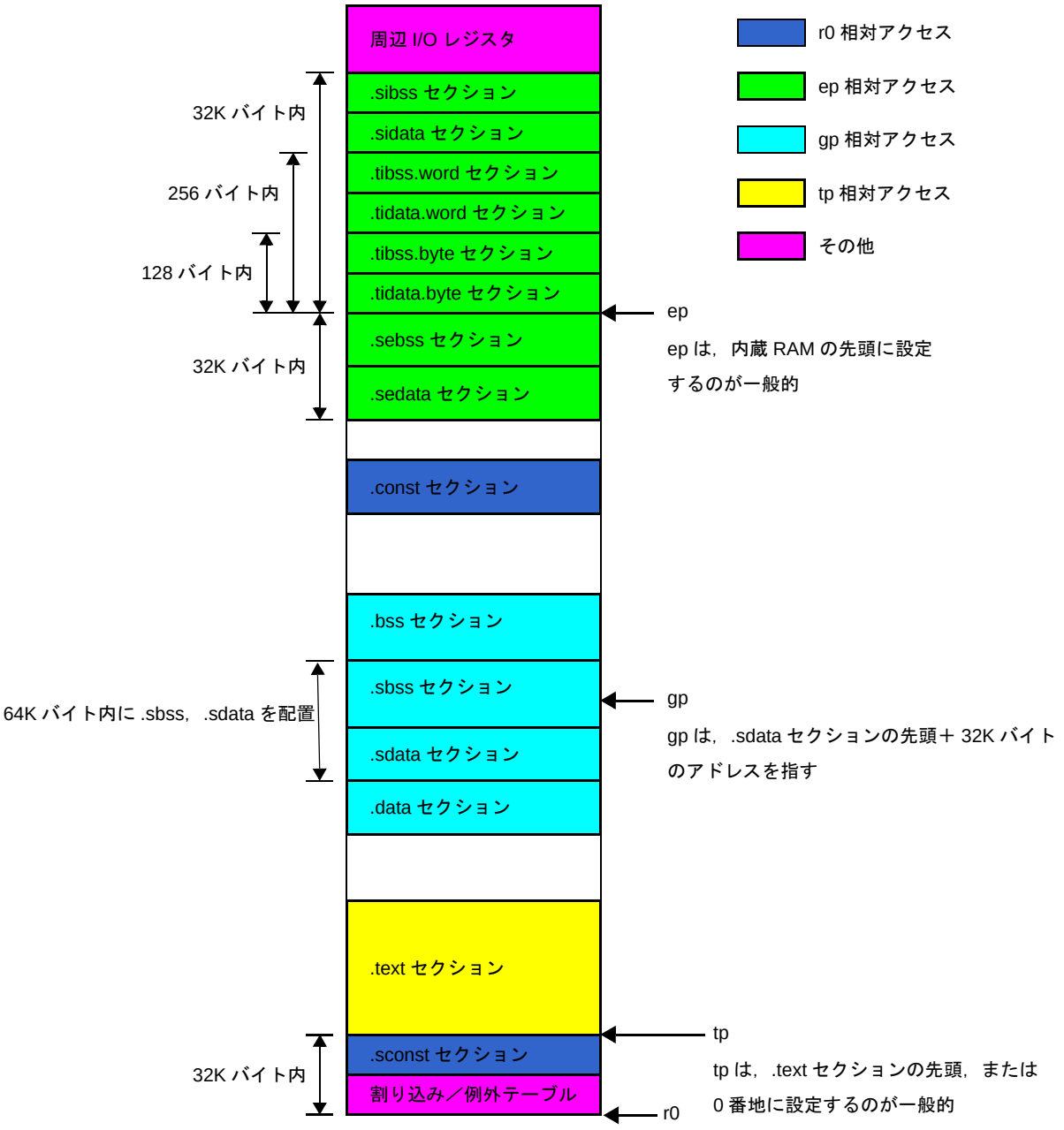
```
#pragma section tidata_word
int     a = 1;          /*tidata.word セクション配置 */
short   b;              /*tibss.word セクション配置 */
#pragma section default
```

```
#pragma section tidata
int     a = 1;          /*tidata.word セクション配置 */
char    b;              /*tibss.byte セクション配置 */
#pragma section default
```

変数／データの中で，特にシステムの中で参照頻度の高いものを選び，tidata（tidata.byte，tidata.word）属性 /tibss（tibss.byte，tibss.word）属性セクションに配置できると，実行速度の面からも効率がよくなります。CX には，この参照頻度を調査する機能があります。その情報を元にして，自動的に tidata（tidata.byte，tidata.word）属性 /tibss（tibss.byte，tibss.word）属性セクションに配置するコードを出力します。

次図に各セクションのメモリ配置イメージの例を示します。

図 3―14　tidata 属性 /tibss 属性セクション



（a）#pragma section 指令

#pragma section 指令を使って，目的のセクションへデータを割り当てる方法を説明します。

- セクション名をデフォルトのまま使用する場合

CX で定義されているセクション名をそのまま使用する場合，#pragma section 指令で次のような書式で記述します。

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

ここで“セクション種別”に指定できるのは，次のとおりです。

data，sdata，sedata，sidata，tidata，tidata_word，tidata_byte，sconst，const

bss 属性のセクション名をセクション種別に指定することはできません。bss 属性については，宣言／定義された変数・データに“初期値がある場合”は data 属性に，“初期値がない場合”は bss 属性に CX が自動的に振り分けます。

```
#pragma section sdata
int a = 1;  /*sdata セクション配置 */
int b;      /*sbss セクション配置 */
#pragma section default
```

上記の場合，“変数 a”は初期値を持つので，data 属性の“.sdata セクション”へ，“変数 b”は初期値を持たないので，bss 属性の“sbss セクション”へ配置します。

“#pragma section セクション種別”と“#pragma section default”の間には，変数宣言／定義を複数記述することができます。そのセクション種別に配置したい変数を列挙します。

なお，tidata.word と tidata.byte をセクション種別に指定する場合，次のように，間の“.（ピリオド）”の代わりに“_（アンダースコア）”を指定してください。

tidata_word，tidata_byte

- セクション名に独自の名前をつける場合

独自のセクション名を指定し，そこに変数やデータを配置することができます。

この場合，#pragma section 指令で次のような書式で記述します。

```
#pragma section セクション種別     “作成するセクション名”
変数宣言／定義
#pragma section default
```

ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが作成するセクション名に“. セクション種別”が付加されたものになります。

```
作成するセクション名．セクション種別
```

これは，初期値の“ある”／“なし”で，data 属性，bss 属性に分かれるためで，同一セクション名で別属性のセクションができてしまうのを防ぐためです。リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「(b) 独自のデータ・セクションのリンク・ディレクティブ指

定」を参照してください。ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3―15 指定するセクション名と生成されるセクション名

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mydata	data 属性	.data/.bss	mydata.data/mydata.bss
mysdata	sdata 属性	.sdata/.sbss	mysdata.sdata/mysdata.sbss
myconst	const 属性	.const	myconst.const

次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata   "mysdata"
int a = 1;  /*mysdata.sdata セクション配置 */
int b;      /*mysdata.sbss セクション配置 */
#pragma section default
```

(b) 独自のデータ・セクションのリンク・ディレクティブ指定

#pragma section 指令で，独自のセクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

C ソースにて“変数 a”と“変数 b”を次のように指定した場合，“変数 a”は初期値を持つので“mysdata.sdata セクション”へ，“変数 b”は初期値を持たないので“mysdata.sbss セクション”へ配置されます。

```
#pragma section sdata   "mysdata"
int a = 1;  /*mysdata.sdata セクション配置 */
int b;      /*mysdata.sbss セクション配置 */
#pragma section default
```

このとき，リンク・ディレクティブ・ファイル内のマッピング・ディレクティブは，次のように記載すると，独自のセクションに配置されます。

```
.sdata          = $PROGBITS     ?AWG .sdata;
.sbss           = $NOBITS       ?AWG .sbss;
mysdata.sdata   = $PROGBITS     ?AWG mysdata.sdata;
mysdata.sbss    = $NOBITS       ?AWG mysdata.sbss;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また，直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mysdata.sdata の属性“$PROGBITS ?AWG”と入力セクション，mysdata.sbss の属性“$NOBITS ?AWG”と入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる“$PROGBITS ?AWG mysdata.sdata ”“ $NOBITS ?AWG mysdata.sbss”のこと）を省略しないでください。

（c）セクション割り当ての注意点

#pragma section 指令，const 修飾子，およびセクション・ファイルによってセクションを割り当てた場合の注意点を次に示します。

- 自動変数に対してセクション指定を行った場合，その指定は無視されます。セクション指定は外部変数／文字列定数／静的変数に対する機能です。

- 初期値を設定しない変数宣言は，通常“仮定義”として扱われますが，セクション指定した場合は“定義”として扱われます。初期値を設定しない変数宣言と定義を混在させないようにしてください。

【#pragma section を使用しない変数宣言】	【#pragma section を使用した変数宣言】
int i; /* 仮定義 */ int i = 10; /* 定義 */ 【エラーになりません】	#pragma section sedata int i; /* 定義 */ int i = 10; /* 定義 */ #pragma section default 【二重定義エラー】

外部変数を参照するファイルでは，必ず extern 宣言してください。次の場合では，file1.c 側の変数の仮定義で extern がないと，二重定義エラーになります。

【file1.c】	【file2.c】
#pragma section sedata extern int i; #pragma section default	#pragma section sedata int i; #pragma section default
【extern がないと，二重定義エラー】	

- セクション指定した変数を，他のファイルで参照する場合，その変数に対する extern 宣言に対しても，同じセクション種別でセクション指定する必要があります。変数定義時に指定したセクションと異なる種別のセクションを指定した場合はエラーになります。
  たとえば，定義側で“#pragma section data ～ #pragma section default”指定して，仮定義側（extern 宣言）で“#pragma section data ～ #pragma section default”指定しなかった場合，仮定義側では sdata に配置されているものとみなされます。つまり，定義側では gp からの 2 命令でアクセスするコードが出力され，仮定義側では gp からの 1 命令でアクセスするコードが出力されることになります。この場合，つじつまが合わなくなるため，リンク時にエラー・メッセージが出力されます。

【正しい例】

【file1.c】	【file2.c】
#pragma section sedata int i = 1; #pragma section default	#pragma section sedata extern int i; #pragma section default

【誤った例 1】

【file1.c】	【file2.c】
int i = 1;	#pragma section sedata extern int i; #pragma section default

file1.c で定義した“変数 i”は sbss セクション，または bss セクションに配置されますが，file2.c では“変数 i”に対して sebss セクションへのアクセス・コードが出力されるため，リンカでエラー・メッセージが出力されます。

【誤った例 2】

【file1.c】	【file2.c】
#pragma section sedata int i = 1; #pragma section default	extern int i;

file1.c で定義した“変数 i”は sebss セクションに配置されますが，file2.c では“変数 i”に対して sbss セクション，または bss セクションへのアクセス・コードが出力されるため，リンカで不整合エラーが出力されます。

- const 指定された変数は，const セクションに割り付けられますが，#pragma section 指令により const/sconst 以外を指定した場合は，指定されたセクションに割り付けられます。

- #pragma section 指令で，sconst 属性，const 属性の変数を定義する場合，変数に必ず“const 指定”をしてください。また，extern 宣言で仮定義している箇所でも，同じく const 指定が必要です。
“#pragma section sconst”や“#pragma section const”で指定しても，変数宣言時に const 宣言がなかった場合，sconst セクション /const セクションに配置されず（#pragma section 指令は無視され），gp 相対セクションである sdata セクションや data セクションに配置されてしまい，意図した配置にならないことになります。

【file1.c】	【file2.c】
#pragma section sconst const int i = 1; #pragma section default	#pragma section sconst int i; #pragma section default

file1.c の方は，“変数 i”が sconst セクションに配置するコードが出力されますが，file2.c の方は，“変数 i”に const 指定がないため，#pragma section 指定が無視され，gp 相対の変数として扱われるため，sdata セクションや data セクションに配置されるコードになります。リンクしても sconst に“変数 i”が配置されません。
また，次のように extern 宣言で仮定義している箇所でも const 指定が必要になります。

【file1.c】	【file2.c】
`#pragma section sconst`	`#pragma section sconst`
`const int i = 10;`	`extern const int i;`
`#pragma section default`	`#pragma section default`

- #pragma section には，サイズ不明な変数，要素数不明な配列，定義されていない構造体，および，それを含む構造体は指定できません。

- -Xsdata，-Xsconst オプション指定とともに #pragma section が指定された場合，#pragma section による指定が有効となります。#pragma section を記述しない場合，または再配置属性に default を指定した場合はオプション指定が有効となります。

（d）#pragma section 指令の例

次に，#pragma section 指令の例を数点示します。

- 変数 a を tidata.word セクションに，変数 b を tibss.word セクションに配置する

```
#pragma section tidata_word
int     a = 1;      /*tidata.word セクションに配置 */
short   b;          /*tibss.word セクションに配置 */
#pragma section default
```

- 変数 c を tidata.byte セクションに，変数 d を tibss.byte セクションに配置する

```
#pragma section tidata_byte
char    c = 0x10;   /*tidata.byte セクションに配置 */
char    d;          /*tibss.byte セクションに配置 */
#pragma section default
```

tidata 属性セクションでは，ワード・データ／ハーフワード・データは tidata.word/tibss.word セクションに配置され，バイト・データは tidata.byte/tibss.byte セクションに配置されます。

- const 指定された変数 e を sconst セクションに，ポインタ p が示す文字列定数データを sconst セクションに配置する

```
#pragma section sconst
const   int     e = 0x10;
const   char    *p = "Hello, World";
#pragma section default
```

上記の記述で，ポインタ p の示す“Hello World”は sconst セクションへ，ポインタ変数“p”自体は sdata セクション，または data セクションへ配置されます。ポインタ変数とポインタの示す内容の配置場所に関しては，const 指定の方法によって変わってきます。

例 1.

```
const   char    *p = "Hello, World";
```

このように宣言すると，次のようになります。

ポインタ変数“p”	書き換え可能（“p = 0”はコンパイル OK）
文字列定数“Hello World”	書き換え不可能（“*p = 0”はコンパイル NG）

ポインタ変数の指す先を const 属性に配置したい場合は，上記のように記述します。

```
#pragma section sconst
const   char    *p = "Hello, World";
#pragma section default
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sdata/data セクション
文字列定数“Hello World”	sconst セクション

2.

```
char    *const  p;
```

ポインタ変数“p”	書き換え不可能（“p = 0”はコンパイル NG）

ポインタ変数を const 属性に配置したい場合は，上記のように記述します。

```
char    *const  p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst
char    *const  p = "Hello World";
#pragma section  default
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

3.

```
const   char    *const  p;
```

ポインタ変数“p”	書き換え不可能（“p=0”はコンパイル NG）

ポインタ変数，およびその指す先を const 属性に配置したい場合は，上記のように記述します。どちらも ROM に固定するような場合に使用します。

```
const   char    *const  p = "Hello World";
```

上記のように記述すると，ポインタ変数，および文字列定数“Hello World”ともに，const 属性のセクションに配置されます。

```
#pragma section sconst
const   char    *const  p = "Hello World";
#pragma section default
```

上記のように定義すると次のようなセクション配置になります。

ポインタ変数“p”	sconst セクション
文字列定数“Hello World”	sconst セクション

なお，const 指定で宣言した変数を sconst セクションに配置する方法として #pragma section 指令のほかに，コンパイラ・オプション“-Xsconst”指定があります。

- #pragma section 指令の extern 宣言する方は，共通に使用するヘッダ・ファイル内で行い，C ソース内にインクルードして使用する

```
【header.h】
#pragma section sidata
extern  int k;
#pragma section default
```

```
【file1.c】
#include    "header.h"
#pragma section sidata
int k;
#pragma section default
```

```
【file2.c】
#include    "header.h"

void func1(void) {
    k = 0x10;
}
```

上記のように #pragma section 指令の extern 宣言する方は，ヘッダ・ファイルでまとめておくと，間違いが少なくなり，ソースも見やすくなります。

（3）関数のセクション割り当て

CX では，C ソースの関数であるプログラム・コードは，デフォルトで“.text セクション”に配置されます。リンク・ディレクティブ・ファイルで，.text セクションの配置アドレスを決定すると，そこからプログラムを配置します。

しかし，「関数ごとに配置アドレスを変えたい」場合や，「メモリの配置上，配置アドレスを分ける必要がある」場合があります。これに対応するため，CX では“#pragma text 指令”を用意しています。#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけ，リンク・ディレクティブ・ファイルにて配置アドレスを変えます。

（a）#pragma text 指令

#pragma text 指令を使って，text 属性を持つセクションに任意の名前をつけることができます。

#pragma text 指令の使い方には，次の 2 通りあります。

- #pragma text 指令に，作成するセクションに配置する“関数名”を指定する方法

```
#pragma text     “作成するセクション名”      関数名 [, 関数名 ]…
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。また，“作成するセクション名”を省略することもできます。その場合“関数名”で指定した関数を，デフォルトの“.text セクション”に配置します。

- 関数本体（関数定義）の前に #pragma text 指令を記述し，関数名は指定しない方法

```
#pragma text     “作成するセクション名”
```

“作成するセクション名”を省略することもできます。その場合，その直前に指定した“#pragma text”作成するセクション名の指定を解除し，これ以降の関数を，デフォルトの“.text セクション”に配置します。
ただし，この指定方法で実際に生成されるセクション名は，次のようにユーザが指定したセクション名に“.text”が付加されたものになります。

```
セクション名 .text
```

リンク・ディレクティブ・ファイルでセクション指定するときは，生成されたセクション名を指定してください。リンク・ディレクティブ・ファイルにおける指定例については，「(b) 独自のデータ・セクションのリンク・ディレクティブ指定」を参照してください。
ユーザが独自に指定するセクション名と生成されるセクション名の具体例は，次のとおりです。

表 3―16 指定するセクション名と生成されるセクション名

ユーザが独自に指定したセクション名	セクション種別	付加される文字列	生成されるセクション名
mytext	text 属性	.text	mytext.text

次のように指定した場合，“関数 func1”は“mytext1.text セクション”へ，“関数 func2”は #pragma text 指令がないので，デフォルトで“.text セクション”へ配置されます。

```
#pragma text    "mytext1"       func1
void func1(void) {
  :
}

void func2(void) {
  :
}
```

また，次のように指定した場合，“関数 func1”，“関数 func2”は“mytext2.text セクション”へ，“関数 func3”は“mytext3.text セクション”へ，“関数 func4”は“#pragma text”で直前の“#pragma text”mytext3”が解除されているので，デフォルトで“.text セクション”へ配置されます。

```
#pragma text    "mytext2"
void func1(void) {
  :
}

void func2(void) {
  :
}

#pragma text    "mytext3"
void func3(void) {
  :
}

#pragma text
void func4(void) {
  :
}
```

（b）独自のテキスト・セクションのリンク・ディレクティブ指定

#pragma text 指令で，独自のテキスト・セクションを作成した場合，そのセクションのリンク・ディレクティブ・ファイルの記述について説明します。

```
#pragma text    "mytext2"
void func1(void) {
  :
}

void func2(void) {
  :
}

#pragma text    "mytext3"
void func3(void) {
  :
}

#pragma text
void func4(void) {
  :
}
```

C ソースにて，上記のように #pragma text 指定した場合，“関数 func1”，“関数 func2”は“mytext2.text セクション”へ，“関数 func3”は“mytext3.text セクション”へ，“関数 func4”は“#pragma text”で直

前の“#pragma text ”mytext3”が解除されているので，デフォルトで“.text セクション”へ配置されます。

```
text = $PROGBITS    ?AX .text;
mytext2 = $PROGBITS ?AX mytext2.text;
mytext3 = $PROGBITS ?AX mytext3.text;
```

なお，記述順に配置されるので，配置変更したい場合は，記述順を変更してください。また，直接アドレス指定することもできます（ただし，セグメントを作成して，その中にマッピング・ディレクティブを記述し，セグメント単位でアドレス指定するのが一般的です）。

ここで注意が必要なのは，mytext2.text/mytext3.text の属性が“$PROGBITS ?AX”なので，これらと同じ属性を持つマッピング・ディレクティブにて，入力セクション（上記で，マッピング・ディレクティブの一番右側に書かれる“.text ”“ mytext2.text ”“ mytext3.text”のこと）を省略しないでください。

例 下記のように同じ“$PROGBITS ?AX”属性のマッピング・ディレクティブで，入力セクションが省略されていると，リンカはその属性のセクションをすべて結合して配置するため，独自に作ったセクションへ配置されない結果となります。
したがって，mytext2.text/mytext3.text に配置しようとしたプログラムは .text に配置されることになります。

```
.text = $PROGBITS       ?AX;
```

（c）#pragma text 指令の注意点

#pragma text 指令の注意点を次に示します。

- #pragma text 指令は，関数の定義と同一ファイル内で，定義より前に記述してください。関数の定義よりも後ろに記述された場合は，警告メッセージを出力し，#pragma text 指令は無視します。ただし，関数のプロトタイプ宣言の順番には関係ありません。
- 関数名を指定した #pragma text の後に関数を指定しない #pragma text を記述した場合，指定された関数は指定されたセクションに割り当てられ，指定されていない関数は後から記述された #pragma text にしたがい配置されます。
- #pragma text 指定した関数が“direct 配置指定された割り込みハンドラ”である場合，警告メッセージを出力し，#pragma text 指令は無視します。なお，direct 配置指定についての詳細は「(8) 割り込み／例外処理ハンドラ」を参照してください。
- #pragma text 指定した関数は，#pragma inline 指定や，最適化オプションによるインライン展開後，関数が不要になっても，指定したセクションに関数を出力します。
- 作成するセクション名を省略した場合，この指定はデフォルトの text 属性セクションに配置されるので，意味をもたないとものとなりますが，すでに名称付きのセクション指定があった場合は，デフォルトに戻ります。
- セクション名を指定する場合，その名前は 256 文字以内にしてください。

(4) 周辺 I/O レジスタへのアクセス

周辺 I/O レジスタとは，各デバイスで内蔵している周辺機能のためのレジスタです。デバイス定義の周辺 I/O レジスタ名を用いることにより，C 言語レベルで，周辺 I/O へアクセスできます。周辺 I/O レジスタ名は，C ソース・プログラム中で，通常の符号なし外部変数（unsigned）のように扱うことができます。

なお，指定可能なレジスタ名，属性などについては，各デバイスのユーザーズ・マニュアルを参照してください。

(a) アクセス方法

周辺 I/O レジスタ名を使用できるようにするには，次の #pragma 指令を記述することにより行います。

```
#pragma ioreg
```

“#pragma ioreg”を記述した C ソース内では，それ以降，周辺 I/O レジスタ名を使用することができます。

上記の指令を行わずに，あるいは，適したデバイス名を指定せずに周辺 I/O レジスタ名を使用すると，「変数が未定義である」というエラーになります。

また，指定したレジスタ固有のアクセス属性に反した使用も，エラーになります。

次の場合，例 1 は正しいですが，例 2，例 3 はエラーとなります。

なお，次の例で，P0，P1，P2，RXBO，OVF0 は V850 マイクロコントローラの周辺 I/O レジスタを示します。このように，“レジスタ名”には，デバイス・ファイルで定義されている略号を指定します。

例 1.

```
#pragma ioreg
void func1(void) {
    int i;
    P0 = 1;      /*P0 に書き込み */
    i = RXB0;    /*RXB0 から読み込み */
}

void func2(void) {
    P1 = 0;      /*P1 に書き込み */
}
```

2.

```
void func(void) {
    P1 = 0;      /* 未定義エラー */
}
```

3.

```
#pragma ioreg
void func(void) {
    RXB0 = 1;    /*RXB0 の属性が読み込み専用のため，エラー */
}
```

(5) アセンブラ命令の記述

CX では，次に示す形式において，C ソース・プログラム中にアセンブラ命令が記述できます。

- asm 宣言
- #pragma 指令

挿入するアセンブラ命令でレジスタを使用する場合，必要な退避／復帰はプログラム内で行ってください。CX では行いません。

また，アセンブラ命令の記述は，関数の中に挿入してください。関数の外に記述した場合，エラーとなります。

(a) asm 宣言

```
__asm( 文字列定数 );
```

- コンパイラは，asm 宣言が指定された場合，指定された文字列定数[注]の後ろにニューライン（\ n）を付けてアセンブラに渡します。

注 “\”は，エスケープ文字となります（例：\ 0 → NULL，\ r →キャリッジ・リターン，\ " → "，\\→ \）。

例

```
__asm("nop");
__asm (".str    \"string\\0\"");
```

- __asm は宣言であり，文として扱われません。このため，次に示す例 1 のように，C 言語の文法上，宣言のみの記述が許されない構文ではエラーとなります。
  そこで，例 2 のように “{}” で囲んで複合文としてください。

例 1.

```
if(i == 0)
__asm("mov     r11, r10");  /* 宣言のみのためエラー */
```

2.

```
if(i == 0) {
    __asm("mov     r11, r10"); /* 複合文となるため記述可能 */
}
```

（b）#pragma 指令

この #pragma 指令で囲まれた範囲では，そのまま，アセンブラ命令が記述できます。複数のアセンブラ命令を記述する場合に有効です。

```
#pragma asm
        アセンブラ命令
#pragma endasm
```

次に示す例 1 の記述は，例 2 の記述と同様の意味となります。

例 1.

```
int i;

void f() {
#pragma asm
        mov     r0, r10
        st.w    r10, $_i
          :
#pragma endasm
}
```

2.

```
int i;

void f() {
    __asm("mov     r0, r10");
    __asm("st.w    r10, $_i");
      :
}
```

“#pragma asm”から“#pragma endasm”までの記述は，そのままアセンブラに渡されます。

したがって，CX が内部的にアセンブラ命令を作成し，アセンブラを起動します。

そのため“#pragma asm”宣言後に，アセンブラ命令の疑似命令を使用することも可能です。また，アセンブラ命令で，C ソース・プログラム中のローカル変数は使用できません。ローカル変数は，CX で“スタック”，または“レジスタ”に割り当てられるためで，インライン・アセンブラでは使用することはできません。

C ソース・ファイルの #define で定義したシンボルも“#pragma asm”から“#pragma endasm”までの記述では使用できません。ファイル内で #define 定義したマクロをアセンブラ命令で展開したい場合，次の方法で回避してください。

- #pragma asm ～ #pragma endasm 指令内で .macro 命令を使用してマクロ定義する
- C ソースから関数コールでアセンブラ命令を呼ぶ

マクロ定義せず，そのままアセンブラ命令で書くのも 1 つの方法です。

(6) 割り込みレベルの制御

(a) __set_il 関数

CX では，V850 マイクロコントローラの割り込みに対して，C ソース上で，次の制御を行うことができます。

- 割り込み優先順位レベルの制御
- マスカブル割り込みの受け付けの許可／禁止（割り込みのマスク）

つまり，“割り込み制御レジスタ”を操作することができます。

“割り込み優先順位レベル”を制御する場合は，“__set_il 関数”を用いて次のように指定します。

```
__set_il( 割り込みの優先順位レベル , “割り込み要求名”);
```

指定できる“割り込み要求名”は，デバイス・ファイルで定義されている“マスカブル割り込みの要求名”です。

“割り込みの優先順位レベル”として指定できる値は“1 ～ 16”の整数値です。命令セット・アーキテクチャが V850E2V3 であるデバイスの割り込み優先順位レベルは“0 ～ 15 までの 16 段階”を指定するため，“命令セット・アーキテクチャが V850E2V3 であるデバイスの割り込みの優先順位レベルを 5 にしたい”場合は，この関数で指定する割り込みの優先順位レベルは“6”と指定します。

例

```
__set_il(2, "INTP0");
```

上記の指定の場合，割り込み“INTP0”の優先順位レベルは“1”になります。

次に，“割り込みに対して，マスカブル割り込みの受け付けの許可／禁止”を制御する場合は，次のように指定します。

```
__set_il( マスカブル割り込みの許可／禁止 , “割り込み要求名”);
```

“マスカブル割り込みの許可／禁止”に設定できるのは“-1”か“0”です。

表 3―17 マスカブル割り込みの許可／禁止

設定値	動作
-1	マスカブル割り込みの受け付け禁止（割り込みをマスク）
0	マスカブル割り込みの受け付け許可（割り込みのマスクを解除）

例

```
__set_il(-1, "INTP0");
```

上記のように指定すると，割り込み“INTP0”のマスカブル割り込みの受け付けを禁止します（INTP0をマスクします）。

なお，__set_il 関数では，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作は行いません。

（b）__set_il 関数と割り込み制御レジスタ

__set_il 関数を使用した場合は，“優先順位レベル”，または“割り込みマスク・フラグ”のいずれか一方の設定になります。したがって，__set_il 関数では，割り込み要求フラグの設定はできません。

（7）割り込み禁止

CX では，C ソースにおいて，マスカブル割り込みを禁止にすることができます。

マスカブル割り込みを禁止する方法には，大きく分けて次の 2 通りがあります。

- 関数内で部分的に割り込みを禁止する方法
- 関数全体の割り込みを禁止する方法

（a）関数内で部分的に割り込みを禁止する方法

C 言語で記述した関数内で，部分的に割り込みを禁止する場合，アセンブラ命令の“di 命令”と“ei 命令”を使用することができますが，CX では C ソースで割り込み制御を行うことのできる関数を用意しています。

表 3―18 割り込み制御関数

割り込み制御関数	動作	CX の処理
__DI	すべてのマスカブル割り込みの受け付けを禁止します。	di 命令を生成
__EI	すべてのマスカブル割り込みの受け付けを許可します。	ei 命令を生成

例 __DI，__EI 関数の記述方法とその出力コード

```
【C ソース】
void func1(void) {
      :
    __DI();
    /* 割り込みを禁止して行いたい処理を記述 */
    __EI();
      :
}
```

```
【出力コード】
_func1:
    -- プロローグ・コード
      :
    di
    -- 割り込みを禁止して行いたい処理
    ei
      :
    -- エピローグ・コード
    jmp     [lp]
```

（b）関数全体の割り込みを禁止する方法

CX では，関数全体のマスカブル割り込み割り込みを禁止する“#pragma block_interrupt”指令を用意しています。

次のような書式で記述します。

```
#pragma block_interrupt 関数名
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。

上記で“関数名”で指定された関数に対し，マスカブル割り込みを禁止します。「（a）関数内で部分的に割り込みを禁止する方法」で説明したように，関数の最初に“__DI ( );”を，最後で“__EI ( );”を記述することもできますが，この場合だと，CX が出力する“プロローグ・コード”，“エピローグ・コード”に対してマスカブル割り込みを禁止／許可することができず，関数全体を完全に割り込み禁止にすることができません。

#pragma block_interrupt 指令を用いると，“プロローグ・コード”実行の直前にマスカブル割り込みが禁止され，“エピローグ・コード”実行直後にマスカブル割り込み割り込みが許可されます。そのため関数全体を完全に割り込み禁止にすることができます。

例 #pragma block_interrupt 指令の使用方法と，出力されるコードは次のとおりです。

```
【C ソース】
#pragma block_interrupt func1
void func1(void) {
      :
    /* 割り込みを禁止して行いたい処理を記述 */
      :
}
```

```
【出力コード】
_func1:
    di
    -- プロローグ・コード
      :
    -- 割り込みを禁止して行いたい処理
      :
    -- エピローグ・コード
    ei
    jmp     [lp]
```

(c) 関数全体の割り込み禁止時の注意事項

関数全体の割り込みを禁止にした場合の注意事項について，次に示します。

- 同じ関数に対して，割り込みハンドラ指定と #pragma block_interrupt 指定された場合，割り込みハンドラ指定の方が優先され，割り込み禁止の設定は無視されます。
- 割り込み禁止となっている関数内で，次の関数を呼び出した場合，その呼び出しからの復帰時に，割り込み許可状態になるため，注意が必要です。
- #pragma block_interrupt 指定された関数
- 関数の先頭で割り込み禁止をし，最後で割り込み許可している関数
- #pragma block_interrupt 指令は，関数の定義と同一ファイル内で，かつ，定義より前に記述してください。
- 関数の定義より後ろに記述された場合，コンパイル時にエラーとなります。
- ただし，関数のプロトタイプ宣言順とは関係ありません。
- #pragma block_interrupt 指定された関数は，#pragma inline 指令を指定したり，最適化オプションでインライン展開指定ができません。インライン展開指定は無視されます。
- #pragma block_interrupt 指定しても，PSW（プログラム・ステータス・ワード）内の EP フラグ（例外処理中を示すフラグ）の操作を行うコードは出力されません。

(8) 割り込み／例外処理ハンドラ

CX では，C 言語で“割り込み”や“例外”が発生したときに呼ばれる“割り込みハンドラ”，“例外ハンドラ”を記述することができます。ここでは，その記述方法などについて説明します。

(a) 割り込み／例外の発生

V850 マイクロコントローラでは，割り込みや例外が発生すると，その割り込みや例外に対応したハンドラ・アドレスにジャンプします。“割り込み要因”と“ハンドラ・アドレス”は一対一で対応しており，ハンドラ・アドレスの集合を“割り込み／例外テーブル”と呼びます。

たとえば，V850E2/MN4 の場合の割り込み／例外テーブルは次のようになっています（一部分のみ掲載）。

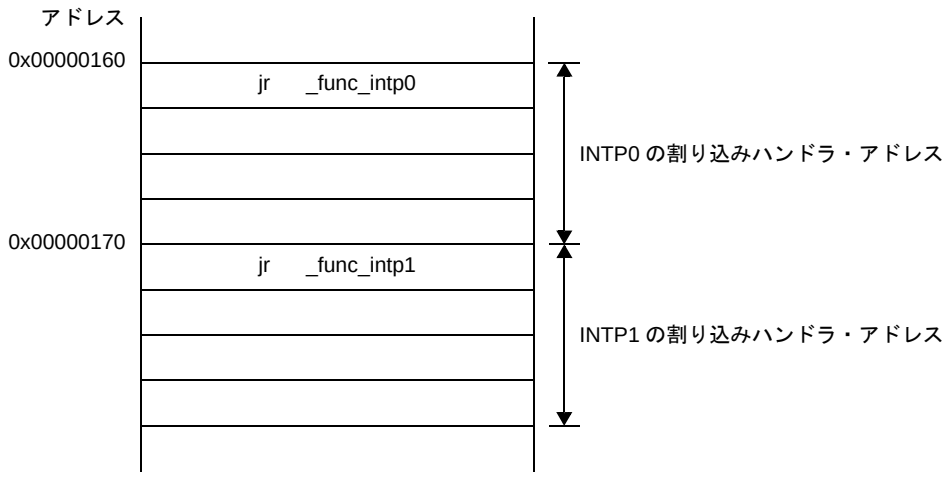
表 3―19 割り込み／例外テーブル（V850E2/MN4）

アドレス	割り込み名称	割り込みのトリガ
0x0000	RESET	リセット入力
0x0010	FEINT	外部 NMI 入力
0x0020	FENMI	WDT0ATNMI/WDT1ATNMI
:	:	:
0x0120	INTWDTA0	WDT0 インターバル・タイマー割込み
0x0130	INTWDTA1	WDT1 インターバル・タイマー割込み
0x0140	INTOSTM0	OS タイマ・アンダフロー割り込み
0x0150	INTOSTM1	OS タイマ・アンダフロー割り込み
0x0160	INTP0	外部割り込み
0x0170	INTP1	外部割り込み
0x0180	INTP2	外部割り込み

なお，ハンドラ・アドレスの並びや，搭載している割り込みは，V850 の品種ごとに異なります。詳細は，使用する各デバイスのユーザーズ・マニュアルを参照してください。

各ハンドラ・アドレスは“16 バイト”の領域を持っており，割り込みが発生すると，その 16 バイトの領域に書かれた命令を実行します。したがって，16 バイトで収まる処理であれば，ハンドラ・アドレス内だけで処理し，収まらなければ，処理の書かれた関数（割り込み／例外ハンドラ）への分岐命令を記述することになります。

図 3―15 ハンドラ・アドレスのイメージ



V850E2/MN4 で INTP0 割り込みが入ると，0x160 番地にジャンプし，そこにあるコードを実行します。上記の例の場合，関数 func_intp0 に分岐するコードが書かれているので。そこへ分岐します。つまり，func_intp0 は INTP0 の割り込みハンドラということになります。

これらはアセンブラ・ソース・レベルでの話になりますが，CX では，C 言語レベルで割り込み／例外処理を記述する場合，この点について留意することなく記述できるようになっています。具体的な記述方法は「(c) 割り込み／例外ハンドラの記述方法」で説明します。

(b) 割り込み／例外発生時に行う必要のある処理

関数実行時やタスク実行時に割り込み／例外が入ると，即座に割り込み／例外処理を行う必要があります。そして割り込み／例外処理が終わると，割り込みが入った時点の関数やタスクに戻る必要があります。**注**

したがって，割り込み／例外発生時には，そのときのレジスタ情報を保存し，割り込み／例外処理が終わった後は，そのレジスタ情報を復帰する必要があります。

注 リアルタイム OS 使用時のタスクの場合，割り込み内でのシステム・コール発行によって，割り込みが入った時点のタスクに戻らないことがあります。詳細は各リアルタイム OS のユーザーズ・マニュアルを参照してください。

通常の関数のプロローグ／エピローグ・コードでは，レジスタ変数用レジスタの退避／復帰を行います。レジスタ変数用レジスタは次のとおりで，必要のあるものに関して退避／復帰を行います。

表 3―20 レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

割り込み／例外ハンドラに移る場合は，レジスタ変数用レジスタのほかに，割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表 3―21 割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	割り込み／例外発生時に退避／復帰するレジスタ
22 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r31（lp），CTPC，CTPSW，BSEL【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】
26 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r31（lp），CTPC，CTPSW，BSEL【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】
32 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，r18，r19，r31（lp），CTPC，CTPSW，BSEL【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】

また，多重割り込み／例外が発生した場合は，レジスタ変数用レジスタのほかに，多重割り込み／例外ハンドラ用のスタック・フレームとして，次のレジスタの必要のあるものに関して退避します。

表 3―22 多重割り込み／例外ハンドラ用のスタック・フレーム

レジスタ・モード	多重割り込み／例外発生時に退避／復帰するレジスタ
22 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r31（lp），EIPC，EIPSW，CTPC，CTPSW，BSEL【V850E2V3】，EIIC【V850E2V3】，EIWR【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】
26 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r31（lp），EIPC，EIPSW，CTPC，CTPSW，BSEL【V850E2V3】，EIIC【V850E2V3】，EIWR【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】
32 レジスタ・モード	r1，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，r18，r19，r31（lp），EIPC，EIPSW，CTPC，CTPSW，BSEL【V850E2V3】，EIIC【V850E2V3】，EIWR【V850E2V3】，FPSR/FPEPC（FPU を持つ場合）【V850E2V3】

上記のレジスタの用途は，次のとおりです。

表 3―23 レジスタの用途

レジスタ	用途
r1	アセンブラ予約レジスタ
r6 ～ r9	引数用レジスタ（関数の引数をセットするためのレジスタ）

レジスタ	用途
r10 ～ r19	作業用レジスタ（CX がコード生成時に使用するレジスタ）
r31	リンク・ポインタ
CTPC	CALLT 命令実行時のプログラム・カウンタ（PC）
CTPSW	CALLT 命令実行時のプログラム・ステータス・ワード（PSW）
EIPC	割り込み／例外処理時のプログラム・カウンタ（PC）
EIPSW	割り込み／例外処理時のプログラム・ステータス・ワード（PSW）
BSEL【V850E2V3】	レジスタ・バンク選択レジスタ
EIIC【V850E2V3】	EI レベル例外要因を保持するレジスタ
EIWR【V850E2V3】	EI レベル例外用作業レジスタ
FPSR【V850E2V3】	浮動小数点演算の設定／ステータス保持用レジスタ
FPEPC【V850E2V3】	浮動小数点演算例外プログラム・カウンタ

割り込み／例外処理が終わると，退避したレジスタを復帰するコードを出力し，最後に eiret 命令を出力します。この命令の発行により，割り込み／例外処理が終了したことを V850 に通知します。

CX では，“レジスタの退避／復帰”，および“reti 命令の出力”を「(c) 割り込み／例外ハンドラの記述方法」に従って記述すると自動的に出力します。“レジスタの退避／復帰”については，「表 3―24　割り込みのレジスタの退避／復帰処理」に従って出力します。つまり，ユーザは，この点について留意する必要なく，割り込み／例外ハンドラ本体処理の記述に専念することができます。

表 3―24　割り込みのレジスタの退避／復帰処理

レジスタ名		レジスタ	説明
アセンブラ予約レジスタ		r1	割り込み時には，必ず退避／復帰します。
引数用レジスタ		r6 ～ r9	割り込み要因が TRAP0/TRAP1 の場合には，r6 は，必ず退避／復帰します。 関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば，退避／復帰します。
作業用レジスタ	22 レジスタ・モード	r10 ～ r14	関数コールが存在した場合には，退避／復帰します。 関数コールが存在しない場合には，割り込み関数で使用していれば退避／復帰します。
	26 レジスタ・モード	r10 ～ r16	
	32 レジスタ・モード	r10 ～ r19	
レジスタ変数用レジスタ	22 レジスタ・モード	r25 ～ r29	通常の関数と同様に，必要に応じて退避／復帰します。
	26 レジスタ・モード	r23 ～ r29	
	32 レジスタ・モード	r20 ～ r29	
リンク・ポインタ		r31（lp）	関数コール（ランタイム関数を含む）が存在した場合には，退避／復帰します。 関数コールが存在しない場合には，退避／復帰しません。

レジスタ名	レジスタ	説明
割り込み関連システム・レジスタ	EIPC，EIPSW	多重割り込み（multi オプションを指定した割り込み関数）では，必ず退避／復帰します。 multi オプションを指定しない割り込み関数の場合には，退避／復帰しません。
callt 命令関連システム・レジスタ	CTPC，CTPSW	nopush オプションを指定しない割り込み関数では，必ず退避／復帰します。 nopush オプションを指定した割り込み関数の場合には，退避／復帰しません。
レジスタ・バンク選択レジスタ【V850E2V3】	BSEL	V850E2V3 のインストラクション・セットをもつデバイスを指定してコンパイルしている割り込み関数では，必ず退避／復帰します。
EI レベル例外要因を保持するレジスタ【V850E2V3】	EIIC	V850E2V3 のインストラクション・セットをもつデバイスを指定してコンパイルしている多重割り込み関数では，必ず退避／復帰します。
EI レベル例外用作業レジスタ【V850E2V3】	EIWR	V850E2V3 のインストラクション・セットをもつデバイスを指定してコンパイルしている多重割り込み関数では，必ず退避／復帰します。
浮動小数点演算の設定／ステータス保持用レジスタ【V850E2V3】	FPSR	V850E2V3 のインストラクション・セットをもつデバイスで FPU を持つデバイスを指定してコンパイルしている割り込み関数では，必ず退避／復帰します。
浮動小数点演算例外プログラム・カウンタ【V850E2V3】	FPEPC	V850E2V3 のインストラクション・セットをもつデバイスで FPU を持つデバイスを指定してコンパイルしている割り込み関数では，必ず退避／復帰します。

（c）割り込み／例外ハンドラの記述方法

割り込み／例外ハンドラの記述上の形態は，通常の C 言語関数と変わりませんが，C 言語で記述した関数を，CX に対して“割り込み／例外ハンドラ”として認識させる必要があります。CX では，割り込み／例外ハンドラの指定を“#pragma interrupt 指令”で行います。

```
#pragma interrupt　割り込み要求名　関数名　[ 配置方法 ]　[ オプション [ オプション ]…]
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。

- 割り込み要求名

デバイス・ファイルに登録されている割り込み要求名を指定できます。デバイス・ファイルに登録されている割り込み要求名は，各デバイスのユーザーズ・マニュアルの“割り込み要求名”に書かれてある文字列になりますので，そちらを参照してください。

なお，ノンマスカブル割り込み（NMI）も，この方法で指定できますが，リセット割り込み（RESET）は指定できません。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。リセット時の処理は，スタートアップ・ルーチンに記述するのが一般的であるため，詳細は「第７章　スタートアップ」を参照してください。
割り込み要求名に，“NO_VECT”と記述した場合は，割り込みハンドラ・アドレスには設定せず，関数を割込み関数としてのみ出力します。

- 関数名

割り込み／例外ハンドラとする関数名を指定します。ここでは“C 言語での関数名”を記述してください。“void func1 (void)”という関数を指定する場合は，関数名に“func1”と指定します。

- 配置方法

ハンドラ・アドレスに関数本体を直接配置するか，それとも割り込み／例外ハンドラ関数への分岐命令だけを配置するかを指定します。直接配置するときだけに“direct”を指定します。直接配置しない場合は，“配置方法”には何も記述しないでください。direct 指定を行うことによって，指定した割り込み要因のハンドラ・アドレスからすべて配置されますが，その結果，以降のハンドラ・アドレスの領域も使用されることになる可能性があるため注意が必要です。
また，direct 指定をする場合は，関数定義より後ろに #pragma interrupt 指令を記述するとコンパイル時にエラーとなります。必ず関数定義より前で行ってください。
割り込み要求名に，“NO_VECT”と記述した場合は，direct 指定はできず，エラーとなります。

- オプション

オプションには，以下のものを指定できます。

multi	多重割り込みハンドラとします。EIPC，EIPSW を退避／復帰するコードを出力します。 また，割り込みを許可状態とするコードを出力するので，__EI() で許可する必要はありません。 関数終了時は，割り込みを禁止にします。禁止にする直前に sync 命令を実施します。
nopush	CTPC，CTPSW の退避／復帰コードを出力しません。単一割り込みで，割込み関数の中に関数コールが存在しない場合は，このオプションでコードを短縮することができます。
push_ei	EIPC，EIPSW を退避するコードを出力します。
nopush_fpu	FPSR，FPEPC を退避するコードを出力しません。

“多重割り込みハンドラの指定”とは“「多重割り込みされる」ことを許可する関数を指定”ということです。“多重割り込みする関数を指定する”ということではありません。

次に，割り込みハンドラとして指定できる“関数の型”について説明します。

- 関数の型

マスカブル割り込み，NMI 割り込みのハンドラに関しては，次のようになります。

void func (void) 型

引数が void 型，戻り値が void 型の関数になります。
ソフトウエア例外処理（トラップ）ハンドラについては，次のようになります。

void func (unsigned int) 型

引数には EI レベル例外要因レジスタ（EIIC）の例外要因コードがセットされます。これらの型で指定しなければ，コンパイル時にエラーとなります。ソフトウエア例外処理関数については，次の項目を参照してください。

- ソフトウエア例外処理（トラップ処理）ハンドラ

ソフトウエア例外処理（トラップ処理）を使用する場合，V850 マイクロコントローラの仕様上，エントリ・ポイントは，“TRAP0（0x40 番地）”と“TRAP1（0x50 番地）”の 2 箇所になります。ソフトウエア例外“trap 0x00 ～ trap 0x0F”が入ったときは TRAP0（0x40 番地）へ，“trap 0x10 ～ trap 0x1F”が入ったときは，TRAP1（0x50 番地）へ分岐します。その際に，“ソフトウエア例外コード”として TRAP0 のときは“0x40 ～ 0x4F”の値が，TRAP1 のときは“0x50 ～ 0x5F”の値が EI レベル例外要因レジスタ（EIIC）にセットされます。

表 3―25 トラップ命令とソフトウエア例外コード

トラップ命令	ソフトウエア例外コード
trap 0x00	0x40
trap 0x01	0x41
trap 0x02	0x42
:	:
trap 0x0A	0x4A
trap 0x0B	0x4B
:	:
trap 0x10	0x50
trap 0x11	0x51
trap 0x12	0x52
:	:
trap 0x1E	0x5E
trap 0x1F	0x5F

TRAP0，TRAP1 に対するソフトウエア例外処理を記述する場合，その関数は，引数を 1 つ持ち，変数の型は“unsigned int 型”になります。その引数には EI レベル例外要因レジスタ（EIIC）にセットされた“ソフトウエア例外コード”が入ります。TRAP0 のときは“0x40 ～ 0x4F”の値が，TRAP1 のときは“0x50 ～ 0x5F”の値のいずかになります。ハンドラ内では，これらの値によって場合分けした処理を記述することになります。

```
#pragma interrupt   TRAP0   trap0_func
void trap0_func(unsigned int codenum) {
      :
    例外コード別に場合分けし，その処理を記述
      :
}
```

（d）割り込み／例外ハンドラの記述時の注意事項

- multi オプションで指定する“多重割り込みハンドラの指定”とは，“「多重割り込みされる」ことを許可する関数を指定”ということです。“多重割り込みする関数を指定する”ということではありません。
- #pragma interrupt 指令で，リセット割り込みは指定できません。

```
#pragma interrupt   RESET   reset_func  /* エラー */
```

上記のように記述をすると，コンパイル時にエラーになります。リセット時の処理に関しては，アセンブラ命令で記述する必要があります。

リセット時の処理は，スタートアップ・ルーチンに記述するのが一般的であるため，詳細は「第 7 章 スタートアップ」を参照してください。

- 多重割り込みを行うハンドラとして指定する関数には，multi オプションを指定して下さい。この場合，EIPC，EIPSW を退避／復帰するコードを出力します。multi オプションを指定しない割り込みハンドラは，EIPC，EIPSW を退避／復帰するコードを出力しません。
- #pragma interrupt 指令では，多重例外や多重 NMI には対応していません。多重例外や多重 NMI を行う場合は，必要となるシステム・レジスタ（FEPC，FEPSW など）の退避／復帰を行うコードを追加してください。必要となるシステム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。
- リンク・ディレクティブ・ファイルへの割り込みハンドラ・アドレスの追加記述は，ユーザで行う必要はありません。CX が内部的に出力します。
- 1 つの割り込み要求名に対し，異なる関数を複数指定することはできません。
- 割り込み／例外ハンドラとして指定された関数はインライン展開できません。#pragma inline 指定しても無視されます。
- 割り込み／例外ハンドラとして指定された関数は割り込み禁止となっているため，#pragma block_interrupt 指定されていても無視されます。
- 割り込み／例外ハンドラとして指定された関数は，通常の関数呼び出しで呼び出すことはできません。ただし，別ファイルから呼び出された場合は，コンパイラでチェックできません。
- 割り込み／例外ハンドラからアセンブラ命令を呼び出し，「表 3―20 レジスタ変数用レジスタ」，および「表 3―21 割り込み／例外ハンドラ用のスタック・フレーム」に示されたレジスタを使用する場合，退避／復帰処理を記述する必要があります。また，SP（r3），GP（r4），TP（r5），EP（r30）を書き換える場合も，退避／復帰処理を記述する必要があります。
- #pragma interrupt 指令では，外部割り込みコントローラに対する処理終了通知（EOI コマンド）は発行していません。必要な場合はユーザで実行してください。
- 多重割り込みの最後は，EIPC，EIPSW の復帰コードが入るので，割り込み禁止にしてください。

- direct 指定をしない場合，ハンドラ・アドレスには“割り込み／例外ハンドラへの分岐命令”が配置されますが，その場合 CX はコード効率の面から“jr 命令”を出力しています。ただし，jr 命令で分岐できる範囲には限界があり，jr 命令から± 21 ビット内になります。もし，割り込みハンドラ本体へ jr 命令で分岐できる範囲になかった場合，リンカでエラーになります。その場合は，コンパイル・オプション“-Xfar_jump オプション”を指定することにより，jr 命令を jmp 命令に置き換える処置をしてください。
- FE レベル割り込みはサポートしていません。
- multi オプションが指定された場合，デバイスの仕様上，push_ei の有無にかかわらず EIPC/EIPSW を退避するコードを出力します。エラーは出力されません。
- nopush_fpu は FPU 非搭載デバイスでは意味を持たず，暗黙に指定されたものとみなします。指定がなくとも FPSR/FPEPC を退避するコードは出力しません（FPU 非搭載デバイスでは FPSR/FPEPC がありません）。エラーは出力されません。

（e）割り込み／例外ハンドラの記述例

割り込み／例外ハンドラの記述例を次に示します。

ただし，割り込み要求名は，デバイスによって異なりますので，各デバイスのユーザーズ・マニュアルを参照してください。

例 1. ノンマスカブル割り込みの場合

```
#pragma interrupt   NMI     func1   /* ノンマスカブル割り込み */
void func1(void) {
  :
}
```

2. トラップの場合

```
#pragma interrupt   TRAP0   func2   /* トラップ */
void func2(unsigned int num) {
    switch(num) {                   /* 例外コード別に場合分け */
      :
    }
}
```

3. 多重割り込みの場合

```
#pragma interrupt   INTP0   func1   multi   /* 多重割り込み */
void func1(void) {
  :
}
```

（9）インライン展開

CX では，関数ごとのインライン展開ができます。ここでは，インライン展開の指定について説明します。

(a) インライン展開とは

インライン展開とは，関数呼び出し部分に関数本体を展開することを言います。これにより，関数呼び出しによるオーバーヘッドが小さくなり，また，最適化の可能性が高められることから，実行速度向上を図ることができます。

ただし，インライン展開を行うと，オブジェクト・サイズは増大することになります。

インライン展開したい関数は，#pragma inline で指定します。

```
#pragma inline  関数名 [, 関数名 , ...]
```

関数名は，C 言語記述の関数名を記述してください。たとえば，"void func1 ( ) { }" という関数であれば "func1" と指定します。また，関数名は“,”（カンマ）で区切って複数指定することができます。

```
#pragma inline  func1, func2
void    func1() {...}
void    func2() {...}
void func(void) {
    func1();    /* インライン展開対象 */
    func2();    /* インライン展開対象 */
}
```

(b) インライン展開の条件

#pragma inline 指定された関数をインライン展開するためには，最低限次の条件が必要となります。

ただし，CX の内部処理の関係により，次の条件を満たしていてもインライン展開されない場合があります。

- インライン展開を“する関数”と“される関数”を同一ファイル内に記述する

  インライン展開を“する関数”と“される関数”，つまり，“関数呼び出し”と“関数定義”は“同一ファイル内”に存在しなければなりません。別の C ソースに書かれてある関数をインライン展開することはできません。この場合，CX はエラーも警告メッセージも出力せず，インライン展開指定を無視します。

- #pragma inline を“関数定義より前”に記述する

  pragma inline が，関数定義よりも後ろに記述されていた場合，警告を出力してインライン展開指定を無視します。ただし，関数のプロトタイプ宣言との記述順序は問いません。次に例を示します。

例

【インライン展開指定：有効】

```
#pragma inline  func1, func2
void    func1();  /* プロトタイプ宣言 */
void    func2();  /* プロトタイプ宣言 */
void    func1() {…}  /* 関数定義 */
void    func2() {…}  /* 関数定義 */
```

【インライン展開指定：無効】

```
void    func1();  /* プロトタイプ宣言 */
void    func2();  /* プロトタイプ宣言 */
void    func1() {…}  /* 関数定義 */
void    func1() {…}  /* 関数定義 */
#pragma inline  func1, func2
```

- インライン展開する関数の“呼び出し”と“定義”の間で，“引数の数”を同じにする
  インライン展開する関数の“呼び出し”と“定義”の間で“引数の数”が違う場合，インライン展開指定を無視します。

- インライン展開する関数の“呼び出し”と“定義”の間で，“戻り値の型”や“引数の型”を同じにする
  インライン展開する関数の“呼び出し”と“定義”の間で，“戻り値の型”や“引数の型”が異なる場合，インライン展開指定を無視します。ただし，引数の型が整数型（enum を含む），またはポインタ型でサイズが同じ場合は，インライン展開を行います。

- インライン展開する関数の引数は“可変個”にしない
  引数が“可変個”の関数にインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。

- “再帰関数”はインライン展開できない
  自分自身を呼び出す“再帰関数”をインライン展開指定した場合，エラーも警告メッセージも出力せず，インライン展開指定を無視します。ただし，関数呼び出しが複数ネストし，そのネストした中に自分自身を呼び出すコードが存在した場合，インライン展開する場合があります。

- “割り込みハンドラ”はインライン展開できない
  #pragma interrupt で記述された関数は“割り込みハンドラ”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

- リアルタイム OS の“タスク”はインライン展開できない
  #pragma rtos_task で指定された関数は，リアルタイム OS の“タスク”として認識されますが，この関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

- #pragma block_interrupt 指定で関数内を割り込み禁止にすると，インライン展開できない
  #pragma block_interrupt で，関数内を割り込み禁止として宣言された関数に対してインライン展開指定した場合，警告メッセージを出力して，インライン展開指定を無視します。

（c）実行速度優先最適化とインライン展開

CX の最適化の 1 つである“実行速度優先最適化（-Ospeed）”をオプション指定した場合，CX はインライン展開を最適化手段の 1 つとします。

したがって，実行速度優先最適化（-Ospeed）が指定されていれば，#pragma inline でインライン展開指定した関数“以外”でも「（b）インライン展開の条件」をクリアしていれば，CX が適切な関数を選択し，インライン展開を行います。

(d) オプション指定によるインライン展開動作の違いの例

#pragma inline 指定とオプション指定による“インライン展開動作の違い”の例は，次のようになります。

“-Osize（サイズ優先最適化）指定”（-Ospeed 以外）

```
#pragma inline  func0
void    func0() {...}    /*#pragma inline 指定により，インライン展開の条件が合致すれば
                           展開 */
void    func1() {...}    /* 展開しない */
void    func2() {...}    /* 展開しない */
```

“-Ospeed（実行速度優先最適化）指定”

```
#pragma inline  func0
void    func0() {...}    /*-Ospeed 指定により，インライン展開の条件が合致すれば展開 */
void    func1() {...}    /*-Ospeed 指定により，インライン展開の条件が合致すれば展開 */
void    func2() {...}    /*-Ospeed 指定により，インライン展開の条件が合致すれば展開 */
```

備考 1. CX では，#pragma inline によりインライン展開指定された関数は，静的関数として扱いません。静的関数とするには，明示的に static 指定をする必要があります。

2. デバッグの際，インライン展開をした関数に対して C ソース・レベルでブレークポイントを設定することはできません。

(10) リアルタイム OS 対応機能

CX は，V850 マイクロコントローラ用リアルタイム OS “RX850V4”を使用したシステムを構築する場合を考慮し，プログラミング記述性向上と，コード削減の機能を備えています。

(a) タスクの記述

リアルタイム OS を使用したアプリケーションは“タスク”を処理単位とします。リアルタイム OS は，そのタスク内で発行された“システム・コール”や“割り込み処理”をきっかけとして，タスクのスケジューリングを行います。タスクを切り替えるとき（コンテキストを切り替えるとき）のレジスタの退避／復帰作業は，リアルタイム OS が行うため，一般の関数としてのプロローグ処理／エピローグ処理とは異なります。

したがって，CX が，関数呼び出し時に生成するプロローグ処理／エピローグ処理は，タスクでは実行されないことになります。

記述された関数を“タスク”とする場合，関数呼び出し時のプロローグ処理／エピローグ処理を削除することにより，コード削減がはかれますが，C 言語の記述上，“一般の関数”と“タスク”は区別がつきません。そこで CX では，関数を“リアルタイム OS のタスク”と認識させるために，次の #pragma 指令を用意しています。

```
#pragma rtos_task    [ 関数名 ]
```

これにより，“関数名”で指定された関数を，リアルタイム OS のタスクとして認識させることができます。“関数名”は，C 言語記述の関数名を指定します。例として，“void func1 (int inicode) { }”という関数をタスクとする場合は，次のように記述します。

例

```
#pragma rtos_task   func1
```

#pragma rtos_task を指定すると，具体的には次のような効果が得られます。

- 通常の関数で出力される“プロローグ／エピローグ処理”を行いません。具体的には次のようなコードを出力しません。
  - レジスタ変数用レジスタの退避／復帰
  - リンク・ポインタ（lp）の退避／復帰
  - 戻り先へのジャンプ

- システム・コール“ext_tsk”を，定義済みの関数として使用することができます。
  特にアプリケーション内でプロトタイプ宣言しなくても，このシステム・コールを使用することができます。#pragma rtos_task の記述以降であれば，タスク指定した関数以外でも，同様に呼び出すことができます。
  このシステム・コールを呼び出したとき，コード・サイズ削減のために“jr 命令”を使用したコードを出力します。システム・コール“ext_tsk”本体が，jr 命令で分岐可能な範囲にない場合は，リンカでエラーになります。この場合は，次の処置が必要になります。
  - リンク・ディレクティブでメモリ配置を変更する
  - アセンブラ・ソースで jmp 命令による分岐に切り替える
  - far jump 指定する

  また，#pragma rtos_task 指定した場合，次のような注意事項があります。
  - 関数と同じように，タスクを呼び出すことはできません。ただし，別のファイルで呼び出された場合はチェックされません。また，関数として呼び出すことができないため，インライン展開することができません。したがって，#pragma rtos_task 指定された関数に，#pragma inline 指定しても，#pragma inline 指定は無視されます。
  - “#pragma rtos_task 関数名”を，同じファイル内の関数定義よりも後ろに記述した場合，エラーとなります。
  - “#pragma rtos_task 関数名”を記述したあと，そのファイル内に関数定義を記述しない場合は，その関数に対する #pragma 指令は無視されます。ただし，"#pragma rtos_task" の記述は有効であり，その後ろで呼ばれる関数において ext_tsk() システム・コールを使用することは可能です。
  - #pragma rtos_task 指定された関数は，通常の割り込み／例外ハンドラ（「(8) 割り込み／例外処理ハンドラ」を参照）として指定することはできません。

なお，リアルタイム OS の機能については，各リアルタイム OS のユーザーズ・マニュアルを参照してください。

（11）組み込み関数

CX 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，CX 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 1 命令を出力します。

組み込み関数の引数に暗黙の型変換が不可能な型の引数が指定された場合は，警告を出力し，通常の関数として扱います。また，ldsr()/stsr()/ldgr()/stgr() に対し，ハードウエアにないレジスタ番号を指定した場合も警告を出力し，通常関数として扱います。

以下に，関数として記述できる命令を示します。

表 3―26 組み込み関数

アセンブラ命令	機能	組み込み関数
di	割り込み制御	`__DI();`
ei		`__EI();`
nop	ノー・オペレーション	`__nop();`
halt	プロセッサの停止	`__halt();`
satadd	飽和加算	`long a, b;` `long __satadd(a, b);`
satsub	飽和減算	`long a, b;` `long __satsub(a, b);`
bsh	ハーフワード・データのバイト・スワップ	`long a;` `long __bsh(a);`
bsw	ワード・データのバイト・スワップ	`long a;` `long __bsw(a);`
hsw	ワード・データのハーフワード・スワップ	`long a;` `long __hsw(a);`
sxb	バイト・データの符号拡張	`char a;` `long __sxb(a);`
sxh	ハーフワード・データの符号拡張	`short a;` `long __sxh(a);`
mul	mul 命令を用いて 32 ビット× 32 ビットの符号つき乗算結果の 64 ビットを変数に代入する命令	`long a, b;` `long long __mul(a, b);`
mulu	mulu 命令を用いて 32 ビット× 32 ビットの符号なし乗算結果の 64 ビットを変数に代入する命令	`unsigned long a, b;` `Unsigned long long __mulu(a,b);`
mul32	mul32 命令を用いて乗算結果の上位 32 ビットを変数に代入する命令	`long a, b;` `long __mul32(a, b);`

アセンブラ命令	機能	組み込み関数
mul32u	mulu32 命令を用いて符号なし乗算結果の上位 32 ビットを変数に代入する命令	unsigned long a, b; unsigned long __mul32u(a, b);
sasf	論理左シフト付きフラグ条件の設定	long a; unsigned int b; long __sasf(a, b);
sch0l	MSB 側からのビット（0）検索【V850E2V3】	long a; long __sch0l(a);
sch0r	LSB 側からのビット（0）検索【V850E2V3】	long a; long __sch0r(a);
sch1l	MSB 側からのビット（1）検索【V850E2V3】	long a; long __sch1l(a);
sch1r	LSB 側からのビット（1）検索【V850E2V3】	long a; long __sch1r(a);
ldsr	システム・レジスタへのロード【V850E2V3】	long a; void __ldsr(regID 注, a);
stsr	システム・レジスタの内容のストア【V850E2V3】	unsigned long __stsr(regID 注);
ldgr	汎用レジスタへのロード【V850E2V3】	long a; void __ldgr(regID 注, a);
stgr	汎用レジスタの内容のストア【V850E2V3】	unsigned long __stgr(regID 注);
caxi	比較と交換【V850E2V3】	long *a; long b, c; void __caxi(a, b, c);

注 regID にはシステム・レジスタ番号（0 ～ 31）を指定してください。
ただし，ldsr では regID として 0 を指定しないでください。

注意 組み込み関数と同名の関数を定義して使用することはできません。
同名の関数を呼び出そうとしても，コンパイラが用意している組み込み関数処理を優先します。

(12) 構造体パッキング

CX は，構造体メンバのアライメントを C 言語レベルで指定できます。この機能は，-Xpack オプションと同等ですが，構造体パッキング指令は C ソース内の任意の位置でアライメント値を指定できます。

注意 構造体をパッキングすると，データ領域を小さくできますが，プログラム・サイズは増え，実行速度も低下します。

(a) 構造体パッキングの形式

構造体パッキング機能は次の形式で指定します。

```
#pragma pack([1248])
```

#pragma pack は，この指令が出現した時点で，構造体メンバのアライメント値に変更します。この数値をパッキング値と呼び，指定できる数値は，1，2，4，または 8 です。パッキング値は省略できません。パッキング値を記述しない場合，次のメッセージが出力されます。

```
E0521605: error: #pragma pack の文法が不正です。
```

なお，この指令は出現した時点で有効となるので C ソース内に複数記述することができます。

例

```
#pragma pack(1) /* 構造体メンバを 1 バイトのアライメントで整列 */
struct  TAG {
    char    c;
    int     i;
    short   s;
};
```

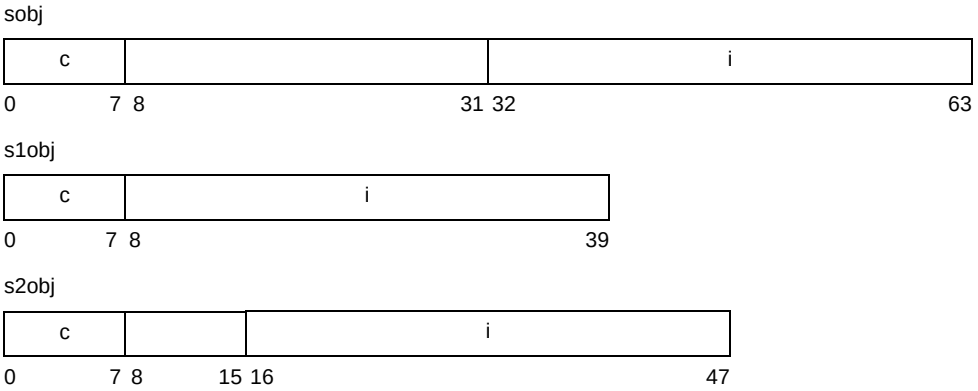
(b) 構造体パッキングのルール

構造体のメンバは，構造体のパッキング値とメンバの持つアライメント値の小さい方の値の整列条件を満たす形で並べられます。

たとえば，構造体のパッキング値が 2 のときメンバの形が int 型ならば 2 バイトの整列条件を満たす形で並べられます。

例

```
struct  S {
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*4 バイトの整列条件を満たす */
};

#pragma pack(1)
struct  S1 {
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*1 バイトの整列条件を満たす */
};

#pragma pack(2)
struct  S2 {
    char    c;      /*1 バイトの整列条件を満たす */
    int     i;      /*2 バイトの整列条件を満たす */
};

struct  S   sobj;   /* サイズ 8 バイト */
struct  S1  s1obj;  /* サイズ 5 バイト */
struct  S2  s2obj;  /* サイズ 6 バイト */
```



(c) 共用体

共用体をパッキングの対象として構造体パッキングと同様に扱います。

例 1.

```
union   U {
    struct  S {
        char    c;
        int     i;
    } sobj;
};

#pragma pack(1)
union   U1 {
    struct  S1 {
        char    c;
        int     i;
    } s1obj;
};

#pragma pack(2)
union   U2 {
    struct  S2 {
        char    c;
        int     i;
    } s2obj;
};

union   U   uobj;   /* サイズ 8 バイト */
union   U1  u1obj;  /* サイズ 5 バイト */
union   U2  u2obj;  /* サイズ 6 バイト */
```

2.

```
union   U {
    int i:7;
};

#pragma pack(1)
union   U1 {
    int i:7;
};

#pragma pack(2)
union   U2 {
    int i:7;
};

union   U   uobj;   /* サイズ 4 バイト */
union   U1  u1obj;  /* サイズ 1 バイト */
union   U2  u2obj;  /* サイズ 2 バイト */
```

(d) ビット・フィールド

ビット・フィールド要素の領域は次のように割り当てます

- 構造体のパッキング値がメンバの型の整列条件値と等しいあるいは大きい場合

  構造体パッキング機能を利用しなかったときと同じように割り当てます。つまり，続けて割り当てるとその領域がメンバの型の整列条件を満たす境界を越えてしまう場合，その整列条件を満たしている領域から割り当てます。

- 構造体のパッキング値が要素の型の整列条件値より小さい場合
  - 続けて割り当てるとその領域を含むバイト数が要素の型よりも大きくなる場合

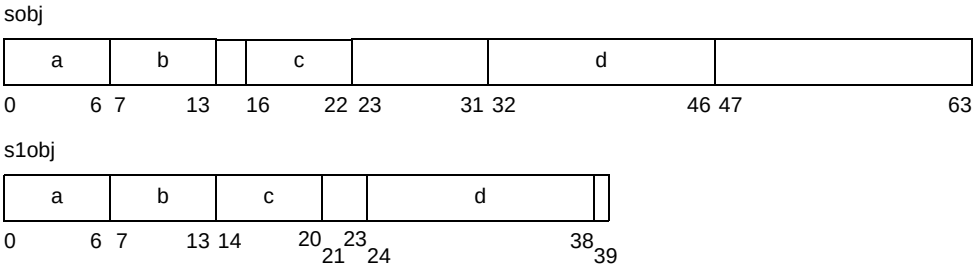
    構造体のパッキング値の整列条件を満たす形で割り当てます。
  - それ以外の場合

    続けて割り当てます。

例

```
struct  S {
    short   a:7;    /*0 ～ 6 ビット目 */
    short   b:7;    /*7 ～ 13 ビット目 */
    short   c:7;    /*16 ～ 22 ビット目（2 バイト境界に整列）*/
    short   d:15;   /*32 ～ 46 ビット目（2 バイト境界に整列）*/
} sobj;

#pragma pack(1)
```

```
struct  S1 {
    short   a:7;    /*0 ～ 6 ビット目 */
    short   b:7;    /*7 ～ 13 ビット目 */
    short   c:7;    /*14 ～ 20 ビット目 */
    short   d:15;   /*24 ～ 38 ビット目（バイト境界に整列）*/
} s1obj;
```



(e) 構造体オブジェクトの先頭の整列条件

構造体オブジェクトの先頭の整列条件は，構造パッキング機能を利用しなかったときと同じです。

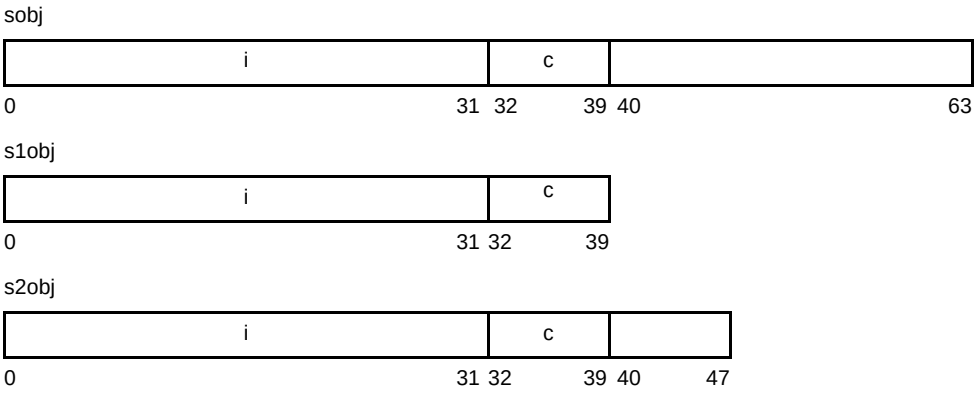
(f) 構造体オブジェクトのサイズ

構造体のサイズが構造体の整列条件の値と構造体のパッキング値の小さい方の値の倍数になるようにパッキングを行います。オブジェクトの先頭の整列条件は構造パッキング機能を利用しなかったときと同じです。

例 1.

```
struct  S {
    int     i;
    char    c;
};

#pragma pack(1)
struct  S1 {
    int     i;
    char    c;
};

#pragma pack(2)
struct  S2 {
    int     i;
    char    c;
};

struct  S   sobj;   /* サイズ 8 バイト */
```

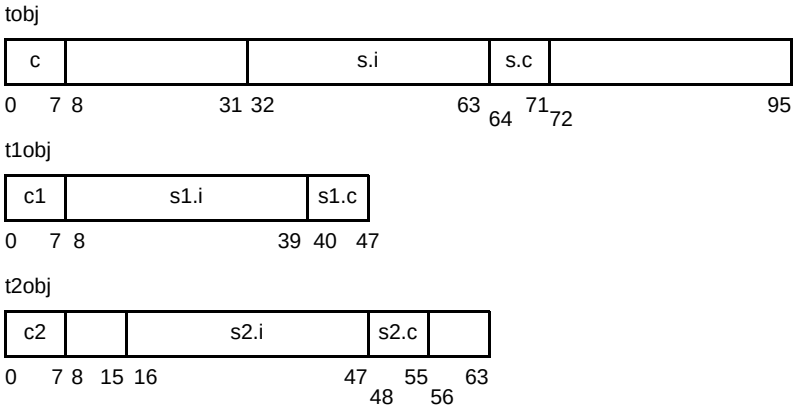
```
struct  S1  s1obj;  /* サイズ 5 バイト */
struct  S2  s2obj;  /* サイズ 6 バイト */
```



2.

```
struct  S {
    int     i;
    char    c;
};
struct  T {
    char    c;
    struct  S   s;
};

#pragma pack(1)
struct  S1 {
    int     i;
    char    c;
};
struct  T1 {
    char    c;
    struct  S1  s1;
};

#pragma pack(2)
struct  S2 {
    int     i;
    char    c;
};
struct  T2 {
    char    c;
    struct  S2  s2;
```

```
};

struct  T   tobj;   /* サイズ 12 バイト */
struct  T1  t1obj;  /* サイズ  6 バイト */
struct  T2  t2obj;  /* サイズ  8 バイト */
```

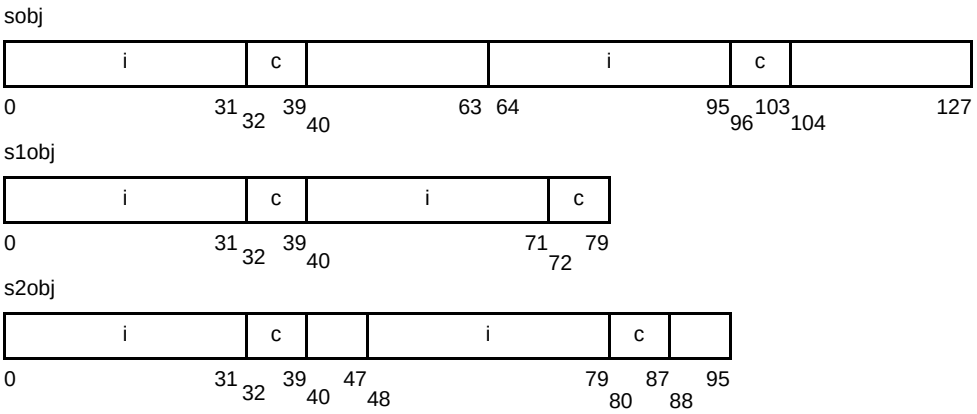


（g）構造体配列のサイズ

構造体オブジェクトの配列のサイズは要素である構造体オブジェクトのサイズに要素数を乗算した値です。

例

```
struct  S {
    int     i;
    char    c;
};

#pragma pack(1)
struct  S1 {
    int     i;
    char    c;
};

#pragma pack(2)
struct  S2 {
    int     i;
    char    c;
};

struct  S   sobj[2];    /* サイズ 16 バイト */
struct  S1  s1obj[2];   /* サイズ 10 バイト */
struct  S2  s2obj[2];   /* サイズ 12 バイト */
```

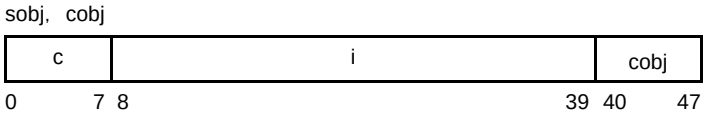


（h）オブジェクト間の領域

たとえば，次のソース・プログラムでは，sobj.c，sobj.i，cobj が隙間なく続いて配置される可能性があります（sobj，cobj の配置順は保証されません）。

例

```
#pragma pack(1)
struct  S {
    char    c;
    int     i;
} sobj;
char    cobj;
```



（i）構造体パッキング機能の注意点

- -Xpack オプションと #pragma pack 指令の同時指定について

C ソース中に #pragma pack 指令で構造体パッキング指定がある時に -Xpack オプションを指定した場合，最初の #pragma pack 指令が出現するまではオプション指定値がすべての構造体に適用されます。それ以降は #pragma pack 指令の値が適用されます。

ただし，#pragma pack 指令の出現後でも指定がデフォルトになった部分は，オプション指定値が適用されます。

例 （-Xpack=2 を指定した場合）

```
struct  S2 {...};   /* オプションでパッキング値 2 を指定している
                       オプション -Xpack=2 が有効：パッキング値 2*/
#pragma pack(1)     /*#pragma 指令でパッキング 1 を指定している
struct  S1 {...};     pragma pack(1) が有効：パッキング値 1*/
#pragma pack()      /*#pragma 指令でパッキング値にデフォルトを指定している
struct  S2_2 {...};   オプション -Xpack=2 が有効：パッキング値 2*/
```

- 制限事項

V850 マイクロコントローラ，V850Ex 製品ミス・アライン・アクセス禁止の設定の CPU をご使用の場合，次の制限があります。

- 構造体メンバのアドレスでのアクセスが正しく行えません。

  次のように構造体メンバのアドレスを取得して，そのアドレスでのアクセスは，デバイスのデータ・アライメントに従い，アドレスをマスクしてアクセスされるため，データの消失や切り捨てが生じます。

例

```
struct  test {
    char    c;      /*offset 0*/
    int     i;      /*offset 1-4*/
} test;
int *ip, i;

void func(void) {
    i = *ip;        /* マスクされたアドレスでアクセスされる */
}

void func2(void) {
    ip = &(test.i); /* 構造体メンバのアドレス取得 */
}
```

- ビット・フィールドへのアクセスは，そのメンバの型で読み込むためデータがない領域もアクセスします。

  次のようにビット・フィールドの幅がメンバの型以下の場合，メンバの型で読み込むのでオブジェクトの外部にアクセスします。実行上，通常は問題ありませんが，I/O などがマップされていた場合に不正なアクセスとなる場合があります。

例

```
struct  S {
    int x:21;
} sobj;  /*3バイト*/
sobj.x = 1;
```

（13）スマート・コレクション機能

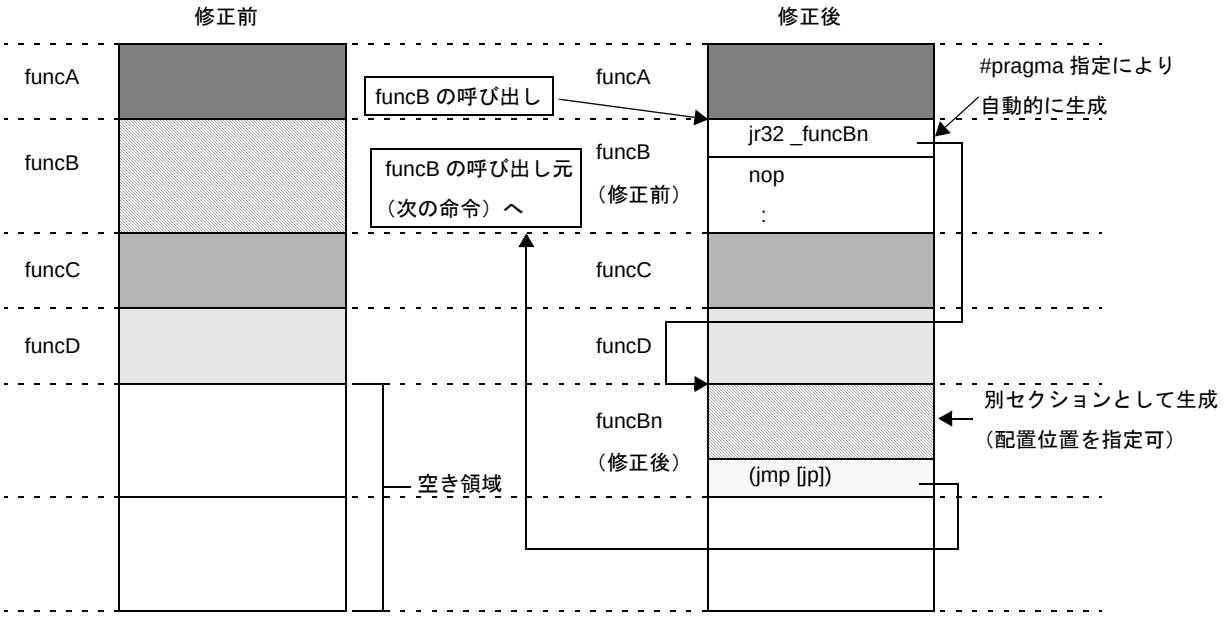
スマート・コレクションとは，特定の関数を修正したい場合に，ほかの関数の内容（コード，およびアドレス）をまったく変更することなく，その関数の実行のみを置き換える機能です。

具体的には，修正する関数のサイズは変更せずに，その内容を修正後の関数へのジャンプ命令（#pragma 指定により，自動的に生成）に置き換えます。

修正後の関数は，元のプログラムに影響しない別のセクションに配置します。

そうすることにより，修正する関数以外は，プログラムは修正前と同一となります。

図 3―16 スマート・コレクションのメモリ・イメージ



スマート・コレクションは，プログラムをフラッシュ領域に書き込んだ後に特定の関数で不具合が見つかった場合に有効な機能です。

不具合の修正後に通常どおりに再コンパイルする場合と比べて，以下のメリットがあります。

- 通常どおりに再コンパイルする場合は，修正する関数以外の関数の内容（配置アドレスや分岐先アドレス）が変わるため，プログラム全体の評価が必要です。
  しかし，スマート・コレクション機能を使用した場合は，修正する関数以外の箇所は変わらないため，再評価を最小限に抑えることができます。
- フラッシュを全面的に書き換える必要がなく，変更箇所のみを書き換えればよいため，セルフ書き換えも可能です。

(a) スマート・コレクションの形式

スマート・コレクション機能は次の形式で指定します。

```
#pragma smart_correct    修正前関数名      修正後関数名
```

修正する関数をそのまま残し，別名で新たにコピーを行い，その関数を修正します。CX は，修正前の関数の位置には修正後の関数を呼び出すコードを出力します。

(b) スマート・コレクションの手順

- 最初のコンパイル時に関数間最適化オプションなど注意すべきオプションが使用されてないことを確認します。
- 修正前のリンク・ディレクティブ・ファイルを準備します。
- 修正する関数をコピーして，その関数を含む C ソース・ファイルの最後に追加します。
  追加した関数を修正して，別の名前に変更します。
- 修正前関数の前に，#pragma smart_correct 指令を追加します。
  この C ソース・ファイルに対して，この #pragma 指令の追加，およびファイルの最後への修正後関数の追加以外の変更は行わないでください。
  #pragma smart_correct 指令により，修正前関数から修正後関数へのジャンプ命令を自動的に生成します。
- 追加した関数に対して，#pragma text 指令で配置セクション名を指定してください。

```
#pragma text " 配置セクション名 " 修正後の関数名
```

また，静的変数を追加した場合には，その変数に対しても #pragma section 指令を用いて配置セクションを指定してください。この場合のセクション名は，元のプログラムに存在しない新たな名前を指定してください。

- リンク・ディレクティブ・ファイルに，修正後関数名の配置アドレスを指定します。
- コンパイル・オプションは，最初のコンパイル時と同じオプションを指定し再コンパイル／アセンブル／リンクします。修正前の関数から修正後の関数へのジャンプ命令は，far jump を指定した関数と同等のコードが生成されます。
- 修正前のヘキサ・ファイルと修正後のヘキサ・ファイルとの差分が，修正分であることを確認します。

(c) スマート・コレクションの記述例

- 元のプログラム prog（prog.lmf/prog.hex）が，file1.c，file2.c，file3.c の 3 個の C ソース・ファイルで構成されているものとします。このうち，file1.c で定義された関数 funcB に不具合があった場合，まず funcB をコピーして file1.c の最後に追加し，関数名を funcBn に変更します。その後で funcBn を修正します。
- 関数 funcB の定義より前に #pragma smart_correct 指令（a）を追加します。
- 修正後の関数 funcBn の定義の前に，#pragma text 指令（b）を追加します。ここでは，funcBn を”text.rc”という名前のセクションに配置することを指定します。

【修正前の file1.c】	【修正後の file1.c】

void funcA()
{
:
funcB();
:
}
 	
void funcB()
{
:
}
 	
void funcC()
{
:
}
	void funcA()
	{
	:
	funcB();
	:
	}
	
	#pragma smart_correct funcB funcBn ← (a)
	void funcB()
	{
	:
	}
	
	void funcC()
	{
	:
	}
	
	#pragma text “text.rc”funcBn ← (b)
	void funcBn()
	{
	:
	}

- リンク・ディレクティブに”text.rc”セクションの配置の指定を追加します。

例 text.rc セクションを 0x2000000 番地に配置します。

```
TEXT.RC: !LOAD ?RX V0x2000000 {
    text.rc = $PROGBITS ?AX text.rc.text;
};
```

- 元のプログラムのコンパイル／アセンブル／リンク時と全く同じオプションを設定して，再コンパイル／アセンブル／リンクします。
- 元の prog.hex と生成された新しい prog.hex とを比較して，funcB，および funcBn 以外の差分がないことを確認してください。

```
【修正後の file1.asm】
_funcA:
    :
    jarl funcB
    :
$smart_correct _funcB, _funcB.End, _funcBn
_funcB:
    :
_funcB.End:
_funcC:
    :
text.rc.text .cseg text
_funcBn:
    jmp [lp]
_funcBn.end:
```

```
【修正後の file1.obj のアセンブラ・イメージ】
_funcA:
    :
    jarl funcB
    :
_funcB:
    jr32 funcBn
    nop
    :
_funcB.End:
_funcC:
    :
text.rc.text .cseg text
_funcBn:
    jmp [lp]
_funcBn.end:
```

元々の funcB と同じサイズ

(d) スマート・コレクション機能の注意事項

- 修正前関数内の変数のみ追加／削除／変更ができます。
- 関数外で定義してある変数の削除／変更はできません。追加は，別セクションで定義することにより可能です。
- 変数を追加する場合は，明示的にセクション，および配置位置を指定して，すでに存在するデータ領域を変更しないように注意してください。
- 初期値あり変数の追加は，ROM 化時にコピー・サイズが変わる場合があるため，初期値あり変数の追加は行わないでください。
- 個別コピーの場合は，該当関数も修正対象として考慮する必要があります。
- 修正前の関数サイズは，修正後関数の呼び出しに必要なコード・サイズ以上としてください。

- 修正前関数は，関数内に閉じた最適化のみ適用されます。
- 修正前関数がインライン展開（最適化）の対象の場合，スマート・コレクション対象とするようにうながすメッセージが出力されます。
- #pragma で指定したセクション名には，コンパイラが自動で ".text" という文字列を付加します。リンク・ディレクティブ内の入力セクションに対象セクション名を指定する場合，自動付加された ".text" まで記述が無いとリンカは対象セクション名を識別できないため，不要なセクションであると判断します。

（14）ポジション・インディペンデント操作

CX では通常，変数／関数にアクセスする場合，相対アドレシングを出力し，ポジション・インディペンデントなコードを出力しています。本機能では，変数／関数にアクセスするアドレシングを，ポジション・インディペンデントで出力するか固定アドレスで出力するかを変更することができます。

たとえば，マルチコア・プログラミングにおいて，コアごとのセクションはコアごとのベース・レジスタからの相対となっていますが，他のコアや共通モジュールにアクセスする場合，ベース・レジスタの値が異なっているため，絶対アドレスで指定する必要があります。本機能では，その制御を行うことができます。

（a）ポジション・インディペンデント操作の形式

変数／関数のポジション・インディペンデント操作は次の形式で指定します。

```
#pragma pic
#pragma nopic
```

（b）ポジション・インディペンデント操作の記述例

#pragma pic 指令が指定されると，それ以降に宣言／定義された変数／関数へのアクセスは相対アドレスになり，#pragma nopic 指令が指定されると，それ以降に宣言／定義された変数／関数へのアクセスは絶対アドレスになります。

例

```
#pramga nopic
extern int i;   /* i は、絶対アドレスによるアクセスになります */
#pragma pic
extetern int j; /* j は、相対アドレシングによるアクセスになります */
```

同じ指定が繰り返され指定された場合は，エラーになりませんが，

```
#pragma nopic
extern int i;
#pragma nopic   /* エラーとなりません */
extern int j;
```

異なった指令が同じ変数に対して行われた場合は，エラーとなります。

```
#pragma nopic
extern int i;
#pragma pic   /* エラーとなります */
int i;
```

マルチコア・プログラミングを行う場合，他のコア用モジュールにて定義された変数にアクセスしたい場合は，#pragma nopic 指令の後に宣言を行います。-Xmult=cmn オプション指定された場合は，#pragma nopic はデフォルトなので記述する必要はありません。

例

各 PE（Processing Element）用プログラム（-Xmulti=pen）

```
#pragma nopic
/* 共通部の宣言 */
extern int cmn_var;
extern int cmn_func();
#pragma pic
/* PE ローカル部の宣言 */
int pe_var;
int pe_func();
```

共通部（-Xmulti=cmn）

```
#pramga nopic   /* 書いても書かなくとも良い */
int cmn_var = 0;
int cmn_func(){
        return 1;
}
```

（c）ポジション・インディペンデント操作の注意事項

- -Xmulti オプションがない場合，あるいは -Xmulti=pen オプションが指定された場合，#pragma pic がファイル先頭に書かれたものとして扱います。この場合，通常の出力コードと同一となります。
- -Xmulti=cmn オプション指定時には，暗黙に #pragma nopic がファイル先頭に書かれたものとして扱います。-Xmulti=cmn オプション指定時に，#pragma pic を記述するとエラーとなります。
- -Xmulti=cmn オプション指定時に，#pragma section にて，sdata/sidata/sedata/tidata/tidata_byte/tidata_word を 指定するとエラーとします。
- -Xmulti 指定なし，あるいは -Xmulti=pen オプション指定時には，#pragma nopic 指定と #pragma section にて，sdata/sidata/sedata/tidata/tidata_byte/tidata_word を 指定するのは，同時利用可能ですが，コード効率は悪くなります。
- 複数の宣言で異なる指定があるものはエラーとします。ヘッダファイルに記述する場合に注意が必要です。

- シンボルファイルに関係する機能（-Xsfg*，-Xsymbol_file）と同時指定はできません。また，指定された場合の動作は未定義となります。

3. 2. 5 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。しかし，拡張機能はV850 マイクロコントローラに則したもので，他に利用するためには修正が必要になる場合があります。

ここでは，他の C コンパイラから CX への移植と，CX 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 C コンパイラから CX ＞

- #pragma 注

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

- 拡張仕様

  他の C 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 C 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。

＜ CX から他の C コンパイラ＞

- CX 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 __CA850__
#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 __V850__
#define bit char        /*bit 型変数を char 型変数にします */
#endif
```

3.3 関数呼び出しインタフェース

この節では，CX 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。

3.3.1 C 言語関数間の呼び出し

- 通常の関数呼び出し
  → jarl 命令
- 関数を指すポインタを用いた関数呼び出し
  → jmp 命令

C 言語関数から C 言語関数を呼び出す際，4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，4 ワードを越えた引数は，呼び出し側の関数のスタック・フレームに格納します。構造体，double/long long 型の引数も同様に下位バイトから順に r6 から格納します。その後，呼び出された関数へ移行（ジャンプ）し，呼び出されるときに格納された“引数用レジスタの値”を，呼び出された関数側で，呼び出された関数側のスタック・フレームに格納します。

構造体を返す関数の場合は，第一引数として呼び出し側の関数で確保した返り値用領域のアドレスを渡します。その場合，ソースで指定された第一引数，第二引数，…はそれぞれ，第二引数，第三引数，…として扱われます。

CX では，関数の戻り値に r10 を用います。関数が double/long long 型の場合は，r10，r11 を用い，r10 に下位 32bit，r11 に上位 32bit を格納します。構造体を返す関数の場合は，第一引数で渡されたアドレスに構造体を格納するため明示的な返り値はありません。

スタック・フレームは，関数のプロローグ・コード，つまり，関数が呼び出されてから，関数本体のコードを実行する前に実行されるコード（「図 3―19　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」，「図 3―21　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）」で示す（4）～（7）の処理がプロローグ・コードになります）において，スタック・ポインタ（sp）を，必要サイズ分だけずらすことによって生成されます。また，関数のエピローグ・コード，つまり，関数本体のコードを実行し終わり，呼び出し側の関数に戻るまでに実行されるコード（「図 3―19　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」，「図 3―21　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）」で示す（i）～（iv）の処理がエピローグ・コードになります）において，スタック・ポインタ（sp）を戻すことにより，スタック・フレームは消滅します。

(1) スタック・フレーム／関数呼び出し

スタック・フレームの形状，および関数呼び出し時のスタック・フレームの生成／消滅状態について説明します。

(a) スタック・フレームの形状

CX では，スタック・フレームは引数の条件によって，引数レジスタ領域を“スタックの先頭”か“スタックの中央”のどちらかに確保します。引数の条件は次のようになります。

- 引数レジスタ領域が，スタックの先頭に配置される場合

  4 ワード分の引数領域を越えて，連続アクセスする必要がある場合で，次の 2 通りが該当します。

  - 引数が可変個引数の場合

- 引数が構造体実体で，その領域が 4 ワード境界をまたぐ場合

- 引数レジスタ領域が，スタックの中央に配置される場合

この場合は上記条件以外の場合になります。

引数レジスタ領域を“スタックの先頭”に持つ場合のスタック・フレームを「図 3―17 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）」，“スタックの中央”に持つ場合のスタック・フレームを「図 3―18 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）」に示します。

図 3―17 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）

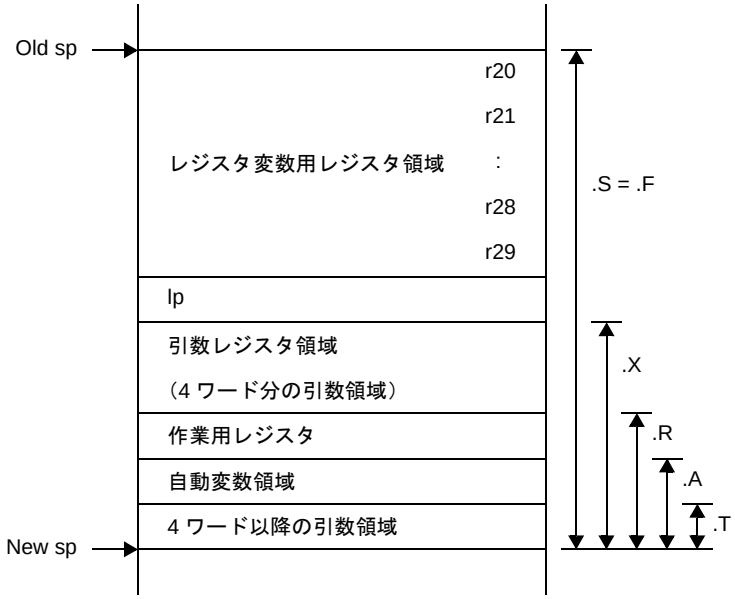
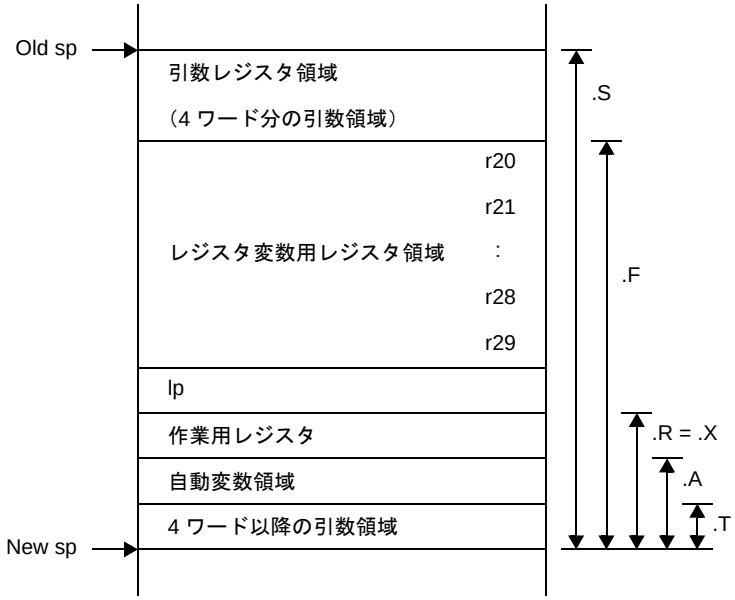


図 3―18 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）



図中にある“.S，.F，.X，.R，.A，.T”は，コンパイラが内部的に出力する関数用マクロです。
それぞれ，使用用途が決まっており，次表のようになります。

表 3―27 関数用マクロ

マクロ名	意味
.S	スタック・サイズ
.F	スタック・サイズ - 引数レジスタ領域のサイズ（スタックの先頭にある場合）
.X	引数レジスタ領域のサイズ（スタックの中央にくる場合）＋ .R
.R	作業用レジスタ領域のサイズ ＋ .A
.A	自動変数領域のサイズ ＋ .T
.T	呼び出す関数の 4 ワード以降の引数領域のサイズ
.P	常に 0（コード生成用のマクロ）**注**

注 .P は常に 0 のため，「図 3―17 スタック・フレーム（引数レジスタ領域がスタック中央になる場合）」，「図 3―18 スタック・フレーム（引数レジスタ領域がスタック先頭になる場合）」には記述してありません。

これらのマクロを使用して，スタック領域にアクセスすることになりますが，具体的なアクセス方法（出力するアクセス・コード）は次表のようになります。

表 3―28 スタック領域のアクセス方法

スタック領域	アクセス方法（ディスプレースメント [sp]）
レジスタ変数用レジスタ領域（lp も含む）	-offset + .F*xx*[sp]
作業用レジスタ領域	-offset + .R*xx*[sp]
自動変数領域	-offset + .A*xx*[sp]
4 ワード以降の引数領域	offset + .P*xx*[sp]
引数レジスタ領域	offset + .F*xx*[sp]
引数レジスタ領域（スタックの中央にくる場合）	offset + .R*xx*[sp]

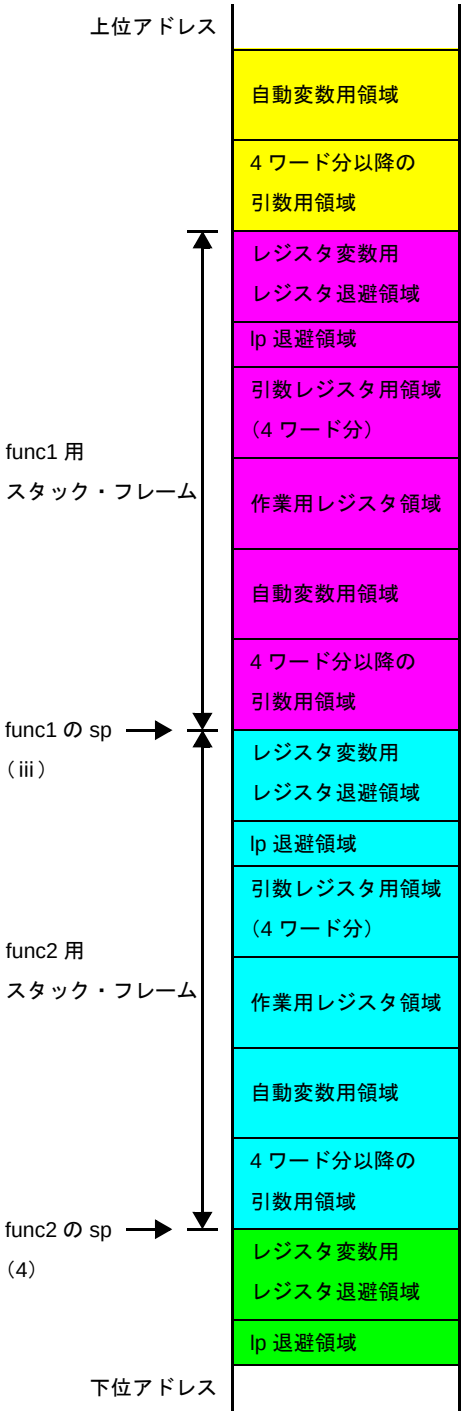
表中で“offset”は正の整数で，各領域中のオフセットを意味します。また，マクロの後に書かれている“*xx*”は正の整数で，関数のフレーム番号を示します。

(b) 関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの中央”にくる場合）

“引数レジスタ領域がスタックの中央にくる場合”の，関数呼び出し時のスタック・フレームの生成と消滅について説明します。ほとんどの関数呼び出しは，このケースになります。

以下に，関数 func1 から関数 func2 を呼び出し，その後関数 func1 へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 3―19　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）



【関数 func2 を呼び出すときの関数 func1 側の処理】

（1）引数を引数用レジスタに格納

呼び出す func2 の引数を r6 ～ r9 へ格納

（2）4 ワード分以降の引数をスタックへ格納

r6 ～ r9 の 4 つに入りきらない引数を格納

（3）jarl 命令で func2 へ分岐

【関数 func1 が呼び出されたときの関数 func2 側の処理】

（4）sp をずらす

func2 が使用するスタックへ移動

（5）lp を退避

func1 の戻りアドレスを格納

（6）レジスタ変数用レジスタを退避

func2 でもレジスタ変数用レジスタを使用するとき，func1 で使用していたレジスタ値を保護する必要があるために退避

（7）引数を“引数レジスタ用領域”に格納

r6 ～ r9 の値を格納

func2 から他関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

V850Ex の場合，（4）～（6）の処理は prepare 命令で実行できるため，CX は prepare 命令を出力します。

【関数 func2 から関数 func1 へ戻るときの関数 func2 側の処理】

（i）レジスタ変数用レジスタの復帰

退避していた“func1 のレジスタ変数用レジスタの値”をレジスタへ戻す

（ii）lp の復帰

func1 の戻りアドレスを復帰

（iii）sp を戻す

func1 が使用するスタックへ移動

（iv）jmp [lp] で戻る

V850Ex の場合，（ i ）～（ iv ）の処理は dispose 命令で実行できるため，CX は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

- 呼び出す側～関数 func1
  - 呼び出す func2 の引数が 4 ワードを越えていた時，越えた分の引数の値

- 呼び出される側～関数 func2
  - 引数用レジスタに入れられた引数の受け渡し（引数用レジスタに入れるのは，呼び出す側（関数 func1））
  - 呼び出した側（関数 func1）のリンク・ポインタ（lp）（= 関数 func1 の戻りアドレス）の退避
  - “レジスタ変数用レジスタ”の退避

    “レジスタ変数用レジスタ”として割り当てられているのは，次のようになります。

    22 レジスタ・モードのとき：“r25，r26，r27，r28，r29”

    26 レジスタ・モードのとき：“r23，r24，r25，r26，r27，r28，r29”

    32 レジスタ・モードのとき：“r20，r21，r22，r23，r24，r25，r26，r27，r28，r29”

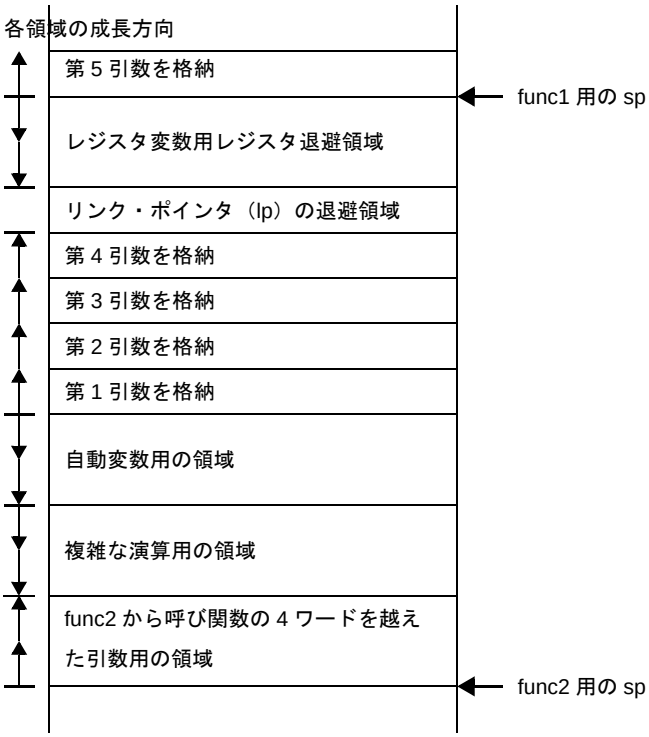
    このうち使用しているものを退避します。
  - “自動変数用”の領域
  - 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保

    この領域を使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます。

また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2 の引数が 5 個あるとします）。

図 3―20　スタック・フレームの各領域のスタック成長方向



次に“C 言語関数から C 言語関数を呼び出したソース”と“それをコンパイルしたときのアセンブラ・ソース”の具体例を示します。

例

```
void func1(void) {
    int a, b, c, d, e;
    func2(a, b, c, d, e);
      :
}
int func2(int a, int b, int c, int d, int e) {
    register int    i;
      :
    return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

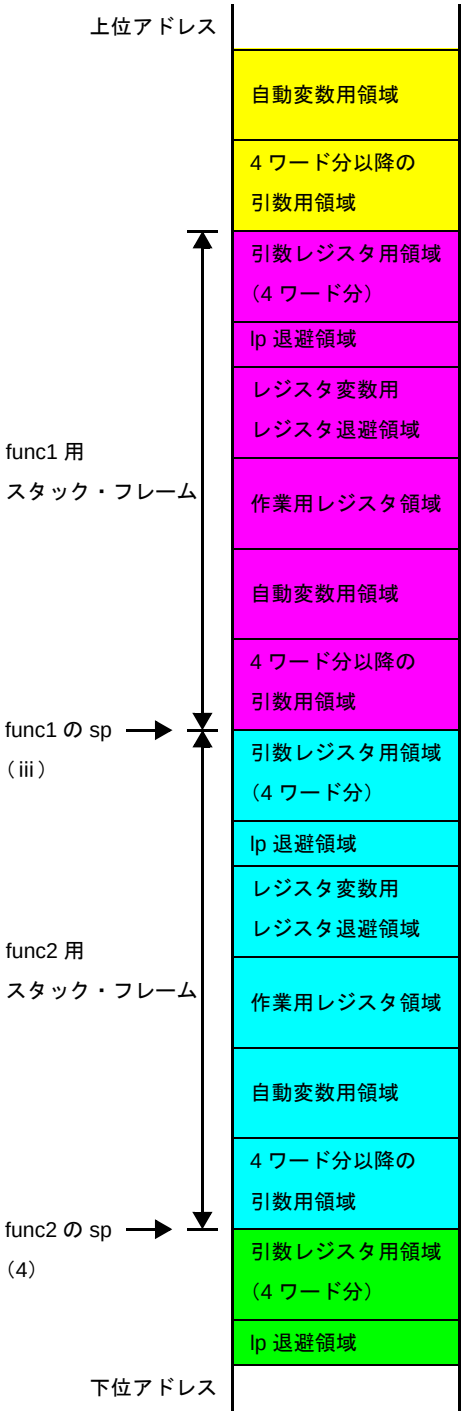
```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    --（ii）から（iv）の処理
.L3:
    --func1 に対するプロローグ
    --(4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .R2[sp]
    st.w    r7, 4 + .R2[sp]
    st.w    r8, 8 + .R2[sp]         -- (7)
    st.w    r9, 12 + .R2[sp]
    st.w    r29, -4 + .A2[sp]
      :
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    dispose .X2, 0x3, [lp]
    --（i）, （ii）, （iii）, （iv）
.L5:
    prepare 0x3, .X2
    -- (4), (5), (6)
    jbr     .L6
```

(c) 関数呼び出し時のスタック・フレームの生成／消滅（引数レジスタ領域が“スタックの先頭”にくる場合）

“引数レジスタ領域がスタックの先頭にくる場合”の関数呼び出し時のスタック・フレームの生成と消滅について説明します。

以下に，関数 func1 から関数 func2 を呼び出し，その後関数 func1 へ戻るときの，スタック・フレームの生成／消滅の例を示します。

図 3―21　スタック・フレームの生成／消滅（引数レジスタ領域がスタックの先頭にくる場合）



【関数 func2 を呼び出すときの関数 func1 側の処理】

（1）引数を引数用レジスタに格納

呼び出す func2 の引数を r6 ～ r9 へ格納

（2）4 ワード分以降の引数をスタックへ格納

r6 ～ r9 の 4 つに入りきらない引数を格納

引数が 5 つ以上の場合は，この処理を行う対象となります。

（3）jarl 命令で func2 へ分岐

【関数 func1 が呼び出されたときの関数 func2 側の処理】

（4）sp をずらす

func2 が使用するスタックへ移動

（5）lp を退避

func1 の戻りアドレスを格納

（6）レジスタ変数用レジスタを退避

func2 でもレジスタ変数用レジスタを使用するとき，func1 で使用していたレジスタ値を保護する必要があるために退避

（7）引数を“引数レジスタ用領域”に格納

r6 ～ r9 の値を格納

func2 から他関数を呼び出すとき，その時の引数をレジスタ（r6 ～ r9）に格納することになるため，現状の引数値をスタックへ格納しておく

V850Ex の場合，（4）～（6）の処理は prepare 命令で実行できるため，CX は prepare 命令を出力します。

【関数 func2 から関数 func1 へ戻るときの関数 func2 側の処理】

（i）レジスタ変数用レジスタの復帰

退避していた“func1 のレジスタ変数用レジスタの値”をレジスタへ戻す

（ii）lp の復帰

func1 の戻りアドレスを復帰

（iii）sp を戻す

func1 が使用するスタックへ移動

（iv）jmp [lp] で戻る

V850Ex の場合，（ i ）～（ iv ）の処理は dispose 命令で実行できるため，CX は dispose 命令を出力します。

スタック・フレームに退避されるものと，使用されるスタック・フレームをまとめると次のようになります。

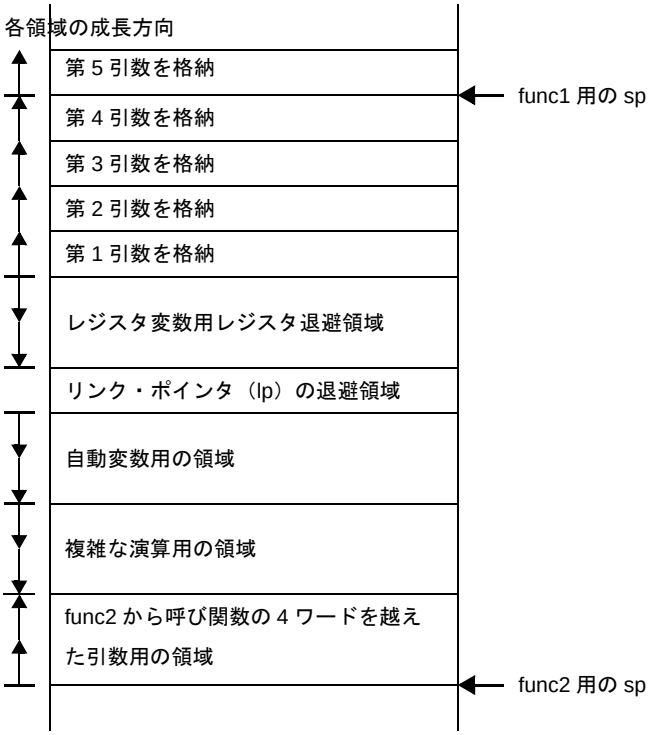
- 呼び出す側～関数 func1
  - 呼び出す func2 の引数が 4 ワードを越えていた時，越えた分の引数の値

- 呼び出される側～関数 func2
  - 引数用レジスタに入れられた引数の受け渡し（引数用レジスタに入れるのは，呼び出す側（関数 func1））
  - 呼び出した側（関数 func1）のリンク・ポインタ（lp）（= 関数 func1 の戻りアドレス）の退避
  - “レジスタ変数用レジスタ”の退避
  - “自動変数用”の領域
  - 関数内で非常に複雑な式が用いられた場合，演算のために使用する領域の確保
    この領域を使用する必要が出てきた場合には，自動変数用の領域の下位側に確保されます。

また，関数に戻り値がある場合，その値は r10 に格納されます。

スタック・フレームの各領域の配置と各領域のスタック成長方向のイメージを図にすると，次のようになります（呼び出す関数 func2 の引数が 5 個あるとします）。

図 3―22 スタック・フレームの各領域のスタック成長方向



次に“C 言語関数から C 言語関数を呼び出したソース”と“それをコンパイルしたときのアセンブラ・ソース”の具体例を示します。

例

```
void func1(void) {
    int a, b, c, d, e;
    func2(a, b, c, d, e);
      :
}
int func2(int a, int b, int c, int d, int e) {
    register int    i;
      :
    return(i);
}
```

例の関数 func2 呼び出しに対して生成されるアセンブラ命令

```
_func1:
    jbr     .L3
.L4:
    ld.w    -8 + .A3[sp], r6
    ld.w    -12 + .A3[sp], r7
    ld.w    -16 + .A3[sp], r8       -- (1)
    ld.w    -20 + .A3[sp], r9
    ld.w    -24 + .A3[sp], r10
    st.w    r10, [sp]               -- (2)
    jarl    _func2, lp              -- (3)
      :
    --func1 に対するエピローグ
    --（ii）から（iv）の処理
.L3:
    --func1 に対するプロローグ
    -- (4), (5) の処理
      :
    jbr     .L4
_func2:
    jbr     .L5
.L6:
    st.w    r6, .F2[sp]
    st.w    r7, 4 + .F2[sp]
    st.w    r8, 8 + .F2[sp]         -- (7)
    st.w    r9, 12 + .F2[sp]
      :
    st.w    r29, -4 + .A2[sp]
```

```
    jbr     .L2
.L2:
    ld.w    -4 + .A2[sp], r10
    dispose .X2, 0x3
    -- (i), (ii), (iii)
    add     .S2 - .F2, sp           -- (iii)
    jmp     [lp]                    -- (iv)
.L5:
    add     .F2 - .S2, sp           -- (4)
    prepare 0x3, .X2
    -- (4), (5), (6)
    jbr     .L6
```

3. 3. 2 関数のプロローグ／エピローグ

CX は，関数のプロローグ／エピローグ処理の一部分を“ランタイム・ライブラリ呼び出し”にすることで，オブジェクト・サイズの削減を行うことができます。これを“プロローグ／エピローグのランタイム化”と呼びます。つまり，関数のプロローグ／エピローグ処理は決まった処理なので，これらを“ランタイム・ライブラリ関数”として CX でまとめて用意し，関数呼び出し時や関数戻り時にこれらの関数を呼び出します。

関数のプロローグ／エピローグ部分のアセンブラ命令の例を次に示します。

なお，例中の“(数字)”は「図 3―19 スタック・フレームの生成／消滅（引数レジスタ領域がスタックの中央にくる場合）」の説明に対応しています。

例

```
int func(int a, int b, int c, int d, int e) {
    register int    i;
      :
    return(i);
}
```

上記例の関数 func のプロローグ／エピローグ部分のアセンブラ命令

【ランタイムを使用しない場合の出力コード】

```
_func:
.BB.LABEL.0:
        prepare 3, 16           --(4)(5)(6)
        st.w    r6, 0[sp]
        st.w    r7, 4[sp]
        st.w    r8, 8[sp]       --(7)
        st.w    r9, 12[sp]
              :
        mov     r29, r11
.BB.LABEL.1:
        mov     r11, r10
        dispose 16, 3, [lp]     --(i),(ii),(iii),(iv)
_func.end:
```

【ランタイムを使用する場合の出力コード】

```
_func:
.BB.LABEL.0:
        callt   9               --(4)(5)(6)
        st.w    r6, 0[sp]
        st.w    r7, 4[sp]
        st.w    r8, 8[sp]       --(7)
        st.w    r9, 12[sp]
              :
        mov     r29, r11
.BB.LABEL.1:
        mov     r11, r10
        callt   39              --(i),(ii),(iii),(iv)
_func.end:
```

(1) 関数のプロローグ／エピローグのランタイム化の指定

プロローグ／エピローグのランタイム化は，コンパイル・オプション“-Xpro_epi_runtime=on”を指定することにより行います。逆にランタイム化しない場合には“-Xpro_epi_runtime=off”を指定します。

ただし，最適化オプション“-Ospeed（実行速度優先最適化）”指定時以外のとき，自動的に関数のプロローグ／エピローグのランタイム化が行われます（自動的にコンパイラ・オプション“-Xpro_epi_runtime=on”が指定されます）。

“-Ospeed オプション”以外を指定し，かつ，ランタイム化をしたくない場合は“-Xpro_epi_runtime=off オプション”を指定する必要があります。

なお，“-Xpro_epi_runtime オプション”は，ソース・ファイル個別に指定することも可能であるため，“ランタイム化するファイル”と“ランタイム化しないファイル”を混在することができます。

また，“-Xpro_epi_runtime=on オプション”で，関数のプロローグ／エピローグのランタイム化を行う場合，専用のセクション“.pro_epi_runtime セクション”が必要となります。

したがって，リンク・ディレクティブでは，次のように定義する必要があります。

```
.pro_epi_runtime = $PROGBITS    ?AX     .pro_epi_runtime;
```

このセクションには，プロローグ／エピローグ・ランタイム関数のテーブル情報が配置されます。

(2) 関数のプロローグ／エピローグのランタイム化

関数のプロローグ／エピローグ・ランタイム関数の呼び出しには，CALLT 命令を使用します。

CALLT 命令は“2 バイト命令”で，関数呼び出しにこの命令を使用することによって，コード・サイズ削減の効果が得られます。CALLT 命令は“CTBP（Callt Base Pointer）レジスタ”に“CALLT 命令の対象となる，関数のテーブル”を指すポインタを設定が必要となります。設定処理がプログラム内になかった場合，リンク時にエラー・メッセージが出力されます。

CTBP レジスタへの値設定は，アセンブラ命令で行う必要があるため，スタートアップ・ルーチンで行うのが適当です。

したがって，次の命令をスタートアップ・ルーチンに追加してください。

```
mov     #___PROLOG_TABLE, r12   --“PROLOG”の前の“_( アンダースコア )”は 3 つ
ldsr    r12, 20
```

このとき，___PROLOG_TABLE が“関数のプロローグ／エピローグのランタイム関数”の関数テーブルの先頭シンボルとなり，CTBP に設定され，関数テーブル自体は“.pro_epi_runtime セクション”に配置されます。また，上記では“汎用レジスタ r12”を使用していますが，特に r12 である必要はありません。

この CX が提供する“関数のプロローグ／エピローグのランタイム化”で CALLT 命令を使用する以外で CALLT 命令を使用する場合，CTBP レジスタを退避／復帰する必要があります。もし，CALLT 命令を他のオブジェクト，たとえば，ミドルウエアやユーザ作成のライブラリで使用しており，かつ，その中に CTBP レジスタの退避／復帰コードがない，または挿入できない場合，“関数のプロローグ／エピローグのランタイム化”は使用できません。その場合は“-Xpro_epi_runtime=off オプション”を指定して，ランタイム化を抑止してください。

なお，CALLT 命令や CTBP レジスタについての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

(3) 関数のプロローグ／エピローグのランタイム化の注意事項

関数のプロローグ／エピローグのランタイム化において，次のような注意事項があります。

- 関数のプロローグ／エピローグのランタイム化は，関数呼び出しが入るため，その分，実行速度は低下します。これを避けたい場合は“-Xpro_epi_runtime=off オプション”を指定してください。ファイル単位で“-Xpro_epi_runtime=off オプション”を指定すると効果的です。
- 呼び出される関数が少ないプログラムの場合，プロローグ／エピローグのランタイム化を行っても，コード・サイズの削減ができない場合があります。効果が見込めない場合は“-Xpro_epi_runtime=off ”を指定してください。
- 割り込み関数の場合は，プロローグ／エピローグ・ランタイム化は行いません。割り込み関数から呼び出される関数については，プロローグ／エピローグ・ランタイム化の対象になります。

3. 3. 3 far jump 機能

CX は，関数を呼び出す際に，次の“jarl 命令”を使用したコードを出力します。

```
jarl    _func1, lp
```

V850 マイクロコントローラのアーキテクチャ上，jarl 命令の第 1 オペランドとして指定できるのは，符号拡張した 22 ビット値まで（22 ビット・ディスプレースメント）となっています。

したがって，分岐点から± 2M バイト範囲内に分岐先がなかった場合，分岐ができず，リンカでエラー・メッセージが出力されます。

これを解決するには，以下のように配置することですぐに解決できますが，ターゲット・システムにおいては，この範囲内に分岐先を配置できないことがあります。これを解決するのが“far jump 機能”です。

- 分岐先を，分岐点から± 2M バイト範囲内に配置する

far jump 機能を使用すると，関数を呼び出すときに“jmp 命令”を使用したコードを出力します。これにより，V850 が持つ 32 ビット空間すべてに分岐できるようになります。ただし，汎用レジスタを 1 つ使うことになります。far jump 機能を使用した関数呼び出しを“far jump 呼び出し”と呼びます。

（1）far jump 指定の方法

far jump 呼び出しを行う場合“far jump 呼び出しを行う関数”を列挙したファイル（far jump 呼び出し関数一覧ファイル）を用意し，コンパイラ・オプション“-Xfar_jump”を使用します。

```
-Xfar_jump=far jump 呼び出し関数一覧ファイル
```

“far jump 呼び出し関数一覧ファイル”の書式については次を参照してください。

（2）far jump 呼び出し関数一覧ファイル

far jump 呼び出しの対象とする関数を列挙する“far jump 呼び出し関数一覧ファイル”の書式について説明します。far jump 機能を適用したい関数を，1 行に 1 関数を記述していきます。記述する名前は，C 言語関数名の先頭に“_（アンダースコア）”を付けた名前になります。

【far jump 呼び出し関数一覧ファイルのサンプル】

```
_func_led
_func_beep
_func_motor
  :
_func_switch
```

“_関数名”のかわりに次のようにを記述すると，すべての関数を far jump 呼び出しの対象にします。

```
{all_function}
```

{all_function} 指定があると，他に“_関数名”があっても，すべての関数が far jump 呼び出しの対象になります。

なお，far jump 機能は，ユーザ関数だけではなく，次のものにも適用できます。

- 標準ライブラリ関数
- ランタイム・ライブラリ関数
- リアルタイム OS のシステム・コール

“_関数名”のかわりに次のようにを記述すると，すべての割り込み関数を far jump 呼び出しの対象にします。

```
{all_interrupt}
```

“far jump 呼び出し関数一覧ファイル”の注意事項は次のとおりです。

- 使用できる文字は ASCII 文字のみです。
- コメントは挿入できません。
- 1 行に 1 関数のみです。
- 関数名の前後に，空白やタブを挿入できます。
- 1 行に 1023 文字まで記述できます。空白やタブも 1 文字と数えます。
- 関数名は C 言語関数の先頭に“_（アンダースコア）”付で記述してください。
- ブートーフラッシュ再リンク機能と併用することはできません。

(3) far jump 機能の使用例

far jump 機能の使用例について説明します。

(a) ユーザ関数（標準関数も同様）

【C ソース・ファイル】

```
extern  void    func3(void);

void func(void)
{
    func3();
}
```

【far jump 呼び出し関数一覧ファイル】

```
_func3
```

【通常の呼び出しコード】

```
#@CALL_ARG
jarl    _func3, lp
```

【far jump 呼び出しコード】

```
#@CALL_ARG
    movea   _func3, tp, r10
    movea   .L18, tp, lp
    jmp     [r10]
.L18:
```

(b) ランタイム関数（マクロ呼び出しの場合）

【far jump 呼び出し関数一覧ファイル】

```
___mul
```

【通常の呼び出しコード】

```
.macro  mul     arg1, arg2
    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    jarl    ___mul, lp
    ld.w    4[sp], r7
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

【far jump 呼び出しコード】

```
.macro  mul     arg1, arg2
    .local  macro_ret
    add     -8, sp
    st.w    r6, [sp]
    st.w    r7, 4[sp]
    mov     arg1, r6
    mov     arg2, r7
    movea   macro_ret, tp, r31
    .option nowarning
    movea   #___mul, tp, r1
    jmp     [r1]
    .option warning
macro_ret:
    ld.w    4[sp], r7
```

```
    mov     r6, arg2
    ld.w    [sp], r6
    add     8, sp
.endm
```

(c) ランタイム関数の場合（ダイレクト呼び出しの場合）

【far jump 呼び出し関数一覧ファイル】

```
___mul
```

【通常の呼び出しコード】

```
mov     r12, r6
mov     r13, r7
#@CALL_ARG      r6, r7
#@CALL_USE      r6, r7
jarl    ___mul, lp
mov     r6, r13
```

【far jump 呼び出しコード】

```
    mov     r12, r6
    mov     r13, r7
    #@CALL_ARG      r6, r7
    #@CALL_USE      r6, r7
    movea   #___mul, tp, r14
    movea   .L13, tp, lp
    jmp     [r14]
.L13:
    mov     r6, r13
```

ランタイムのマクロ呼び出しとダイレクト呼び出しはコンパイラが最適化の過程でレジスタ効率などを判定し自動的に切り替えます。

(d) リアルタイム OS のシステム・コールの場合

【far jump 呼び出し関数一覧ファイル】

```
_ext_tsk
```

【通常の呼び出しコード】

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
jr      _ext_tsk    --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

【far jump 呼び出しコード】

```
#@B_EPILOGUE
#@BEGIN_NO_OPT
add     .S4, sp
movea   #_ext_tsk, tp, r10
jmp     [r10]       --C NR
#@END_NO_OPT
#@E_EPILOGUE
```

3.4 セクション名一覧

以下に，予約されているセクションの名前とそれらのセクション・タイプ，およびセクション属性を示します。

表 3―29 予約セクション

名前[注1]	内容	セクション・タイプ	セクション属性
.bss	.bss セクション	NOBITS	AW
.const	.const セクション	PROGBITS	A
.data	.data セクション	PROGBITS	AW
.ext_info .ext_info_boot	ブートーフラッシュ再リンク機能用情報セクション	PROGBITS	なし
.ext_table	ブートーフラッシュ再リンク機能用分岐テーブル・セクション	PROGBITS	AX
.ext_tgsym	ブートーフラッシュ再リンク機能用情報セクション	PROGBITS	なし
.gptabname	グローバル・ポインタ・テーブル[注2]	GPTAB	なし
.pro_epi_runtime	プロローグ／エピローグ・ランタイム呼び出しセクション	PROGBITS	AX
.regmode	レジスタ・モード情報	REGMODE	なし
.relname	リロケーション情報	REL	なし
.relaname	リロケーション情報	RELA	なし
.sbss	.sbss セクション	NOBITS	AWG
.sconst	.sconst セクション	PROGBITS	A
.sdata	.sdata セクション	PROGBITS	AWG

名前[注1]	内容	セクション・タイプ	セクション属性
.sebss	.sebss セクション	NOBITS	AW
.sedata	.sedata セクション	PROGBITS	AW
.shstrtab	セクション名を保持しているストリング・テーブル	STRTAB	なし
.sibss	.sibss セクション	NOBITS	AW
.sidata	.sidata セクション	PROGBITS	AW
.strtab	ストリング・テーブル	STRTAB	なし
.symtab	シンボル・テーブル	SYMTAB	なし
.text	.text セクション	PROGBITS	AX
.tibss	.tibss セクション	NOBITS	AW
.tibss.byte	.tibss.byte セクション	NOBITS	AW
.tibss.word	.tibss.word セクション	NOBITS	AW
.tidata	.tidata セクション	PROGBITS	AW
.tidata.byte	.tidata.byte セクション	PROGBITS	AW
.tidata.word	.tidata.word セクション	PROGBITS	AW
.debug_info	デバッグ情報	PROGBITS	なし
.debug_line	行番号情報	PROGBITS	なし
.debug_loc	位置リスト情報	PROGBITS	なし
.version	バージョン情報	PROGBITS	なし
.float_info	浮動小数点演算情報	FLOATINFO	なし
.multi	マルチコア情報	MULTI	なし

注 1. .gptabname，.relname，および .relaname の name の部分は，それぞれそのセクションに対応するセクションの名前を示します。

2. リンカにおける -Xsdata_info オプションの処理において用いられる情報です。

備考 マルチコア用の予約セクション名（デフォルトのセクション名）は，末尾に“.cmn/.pe*n* (n=1…N)”が付加されます。

マルチコア用の予約セグメント名（デフォルトのセグメント名）は，末尾に“_CMN/_PE*n* (n=1…N)”が付加されます。

以下にマルチコア用の予約セクション名と予約セグメント名を示します。

- 予約セクション名

.sconst.cmn, .pro_epi_runtime, .const.cmn, .text.cmn, .data.cmn, .bss.cmn,
.sconst.pe1, .const.pe1, .text.pe1, .data.pe1, .sdata.pe1, .sbss.pe1, .bss.pe1 ,.sedata.pe1, .sebss.pe1,
.tidata.byte.pe1, .tibss.byte.pe1, .tidata.word.pe1, .tibss.word.pe1, .sidata.pe1, .sibss.pe1

:

.sconst.pe*n*, .const.pe*n*, .text.pe*n*, .data.pe*n*, .sdata.pe*n*, .sbss.pe*n*, .bss.pe*n*,.sedata.pe*n*, .sebss.pe*n*,
.tidata.byte.pe*n*, .tibss.byte.pe*n*, .tidata.word.pe*n*,.tibss.word.pe*n*,.sidata.pe*n*, .sibss.pe*n*

- 予約セグメント名

SCONST_CMN, CONST_CMN, TEXT_CMN, DATA_CMN,

SCONST_PE1, CONST_PE1, TEXT_PE1, DATA_PE1, SEDATA_PE1, SIDATA_PE1,

:

SCONST_PE*n*, CONST_PE*n*, TEXT_PE*n*, DATA_PE*n*, SEDATA_PE*n*, SIDATA_PE*n*

第 4 章　アセンブラ言語仕様

この章では，CX 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

アセンブリ言語文は，“シンボル”，“ニモニック”，“オペランド”，および“コメント”から構成されます。

```
[ シンボル ][: △ ]　　[ ニモニック ]　　[ オペランド ], [ オペランド ]　　;[ コメント ]
```

ラベルを記述する場合は，コロン，または 1 つ以上の空白で区切ります。ただし，コロンか空白かはニモニックで記述する命令によります。

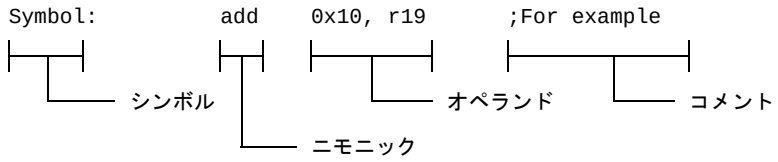
以下の箇所には，1 つ以上の空白があってもなくても問題ありません。

- シンボルとコロンの間
- コロンとニモニックの間
- 2 つ目以降のオペランドの前
- コメントの始まりのセミコロンの前

以下の箇所には，1 つ以上の空白が必要です。

- ニモニックとオペランドの間

図 4―1　アセンブリ言語文の構成



アセンブリ言語文は，1 行に 1 文を記述します。文の最後は改行（リターン）します。

(1) 文字セット

アセンブラがサポートするソース・プログラムで，使用できる文字は次の 3 つから構成されます。

- 言語文字
- 文字データ
- 注釈（コメント）用文字

(a) 言語文字

ソース上で命令を記述するために使用する文字です。

表 4―1 言語文字とその用途

文字	用途
英小文字（a ～ z）	ニモニック，識別子，および定数の構成
英大文字（A ～ Z）	識別子，および定数の構成
_（アンダースコア）	識別子の構成
.（ピリオド）	識別子，および定数の構成
@	識別子，および定数の構成
~	識別子，および定数の構成
数字	識別子，および定数の構成
,（カンマ）	オペランドの区切り
:（コロン）	ラベルの区切り
;（セミコロン）	コメントの開始
*	乗算演算子
/	除算演算子
+	正符号，および加算演算子
-（ハイフン）	負符号，および減算演算子
'（シングルクォーテーション）	文字定数，およびマクロのパラメータを 1 つにまとめる記号
<	比較演算子
>	比較演算子
( )	演算順序の指定
$	ロケーション・カウンタの値 ソース・プログラム中の制御命令の前に付ける記号 相対アドレッシング開始記号 ラベルの gp オフセット参照
=	比較演算子
!	絶対アドレッシングの開始，および否定演算子
△（空白）	各欄の区切り
~	連結記号（マクロボディ内）
&	論理積演算子

文字	用途
#	イミディエート値の開始，およびコメントの開始
[ ]	インダイレクト表示記号
"（ダブルクォーテーション）	文字列定数の開始と終了
%	ラベルの ep オフセット参照，および剰余演算子
<<	左論理シフト演算子
>>	右算術シフト演算子
|	論理和演算子
^	排他的論理和演算子

（b）文字データ

文字データは，文字列定数，文字定数，および制御命令部を記述するために使用する文字です。

注意　0x00 を除くすべての文字（漢字かなを含みます。ただし，OS によってコードは異なります）が記述可能です。0x00 を記述するとエラーとなり，それ以降，引用符（’，または”）で閉じるまで無視されます。

（c）注釈（コメント）用文字

コメントを記述するために使用する文字です。

注意　文字データの文字セットと同一です。

（2）シンボル

シンボル欄には，シンボルを記述します。シンボルとは，数値データやアドレスなどに付けた名前のことです。

シンボルを使用することにより，ソースの内容がわかりやすくなります。

（a）シンボルの種類

シンボルは，その使用目的，定義方法によって，次に示す種類に分けられます。

シンボルの種類	使用目的	定義方法
ラベル	ソース中で，アドレス・データとして使用	シンボルのあとにコロン（：）を付けることにより定義します。
外部参照名	あるモジュールで定義されたシンボルをほかのモジュールで参照するときに使用	.extern 疑似命令のオペランド欄に記述します。
セクション名	リンク時に使用	.cseg，.dseg，.org 疑似命令のシンボル欄に定義します。
マクロ名	ソース中で，マクロ参照時に使用	マクロ疑似命令のシンボル欄に記述します。

(b) シンボル記述上の規則

シンボルは，次の規則に基づいて記述します。

- シンボルは，英数字，および英字相当文字（?，@，_，.）で構成します。
  ただし，先頭文字に数字（0 ～ 9）は使用できません。
- シンボルの最大文字数は 4,294,967,294（=0xFFFFFFFE）（理論値）です。ただし，実際には利用可能なメモリ量に依存します。
- シンボルとして，予約語は使用できません。
  予約語については，「4.5 予約語」を参照してください。
- 同一シンボルを二度以上定義することはできません。
  ただし，.set 疑似命令で定義したシンボルは，.set 疑似命令で再定義することができます。
- アセンブラは，シンボルの大文字／小文字を区別します。
- シンボル欄にラベルを記述する場合は，ラベルの直後にコロン（:）を記述します。

正しいシンボルの例を以下に示します。

```
CODE01  .cseg   text        ; “CODE01”はセグメント名
VAR01   .set    0x10        ; “VAR01”はシンボル
LAB01:  .dw     0           ; “LAB01”はラベル
```

誤ったシンボルの例を以下に示します。

```
1ABC    .set    3           ; 先頭文字に数字は使用できません。
LAB     mov     r10, r11    ; “LAB”ラベルです。ニモニック欄とコロン（ : ）で区切ります。
FLAG:   .set    0x10        ; シンボルにはコロン（ : ）が必要ありません。
```

シンボルのみからなる文の例を以下に示します。

```
ABCD:                       ; ABCD がラベルとして定義されます。
```

(c) シンボルに関する注意事項

セクション定義疑似命令でセクション名が指定されなかったときは，アセンブラがセクション名を自動生成します。次に，そのセクションを示します。

同名で定義するとエラーとなります。

セクション名	疑似命令	再配置属性
.text	.cseg 疑似命令	TEXT
.const		CONST
.sconst		SCONST

セクション名	疑似命令	再配置属性
.bss	.dseg 疑似命令	BSS
.data		DATA
.sbss		SBSS
.sdata		SDATA
.sebss		SEBSS
.sedata		SEDATA
.sibss		SIBSS
.sidata		SIDATA
.tibss		TIBSS
.tibss.byte		TIBSS.BYTE
.tibss.word		TIBSS.WORD
.tidata		TIDATA
.tidata.byte		TIDATA.BYTE
.tidata.word		TIDATA.WORD
SECUR_ID	.cseg 疑似命令	SECUR_ID
OPT_BYTE	.dseg 疑似命令	OPT_BYTE

（d）シンボルの属性

シンボル，およびラベルは，値と属性を持ちます。

値とは，定義された数値データやアドレス・データの値そのものです。

セクション名，モジュール名，およびマクロ名は，値を持ちません。

属性とは，次に示すシンボル属性のことです。

属性の種類	区分	値
BIT	- ビット値として定義されたシンボル - .set 疑似命令で定義されたシンボル	10 進表現 ： -2147483648 ～ 2147483647 16 進表現 ： 0x80000000 ～ 0x7FFFFFFF （符号あり）
CSEG	.cseg 疑似命令で定義されたセクション名	値を持ちません
DSEG	.dseg 疑似命令で定義されたセクション名	
MACRO	マクロ疑似命令で定義されたマクロ名	値を持ちません
FNUMBER	FLOAT 疑似命令で定義したシンボル（単精度浮動小数点)	1.40129846e-45 ～ 3.40282347e+38
DFNUMBER	DFLOAT 疑似命令で定義したシンボル（倍精度浮動小数点)	4.9406564584124654e-324 ～ 1.7976931348623157e+308

例を以下に示します。

```
BIT1    .set    0xFFE20.0   ; シンボル“BIT1”は，BIT 属性と値 0xFFE20.0 を持ちます。
```

(3) ニモニック

ニモニック欄には，インストラクションのニモニック，疑似命令，およびマクロ参照を記述します。

オペランドの必要なインストラクションや疑似命令，マクロ参照の場合，ニモニック欄とオペランド欄を 1 つ以上の空白，または TAB で区切ります。

ただし，インストラクションの第 1 オペランドの先頭が“#”，“$”，“！”，“［”の場合には，ニモニックと第 1 オペランドの間に何もなくても，正常にアセンブルが行われます。

正しい例を以下に示します。

```
add     r11, r12
reti
di
```

誤った例を以下に示します。

```
addr11, r12     ; ニモニック欄とオペランド欄の間に，空白がありません。
r eti           ; ニモニック中に空白があります。
HLT             ; ニモニック欄に記述できない命令です。
```

(4) オペランド

オペランド欄には，インストラクションや疑似命令，およびマクロ参照の実行に必要なデータ（オペランド）を記述します。

各インストラクションや疑似命令により，オペランドを必要としないものや，複数のオペランドを必要とするものがあります。

2 個以上のオペランドを記述する場合には，各オペランドをコンマ（ , ）で区切ります。

オペランド欄に記述できるものは，次のものです。

- 定数（数値定数，文字定数，文字列定数）
- レジスタ名
- セクション定義疑似命令の再配置属性
- シンボル
- 式

なお，インストラクション・セットにおけるオペランドの表現形式と記述方法については，開発対象となる各デバイスのユーザーズ・マニュアルを参照してください。

以降に，オペランド欄に記述可能な各項目について説明します。

(a) 定数

定数は，それ自身で定まる値を持つもので，イミーディエト・データとも呼びます。

定数には，数値定数と文字定数，文字列定数があります。

- 数値定数

整定数として，2 進数，8 進数，10 進数，16 進数が記述可能です。

整定数，つまり，整数の定数は，32 ビット幅をもつ定数です。負の数は 2 の補数で表現されます。32 ビットで表現することのできる値を越える整数値が指定された場合，アセンブラはその整数値の下位 32 ビットの値を用いて処理を続行します（メッセージ等は出力しません）。

整定数の種類	表記方法	表記例
2 進数	- 数値の最後に文字“B”，または“Y”を付加 - 数値の前に“0b”を記述	1101B 1101Y 0b1101
8 進数	数値の前に“0”を記述	074
10 進数	数値をそのまま記述	128
16 進数	数値の前に“0x”を記述	0xA6

浮動小数点定数は次に示す要素で構成されます。指数値，および仮数値は 10 進定数で指定します。ただし，指数表現を用いない場合（3），（4），（5）は使用しません。

（1）仮数部の符号（+ は省略可）

（2）仮数部

（3）指数部を示す’e’，または’E’

（4）指数部の符号（+ は省略可）

（5）指数部

例

```
123.4
-100.
10e-2
-100.2E+5
```

仮数値の頭に”0f”，または”0F”を置くことにより浮動小数点定数であることを示すこともできます。

例

```
0f10
```

- 文字定数

文字定数は，1 つの文字をシングル・クォート“’”で囲むことにより構成され，囲まれた文字の値を示します。**注**

以下に示したエスケープ・シーケンスを“’”と“’”の間に指定した場合，1 つの文字を表すものとして扱われます。

例

```
"ab"            ; 0x6162
"A"             ; 0x41
"A\""           ; 0x4122
" "             ; 0x20（空白１個）
""              ;
```

注 文字定数を指定した場合，その文字定数の値を持つ整定数を指定したものとみなして扱われます。

表 4―2 エスケープ・シーケンスの値と意味

エスケープ・シーケンス	値	意味
\ 0	0x00	null 文字
\ a	0x07	アラート
\ b	0x08	バックスペース
\ **f**	0x0C	フォーム・フィード
\ n	0x0A	改行
\ r	0x0D	キャリッジ・リターン
\ t	0x09	水平タブ
\ **v**	0x0B	垂直タブ
\\	0x5C	バックスラッシュ
\ '	0x27	シングル・クォート
\ "	0x22	ダブル・クォート
\ ?	0x3F	疑問符
\ ddd	0 ～ 0377	3 桁までの 8 進数（0 < d < 7）**注**
\ xhh	0 ～ 0xFF	2 桁までの 16 進数 （0 < h < 9，a < h < f，または A < h < F）

注 “\ 377”を越える指定の場合，そのエスケープ・シーケンスの値は，下位 1 バイト分のみの値となります。0377 より大きい値にはなりません。たとえば“\ 777”の値は 0377 です。

- 文字列定数

文字列定数は，「(1) 文字セット」で示した文字を引用符（“ ）で囲んだものです。

文字列定数は，-Xcharacter_set オプションで指定した文字コード値でアセンブルされた結果，文字列の終端に”￥0”を付加します。

引用符自体を文字列定数とする場合には，引用符を 2 個続けて記述します。

例を以下に示します。

```
"ab"            ; 0x616200
"A"             ; 0x4100
"A\""           ; 0x412200
" "             ; 0x2000（空白１個）
""              ; 0x00
```

(b) レジスタ名

オペランド欄に記述可能なレジスタとして，次のものがあります。

- 汎用レジスタ
- 汎用レジスタ・ペア
- 特殊機能レジスタ
- その他（PSW，CY，RBn，[BC]，[DE]，[HL]，[DE+byte]，[HL+byte]，[HL+B]，[HL+C]）

汎用レジスタや汎用レジスタ・ペアは，絶対名称での記述のほかに，機能名称での記述も可能です。

なお，インストラクションの種類により，オペランド欄に記述可能なレジスタ名が異なります。各レジスタの記述方法の詳細については，開発対象となる各デバイスのユーザーズ・マニュアルを参照してください。

(c) セクション定義疑似命令の再配置属性

オペランド欄には，再配置属性を記述するとこができます。

再配置属性の詳細については，「4. 2. 2 セクション定義疑似命令」を参照してください。

(d) シンボル

シンボルをオペランド欄に記述した場合は，そのシンボルに割り付けられたアドレス（または値）がオペランドの値になります。

使用例を以下に示します。

```
HERE:   jmp32   #THEREE         ; THERE はラベル THERE のアドレスを示します。
         :
THERE:  add     r11, r12
VALUE   .set    0x100
        movea   VALUE, r11, r12 ; VALUE はシンボル VALUE の値を示します。
```

(e) 式

式は，定数，ロケーション・カウンタを示す $，またはシンボルを演算子で結合したものです。

インストラクションのオペランドとして数値表現可能なところに記述することができます。

式と演算子については，「4. 1. 2 式と演算子」を参照してください。

例を以下に示します。

```
TEN     .set    0x10
        mov     TEN - 0x05, r12
```

この記述例では，“TEN - 0x05”が式です。

この式は，シンボルと数値定数が -（マイナス）演算子で結合されています。式の値は“0x0B”です。

したがって，この記述は“mov 0x0B, r12”と書き換えることが可能です。

(5) コメント

コメント欄には，セミコロン（ ; ）のあとにコメント（注釈）を記述します。

コメント欄は，セミコロンからその行の改行コード，または EOF までです。

コメントを記述することにより，理解しやすいソースを作成できます。

コメント欄の記述は，機械語変換というアセンブル処理の対象とはならず，そのままアセンブル・リストに出力されます。

記述可能な文字は，「(1) 文字セット」に示すものです。

例を以下に示します。

```
; sample program                                    コメント欄のみの行
        .extern __tp_TEXT, 4
        .extern __gp_DATA, 4
        .extern _main
RESET   .cseg   text                    ; Reset Handler address
        jr      __boot                  ; Jump to __boot
.text   .cseg   text                    ; Text section
        .align  4                       ; Code alignment
        .public  __boot                 ; Alignment
 __boot:
        mov     #__tp_TEXT, tp          ; Set tp
        mov     #__gp_DATA, gp          ; Set gp
        .extern __ssbss, 4
        .extern __esbss, 4
   ; start of bss initialize                       コメント欄のみの行
        mov     #__ssbss, r13
        mov     #__esbss, r13
        cmp     r12, r13
        jnl     sbss_init_end

 sbss_init_loop:
        st.w    r0, 0[r13]
        add     4, r13
        cmp     r12, r13
        jl      sbss_init_loop
 sbss_init_end:
   ; end of bss initialize                         コメント欄のみの行
        jarl    _main, lp               ; Call main function
.data   .dseg  data
        .align 4
 data_area:
        .dw     0x00                    ; data1
        .dhw    0x01                    ; data2
        .db     0xFF                    ; data3
        .db     0xFE                    ; data4
```

コメント欄にコメントが記述されている行

コメント欄にコメントが記述されている行

4. 1. 2 式と演算子

式とは，シンボル，定数，ロケーション・カウンタを示す $，前述の 3 つに演算子を付加したもの，または演算子で結合したものです。

式を構成する演算子以外の要素を項といい，記述された左側から順に第 1 項，第 2 項，…と呼びます。

演算子には「表 4―3 演算子の種類」に示すものがあり，演算実行上の優先順位が「表 4―4 演算子の優先順位」のように決められています。

演算の順序を変更するには，かっこ“( )”を使用します。

例を示します。

```
mov32   5 * (SYM + 1), r12
```

上記の例では，“5 * (SYM+1)”が式です。“5”が第 1 項，“SYM”が第 2 項，“1”が第 3 項です。“*”，“+”，“( )”が演算子です。

表 4―3 演算子の種類

演算子の種類	演算子
算術演算子	+，-，*，/，MOD（%），+ 符号，- 符号
論理演算子	!，&，|，^
比較演算子	==，!=，>，>=，<，<=，&&，||
シフト演算子	>>，<<
バイト分離演算子	HIGH，LOW
2 バイト分離演算子	HIGHW，LOWW，HIGHW1
特殊演算子	DATAPOS，BITPOS
その他の演算子	( )

上記の演算子は，単項演算子，特殊単項演算子，2 項演算子に分けられます。

単項演算子	+ 符号，- 符号，!，HIGH，LOW，HIGHW，LOWW，HIGHW1
特殊演算子	DATAPOS，BITPOS
2 項演算子	+，-，*，/，MOD（%），&，|，^，==，!=，>，>=)，<)，<=，>>，<<，&&，||

表 4―4 演算子の優先順位

優先度	優先順位	演算子
高い	1	+ 符号，- 符号，!
	2	*，/，MOD（%），>>，<<
	3	&，|，^
	4	+，-
	5	==，!=，>，>=，<，<=)
低い	6	&&，||

式の演算は，次の規則に従います。

- 演算の順序は，演算子の優先順位に従います。
  同一順位の場合は，左から右に演算されます。単項演算子の場合は，右から左に演算されます。
- かっこ“( )”の中の演算は，かっこの外の演算に先立って行われます。
- 式の演算は，符号なし 32 ビットで行います。
  演算中に 32 ビットを越えてオーバーフローした場合，オーバーフローした値は無視されます。
- 定数が 32 ビットを越える場合には，エラーとなり，その値は 0 とみなされて計算されます。
- 除算では，小数部分を切り捨てます。
  除算がゼロの場合は，エラーとなり，結果は 0 となります。
- 負の値は，2 の補数形式となります。
- 外部参照記号のアセンブル時の評価値はゼロです（評価値はリンク時に決定されます）。

(1) 式評価の例

式	評価値
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0xFFFFFFFF
-1 > 1	0x0（偽）
EXT **注** + 1	1

注 EXT ： 外部参照記号

4. 1. 3　算術演算子

算術演算子には，次のものがあります。

演算子	概要
+	第 1 項と第 2 項の値の加算
-	第 1 項と第 2 項の値の減算
*	第 1 項と第 2 項の値の乗算
/	第 1 項と第 2 項の値で剰余算を行い，整数部を求める
MOD（%）	第 1 項と第 2 項の値で剰余算を行い，余りを求める
+ 符号	項の値をそのまま返す
- 符号	項の値の 2 の補数を求める

+

第 1 項と第 2 項の値の加算を行います。

［機能］

第 1 項と第 2 項の値の和を返します。

［使用例］

```
        .org    0x100
START:  jmp     #START + 6      ; (1)
```

(1) jmp 命令により，“START ラベルのアドレス＋ 6 番地”へジャンプします。
つまり，START ラベルが 0x100 番地の場合，“0x100 + 0x6 = 0x106”へジャンプします。
したがって，“START: jmp #0x106”と記述することもできます。

-

第 1 項と第 2 項の値の減算を行います。

［機能］

第 1 項と第 2 項の値の差を返します。

［使用例］

```
        .org    0x100
BACK:   jmp     !BACK - 6       ; (1)
```

(1) jmp 命令により，「“BACK”に割り付けられたアドレス－ 6 番地」へジャンプします。
つまり，BACK ラベルが 0x100 番地の場合，“0x100 - 0x6 = 0xFA”へジャンプします。
したがって，“BACK: jmp !0xFA”と記述することもできます。

*

第 1 項と第 2 項の値の乗算を行います。

［機能］

第 1 項と第 2 項の値の積を返します。

［使用例］

```
TEN     .set    0x10
        mov     TEN * 3, r12    ; (1)
```

(1) .set 疑似命令により，シンボル“TEN”に 0x10 という値が定義されます。
“TEN * 3”という式は“0x10 * 3”のことで，0x30 を返します。
したがって，“mov 0x30, r12”と記述することもできます。

/

第 1 項と第 2 項の値で剰余算を行い，整数部を求めます。

［機能］

第 1 項の値を第 2 項の値で割り，その値の整数部を返します。

小数部は切り捨てられます。

除数（第 2 項）が 0 の場合は，エラーとなります。

［使用例］

```
mov     A, #256 / 50            ; (1)
```

(1)“256 / 50 = 5　余り 6”となります。

よって，整数部の 5 を返します。

したがって，“mov A, #5”と記述することもできます。

MOD（%）

第 1 項と第 2 項の値で剰余算を行い，余りを求めます。

［機能］

第 1 項の値を第 2 項の値で割り，その値の余りを返します。

除数が 0 の場合は，エラーとなります。

MOD の前後には，空白が必要です。

［使用例］

```
mov     256 % 50, r12           ; (1)
```

(1)“256 / 50 = 5　余り 6”となります。

よって，余りの 6 を返します。

したがって，“mov 6, r12”と記述することもできます。

+ 符号

項の値をそのまま返します。

［機能］

項の値をそのまま返します。

［使用例］

```
FIVE    .set    +5              ; (1)
```

(1) 項の値“5”をそのまま返します。

.set 疑似命令により，シンボル“FIVE”に 5 という値が定義されます。

- 符号

項の値の 2 の補数を求めます。

［機能］

項の値の 2 の補数をとった値を返します。

［使用例］

```
NO      .set    -1              ; (1)
```

(1)“-1”は 1 の 2 の補数となります。

0000　0000　0000　0000　0000　0000　0000　0001 の 2 の補数は

1111　1111　1111　1111　1111　1111　1111　1111

となります。

よって，.set 疑似命令により，シンボル“NO”に 0xFFFFFFFF が定義されます。

4. 1. 4　論理演算子

論理演算子には，次のものがあります。

演算子	概要
!	項のビットごとの論理否定を求める
&	第 1 項の値と第 2 項の値のビットごとの論理積を求める
|	第 1 項の値と第 2 項の値のビットごとの論理和を求める
^	第 1 項の値と第 2 項の値のビットごとの排他的論理和を求める

!

項のビットごとの論理否定を求めます。

［機能］

項のビットごとの論理否定をとり，その値を返します。

! と項との間には，空白が必要です。

［使用例］

```
mov32   !0x3, r12               ; (1)
```

(1)“0x3”の論理否定をとります。

よって，0xFFFFFFFC を返します。

したがって，“mov32 0xFFFFFFFC, r12”と記述することもできます。

!)	0000	0000	0000	0000	0000	0000	0000	0011
	1111	1111	1111	1111	1111	1111	1111	1100

&

第 1 項の値と第 2 項の値のビットごとの論理積を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの論理積をとり，その値を返します。

& の前後には，空白が必要です。

［使用例］

```
mov32   0x6FA & 0xF, r12        ; (1)
```

(1)“0x6FA”と“0xF”の論理積をとります。

よって，“0xA”を返します。

したがって，(1)は“mov32 0xA, r12”と記述することもできます。

	0000	0000	0000	0000	0000	0110	1111	1010
&)	0000	0000	0000	0000	0000	0000	0000	1111
	0000	0000	0000	0000	0000	0000	0000	1010

|

第 1 項の値と第 2 項の値のビットごとの論理和を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの論理和をとり，その値を返します。

| の前後には，空白が必要です。

［使用例］

```
mov32   0xA | 0b1101, r12       ; (1)
```

(1)“0xA”と“0b1101”の論理和をとります。

よって，“0xF”を返します。

したがって，(1)は“mov32 0xF, r12”と記述することもできます。

	0000	0000	0000	0000	0000	0000	0000	1010
|	0000	0000	0000	0000	0000	0000	0000	1101
	0000	0000	0000	0000	0000	0000	0000	1111

^

第 1 項の値と第 2 項の値のビットごとの排他的論理和を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの排他的論理和をとり，その値を返します。

^ の前後には，空白が必要です。

［使用例］

```
mov32   0x9A ^ 0x9D, r12        ; (1)
```

(1)“0x9A”と“0x9D”の排他的論理和をとります。

よって，“0x7”を返します。

したがって，(1)は“mov32 0x7, r12”と記述することもできます。

	0000	0000	0000	0000	0000	0000	1001	1010
^)	0000	0000	0000	0000	0000	0000	1001	1101
	0000	0000	0000	0000	0000	0000	0000	0111

4. 1. 5　比較演算子

比較演算子には，次のものがあります。

演算子	概要
==	第 1 項の値と第 2 項の値が等しいかどうか比較
!=	第 1 項の値と第 2 項の値が等しくないかどうか比較
>	第 1 項の値が第 2 項の値より大きいかどうか比較
>=	第 1 項の値が第 2 項の値より大きい，または等しいかどうか比較
<	第 1 項の値が第 2 項の値より小さいかどうか比較
<=	第 1 項の値が第 2 項の値より小さい，または等しいかどうか比較
&&	第 1 項の値と第 2 項の値の論理積を求める
||	第 1 項の値と第 2 項の値の論理和を求める

==

第 1 項の値と第 2 項の値が等しいかどうか比較します。

［機能］

第 1 項の値と第 2 項の値が等しいときに ~0（真），等しくないときに 0（偽）を返します。

== の前後には，空白が必要です。

!=

第 1 項の値と第 2 項の値が等しくないかどうか比較します。

［機能］

第 1 項の値と第 2 項の値が等しくないときに ~0（真），等しいときに 0（偽）を返します。

!= の前後には，空白が必要です。

>

第 1 項の値が第 2 項の値より大きいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より大きいときに ~0（真），等しいか小さいときに 0（偽）を返します。

> の前後には，空白が必要です。

>=

第 1 項の値が第 2 項の値より大きい，または等しいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より大きいか，等しいときに ~0（真），小さいときに 0（偽）を返します。

>= の前後には，空白が必要です。

<

第 1 項の値が第 2 項の値より小さいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より小さいときに ~0（真），等しいか大きいときに 0（偽）を返します。

< の前後には，空白が必要です。

<=

第 1 項の値が第 2 項の値より小さい，または等しいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より小さいか等しいときに ~0（真），大きいときに 0（偽）を返します。

<= の前後には，空白が必要です。

&&

第 1 項の値と第 2 項の値の論理積を求めます。

［機能］

第 1 項の論理値と第 2 項の論理値の論理積を求めます。

||

第 1 項の値と第 2 項の値の論理和を求めます。

［機能］

第 1 項の論理値と第 2 項の論理値の論理和を求めます。

4. 1. 6　シフト演算子

シフト演算子には，次のものがあります。

演算子	概要
>>	第 1 項の値を第 2 項で示す値分だけ右シフトした値を求める
<<	第 1 項の値を第 2 項で示す値分だけ左シフトした値を求める

>>

第 1 項の値を第 2 項で示す値分だけ右シフトした値を求めます。

［機能］

第 1 項の値を第 2 項で示す値（ビット数）分だけ算術右シフトし，その値を返します。

符号ビットはシフトしません。

上位ビットには，シフトされたビット数だけ符号ビットの値が挿入されます。

シフト数が 0 の場合は，第 1 項の値がそのまま返されます。シフト数が 31 を越えた場合は，0 が返されます。

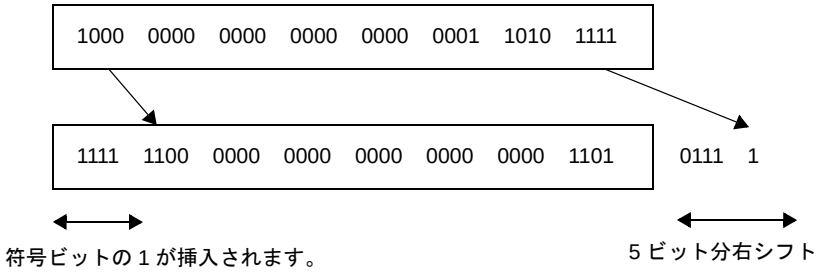
［使用例］

```
MOV32   0x800001AF >> 5, r20    ; (1)
```

(1)“0x800001AF”を符号ビットを残して 5 ビット分右シフトします。

よって，“0xFC00000D”を r20 に転送します。

したがって，“mov32 0xFC00000D, r20”と記述することもできます。



<<

第 1 項の値を第 2 項で示す値分だけ左シフトした値を求めます。

［機能］

第 1 項の値を第 2 項で示す値（ビット数）分だけ左シフトし，その値を返します。

下位ビットには，シフトされたビット数だけ 0 が挿入されます。

シフト数が 0 の場合は，第 1 項の値がそのまま返されます。シフト数が 31 を越えた場合は，0 が返されます。

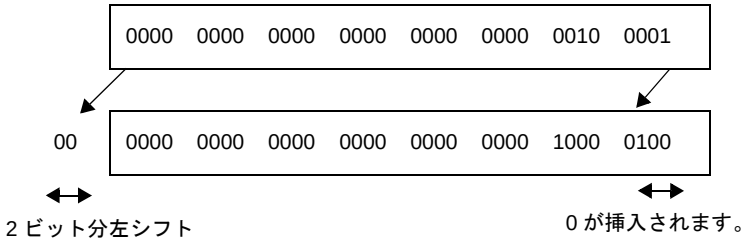
［使用例］

```
mov32   0x21 << 2,r20           ; (1)
```

(1)“0x21”を 2 ビット分左シフトします。

よって，“0x84”を r20 に転送します。

したがって，“mov32 0x84, r20”と記述することもできます。

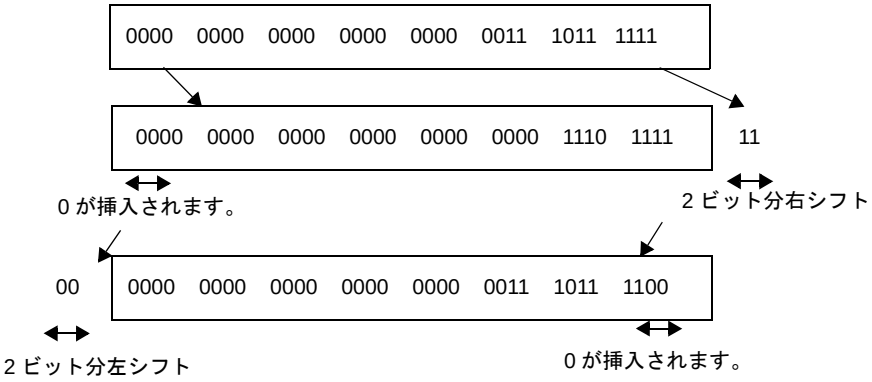



```
mov32   0x3BF >> 2 << 2, r20    ; (2)
```

(2)“0x3BF”を 2 ビット分右シフトし，その後 2 ビット分左シフトします。

よって，“0x3BC”を r20 に転送します。

したがって，“mov32 0x3BC, r20”と記述することもできます。



4. 1. 7　バイト分離演算子

バイト分離演算子には，次のものがあります。

演算子	概要
HIGH	項の上位 8 ビットを求める
LOW	項の下位 8 ビットを求める

HIGH

項の上位 8 ビットを求めます。

［機能］

項の上位 8 ビットを返します。

HIGH と項との間には，空白が必要です。

［使用例］

```
mov32   HIGH 0x1234, r12        ; (1)
```

(1) mov32 命令実行により，0x1234 の上位 8 ビット値 0x12 を返します。
したがって，(1) は“mov32 0x12, r12”と記述することもできます。

```
mov32   HIGH P0, r12            ; (2)
```

(2) mov32 命令実行により，P0 の上位 8 ビット値 0xFF を返します。
したがって，(2) は“mov32 0xFF, r12”と記述することもできます。

LOW

項の下位 8 ビットを求めます。

［機能］

項の下位 8 ビットを返します。

LOW と項との間には，空白が必要です。

［使用例］

```
mov32   LOW 0x1234, r12         ; (1)
```

(1) mov32 命令実行により，0x1234 の下位 8 ビット値 0x34 を返します。
したがって，(1) は“mov32 0x34, r12”と記述することもできます。

4. 1. 8　2 バイト分離演算子

2 バイト分離演算子には，次のものがあります。

演算子	概要
HIGHW	項の上位 16 ビットを求める
LOWW	項の下位 16 ビットを求める
HIGHW1	項の上位 16 ビットに項の第 15 ビットの値を加算した値を求める

HIGHW

項の上位 16 ビットを求めます。

［機能］

項の上位 16 ビットを返します。

HIGHW と項との間には，空白が必要です。

［使用例］

```
mov32   HIGHW(0x12345678), r12          ; (1)
```

(1) mov32 命令実行により，0x12345678 の上位 16 ビット値 0x1234 を返します。
したがって，“mov32 0x1234, r12”と記述することもできます。

LOWW

項の下位 16 ビットを求めます。

［機能］

項の下位 16 ビットを返します。

LOWW と項との間には，空白が必要です。

［使用例］

```
mov32   LOWW(0x12345678), r12           ; (1)
```

(1) mov32 命令実行により，0x12345678 の下位 16 ビット値 0x5678 を返します。
したがって，“mov32 0x5678, r12”と記述することもできます。

HIGHW1

項の上位 16 ビットに項の第 15 ビットの値を加算した値を求めます。

［機能］

項の上位 16 ビットに項の第 15 ビットの値を加算した値を返します。

HIGHW1 と項との間には，空白が必要です。

［使用例］

```
mov32   HIGHW1(0x12345678), r12         ; (1)
```

(1) mov32 命令実行により，0x12345678 の上位 16 ビット値 0x1234 に，0x12345678 の第 15 ビットの値 1 を加算した値 0x1235 を返します。

したがって，“mov32 0x1235, r12”と記述することもできます。

4. 1. 9　特殊演算子

特殊演算子には，次のものがあります。

演算子	概要
DATAPOS	ビット・シンボルのアドレス部を求める
BITPOS	ビット・シンボルのビット部を求める

DATAPOS

ビット・シンボルのアドレス部を求めます。

［機能］

ビット・シンボルのアドレス部を返します。

［使用例］

```
mov32   DATAPOS( DNFA2NFEN2 ), r10      ; (1)
mov32   BITPOS( DNFA2NFEN2 ), r12
clr1    r12, [r10]
```

(1)“DATAPOS DNFA2NFEN2”は“DATAPOS 0xFF41020C.2”ということで，0xFF41020C を返します。したがって，“mov32 0xFF41020C, r10”と記述することもできます。

BITPOS

ビット・シンボルのビット部を求めます。

［機能］

ビット・シンボルのビット部（ビット位置）を返します。

［使用例］

```
mov32   DATAPOS( DNFA2NFEN2 ), r10
mov32   BITPOS( DNFA2NFEN2 ), r12       ; (1)
clr1    r12, [r10]
```

(1)“BITPOS DNFA2NFEN2”は“BITPOS 0xFF41020C.2”ということで，2 を返します。
したがって，“mov32 2, r12”と記述することもできます。

4. 1. 10　その他の演算子

その他の演算子には，次のものがあります。

演算子	概要
( )	( ) 内の演算を優先して行う

( )

( ) 内の演算を優先して行います。

［機能］

( ) 内の演算を ( ) 外の演算に先立って行います。

演算の優先順位を変更したいときに使用します。

( ) が多重になっている場合は，一番内側の ( ) 内の式から演算します。

［使用例］

```
mov     A, #(4 + 3) * 2
```

（1），（2）の順で演算を行い，14 という値を返します。

( ) がなければ

（1），（2）の順で演算を行い，10 という値を返します。

演算子の優先順位については，「表 4―4　演算子の優先順位」を参照してください。

4. 1. 11 演算の制限

式は“定数”，“シンボル”，“ラベルの参照”，“演算子”，および“かっこ”を要素とし，これらの要素で構成される値を示します。式は，絶対値式と相対値式に分けて扱います。

(1) 絶対値式

定数値を示す式を“絶対値式”と呼びます。絶対値式は，命令においてオペランドを指定する場合，または疑似命令において値などを指定する場合に用いることができます。通常，絶対値式は，定数，またはシンボルによって構成されます。次に示した形式が絶対値式として扱われます。

(a) 定数式

定義済みのシンボル参照を指定した場合，そのシンボルに対して定義した値の定数が指定されたものとして扱われます。したがって，定数式は，定義済みのシンボル参照を，その構成要素として持つことができます。

例

```
sym1    .set    0x100           ; シンボル sym1 を定義
        mov     sym1, r10       ; 定義済みの sym1 は定数式として扱う
```

(b) シンボル

シンボルに関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- シンボル
- シンボル±定数式
- シンボル - シンボル
- シンボル - シンボル±定数式

ここで言う“シンボル”とは，その時点において未定義なシンボル参照を指します。定義済みのシンボル参照を指定した場合は，そのシンボルに対して定義した値の“定数”が指定されたものとして扱います。

例

```
add     SYM1 + 0x100, r11       ; この時点で SYM1 は未定義シンボル
SYM1    .set 0x10               ;SYM1 を定義
```

(c) ラベル参照

ラベル参照に関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- ラベル参照 - ラベル参照
- ラベル参照 - ラベル参照±定数式

ラベル参照に関する式の例は，次のようになります。

例

```
mov     $label1 - $label2, r11
```

上記の例のような“2 つのラベル参照”は，次のように参照にする必要があります。

- 指定したファイル内で，同じセクションに定義をもつ
- 同じ参照方法（$label は $label 同士，#label は #label 同士など）

これらの条件を満たさない場合は，メッセージが出力され，アセンブルが中止されます。

ただし，現在のアセンブラの構成上，“ラベル参照に関する式”の中の“- ラベル参照”側のラベル参照に，指定されたファイル内に定義を持たないラベルの絶対アドレス参照が指定された場合，他方のラベル参照の参照方法と同じ参照方法が用いられたものとみなして扱われます。なお，分岐命令に対して，この形式の絶対値式は指定できません。指定した場合は，メッセージが出力され，アセンブルが中止されます。

(2) 相対値式

特定のアドレスからのオフセット値[注1]を示す式を“相対値式”と呼びます。相対値式は，命令においてオペランドを指定する場合，データ定義疑似命令において値を指定する場合に用いることができます。通常，相対値式は，ラベル参照によって構成されます。次に示した形式[注2]が相対値式として扱われます。

注 1. このアドレスはリンク時に定められます。このため，このオフセットの値もリンク時に定められます。

2. 絶対値式，相対値式ともに，“- シンボル + ラベルの参照”の形式の式を“ラベルの参照 - シンボル”の形式の式とみなすことはできますが，“ラベルの参照 -（+ シンボル）”の形式の式を“ラベルの参照 - シンボル”の形式の式とみなすことはできません。このため，かっこ“( )”は定数式においてのみ用いるようにしてください。

(a) ラベル参照

ラベル参照に関する式には，次のものがあります（“±”は“+”か“-”のどちらかになります）。

- ラベル参照
- ラベル参照±定数式
- ラベル参照 - シンボル
- ラベル参照 - シンボル±定数式

ラベル参照に関する式の例は次のようになります。

例

```
add     #labe11 + 0x10, r10
add     #label2 - SIZE, r10
SIZE    .set 0x10
```

4. 1. 12　識 別 子

識別子とは，シンボル，ラベル，マクロ名などに使用する名前です。

識別子は，次の規則に基づいて記述します。

- 識別子は，英数字，および英字相当文字（?，@，_）で構成します。
  ただし，先頭文字に数字（0 ～ 9）は使用できません。
- 識別子として，予約語は使用できません。
  予約語については，「4. 5　予 約 語」を参照してください。
- アセンブラは，識別子の大文字／小文字を区別します。

4.2 疑似命令

この章では，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。

表 4―5 疑似命令一覧

種類	疑似命令
セクション定義疑似命令	.cseg，.dseg，.org，.vseg
シンボル定義疑似命令	.set，.file，.func
データ定義，領域確保疑似命令	.db，.db2/.dhw，.dshw，.db4/.dw，.db8/.ddw，.float，.double，.ds，.align
外部定義，外部参照疑似命令	.public，.extern，.comm
マクロ疑似命令	.macro，.local，.rept，.irp，.exitm，.exitma，.endm

以降，各疑似命令について詳細な説明を行います。

説明の中で，[　] は大かっこの中が省略可能であることを，…は同一の形式を繰り返すことを示します。

4. 2. 2 セクション定義疑似命令

セクションとは，同種のルーチン，またはデータなどをブロック化したものです。セクション定義疑似命令は，セクションの開始，および終了を宣言するための疑似命令です。

セクションは，リンカにおける配置単位です。

例

```
.cseg
  :
.dseg
  :
```

同じセクション名のセクション同士は，同一の再配置属性でなければなりません。したがって，再配置属性の異なる複数のセクションに，同一のセクション名を付けることはできません。同一のセクション名でありながら再配置属性が異なる場合はエラーとなり，本疑似命令を無視します。

1 つのソース・プログラム・ファイル内に記述した同一再配置属性，同一セクション名のセクションは，アセンブラ内部で連続したひとつのセクションとして処理を行います。

別々のソース・プログラム・ファイル内に分割記述した場合は，リンカが処理を行います。

セクション名はシンボルとして参照できません。

セクション定義疑似命令には，次のものがあります。

表 4―6 セクション定義疑似命令

疑似命令	概要
.cseg	アセンブラにコード・セクション（ROM 領域へ配置）の開始を指示
.dseg	アセンブラにデータ・セクション（RAM 領域へ配置）の開始を指示
.org	ロケーション・カウンタの値を進める
.vseg	アセンブラにデバッグ情報用のセクションの開始を指示

.cseg

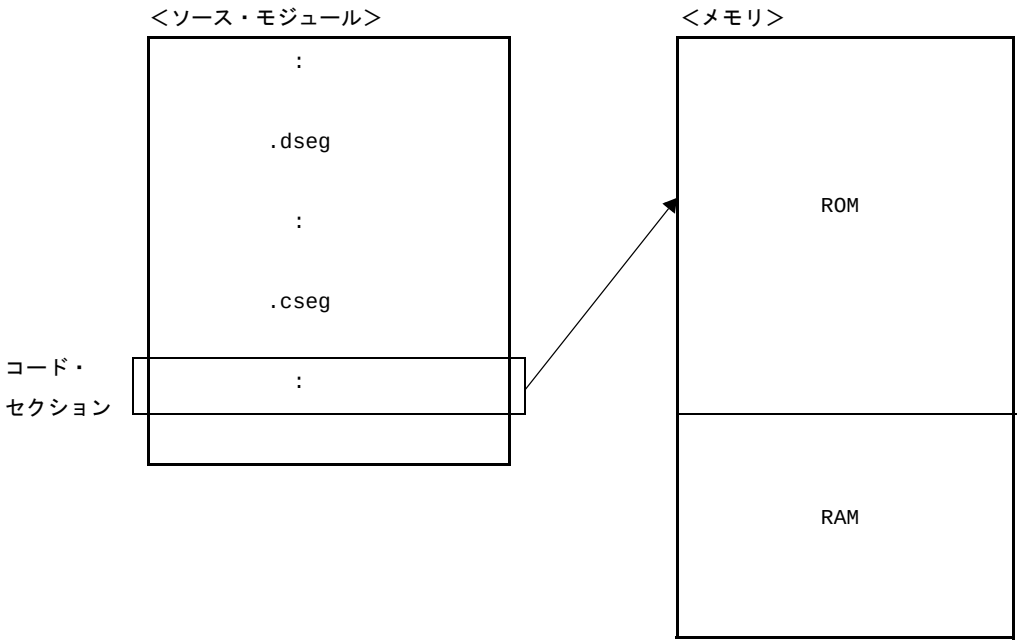
アセンブラにコード・セクション（ROM 領域へ配置）の開始を指示します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セクション名*]	.cseg	[*再配置属性*]	[; *コメント*]

［機能］

- .cseg 疑似命令は，アセンブラにコード・セクションの開始を指示します。
- .cseg 疑似命令以降に記述した命令は，再びセクション定義疑似命令（.cseg，.dseg，.org）が現れるまでコード・セクションに属し，最終的に機械語に変換された時点で ROM アドレス内に配置されます。



［用途］

- .cseg 疑似命令で定義するコード・セクションには，インストラクションや .db，.dw 疑似命令等を記述します。
- サブルーチンなどの 1 つの機能を持つ単位の記述は，1 つのコード・セクションとして定義します。

［詳細説明］

- 再配置属性とは，セクションの配置アドレスの範囲を限定するものです。

  次に，.cseg の再配置属性を示します。

表 4―7 .cseg の再配置属性

再配置属性	記述形式	説明
OPT_BYTE	OPT_BYTE	ユーザ・オプション・バイト，およびオンチップ・デバッグ指定専用の属性です。ユーザ・オプション・バイト，およびオンチップ・デバッグ以外は指定しないでください。 V850E1, V850E2 コアでは，指定セクションを 0x7A ～ 0x7F 番地へ配置します。 V850E2V3 のインストラクション・セットをもつデバイスでは指定できません。
SECUR_ID	SECUR_ID	セキュリティ ID 指定専用の属性です。セキュリティ ID 以外は指定しないでください。 V850E1, V850E2 コアでは，指定セクションを 0x70 ～ 0x79 番地へ配置します。 V850E2V3 のインストラクション・セットをもつデバイスでは指定できません。
TEXT	TEXT	プログラムを配置します。 セクション名 .text，セクション・タイプ PROGBITS，およびセクション属性 AX を持つ予約セクションです。 ひとつのアセンブラ・ソース・ファイル中のアセンブラ・ソース・プログラムの前には TEXT が 2 回指定されているものとみなされます（たとえば，セクション定義疑似命令を指定する前に“.dw 1”を指定した場合，それは .text セクションに割り当てられます）。ただし，.text セクションを明示的に指定せず，かつデフォルトで指定された TEXT セクションに対しラベルの定義，命令，ロケーション・カウンタ制御疑似命令，領域確保疑似命令のいずれも指定しなかった場合，.text セクションが生成されません。
CONST	CONST	定数データ用（読み出し専用）のセクションで，r0 と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲へ配置します。 セクション名 .const，セクション・タイプ PROGBITS，およびセクション属性 A を持つ予約セクションです。
SCONST	SCONST	定数データ用（読み出し専用）のセクションで，r0 と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（r0 からプラス方向に最大 32K バイト）へ配置します。 セクション名 .sconst，セクション・タイプ PROGBITS，およびセクション属性 A を持つ予約セクションです。

- 再配置属性を省略した場合，“TEXT”と解釈されます。

- 各セクションのサイズが領域のサイズを越えた場合には，エラーとなります。この時，ロケーション・カウンタを進めながら，アセンブルは続行します。
- .cseg 疑似命令のシンボル欄にセクション名を記述することにより，そのコード・セクションにシンボル（名前）を付けることができます。セクション名が省略されたコード・セクションには，アセンブラが自動的にデフォルトのセクション名を与えます。

  次に，.cseg のデフォルト・セクション名を示します。

再配置属性	デフォルト・セクション名
OPT_BYTE 注	OPTION_BYTES
SECUR_ID 注	SECURITY_ID
TEXT	.text
CONST	.const
SCONST 注	.sconst

注 これらの再配置属性に指定可能なセクション名は，デフォルト・セクション名のみです。

- 再配置属性が同一であれば，複数のコード・セクションに同一のセクション名を与えることができます。

  これらは，アセンブラ内部で 1 つのセクションとして処理されます。同名セクションの再配置属性が異なる場合には，エラーとなります。したがって，再配置属性ごとの同名セクションの数は 1 つです。
- コード・セクションは分割記述が可能です。つまり，同一モジュール内に記述された同一再配置属性，同一セクション名のコード・セクションは，アセンブラ内部で連続したひとつのセクションとして処理されます。
- 別モジュール間での同名セクションは，SECUR_ID の場合のみ記述することができ，リンク時に連続した 1 つのセクションとして結合されます。
- セクション名は，シンボルとして参照できません。
- OPT_BYTE で，ユーザ・オプション・バイト，およびオンチップ・デバッグを指定します。

  ユーザ・オプション・バイト指定機能を持つデバイスに対して，ユーザ・オプション・バイトを指定していない場合には，各番地に“OPT_BYTE”というデフォルト・セクションが定義され，デバイス・ファイルから読み込んだ初期値が設定されます。
- マルチコアでのセクション名の指定がないコード・セクションには，アセンブラが再配置属性ごとにデフォルトのセクション名を自動的に与えます。

  次に，デフォルト・セクション名を示します。

  - CSEG のデフォルト・セクション名 (-Xmulti=pe*n* の場合)

再配置属性	デフォルト・セクション名
TEXT	.text.pe*n*
CONST	.const.pe*n*
SCONST	.sconst.pe*n*

- CSEG のデフォルト・セクション名 (-Xmulti=cmn の場合)

再配置属性	デフォルト・セクション名
TEXT	.text.cmn
CONST	.const.cmn
SCONST	.sconst.cmn

.dseg

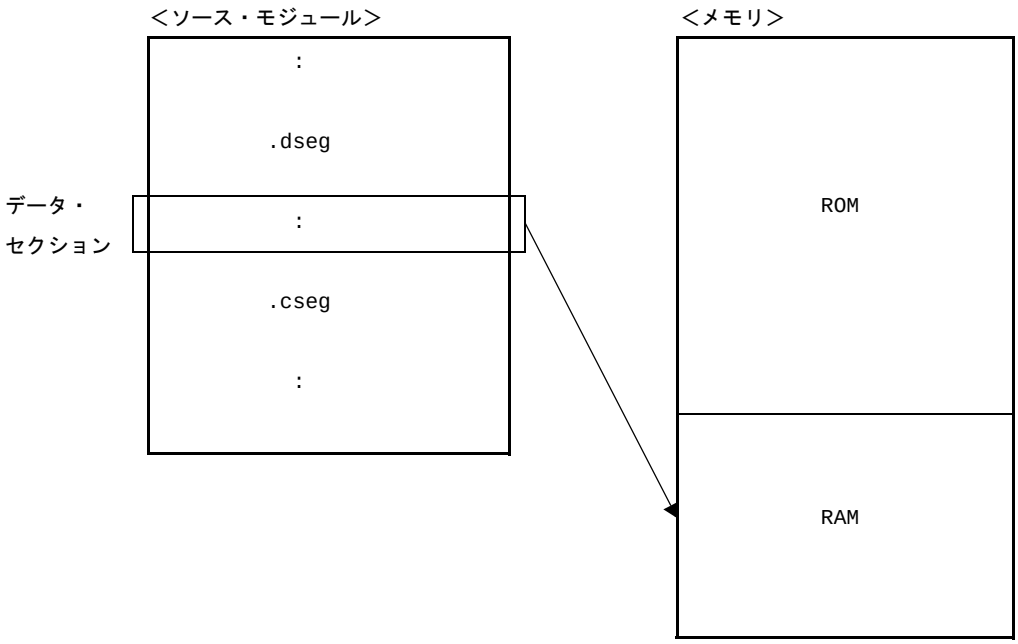
アセンブラにデータ・セクション（RAM 領域へ配置）の開始を指示します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セクション名*]	.dseg	[*再配置属性*]	[; *コメント*]

［機能］

- .dseg 疑似命令は，アセンブラにデータ・セクションの開始を指示します。
- .dseg 疑似命令以降，再びセクション定義疑似命令（.cseg，.dseg，.org）が現れるまで，データ・セクションに属し，最終的に，RAM アドレス内に確保されます。



［用途］

- .dseg 疑似命令で定義するデータ・セクションには，主に .ds 疑似命令を記述します。
  データ・セクションは，RAM 内に配置されます。したがって，データ・セクション内にインストラクションを記述することはできません。
- データ・セクションでは，プログラムで使用する RAM の作業領域を .ds 疑似命令により確保し，それぞれの作業領域のアドレスにラベルを付けます。プログラムを記述する場合，このラベルを利用します。
  データ・セクションとして確保された領域は，RAM 上でほかの作業領域（スタック領域，ほかのモジュールで定義された作業領域など）と重複しないよう，リンカにより配置されます。

［詳細説明］

- 再配置属性とは，データ・セクションの配置アドレスの範囲を限定するものです。

次に，.dseg の再配置属性を示します。

表 4―8 DSEG の再配置属性

再配置属性	記述形式	説明
BSS	BSS	初期値を持たず，gp と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲に配置します。
DATA	DATA	初期値を持ち，gp と 2 命令によって構成される，32 ビットのディスプレースメントを用いて参照されるメモリ範囲に配置します。
SBSS	SBSS	初期値を持たず，gp と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（SDATA セクションとあわせて最大 64K バイト）に配置します。
SDATA	SDATA	初期値を持ち，gp と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（SDATA セクションとあわせて最大 64K バイト）に配置します。
SEBSS	SEBSS	初期値を持たず，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からマイナス方向に最大 32K バイト）のうち，SEDATA セクションのサイズ分だけ上位のアドレスに配置します。
SEDATA	SEDATA	初期値を持ち，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からマイナス方向に最大 32K バイト）のうち，SEBSS セクションのサイズ分だけ下位のアドレスに配置します。
SIBSS	SIBSS	初期値を持たず，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からプラス方向に最大 32K バイト）のうち，SIDATA，TI* セクションのサイズ分だけ上位のアドレスに配置します。
SIDATA	SIDATA	初期値を持ち，ep と 16 ビットのディスプレースメントを用いて 1 命令で参照されるメモリ範囲（ep からプラス方向に最大 32K バイト）のうち，SIBSS，TI* セクションのサイズ分だけ上位のアドレスに配置します。

再配置属性	記述形式	説明
TIBSS	TIBSS	初期値を持たず，内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 TIDATA.BYTE，TIBSS.BYTE，TIDATA.WORD，TIBSS.WORD，TIDATA が使用されていない場合は，TIBSS を ep の示すアドレスに配置します。 TIDATA.BYTE，TIBSS.BYTE，TIDATA.WORD，TIBSS.WORD，TIDATA が使用されている場合は，TIBSS を ep の示すアドレスに TIDATA.BYTE，TIBSS.BYTE，TIDATA.WORD，TIBSS.WORD，TIDATA のサイズを加えたアドレスへ配置します。 sld/sst 命令では，データのサイズによってアクセスする領域の範囲が異なるため，より有効に sld/sst 命令を用いるためには，バイトデータは TIBSS.BYTE セクションに，バイトデータより大きいデータは TIBSS.WORD セクションに置くことを推奨します。このようにアクセス領域を細かく考慮する必要がない場合に，TIBSS セクションを用います。
TIBSS.BYTE	TIBSS.BYTE	内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 sld/sst 命令では，アクセスするデータがバイトデータの場合 128 バイト以内の領域をアクセスできるため，初期値を持たないバイトデータを TIBSS.BYTE セクションに置くことを推奨します。
TIBSS.WORD	TIBSS.WORD	内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 sld/sst 命令では，アクセスするデータがバイトデータより大きい場合 256 バイト以内の領域をアクセスできるため，初期値を持たないバイトデータより大きいデータを TIBSS.WORD セクションに置くことを推奨します。
TIDATA	TIDATA	初期値を持ち，内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 TIDATA.BYTE，TIDATA.WORD が使用されていない場合は，TIDATA を ep の示すアドレスに配置します。 TIDATA.BYTE，TIDATA.WORD が使用されている場合は，TIDATA を ep の示すアドレスに両セクションのサイズを加えたアドレスへ配置します。 sld/sst 命令では，データのサイズによってアクセスする領域の範囲が異なるため，より有効に sld/sst 命令を用いるためには，バイトデータは TIDATA.BYTE セクションに，バイトデータより大きいデータは TIDATA.WORD セクションに置くことを推奨します。このようにアクセス領域を細かく考慮する必要がない場合に，TIDATA セクションを用います。
TIDATA.BYTE	TIDATA.BYTE	内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 sld/sst 命令では，アクセスするデータがバイトデータの場合 128 バイト以内の領域をアクセスできるため，初期値を持つバイトデータを TIDATA.BYTE セクションに置くことを推奨します。

再配置属性	記述形式	説明
TIDATA.WORD	TIDATA.WORD	内蔵 RAM に置かれ，sld/sst 命令を用いて ep 相対でアクセスされることを前提とします。 sld/sst 命令では，アクセスするデータがバイトデータより大きい場合 256 バイト以内の領域をアクセスできるため，初期値を持つバイトデータより大きいデータを TIDATA.WORD セクションに置くことを推奨します。

- 再配置属性を省略した場合，“DATA”と解釈されます。
- 各セクションのサイズが領域のサイズを越えた場合には，エラーとなります。この時，ロケーション・カウンタを進めながら，アセンブルは続行します。
- データ・セクション中に機械語命令は記述できません。記述した場合はエラーとなり，その行は無視されます。
- .dseg 疑似命令のシンボル欄にセクション名を記述することにより，そのデータ・セクションにシンボル（名前）を付けることができます。セクション名が省略されたデータ・セクションには，アセンブラが自動的にデフォルトのセクション名を与えます。

  次に，.dseg のデフォルト・セクション名を示します。

再配置属性	デフォルト・セクション名
BSS	.bss
DATA	.data
SBSS	.sbss
SDATA	.sdata
SEBSS 注	.sebss
SEDATA 注	.sedata
SIBSS 注	.sibss
SIDATA 注	.sidata
TIBSS 注	.tibss
TIBSS.BYTE 注	.tibss.byte
TIBSS.WORD 注	.tibss.word
TIDATA 注	.tidata
TIDATA.BYTE 注	.tidata.byte
TIDATA.WORD 注	.tidata.word

注 これらの再配置属性に指定可能なセクション名は，デフォルト・セクション名のみです。

- 再配置属性が同一であれば，複数のデータ・セクションに同一のセクション名を与えることができます。これらは，アセンブラ内部で 1 つのセクションとして処理されます。
- データ・セクションは分割記述が可能です。つまり，同一モジュール内に記述された同一再配置属性，同一セクション名のデータ・セクションは，アセンブラ内部で連続したひとつのセクションとして処理されます。
- 同名セクションの再配置属性が異なる場合には，エラーとなります。したがって，再配置属性ごとの同名セクションの数は 1 つです。

- セクション名はシンボルとして参照できません。
- マルチコアの場合は次のようになります。【V850E2V3】
  - -Xmulti=pen オプションを指定する場合

    各コア用プログラムでは，シングルコア用プログラムと同様にすべての再配置属性のデータ・セクションに配置することが可能です。
  - -Xmulti=cmn オプションを指定する場合

    共通部のデータ・セクションには再配置属性 DATA/BSS セクションのみ配置することが可能です。再配置属性 DATA/BSS セクション以外を指定するとエラーになります。

    マルチコアでのセクション名の指定がないデータ・セクションには，アセンブラが再配置属性ごとにデフォルトのセクション名自動的に与えます。

    次に，デフォルト・セクション名を示します。

    - DSEG のデフォルト・セクション名 (-Xmulti=pen の場合)

再配置属性	デフォルト・セクション名
BSS	.bss.pen
DATA	.data.pen
SBSS	.sbss.pen
SDATA	.sdata.pen
SEBSS	.sebss.pen
SEDATA	.sedata.pen
SIBSS	.sibss.pen
SIDATA	.sidata.pen
TIBSS	.tibss.pen
TIBSS.BYTE	.tibss.byte.pen
TIBSS.WORD	.tibss.word.pen
TIDATA	.tidata.pen
TIDATA.BYTE	.tidata.byte.pen
TIDATA.WORD	.tidata.word.pen

    - DSEG のデフォルト・セクション名 (-Xmulti=cmn の場合)

再配置属性	デフォルト・セクション名
BSS	.bss.cmn
DATA	.data.cmn

.org

ロケーション・カウンタの値を進めます。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
	.org	*値*	

[機能]

ロケーション・カウンタ値を，オペランドで指定した値まで進めます。

[詳細説明]

前に指定されたセクション定義疑似命令によって指定される現在のセクションに対するロケーション・カウンタ値を，オペランドで指定した値（2^{31} 未満）まで進めます。なお，ロケーション・カウンタ値を進めることによりホールが生じた場合，生じたホールを 0 で埋めます。

[使用例]

ロケーション・カウンタ値を 16 バイトに進める。

```
.org    16
```

[注意事項]

- 現在のロケーション・カウンタ値より小さな値を指定した場合，メッセージが出力され，アセンブルが中止されます。
- sdata 属性セクションにおいて本疑似命令を用いた場合，sdata/sbss 属性セクションに割り当てるデータのサイズを指定するための目安を表示する（-Xsdata_info オプション）際に，正しい情報が得られなくなります。
- 本疑似命令は，そのセクションに対する指定したファイル内でのロケーション・カウンタ値を進めるだけであり，絶対アドレス[注1]を指定するものでも，セクション内オフセット[注2]を指定するものでもありません。

注 1. リンク後のオブジェクト・モジュール・ファイルにおけるアドレス 0 からのオフセットです。

2. リンク後のオブジェクト・モジュール・ファイルにおいてそのセクションが割り当てられたセクション（出力セクション）の先頭アドレスからのオフセットです。

.vseg

アセンブラにデバッグ情報用のセクションの開始を指示します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セクション名*]	.vseg	[*コメント*]	[; *コメント*]

［機能］

- .vseg 疑似命令は，アセンブラにデバッグ情報用のセクションの開始を指示します。
  デバッグ情報用のセクションであるため，変更は行わないでください。

4. 2. 3　シンボル定義疑似命令

シンボル定義疑似命令は，ソース・モジュールを記述する際に使用するデータにシンボル（名前）を割り付けます。これにより，データ値の意味がはっきりし，ソース・モジュールの内容がわかりやすくなります。

シンボル定義疑似命令は，ソース・モジュール中で使用するシンボルの値をアセンブラに知らせるものです。

シンボル定義疑似命令には，次のものがあります。

表 4―9　シンボル定義疑似命令

疑似命令	概要
.set	シンボルの定義
.file	シンボル・テーブル・エントリを生成
.func	シンボル・テーブル・エントリを生成

.set

シンボルの定義をします。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
シンボル	.set	*値*	

[機能]

シンボル欄に指定したシンボル名と，オペランド欄に指定した値（整数値）を持つシンボルを定義します。

[詳細説明]

シンボル欄に指定したシンボル名と，オペランド欄に指定した値（整数値）を持つシンボルを定義します。1 つのアセンブラ・ソース・ファイルにおいて，あるシンボルに対して複数回 .set 疑似命令を指定した場合，そのシンボルの参照は，その参照が現れた位置に従って，次の値を持ちます。

- ファイルの先頭からそのシンボルに対する最初の .set 疑似命令までの間に現れた場合
  そのシンボルに対する最後の .set 疑似命令で指定した値
- ある .set 疑似命令から次の .set 疑似命令までの間，または次の .set 疑似命令が現れなかった場合には，そのアセンブラ・ソース・ファイルの終わりまでの間に現れた場合
  その .set 疑似命令で指定した値

[使用例]

シンボル sym1 の値を 0x10 として定義する。

```
.set    sym1, 0x10
```

[注意事項]

- 値の指定にラベル参照や，その時点において未定義なシンボル参照を用いることはできません。
  値の指定にラベル参照や，その時点において未定義なシンボル参照を用いた場合，次のメッセージが出力され，アセンブルが終了されます。

```
E0550203: illegal expression (string)
```

- ラベル名，.macro 疑似命令で定義済みのマクロ名，およびマクロの仮パラメータと同名のシンボルを指定した場合，次のメッセージが出力され，アセンブルが終了されます。

```
E0550212: symbol already define as string
```

.file

シンボル・テーブル・エントリを生成（file タイプ）します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
	.file	" *ファイル名* "	[; *コメント*]

［機能］

- オブジェクト・モジュール・ファイルの生成時，オペランドに指定したファイル名と，タイプ file を持つシンボル・テーブル・エントリ注を生成します。なお，入力したソース・ファイル中に本疑似命令が存在しない場合，「.file“入力ファイル名”」を指定したものとみなし，入力ファイル名とタイプ file を持つシンボル・テーブル・エントリを生成します。

注 バインディング・クラスは LOCAL とします。

［用途］

- .file 疑似命令はコンパイラのデバッグ情報です。

［詳細説明］

- ファイル名は，指定イメージで記述します。
- コンパイラが出力する C ソース・プログラムのファイル名です。

.func

シンボル・テーブル・エントリを生成（FUNC タイプ）します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
	.func	*関数名*，*関数サイズ*，*スタック・サイズ*	[; *コメント*]

［機能］

- オブジェクト・モジュール・ファイル生成時，タイプ FUNC を持つシンボル・テーブル・エントリを生成します。

［用途］

- .func 疑似命令はコンパイラのデバッグ情報です。

［詳細説明］

- 第 1 オペランドはコンパイラが出力する C 言語の関数名，第 2 オペランドはその関数サイズを示す式，第 3 オペランドはその関数のスタック・サイズを示す数値です。
- .func 疑似命令は，コンパイラが出力する C ソース・プログラムの関数情報です。

4. 2. 4　データ定義，領域確保疑似命令

データ定義疑似命令は，プログラムで使用する定数データを定義します。

定義したデータの値は，オブジェクト・コードとして生成されます。

領域確保疑似命令は，プログラムで使用するメモリの領域を確保します。

データ定義，領域確保疑似命令には，次のものがあります。

表 4―10　データ定義，領域確保疑似命令

疑似命令	概要
.db	1 バイト領域を初期化
.db2/.dhw	2 バイト領域を初期化
.dshw	2 バイトの領域を，指定した値を右に 1 ビット・シフトした値で初期化
.db4/.dw	4 バイト領域を初期化
.db8/.ddw	8 バイト領域を初期化
.float	4 バイト領域を初期化
.double	8 バイト領域を初期化
.ds	オペランドで指定したバイト数分のメモリ領域を確保
.align	ロケーション・カウンタの値を整列

.db

1バイト領域を初期化します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.db	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.db	*式* [, …]	[; *コメント*]
		または	
[*ラベル* :]	.db	*" 文字列定数 "*	[; *コメント*]

[機能]

- 1バイト領域を初期化します。
  初期化するバイト数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリをバイト単位で初期化します。

[用途]

- プログラムで使用する式や文字列を定義するときに，.db 疑似命令を使用します。

[詳細説明]

- オペランドがカッコ“(”，“)”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは，指定されたバイト数分の領域を“0”で初期化します。

(b) サイズには，絶対式を記述できます。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

(2) 初期値指定の場合

(a) 式

式の値は1バイトのデータとして確保されます。したがって，式の値は 0x0 ～ 0xFF の間でなければなりません。1バイトを越えた場合，下位1バイトがデータとして確保されます。

（b）文字列定数

第 1 オペランドが対応する’“’ で囲まれているときは文字列定数を記述したとみなします。

文字列定数が記述された場合，1 文字に対して，それぞれ 8 ビット ASCII コードが確保されます。

- .db 疑似命令は，ビット・セクション内では記述することはできません。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式が記述することができます。
- .db 疑似命令を記述するセクションの再配置属性が BSS，SBSS の場合には初期値指定はできないものとして，エラーを出力します。

［使用例］

```
                .cseg   text
WORK1:          .db     (1)                 ; (1)
WORK2:          .db     (2)                 ; (1)
                .cseg   text
MASSAG:         .db     "ABCDEF"            ; (2)
DATA1:          .db     0xA, 0xB, 0xC       ; (3)
DATA2:          .db     (3 + 1)             ; (4)
DATA3:          .db     "AB" + 1            ; (5)   ←エラー
```

（1）サイズを指定しているので，それぞれのバイト領域を値“0”で初期化します。

（2）6 バイトの領域を文字列“ABCDEF”で初期化します。

（3）3 バイトの領域を 0xA，0xB，0xC で初期化します。

（4）4 バイトの領域を 0x0 で初期化します。

（5）この記述はエラーとなります。

.db2/.dhw

2バイト領域を初期化します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.db2	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.db2	*式* [, …]	[; *コメント*]
		または	
[*ラベル* :]	.dhw	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.dhw	*式* [, …]	[; *コメント*]

[機能]

- 2バイト領域を初期化します。
  初期化する2バイト・データ数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリを2バイト単位に初期化します。

[用途]

- プログラムで使用するアドレスやデータなどの2バイトの定数を定義するときに，.db2，.dhw 疑似命令を使用します。

[詳細説明]

- オペランドがカッコ“(”，“)”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは指定された2バイト・データ数分の領域を“0”で初期化します。

(b) サイズには，絶対式を記述できます。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

(2) 初期値指定の場合

(a) 式

式の値は，2 バイト・データとして確保されます。したがって，式の値は 0x0 ～ 0xFFFF の間でなければなりません。2 バイトを越えた場合，下位 2 バイトがデータとして確保されます。

文字列定数は，初期値として記述できません。

- .db2，.dhw 疑似命令は，ビット・セクション内では記述できません。
- .db2，.dhw 疑似命令を記述するセクションの再配置属性が BSS，SBSS の場合には初期値指定はできないものとして，エラーを出力します。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式が記述することができます。

.dshw

2 バイトの領域を，指定した値を右に 1 ビット・シフトした値で初期化します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.dshw	*式* [, …]	[; *コメント*]

［機能］

- 2 バイトの領域を，指定した値を右に 1 ビット・シフトした値で初期化します。

［詳細説明］

- 式の値を右に 1 ビット・シフトした値を 2 バイトデータとして確保します。
- .dshw 疑似命令はビット・セクション内では記述できません。
- .dshw 疑似命令はセクションの再配置属性が BSS，SBSS の場合には記述できないものとし，エラーを出力します。
- オペランドの式には絶対式が記述可能です。
- 式の値は，右に 1 ビット・シフトした値が 0x0 ～ 0xFFFF の範囲でなければなりません。それ以外の場合は，下位 2 バイト分のデータを確保します。
- 式は，カンマで区切ることにより，1 行の範囲ならばいくつでも指定できます。
- オペランドに文字列定数を記述することはできません。

.db4/.dw

4バイト領域を初期化します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.db4	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.db4	*式* [, …]	[; *コメント*]
		または	
[*ラベル* :]	.dw	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.dw	*式* [, …]	[; *コメント*]

[機能]

- 4バイト領域を初期化します。
  初期化する4バイト・データ数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリを4バイト単位に初期化します。

[用途]

- プログラムで使用するアドレスやデータなどの4バイトの定数を定義するときに，.db4，.dw 疑似命令を使用します。

[詳細説明]

- オペランドがカッコ“(”，“)”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは指定された4バイト・データ数分の領域を“0”で初期化します。

(b) サイズには，絶対式を記述できます。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

（2）初期値指定の場合

（a）式

式の値は，4 バイト・データとして確保されます。したがって，式の値は 0x0 ～ 0xFFFFFFFF の間でなければなりません。4 バイトを越えた場合，下位 4 バイトがデータとして確保されます。

文字列定数は，初期値として記述できません。

- .db4，.dw 疑似命令は，ビット・セクション内では記述できません。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式が記述することができます。
- .db4，.dw 疑似命令を記述するセクションの再配置属性が BSS，SBSS の場合には初期値指定はできないものとして，エラーを出力します。

.db8/.ddw

8 バイト領域を初期化します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.db8	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.db8	*絶対式* [, …]	[; *コメント*]
		または	
[*ラベル* :]	.ddw	(*絶対式*)	[; *コメント*]
		または	
[*ラベル* :]	.ddw	*絶対式* [, …]	[; *コメント*]

[機能]

- 8 バイト領域を初期化します。
  初期化する 8 バイト・データ数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリを 8 バイト単位に初期化します。

[用途]

- プログラムで使用するアドレスやデータなどの 8 バイトの定数を定義するときに，.db8，.ddw 疑似命令を使用します。

[詳細説明]

- オペランドがカッコ“(”，“)”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは指定された 8 バイト・データ数分の領域を“0”で初期化します。

(b) サイズには，絶対式を記述できます。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

(2) 初期値指定の場合

(a) 式

式の値は，8 バイト・データとして確保されます。したがって，式の値は 0x0 ～ 0xFFFFFFFFFFFFFFFF の間でなければなりません。8 バイトを越えた場合，下位 8 バイトがデータとして確保されます。

文字列定数は，初期値として記述できません。

- .db8，.ddw 疑似命令は，ビット・セクション内では記述できません。
- .db8，.ddw 疑似命令はセクションの再配置属性が BSS，SBSS の場合には記述できないものとし，エラーを出力します。
- 初期値は，1 行の範囲であれば複数指定することができます。

.float

4 バイト領域を初期化します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.float	*絶対式* [, …]	[; *コメント*]

[機能]

- 4 バイト領域を初期化します。
- オペランドで指定された絶対式で，メモリを 4 バイト単位に初期化します。

[詳細説明]

- 絶対式の値は，単精度浮動小数点数として確保されます。したがって，式の値は 1.40129846e-45 ～ 3.40282347e+38 の間でなければなりません。それ以外の場合は，下位 4 バイト分のデータが単精度浮動小数点数として確保されます。
- .float 疑似命令は，ビット・セクション内では記述できません。
- .float 疑似命令はセクションの再配置属性が BSS，SBSS の場合には記述できないものとし，エラーを出力します。
- 絶対式は，1 行の範囲であれば複数指定することができます。

.double

8 バイト領域を初期化します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.double	*絶対式* [, …]	[; *コメント*]

［機能］

- 8 バイト領域を初期化します。
- オペランドで指定された初期値で，メモリを 8 バイト単位に初期化します。

［詳細説明］

- 絶対式の値は，倍精度浮動小数点数として確保されます。したがって，式の値は 4.9406564584124654e-324 ～ 1.7976931348623157e+308 の間でなければなりません。それ以外の場合は，下位 8 バイト分のデータが倍精度浮動小数点数として確保されます。
- .double 疑似命令は，ビット・セクション内では記述できません。
- .double 疑似命令はセクションの再配置属性が BSS，SBSS の場合には記述できないものとし，エラーを出力します。
- 絶対式は，1 行の範囲であれば複数指定することができます。

.ds

オペランドで指定したバイト数分のメモリ領域を確保します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.ds	(*絶対式*)[, …]	[; *コメント*]
		または	
[*ラベル* :]	.ds	*絶対式*	[; *コメント*]

［機能］

- オペランドで指定したバイト数分のメモリ領域を確保します。

［用途］

- .ds 疑似命令は，主にプログラムで使用するメモリ（RAM）の領域を確保するときに使用します。
  ラベルがある場合は，確保したメモリ領域の先頭アドレスの値をそのラベルに割り付けます。ソース・モジュールでは，このラベルを使用してメモリを操作する記述をします。

［詳細説明］

- 第 1 オペランドがカッコ“ (”，“) ”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。
- 第 1 オペランドがサイズ指定であり，さらに第 2 オペランドが指定されているときはその値を初期値指定とみなされます。

（1）サイズ指定の場合

（a）オペランドにサイズを記述した場合，アセンブラは指定されたバイト数分の領域を，初期値指定があればその値で，初期値指定がなければ“0”で初期化します。ただし，サイズ指定のバイト数が 0 の場合は，領域の確保は行われません。

（b）サイズには，絶対式を記述します。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

（2）初期値指定の場合

（a）式

式の値は，バイト・データとして確保されます。式の値は 0x0 ～ 0xFF の間でなければなりません。それ以外の場合，下位 1 バイトがデータとして確保され，エラーが出力されます。

- .ds 疑似命令は，ビット・セクション内では記述できません。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式が記述することができます。
- 本疑似命令を記述するセクションの再配置属性が BSS，SBSS の場合には初期値指定はできないものとして，エラーを出力し，本疑似命令を無視します。

.align

ロケーション・カウンタの値を整列します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.align	*整列条件* [, *絶対式*]	[; *コメント*]

［機能］

- ロケーション・カウンタの値を整列します。

［詳細説明］

- 前に指定されたセクション定義疑似命令によって指定される現在のセクションに対するロケーション・カウンタ値を，第 1 オペランドで指定した整列条件で整列します。なお，ロケーション・カウンタ値を整列したことによりホールが生じた場合，生じたホールを第 2 オペランドで指定した絶対式の値，またはデフォルト値の 0 で埋めます。
- .align 疑似命令は，ビット・セクション内では記述できません。
- 整列条件は 2 以上 2^{31} 未満の偶数にしてください。それ以外のものを指定した場合，エラーが出力され，アセンブルが中止されます。
- 第 2 オペランドの絶対式の値は 0x0 ～ 0xFF の間でなければなりません。それ以上の値を指定した場合，下位 1 バイト分の値が用いられます。
- 本疑似命令は，そのセクションに対する指定したファイル内でのロケーション・カウンタ値を整列するだけであり，配置後のアドレスを整列するものでもありません。

4. 2. 5　外部定義，外部参照疑似命令

外部定義，外部参照疑似命令は，ほかのモジュールで定義されているシンボルを参照する場合に，その関連性を明白にさせるためのものです。

1 つのプログラムがモジュール 1 とモジュール 2 に分けて作成されている場合を考えます。モジュール 1 中において，モジュール 2 中で定義されているシンボルを参照したい場合，お互いのモジュールで何の宣言もなくそのシンボルを使うわけにはいきません。このため，「使いたい」，「使ってもよい」の表示をそれぞれのモジュールで行う必要があります。

モジュール 1 では，「ほかのモジュール中で定義されているシンボルを参照したい」というシンボルの外部参照宣言をします。一方，モジュール 2 では，「そのシンボルは，ほかのシンボルで参照してもよい」というシンボルの外部定義宣言をします。

外部参照と外部定義という 2 つの宣言が有効に行われて，はじめてそのシンボルを参照することができます。

この相互関係を成立させるのが，外部定義，外部参照疑似命令であり，次の命令があります。

表 4―11　外部定義，外部参照疑似命令

疑似命令	概要
.public	オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言
.extern	本モジュールで参照するほかのモジュールのビット・シンボル以外のシンボルを宣言
.comm	未定義外部シンボルを宣言

.public

オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言します。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.public	*ラベル名* [, *サイズ*]	[; *コメント*]

［機能］

- オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言します。

［用途］

- ほかのモジュールから参照されるシンボルを定義している場合には，必ず，そのシンボルを .public 疑似命令で外部定義宣言します。

［詳細説明］

- 第 1 オペランドで指定したラベル名と同名のラベルを外部ラベル[注]として宣言します。
  なお，第 2 オペランドを指定した場合，指定した値をそのラベルの示すデータのサイズとして指定します。

  注 外部シンボル（GLOBAL のバインディング・クラスを持つシンボル）です。

- 本疑似命令と .extern 疑似命令は外部ラベルを宣言するという機能において変わりませんが，指定したファイル内に定義を持つラベルを外部ラベルとして宣言する場合は本疑似命令を用い，指定したファイル内に定義を持たないラベルを外部ラベルとして宣言する場合は .extern 疑似命令を用いるようにしてください。
- .public 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
- .public 疑似命令は 1 行で 1 つのシンボルのみ定義可能です。
- オペランドに記述するシンボルは，同一モジュール内で定義していなければなりません。定義されていない場合はエラーを出力し，該当シンボルの .public 宣言を無視します。
  エラーになったシンボル名はエラー・メッセージ中に出力されます。
- 次のシンボルは，オペランドとして記述することはできません。

(1) .set 疑似命令で定義したシンボル

(2) セクション名

［使用例］

- モジュール 1

```
        .public A1              ; (1)
        .extern B1

A1      .set    0x10

        .cseg   text
        jr      B1
```

- モジュール 2

```
        .public B1              ; (2)
        .extern A1
        .cseg   text
B1:
        mov     A1, r12
```

(1) シンボル A1 が，ほかのモジュールから参照されるシンボルであることを宣言します。

(2) シンボル B1 が，ほかのモジュールから参照されるシンボルであることを宣言します。

.extern

本モジュールで参照するほかのモジュールのビット・シンボル以外のシンボルを宣言します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.extern	*ラベル名* [, *サイズ*]	[; *コメント*]

[機能]

- 本モジュールで参照するほかのモジュールのシンボルを宣言します。

[用途]

- ほかのモジュールの中で定義されているシンボルを参照する場合には，必ずそのシンボルを .extern 疑似命令で外部参照宣言します。

[詳細説明]

- 第 1 オペランドで指定したラベル名と同名のラベルを，外部ラベル[注]として宣言します。
  なお，第 2 オペランドを指定した場合，指定した値をそのラベルの示すデータのサイズとして指定します。

  注 外部シンボル（GLOBAL のバインディング・クラスを持つシンボル）です。

- 本疑似命令と .public 疑似命令は外部ラベルを宣言するという機能において変わりませんが，指定したファイル内に定義を持たないラベルを外部ラベルとして宣言する場合は本疑似命令を用い，指定したファイル内に定義を持つラベルを外部ラベルとして宣言する場合は .public 疑似命令を用いるようにしてください。
- .extern 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
- .extern 疑似命令は 1 行で 1 つのシンボルのみ定義可能です。
- .extern 疑似命令で宣言されたシンボルをモジュール中で参照しなくても，エラーにはなりません。
- すでに宣言されたシンボルは，.extern 疑似命令のオペランドに記述することはできません。逆に，.extern 宣言したシンボルも，ほかの疑似命令により再定義，宣言することはできません。

.comm

未定義外部シンボルを宣言します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.comm	*シンボル名* , *サイズ* , *整列条件*	[; *コメント*]

[機能]

- 第 1 オペランドで指定したシンボル名，第 2 オペランドで指定したサイズ，および第 3 オペランドで指定した整列条件を持つ未定義外部シンボルを宣言します。

[詳細説明]

(1) 起動時に -Xsdata オプションを指定した場合

(a) 第 2 オペランドで指定したサイズが 1 以上 *num* バイト以下の場合

オブジェクト・モジュール・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GPCOMMON のセクション・ヘッダ・テーブル・インデクスの値を持つシンボルを生成します。

(b) 第 2 オペランドで指定したサイズが 0 であるか *num* バイトより大きい場合

オブジェクト・モジュール・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，COMMON セクション・ヘッダ・テーブル・インデクスの値を持つシンボルを生成します。

(2) 起動時に -Xsdata オプションを指定しなかった場合

(a) オブジェクト・モジュール・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GPCOMMON のセクション・ヘッダ・テーブル・インデクスの値を持つシンボルを生成します。

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において通常のラベル定義により定義されている場合

  - そのラベルが本疑似命令により GPCOMMON のシンボル・テーブル・エントリ・インデクスを持つよう宣言され data 属性セクションにおいて通常のラベル定義により定義されている場合，または本疑似命令により COMMON のシンボル・テーブル・エントリ・インデクスを持つように宣言され sdata 属性セクションにおいて通常のラベル定義により定義されている場合

```
    .comm   lab1, 4, 4  --“-G”なしでアセンブルするとGPCOMMON
      :
    .data   .dseg   data
lab1:                   --.dataセクションで通常のラベル定義
```

次のメッセージが出力され，アセンブルが中止されます。

```
E0550213: label identifier redefined
```

- 上記以外の場合

通常のラベル定義により定義されたラベルが外部ラベルとみなされ，本疑似命令の指定が無視されます。オブジェクト・モジュール・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GLOBAL のバインディング・クラスを持つシンボル・テーブル・エントリが生成されます。

```
    .comm   lab1, 4, 4  --“-G”なしでアセンブルするとGPCOMMON
      :
    .sdata  .dseg   sdata
lab1:                   --.sdataセクションで通常のラベル定義
```

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において .lcomm 疑似命令により定義されている場合

- .lcomm 疑似命令で指定されたサイズ，または整列条件と，本疑似命令において指定されたサイズ，または整列条件が異なる場合

```
.comm   lab1, 4, 4
  :
.sbss   .dseg   sbss
.lcomm  lab1, 4, 2      -- 整列条件が異なる
```

次のメッセージが出力され，アセンブルが中止されます。

```
E0550213: label identifier redefined
```

- そのラベルが本疑似命令により GPCOMMON のセクション・ヘッダ・テーブル・インデクスを持つよう宣言され bss 属性セクションにおいて .lcomm 疑似命令により定義されている場合，または本疑似命令により COMMON のセクション・ヘッダ・テーブル・インデクスを持つよう宣言され sbss 属性セクションにおいて .lcomm 疑似命令により定義されている場合

```
.comm   lab1, 4, 4      --“-G”なしでアセンブルするとGPCOMMON
  :
.bss    .dseg   bss
.lcomm  lab1, 4, 4      --.bssセクションで定義
```

次のメッセージが出力され，アセンブルが中止されます。

```
E0550213: label identifier redefined
```

- 上記以外の場合

.lcomm により定義されたラベルが外部ラベルとみなされ，本疑似命令の指定が無視されます。オブジェクト・モジュール・ファイル生成時のそのラベルに対するシンボル・テーブル・エントリの生成において，GLOBAL のバインディング・クラスを持つシンボル・テーブル・エントリが生成されます。

```
.comm   lab1, 4, 4      --“-G”なしでアセンブルすると GPCOMMON
  :
.sbss   .dseg   sbss
.lcomm  lab1, 4, 4      --.sbss セクションで定義
```

- 第 1 オペランドで指定したラベル名と同名のラベルが，本疑似命令と同じファイル内において本疑似命令により（再）定義されている場合

- サイズ，または境界条件において異なっている場合

```
.comm   lab1, 4, 4
  :
.comm   lab1, 2, 4      -- サイズが異なる
```

次のメッセージが出力され，アセンブルが中止されます。

```
E0550213: label identifier redefined
```

- サイズ，および境界条件が同じ場合

.comm 疑似命令が 1 回だけ指定されたものと見なされます。

［使用例］

サイズ 4 の未定義外部ラベルを整列条件 4 で宣言する。

```
.sbss   .dseg   sbss
.comm   _p, 4, 4
```

4. 2. 6 マクロ疑似命令

ソースを記述する場合，使用頻度の高い一連の命令群をそのつど記述するのは面倒です。また，記述ミス増加の原因ともなります。

マクロ疑似命令により，マクロ機能を使用することにより，同じような一連の命令群を何回も記述する必要がなくなり，コーディングの効率を上げることができます。

マクロの基本的な機能は，一連の文の置き換えにあります。

マクロ疑似命令には，次のものがあります。

表 4―12 マクロ疑似命令

疑似命令	概要
.macro	.macro 疑似命令と .endm 疑似命令の間に記述された一連の文に対し，シンボル欄で指定したマクロ名を付け，マクロを定義
.local	指定した文字列を特有の識別子として置き換えられるローカル・シンボルとして宣言
.rept	.rept 疑似命令と .endm 疑似命令の間に記述された一連の文をオペランド欄で指定した式の値分だけ，繰り返し展開
.irp	.irp 疑似命令と .endm 疑似命令の間にある一連の文をオペランドで指定された実パラメータで仮パラメータを置き換えながら，実パラメータの数だけ繰り返し展開
.exitm	.exitm 疑似命令を囲んでいる最も内側の .irp，.rept 疑似命令の繰り返しアセンブルをスキップ
.exitma	.exitma 疑似命令を囲んでいる最も外側の .irp，.rept 疑似命令の繰り返しアセンブルをスキップ
.endm	マクロの機能として定義される一連のステートメントを終了

.macro

.macro 疑似命令と .endm 疑似命令の間に記述された一連の文に対し，シンボル欄で指定したマクロ名を付け，マクロを定義します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
マクロ名	.macro	[*仮パラメータ* [, …]]	[; *コメント*]
	:		
	マクロ・ボディ		
	:		
	.endm		[; *コメント*]

[機能]

- .macro 疑似命令と .endm 疑似命令の間に記述された一連の文（マクロ・ボディと呼びます）に対し，シンボル欄で指定したマクロ名を付け，マクロの定義を行います。

[用途]

- ソース中で，使用頻度の高い一連の文をマクロ定義しておきます。その定義以降では，定義されたマクロ名を記述するだけで，そのマクロ名に対応するマクロ・ボディが展開されます。

[詳細説明]

- .macro 疑似命令には，対応する .endm 疑似命令がなければ，エラーとなります。
- シンボル欄に記述するマクロ名の規則については，「(2) シンボル」を参照してください。
- マクロを参照する場合は，ニモニック欄に定義済みのマクロ名を記述します。
- オペランド欄に記述する仮パラメータの規則については，シンボル記述上の規則と同じです。
- 仮パラメータは，マクロ・ボディ内でのみ有効です。
- 仮パラメータとして予約語を記述すると，エラーとなります。ただし，ユーザ定義シンボルを記述した場合には，仮パラメータとしての認識が優先されます。
- 仮パラメータと実パラメータの個数は同じでなければなりません。実パラメータが足りない場合はエラーが出力されます。
- マクロ・ボディ内で定義したシンボル／ラベルを .local 疑似命令で宣言すれば，そのシンボル／ラベルは 1 回のマクロ展開でのみ有効になります。
- 1 つのモジュール内でのマクロ定義の最大数には，特に制限はありません。メモリが使えるかぎり定義することができます。
- クロスリファレンス・リストには，仮パラメータの定義行，参照行，シンボル名は出力されません。

- マクロ・ボディ中に，2 つ以上のセクションを定義することはできません。定義した場合は，エラーが出力されます。
- マクロ・ボディ内で参照されていない余分な仮パラメータがあった場合はエラーが出力されます。
- マクロ・ボディ内で，未定義なマクロの呼び出しが行われた場合，メッセージが出力され，アセンブルが中止されます。
- マクロ・ボディ内で，現在定義中のマクロの呼び出しが行われた場合，メッセージが出力され，アセンブルが中止されます。
- 仮パラメータにラベルや疑似命令で定義済みのパラメータを指定した場合，メッセージが出力され，アセンブルが中止されます。
- マクロ呼び出しにおいて実パラメータに指定可能なものは，ラベル名，シンボル名，数値，レジスタ，および命令ニモニックのみです。
  ラベル式（LABEL-1），参照方法指定ラベル（#LABEL），またはベース・レジスタ指定（[gp]）などを指定した場合，指定された実パラメータに依存したメッセージが出力され，アセンブルが中止されます。
- マクロ・ボディ内には，文の並びが指定可能です。オペランドなど，文の一部を指定することはできません。オペランドにマクロ呼び出しが存在する場合，マクロ名の未定義なラベル参照として扱われるか，メッセージが出力され，アセンブルが中止されます。
- マクロ定義中のマクロ・ボディ内で，マクロ定義を記述した場合は，エラーを出力し，処理が継続されます（対応する .endm 疑似命令までに記述された内容は無視されます）。そのマクロ名を参照した場合は定義エラーとなります。

［使用例］

```
ADMAC   .macro  PARA1, PARA2    ; (1)
        mov     PARA1, r12
        add     PARA2, r12
        .endm                   ; (2)

        ADMAC   0x10, 0x20      ; (3)
```

(1) マクロ名“ADMAC”，2 つの仮引数“PARA1”，“PARA2”を指定したマクロ定義をしています。

(2) マクロ定義の終わりを示します。

(3) マクロ ADMAC を参照しています。

.local

指定した文字列を特有の識別子として置き換えられるローカル・シンボルとして宣言します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
	.local	*シンボル名* [, …]	[; *コメント*]

[機能]

- 指定した文字列を特有の識別子として置き換えられるローカル・シンボルとして宣言します。

[用途]

- マクロ・ボディ内でシンボルを定義しているマクロを 2 回以上参照すると，シンボルは二重定義エラーとなります。
  .local 疑似命令を使用することにより，シンボルを定義しているマクロを複数回，参照することができます。

[詳細説明]

- .local 疑似命令の仮パラメータに 4294967294 個以上のローカル・シンボルを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
F0550514: 実パラメータが 4294967294 個以上用いられています。
```

- アセンブラによって生成されるローカル・シンボル名は，.??00000000 から .??FFFFFFFF までの範囲で生成されます。

[使用例]

```
m1      .macro  x
        .local  a, b
        a:      .dw     a
        b:      .dw     x
.endm
m1      10
m1      20
```

展開すると次のようになります。

```
.??00000000:    .dw     .??00000000
.??00000001:    .dw     10
.??00000002:    .dw     .??00000002
.??00000003:    .dw     20
```

.rept

.rept 疑似命令と .endm 疑似命令の間に記述された一連の文をオペランド欄で指定した式の値分だけ，アセンブラが繰り返し展開します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.rept	*絶対式*	[; *コメント*]
	:		
	.endm		[; *コメント*]

[機能]

- .rept 疑似命令と .endm 疑似命令の間に記述された一連の文（REPT-ENDM ブロックと呼びます）をオペランド欄で指定した式の値分だけ，アセンブラが繰り返し展開します。

[用途]

- ソース中で一連の文を連続して繰り返し記述する場合に，.rept，.endm 疑似命令を使用します。

[詳細説明]

- .rept 疑似命令に対応する .endm 疑似命令がなければ，エラーとなります。
- REPT-ENDM のブロックの途中で .exitm が現れると，展開を中止します。
- REPT-ENDM のブロック内に，アセンブル制御命令を記述することができます。
- REPT-ENDM のブロック内に，マクロ定義を記述することはできません。
- 値は，32 ビットの符号付き整数として評価されます。
- 文の並び（ブロック）がない場合，何も行いません。
- 式の評価結果が負になった場合は，メッセージが出力され，アセンブルが中止されます。
- マクロ定義中のマクロ・ボディ内でマクロ定義を記述した場合は，エラーを出力し，展開処理を行わずに処理が継続されます。

［使用例］

```
.cseg   text
    ; REPT-ENDM ブロック
.rept   3               ; (1)
        nop
        ; ソース本文
.endm                   ; (2)
```

(1) REPT-ENDM ブロックを 3 回連続して展開するよう，指示しています。

(2) REPT-ENDM ブロックの終了を示します。

.irp

.irp 疑似命令と .endm 疑似命令の間にある一連の文をオペランドで指定された実パラメータ（左から順）で仮パラメータを置き換えながら，実パラメータの数だけ繰り返し展開します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.irp	*仮パラメータ* [*実パラメータ* [, …]]	[; *コメント*]
	:		
	.endm		[; *コメント*]

[機能]

- .irp 疑似命令と .endm 疑似命令の間にある一連の文（IRP-ENDM ブロックと呼びます）をオペランドで指定された実パラメータ（左から順）で仮パラメータを置き換えながら，実パラメータの数だけ繰り返し展開します。

[用途]

- ソース中で，一部分だけ変数となる一連の文を連続して繰り返し記述したい場合に，IRP-ENDM 疑似命令を使用します。

[詳細説明]

- .irp 疑似命令には対応する .endm 疑似命令がなければ，エラーとなります。
- IRP-ENDM ブロックの途中に .exitm を記述すると，そこで展開を中止します。
- IRP-ENDM ブロックで，マクロを定義することはできません。
- IRP-ENDM ブロック内に，アセンブル制御命令を記述することができます。
- 仮パラメータと実パラメータに同じパラメータ名を指定した場合，メッセージが出力され，アセンブルが中止されます。
- 仮パラメータと実パラメータにラベルや他の疑似命令で定義済みのパラメータ名を指定した場合，メッセージが出力され，アセンブルが中止されます。
- マクロ定義中のマクロ・ボディ内でマクロ定義を記述した場合は，エラーを出力し，展開処理を行わずに処理が継続されます。

［使用例］

```
.cseg   text

.irp    PARA 0xA, 0xB, 0xC      ; (1)
    ; IRP-ENDM ブロック
        add     PARA, r12
        mov     r11, r12
.endm                           ; (2)
    ; ソース本文
```

(1) 仮パラメータが PARA，実パラメータが 0xA，0xB，0xC の 3 個です。
仮パラメータ“PARA”を実引数“0xA”，“0xB”，“0xC”に置き換えながら，IRP-ENDM ブロックを実パラメータの数 3 回分展開することを指示します。

(2) IRP-ENDM ブロックの終了を示します。

.exitm

本疑似命令を囲んでいる最も内側の .irp，.rept 疑似命令の繰り返しアセンブルをスキップします。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.exitm		[; *コメント*]

［機能］

- 本疑似命令は，本疑似命令を囲んでいる最も内側の .irp，.rept 疑似命令の繰り返しアセンブルをスキップします。

［詳細説明］

- 本疑似命令が .irp，.rept 疑似命令に囲まれていない場合，メッセージが出力され，アセンブルが中止されます。

.exitma

本疑似命令は，本疑似命令を囲んでいる最も外側の .irp，.rept 疑似命令の繰り返しをスキップします。

［指定形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	.exitma		[; *コメント*]

［機能］

- 本疑似命令は，本疑似命令を囲んでいる最も外側の .irp，.rept 疑似命令の繰り返しをスキップします。

［詳細説明］

- 本疑似命令が .irp，.rept 疑似命令に囲まれていない場合，メッセージが出力され，アセンブルが中止されます。

.endm

マクロの機能として定義される一連のステートメントの終了をアセンブラに指示します。

[指定形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
	.endm		[; コメント]

[機能]

- マクロの機能として定義される一連のステートメントの終了をアセンブラに指示します。

[用途]

- .macro 疑似命令，.rept 疑似命令，および .irp 疑似命令に続く一連のマクロ・ステートメントの最後には，必ず .endm 疑似命令を記述します。

[詳細説明]

- .macro 疑似命令と .endm 疑似命令の間に記述された一連のマクロ・ステートメントがマクロ・ボディとなります。
- .rept 疑似命令と .endm 疑似命令の間に記述された一連のステートメントが，REPT-ENDM ブロックとなります。
- .irp 疑似命令と .endm 疑似命令の間に記述された一連のステートメントが，IRP-ENDM ブロックとなります。
- 本疑似命令に対応する .macro 疑似命令，.rept 疑似命令，および .irp 疑似命令が存在しない場合，メッセージが出力され，アセンブルが中止されます。

[使用例]

(1) MACRO-ENDM

```
ADMAC   .macro  PARA1, PARA2
                mov     A, #PARA1
                add     A, #PARA2
        .endm
```

（2）REPT-ENDM

```
.cseg   text
  :
.rept   3
        inc     B
        DEC     C
.endm
```

（3）IRP-ENDM

```
.cseg   text
  :
.irp    PARA, <1, 2, 3>
        add     A, #PARA
        mov     [DE], A
.endm
```

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。

次に，制御命令の種類を示します。

表 4―13　制御命令一覧

制御命令の種類	制御命令
コンパイル対象品種指定制御命令	PROCESSOR
シンボル制御命令	EXT_ENT_SIZE，EXT_FUNC
アセンブラ制御命令	CALLT，REG_MODE，EP_LABEL，NO_EP_LABEL，NO_MACRO，MACRO，DATA，SDATA，NOWARNING，WARNING
ファイル入力制御命令	INCLUDE，BINCLUDE
スマート・コレクション制御命令	SMART_CORRECT
条件アセンブル制御命令	IFDEF，IFNDEF，IF，IFN，ELSEIF，ELSEIFN，ELSE，ENDIF

制御命令は，疑似命令と同様に，ソース中に記述します。

また，「表 4―13　制御命令一覧」で示した制御命令のうち，次に示すものは，CX を起動するときにオプションとしてコマンド行でも指定することができます。

表 4―14　制御命令とアセンブラ・オプション

制御命令	オプション
PROCESSOR	-C

4. 3. 2　コンパイル対象品種指定制御命令

コンパイル対象品種指定制御命令は，ソース・モジュール・ファイル中でコンパイル対象品種を指定します。

コンパイル対象品種指定制御命令には，次のものがあります。

表 4―15　コンパイル対象品種指定制御命令

制御命令	概要
PROCESSOR	ソース・モジュール・ファイル中でコンパイル対象品種を指定

PROCESSOR

ソース・モジュール・ファイル中でコンパイル対象品種を指定します。

［指定形式］

```
[△]$[△]PROCESSOR[△]([△]品種名[△])[△][;コメント]
```

［機能］

- PROCESSOR 制御命令は，ソース・モジュール・ファイル中でコンパイル対象品種を指定します。

［用途］

- コンパイル対象品種指定は，ソース・モジュール・ファイル，またはコマンド・ラインのどちらかで必ず指定しなければなりません。
- ソース・モジュール・ファイル中でコンパイル対象品種指定の記述がない場合，コンパイルのたびにコンパイル対象品種を指定しなければなりません。したがって，ソース・モジュール・ファイル中でコンパイル対象品種を指定しておくことにより，コンパイラ起動時の手間を省くことができます。

［詳細説明］

- 指定可能な品種名ついては，各デバイスのユーザーズ・マニュアル，または「デバイス・ファイル　使用上の留意点」を参照してください。
- ソース・モジュール・ファイル中とオプションでの指定が異なる場合，コンパイラはワーニングを出力し，オプションでの指定を優先します。

［使用例］

```
$       PROCESSOR (f3507)
```

4. 3. 3　シンボル制御命令

シンボル制御命令を用いることにより，シンボル・テーブル・エントリの生成，シンボルの定義，およびラベルの示すデータ・サイズの指定ができます。

シンボル制御命令には，次のものがあります。

表 4―16　シンボル制御命令

制御命令	概要
EXT_ENT_SIZE	フラッシュ・テーブル・エントリのサイズ指定
EXT_FUNC	フラッシュ・テーブル・エントリの生成

EXT_ENT_SIZE

フラッシュ・テーブル・エントリのサイズ指定をします。

[指定形式]

```
[△]$[△]EXT_ENT_SIZE[△]サイズ[△][;コメント]
```

[機能]

- フラッシュ・テーブル・エントリ・サイズにオペランドに指定した値を設定します。

[用途]

- オブジェクト・モジュール・ファイルの生成時，フラッシュ・テーブル・エントリ・サイズにオペランドに指定した値を設定します。フラッシュ／外部 ROM 再リンク機能を使用する際に指定します。

[詳細説明]

- 書き換え／取り換え不可能領域（以降，ブート領域）から，可能領域（以降，フラッシュ領域）へ分岐する場合，本制御命令を指定することにより，フラッシュ領域側の指定したアドレスに分岐テーブルが作成され，テーブルを介しての二段分岐が行われます。
- テーブルのエントリ・サイズはデフォルトで 4 バイトであり，jr 命令が生成され，分岐命令から 22 ビットの範囲に分岐可能です。
- テーブルの分岐命令から 22 ビットの範囲を越えるアドレスに分岐する必要がある場合，本制御命令でエントリ・サイズに V850Ex コアの場合には 8 を指定することにより，32 ビットアドレス空間全体へ分岐可能です。
- 本制御命令は，ブート領域側の該当する分岐命令のあるソース・ファイル，およびフラッシュ領域側の該当するラベル定義のあるソース・ファイルに記述する必要があります。
- 本制御命令にて指定するサイズは，ブート領域／フラッシュ領域を含め，全体で唯一の値となります。
  異なるサイズを指定した場合，メッセージを出力し，アセンブルを続行します。
- サイズは，4（デフォルト），8 のいずれかを指定してください。

EXT_FUNC

フラッシュ・テーブル・エントリを生成します。

[指定形式]

```
[△]$[△]EXT_FUNC[△] ラベル名,ID 値[△][;コメント]
```

[機能]

- オペランドに指定したラベル名と，ID 値を持つフラッシュ・テーブル・エントリを生成します。

[用途]

- オブジェクト・モジュール・ファイルの生成時，オペランドに指定したラベル名と，ID 値を持つフラッシュ・テーブル・エントリを生成します。フラッシュ／外 ROM 再リンク機能を使用する際に指定します。

[詳細説明]

- 書き換え／取り換え不可能領域（以降，ブート領域）から，可能領域（以降，フラッシュ領域）へ分岐する場合，本制御命令を指定することにより，フラッシュ領域側の指定したアドレスに分岐テーブルが作成され，テーブルを介しての二段分岐が行われます。
- 本制御命令は，ブート領域側の該当する分岐命令のあるソース・ファイル，およびフラッシュ領域側の該当するラベル定義のあるソース・ファイルに記述する必要があります。
- 同じラベル名で異なる ID 値を指定した場合，メッセージが出力され，アセンブルが中止されます。
- 同じ ID 値で異なるラベル名を指定した場合，メッセージが出力され，アセンブルが中止されます。
- 上記不整合を防止するため，該当するラベル名すべてを 1 ファイルにまとめて記述し，INCLUDE 制御命令を用いて，ブート／フラッシュ両側のソース・ファイルにインクルードすることを推奨します。
- ID 値は正数を指定してください。また，確保される分岐テーブルのサイズは ID 値の最大に依存します。ID 値は詰めて使用することを推奨します。

4. 3. 4　アセンブラ制御命令

アセンブラ制御命令を用いることにより，アセンブラが行う処理を制御できます。

アセンブラ制御命令には，次のものがあります。

表 4―17　アセンブラ制御命令

制御命令	概要
CALLT	コンパイラ予約の制御命令
REG_MODE	レジスタ・モード情報セクションを出力
EP_LABEL	%label による参照をすべて ep オフセット参照として扱う
NO_EP_LABEL	EP_LABEL による指定を解除
NO_MACRO	命令展開の抑制
MACRO	NO_MACRO による指定を解除
DATA	シンボル名の外部データが data 再配置属性，または bss 再配置属性のセクションに割り当てられているとみなした命令展開
SDATA	シンボル名の外部データが sdata 再配置属性，または sbss 再配置属性のセクションに割り当てられているとみなした命令展開の抑制
NOWARNING	警告メッセージの出力を抑止
WARNING	警告メッセージを出力

CALLT

コンパイラ予約の制御命令です。

［指定形式］

```
[△]$[△]CALLT[△][;コメント]
```

［機能］

- コンパイラ予約の制御命令です。

［詳細説明］

- コンパイラが出力したアセンブラ・ソース・ファイル中に存在する場合は，削除しないでください。削除した場合，プロローグ／エピローグ・ランタイムのリンクを確認する機能が動作しません。

REG_MODE

レジスタ・モード情報セクションを出力します。

[指定形式]

```
[△]$[△]REG_MODE[△]レジスタ・モード指定[△][;コメント]
```

[機能]

- アセンブラが生成するオブジェクト・モジュール・ファイル中に，レジスタ・モード情報セクションを出力します。

[詳細説明]

- レジスタ・モード指定には，レジスタ・モード 22 を示す“22”，レジスタ・モード 26 を示す“26”，レジスタ・モード 32 を示す“32”，共通レジスタ・モードを示す“common”のいずれかを指定します。
- レジスタ・モード情報セクションとは，コンパイラが使用する作業用レジスタとレジスタ変数用レジスタの本数情報を保持するもので，本制御命令によりオブジェクト・モジュール・ファイルに設定されます。
- レジスタ・モード 22 を使用する場合，作業用レジスタとレジスタ変数用レジスタはそれぞれ 5 本ずつ，レジスタ・モード 26 を使用する場合はそれぞれ 7 本ずつ，レジスタ・モード 32 を使用する場合はそれぞれ 10 本ずつとなります。
- レジスタ・モード 32 を使用する場合は，アセンブラが生成するオブジェクト・モジュール・ファイル中に，レジスタ・モード情報セクションを出力しません。

EP_LABEL

%label による参照をすべて ep オフセット参照として扱います。

［指定形式］

```
[ △ ]$[ △ ]EP_LABEL[ △ ][; コメント ]
```

［機能］

- それ以降の命令に対し，%label による参照をすべて ep オフセット参照として扱います。
- $EP_LABEL を省略した場合，アセンブラでは，$EP_LABEL を指定したものとみなします。

NO_EP_LABEL

EP_LABEL による指定を解除します。

［指定形式］

```
[ △ ]$[ △ ]NO_EP_LABEL[ △ ][; コメント ]
```

［機能］

- それ以降の命令に対し，EP_LABEL による指定を解除します。
- $NO_EP_LABEL を省略した場合，アセンブラでは，$EP_LABEL を指定したものとみなします。

NO_MACRO

命令展開を行いません。

［指定形式］

```
[△]$[△]NO_MACRO[△][;コメント]
```

［機能］

- それ以降の命令に対し，命令展開を行いません。

MACRO

NO_MACRO による指定を解除します。

［指定形式］

```
[ △ ]$[ △ ]MACRO[ △ ][; コメント ]
```

［機能］

- それ以降の命令に対し，NO_MACRO による指定を解除します。

DATA

シンボル名の外部データが data 再配置属性，または bss 再配置属性のセクションに割り当てられているとみなし，命令展開を行います。

［指定形式］

```
[ △ ]$[ △ ]DATA[ △ ] シンボル名 [ △ ][; コメント ]
```

［機能］

- シンボル名の外部データが，-Xsdata オプションで指定したサイズ値に関わらず，data 再配置属性，または bss 再配置属性のセクションに割り当てられているとみなし，そのデータを参照する命令に対して命令展開を行います。
- #pragma section やセクション・ファイルで“data”を指定した変数を，アセンブリ言語ソース・ファイルで外部参照する場合に利用します。

SDATA

シンボル名の外部データが sdata 再配置属性，または sbss 再配置属性のセクションに割り当てられているとみなし，命令展開を行いません。

［指定形式］

```
[ △ ]$[ △ ]SDATA[ △ ] シンボル名 [ △ ][; コメント ]
```

［機能］

- シンボル名の外部データが，-Xsdata オプションで指定したサイズ値に関わらず，sdata 再配置属性，または sbss 再配置属性のセクションに割り当てられているとみなし，そのデータを参照する命令に対して命令展開を行いません。
- #pragma section やセクションフィルで“sdata”を指定した変数を，アセンブリ言語ソース・ファイルで外部参照する場合に利用します。

NOWARNING

警告メッセージの出力を抑止します。

［指定形式］

```
[△]$[△]NOWARNING[△][;コメント]
```

［機能］

- それ以降の命令に対し，警告メッセージの出力を抑止します。

WARNING

警告メッセージを出力します。

［指定形式］

```
[△]$[△]WARNING[△][;コメント]
```

［機能］

- それ以降の命令に対し，警告メッセージを出力します。

4. 3. 5　ファイル入力制御命令

ファイル入力制御命令を用いることにより，アセンブラ・ソース・ファイル，またはバイナリ・ファイルを，指定した位置に取り込むことができます。

ファイル入力制御命令には，次のものがあります。

表 4―18　ファイル入力制御命令

制御命令	概要
INCLUDE	ほかのソース・モジュール・ファイルの一連のステートメントを引用
BINCLUDE	バイナリ・ファイルの入力

INCLUDE

ほかのソース・モジュール・ファイルの一連のステートメントを引用します。

[指定形式]

```
[△]$[△]INCLUDE[△]([△]ファイル名[△])[△][;コメント]
```

[機能]

- 指定されたファイルの内容を指定された行以降に挿入展開し，アセンブルします。

[用途]

- 複数のソース・モジュール中で共通に記述する比較的大きな一連のステートメントを 1 つのファイル（インクルード・ファイル）としてまとめておきます。
  各ソース・モジュール中で，その一連のステートメントを引用する必要があるとき，INCLUDE 制御命令により，必要とするインクルード・ファイル名を指定します。
  これにより，ソース・モジュールの記述作業を軽減することができます。

[詳細説明]

- INCLUDE 制御命令は，通常のソースにのみ記述することができます。
- オプション（-I）で，インクルード・ファイルのサーチ・パスを指定することができます。
- インクルード・ファイルの読み込みパスのサーチの順番は，次のとおりです。

(1) オプション（-I）で指定されたフォルダ

(2) ソース・ファイルのあるフォルダ

(3)（元の）C ソース・ファイルのあるフォルダ

(4) カレント・フォルダ

- インクルード・ファイルは，ネスティングすることが可能です（ネスティングとは，インクルード・ファイル中で，別のインクルード・ファイルを指定することです）。
- インクルード・ファイルのネスト・レベルの最大値は 4,294,967,294（=0xFFFFFFFE）（理論値）です。ただし，実際には利用可能なメモリ量に依存します。
- インクルード・ファイルがオープンできない場合，メッセージが出力され，アセンブルが中止されます。

- 1 つのインクルード・ファイル中に，セクション定義疑似命令やマクロ定義疑似命令や条件アセンブル制御命令のような開始から終了までのブロックがあるものは，その対応が取れた状態で，閉じなければなりません。対応が取れてないあるいは閉じていない場合はエラーが出力され，閉じられたものとしてアセンブルを継続します。
- アセンブル対象とならないセクション定義疑似命令やマクロ定義疑似命令や条件アセンブル制御命令についてはチェックを行いません。

BINCLUDE

バイナリ・ファイルの入力をします。

［指定形式］

```
[△]$[△]BINCLUDE[△]([△]ファイル名[△])[△][;コメント]
```

［機能］

- オペランドに指定したバイナリ・ファイルの内容を，本制御命令の置かれている位置に置かれたソース・プログラムのアセンブル結果であるとみなして扱います。

［詳細説明］

- オプション（-I）で，インクルード・ファイルのサーチ・パスを指定することができます。
- インクルード・ファイルの読み込みパスのサーチの順番は，次のとおりです。

（1）オプション（-I）で指定されたフォルダ

（2）ソース・ファイルのあるフォルダ

（3）（元の）C ソース・ファイルのあるフォルダ

（4）カレント・フォルダ

- 本制御命令は，バイナリ・ファイルの内容全体を扱います。リロケータブル・ファイルを指定した場合には，ELF フォーマットで構成されたファイル全体を扱います。.text セクション等の内容のみを扱うということではありません。
- 存在しないファイルを指定した場合，メッセージが出力され，アセンブルが中止されます。

4. 3. 6　スマート・コレクション制御命令

スマート・コレクション制御命令を用いることにより，オブジェクト・モジュール・ファイルにて，修正前関数を修正後関数へ変更することを指示することができます。

スマート・コレクション制御命令には，次のものがあります。

表 4―19　スマート・コレクション制御命令

制御命令	概要
SMART_CORRECT	修正前関数を修正後関数へ変更

SMART_CORRECT

オブジェクト・モジュール・ファイルにて，修正前関数を修正後関数へ変更することを指示します。

［指定形式］

```
[△]$[△]SMART_CORRECT △修正前関数の先頭ラベル , 修正前関数の終了ラベル , 修正後関数の先頭ラベル△ [;コメント]
```

［機能］

- オブジェクト・モジュール・ファイルにて，修正前関数を修正後関数へ変更することを指示します。

［詳細説明］

- オブジェクト・モジュール・ファイルにて，修正前関数を修正後関数へ変更することを指示します。
- アセンブラは，オペランドにて指示した修正前関数の先頭部分に 修正後関数へ分岐するための分岐命令を出力します
- 修正後関数（_func）へ分岐するための分岐命令は，次のようになります。

```
jr32    _func
```

- 修正前関数のコード・サイズが，修正後関数の呼び出しを行うのに必要なコード・サイズより小さい場合は，エラー・メッセージを出力し，アセンブルを中止します。

4. 3. 7　条件アセンブル制御命令

条件アセンブル制御命令を用いることにより，条件式の評価結果に従って，アセンブルを行う範囲が制御できます。

条件アセンブル制御命令には，次のものがあります。

表 4―20　条件アセンブル疑似命令

疑似命令	意味
IFDEF	シンボルによる制御（定義されているときアセンブル）
IFNDEF	シンボルによる制御（定義されていないときアセンブル）
IF	絶対値式による制御（真のときアセンブル）
IFN	絶対値式による制御（偽のときアセンブル）
ELSEIF	絶対値式による制御（真のときアセンブル）
ELSEIFN	絶対値式による制御（偽のときアセンブル）
ELSE	絶対値式／シンボルによる制御
ENDIF	制御範囲の終わり

条件アセンブル制御命令のネスト・レベルの最大値は 4,294,967,294（=0xFFFFFFFE）（理論値）です。ただし，実際には利用可能なメモリ量に依存します。

IFDEF

シンボルによる制御（定義されているときアセンブル）をします。

［指定形式］

```
[ △ ]$[ △ ]IFDEF[ △ ] スイッチ名 [ △ ][; コメント ]
```

［機能］

- オペランドで指定したスイッチ名が定義されている場合

（a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

（b）それらの制御命令が存在しない場合は，本制御命令とその疑似命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

- 指定したスイッチ名が定義されていない場合

本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(2) シンボル」を参照してください）と同じです。
- スイッチ名は，予約語以外のユーザ定義シンボルと重複してもかまいません。ただし，スイッチ名同士の重複チェックは行われます。
- スイッチ名は，アセンブル・リスト・ファイルのシンボル・リスト情報，およびクロスリファレンス情報には出力されません。

IFNDEF

シンボルによる制御（定義されていないときアセンブル）をします。

［指定形式］

```
[△]$[△]IFNDEF[△]スイッチ名[△][;コメント]
```

［機能］

- オペランドで指定したスイッチ名が定義されている場合

　本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

- 指定したスイッチ名が定義されていない場合

（a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

（b）それらの制御命令が存在しない場合は，本制御命令と本制御命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(2) シンボル」を参照してください）と同じです。
- スイッチ名は，予約語以外のユーザ定義シンボルと重複してもかまいません。ただし，スイッチ名同士の重複チェックは行われます。
- スイッチ名は，アセンブル・リスト・ファイルのシンボル・リスト情報，およびクロスリファレンス情報には出力されません。

IF

絶対値式による制御（真のときアセンブル）をします。

［指定形式］

[△]$[△]IF[△] *絶対値式* [△][; コメント]

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

（a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

（b）それらの制御命令が存在しない場合は，本制御命令と本制御命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

- 偽（=0）に評価された場合

本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- 絶対式の値は 32 ビットの符号付き整数として評価されます。

IFN

絶対値式による制御（偽のときアセンブル）をします。

［指定形式］

```
[ △ ]$[ △ ]IFN[ △ ] 絶対値式 [ △ ][; コメント ]
```

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

  本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

- 偽（=0）に評価された場合

  （a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

  （b）それらの制御命令が存在しない場合は，本制御命令と本制御命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- 絶対式の値は 32 ビットの符号付き整数として評価されます。

ELSEIF

絶対値式による制御（真のときアセンブル）をします。

［指定形式］

[△]$[△]ELSEIF[△] *絶対値式* [△][; コメント]

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

（a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

（b）それらの制御命令が存在しない場合は，本制御命令とその制御命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

- 偽（=0）に評価された場合

本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- 絶対式の値は 32 ビットの符号付き整数として評価されます。

ELSEIFN

絶対値式による制御（偽のときアセンブル）をします。

［指定形式］

[△]$[△]ELSEIFN[△] *絶対値式* [△][; コメント]

［機能］

- オペランドで指定した絶対値式が真（≠ 0）に評価された場合

  本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，ENSE 制御命令，または ENDIF 制御命令までスキップします。

- 偽（=0）に評価された場合

（a）本制御命令と本制御命令に対応する ELSEIF 制御命令，ELSEIFN 制御命令，または ELSE 制御命令が存在する場合は，本制御命令とその制御命令とで囲まれるブロックをアセンブルします。

（b）それらの制御命令が存在しない場合は，本制御命令とその制御命令に対応する ENDIF 制御命令とで囲まれるブロックをアセンブルします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。
- 絶対式の値は 32 ビットの符号付き整数として評価されます。

ELSE

絶対値式／シンボルによる制御をします。

［指定形式］

[△]$[△]ELSE[△] *絶対値式* [△][; コメント]

［機能］

- IFDEF 制御命令においてスイッチ名が定義されていない場合，IF 制御命令，または ELSEIF 制御命令において絶対値式が偽（=0）に評価された場合，あるいは IFN 制御命令，ELSEIFN 制御命令において絶対値式が真（≠ 0）に評価された場合，本制御命令と本制御命令に対応する ENDIF 制御命令 . とで囲まれる文の並び（ブロック）をアセンブルします。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。

ENDIF

制御範囲の終わりを示します。

［指定形式］

```
[ △ ]$[ △ ]ENDIF[ △ ] 絶対値式 [ △ ][; コメント ]
```

［機能］

条件アセンブル制御命令による制御の範囲の終わりを示します。

［用途］

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

［詳細説明］

- 本制御命令は，通常のソース・プログラム中に記述することができます。

4.4 マクロ

この節では，マクロ機能の使い方について説明します。
プログラムの中で一連の命令群を何回も記述する場合に使用すると，便利な機能です。

4.4.1 概　　要

ソースの中で一連の命令群を何回も記述する場合，マクロ機能を使用すると便利です。

マクロ機能とは，.macro，.endm 疑似命令により，マクロ・ボディとして定義された一連の命令群をマクロ参照している箇所に展開することです。

マクロは，ソースの記述性を向上させるために使用するもので，サブルーチンとは異なります。

マクロとサブルーチンには，それぞれ次のような特徴があります。それぞれ目的に応じて有効に使用してください。

- サブルーチン

  プログラム中で何回も必要となる処理を 1 つのサブルーチンとして記述します。サブルーチンは，アセンブラにより一度だけ機械語に変換されます。

  サブルーチンの参照には，サブルーチン・コール命令（一般にはその前後に引数設定の命令が必要）を記述するだけで済みます。したがって，サブルーチンを活用することにより，プログラムのメモリを効率よく使用することができます。

  プログラム中の一連のまとまった処理をサブルーチン化することにより，プログラムの構造化を図ることができます（プログラムを構造化することにより，プログラム全体の構造が分かりやすくなり，プログラムの設計が容易になります）。

- マクロ

  マクロの基本的な機能は，命令群の置き換えです。

  .macro，.endm 疑似命令によりマクロ・ボディとして定義された一連の命令群が，マクロ参照時にその場所に展開されます。アセンブラは，マクロ参照を検出するとマクロ・ボディを展開し，マクロ・ボディの仮パラメータを参照時の実パラメータに置き換えながら，命令群を機械語に変換します。

  マクロは，パラメータを記述することができます。

  たとえば，処理手順は同じであるがオペランドに記述するデータだけが異なる命令群がある場合，そのデータに仮パラメータを割り当ててマクロを定義します。マクロ参照時には，マクロ名と実パラメータを記述することにより，記述の一部分だけが異なる種々の命令群に対処することができます。

サブルーチン化の手法が，メモリ・サイズの削減やプログラムの構造化を図るために用いられるのに対し，マクロは，コーディングの効率を向上させるために用いられます。

4. 4. 2 マクロの利用

マクロとは，一連の決まった手順パターンを登録し，それを利用して記述するものです。マクロはユーザが定義します。マクロの定義方法は次のように，マクロ本体を“.macro”と“.endm”，で囲む形になります。

```
PUSHMAC .macro  REG     -- 次の 2 つの文がマクロ本体
        add     -4, sp
        st.w    REG, 0x0[sp]
.endm
```

上記を定義したあと，次のように記述した場合，「r19 をスタックに格納する」というコードに置き換えられます。

```
PUSHMAC r19
```

したがって，次のようなコードに展開されます。

```
add     -4, sp
st.w    r19, 0x0[sp]
```

4. 4. 3 マクロ・オペレータ

この項では，マクロ本体内において文字列と文字列を連結するコンカティネート記号“~”，およびダラー記号“$”について説明します。

(1) ~（コンカティネート）

- コンカティネート記号は，マクロ・ボディ内で文字，または文字列と文字，または文字列を連結します。マクロ展開時には，コンカティネート記号の左右の文字，または文字列を連結し，コンカティネート自身は消滅します。
- コンカティネート記号は，マクロ定義時にシンボル中の“~”の前後を仮パラメータ，あるいはローカル・シンボルとして認識することが可能であり，区切り記号として利用することもできます。マクロ展開時にシンボル中の“~”の前後の仮パラメータ，あるいはローカル・シンボルを評価してシンボル中に連結することができます。
- コンカティネート記号としての“~”は，マクロ定義時のみ有効です。
- 文字列中，およびコメント中の“~”は，単なるデータとして扱われます。
- “~”を 2 つ続けると，単なるデータとして扱われます。

例 1.

```
abc     .macro  x
        abc~x:  mov     r10, r20
                sub     def~x, r20
.endm
abc NECEL
```

```
【展開結果】
abcNECEL:   mov     r10, r20
            sub     defNECEL, r20
```

2.

```
abc     .macro  x, xy
        a_~xy:  mov     r10, r20
        a_~x~y: mov     r20, r10
.endm
abc necel, NECEL
```

```
【展開結果】
a_NECEL:    mov     r10, r20
a_necely:   mov     r20, r10
```

3.

```
abc     .macro  x, xy
        ~ab:    mov r10, r20
.endm
abc necel, NECEL
```

```
【展開結果 ]
ab:     mov     r10, r20
```

(2) $（ダラー記号）

アセンブラでは，マクロ呼び出しにおける実引数としてダラー記号“$”を前に置いたシンボルを指定した場合，そのシンボルの値が実引数として指定されたものみなされます。ただし，ダラー記号 $ に続けてシンボル以外の識別名，または未定義シンボル名を指定した場合，メッセージが出力され，アセンブルが中止されます。

例

```
mac1    .macro  x
        mov     x, r10
.endm
.macro  mac2
        .set    value, 10
        mac1    value
        mac1    $value
.endm
mac2
```

```
【展開結果】
.set    value, 10
mov     value, r10
mov     10, r10
```

4.5 予約語

アセンブラには予約語が存在します。予約語をシンボル，ラベル，セクション名，マクロ名に使用することはできません。予約語を指定した場合は，メッセージが出力され，アセンブルが中止されます。予約語は大文字／小文字を区別しません。

予約語は次のとおりです。

- 命令（add，sub，mov など）
- 疑似命令
- 制御命令
- レジスタ名，内部レジスタ名

4.6 アセンブラ生成シンボル

次に，アセンブラが内部処理で利用するために生成するシンボルの一覧を示します。
ただし，予約セクション名は除きます。以下のシンボルと同名のシンボルは利用できません。

表 4―21 アセンブラ生成シンボル

シンボル名	説明
.__multi_*N* .__multi_*N*.end (*N* : 0~4294967294)	マルチコア情報用シンボル
.??00000000 ～ .??FFFFFFFF	.local 疑似命令生成ローカル・シンボル
__s_PPPP_SSSS0000 (PPPP : ファイルのプライマリ名) (SSSS : text セクション名)	アセンブラ・デバッグ情報用シンボル 例 : __s_src_sub_sample0000

4.7 インストラクション

この節では，V850 マイクロコントローラ製品の持つ各種命令機能を説明します。
V850E2V3 のインストラクション・セットをもつデバイスの詳細については，V850E2V3 のインストラクション・セットをもつデバイス製品のユーザーズ・マニュアル，およびアーキテクチャ編を参照してください。

4.7.1 メモリ空間

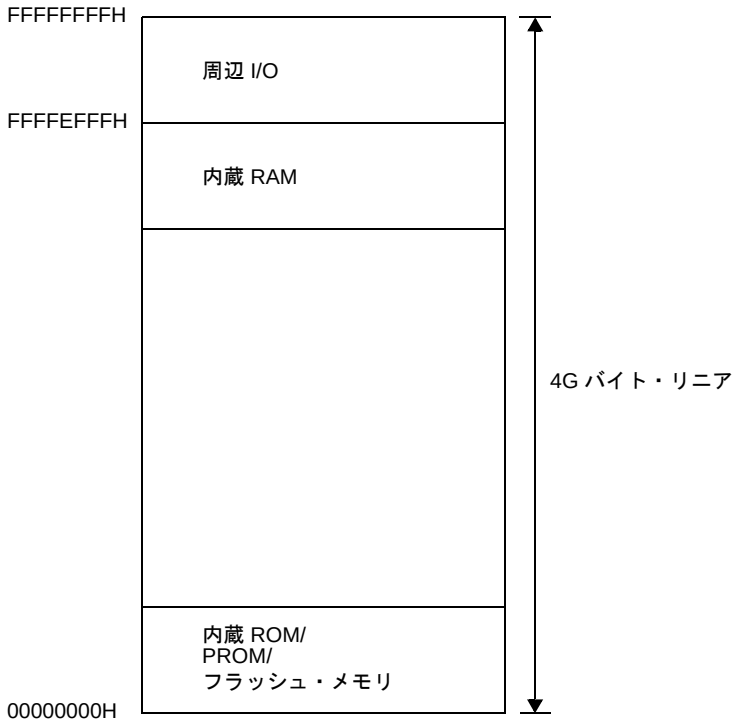
V850 マイクロコントローラは，32 ビット・アーキテクチャであり，オペランド・アドレシングにおいては，最大 4G バイトのリニア・アドレス空間（データ空間）をサポートしています。

一方，命令アドレスのアドレスにおいては，最大 16M バイトのリニア・アドレス空間（プログラム空間）をサポートしています。

以下に，V850 マイクロコントローラのメモリ・マップを示します。

ただし，内蔵 ROM，内蔵 RAM などの容量については，製品ごとにことなるため，詳細は，各製品のユーザーズ・マニュアルを参照してください。

図 4―2　V850 マイクロコントローラのメモリ・マップ



4.7.2 レジスタ

レジスタは，一般のプログラム用として使用するプログラム・レジスタと，実行環境の制御用として使用するシステム・レジスタの 2 種類に大別することができます。レジスタは，どれも 32 ビット幅を持っています。

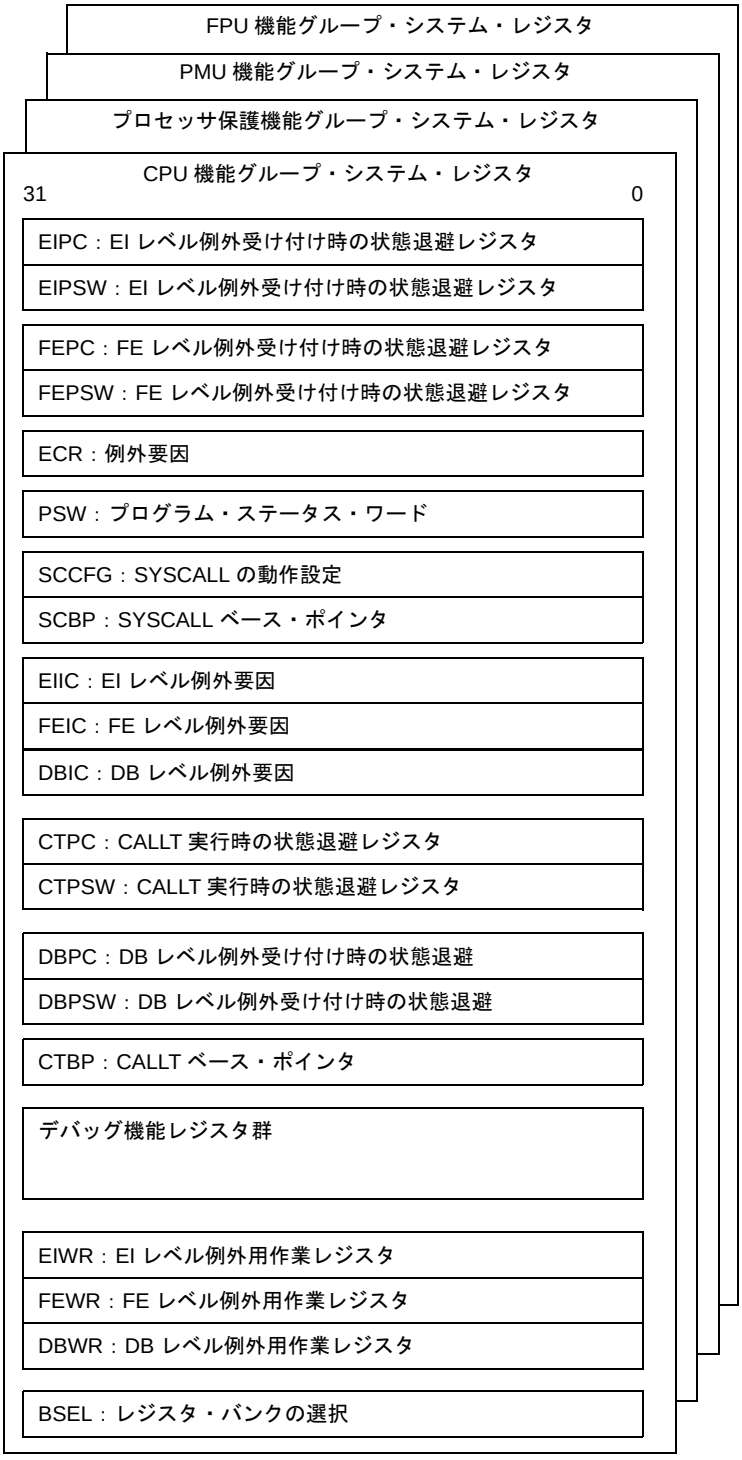
図 4―3 プログラム・レジスタ

31 0

r0：ゼロ・レジスタ
r1：アセンブラ予約レジスタ
r2
r3：スタック・ポインタ（SP）
r4：グローバル・ポインタ（GP）
r5：テキスト・ポインタ（TP）
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15
r16
r17
r18
r19
r20
r21
r22
r23
r24
r25
r26
r27
r28
r29
r30：エレメント・ポインタ（EP）
r31：リンク・ポインタ（LP）

PC：プログラム・カウンタ

図 4―4 システム・レジスタ



詳細については，V850E2V3 のインストラクション・セットをもつデバイス製品のユーザーズ・マニュアル，およびアーキテクチャ編を参照してください。

(1) プログラム・レジスタ

プログラム・レジスタには，汎用レジスタ（r0 ～ r31）とプログラム・カウンタ（PC）があります。

表 4―22 プログラム・レジスタ

名称	用途	動作
r0	ゼロ・レジスタ	常に，0 を保持
r1	アセンブラ予約レジスタ	アドレスを生成する際のワーキング・レジスタ
r2	アドレス／データ変数用レジスタ（使用するリアルタイム OS が r2 を使用していない場合）	
r3	スタック・ポインタ	関数コール時，スタック・フレームを生成する際に使用
r4	グローバル・ポインタ	データ領域のグローバル変数をアクセスする際に使用
r5	テキスト・ポインタ	テキスト領域（プログラム・コードを配置する領域）の先頭を示すレジスタとして使用
r6 ～ r29	アドレス／データ変数用レジスタ	
r30	エレメント・ポインタ	メモリ・アクセス時，アドレスを生成する際のベース・ポインタとして使用
r31	リンク・ポインタ	コンパイラが関数コールをする際に使用
PC	プログラム・カウンタ	プログラム実行中の命令アドレスを保持

(a) 汎用レジスタ：r0 ～ r31

汎用レジスタとして，r0 ～ r31 の 32 本が用意されています。これらのレジスタは，どれもアドレス変数用，またはデータ変数用として利用できます。

ただし，r0 ～ r5，r30 ～ r31 を使用する際には，以下の注意が必要です。

- r0，r30

命令により暗黙的に使用されます。

r0 は常に 0 を保持しているレジスタであり，0 を使用する演算やオフセット 0 のアドレッシングで使用されます。

r30 は SLD 命令，または SST 命令により，メモリをアクセスする際のベース・ポインタとして使用されます。

- r1，r3 ～ r5，r31

アセンブラ，および C コンパイラにより暗黙的に使用されます。

これらのレジスタを使用する際には，レジスタの内容を破壊しないように退避してから使用し，使用後には元の状態に戻す必要があります。

- r2

リアルタイム OS が使用する場合があります。

リアルタイム OS が r2 を使用していない場合は，アドレス変数用，またはデータ変数用として利用できます。

(b) プログラム・カウンタ：PC

プログラム実行中の命令アドレスを保持しています。

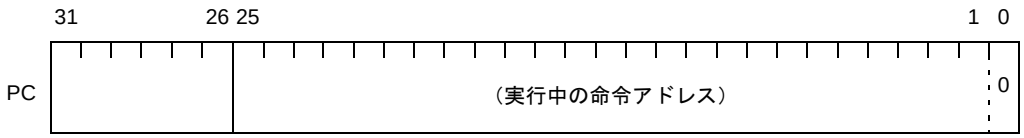
なお，PC の各ビットの意味は，CPU の種類（V850ES，V850E1，V850E2）により異なります。

- V850ES，V850E1

ビット 25 ～ 0 が有効で，ビット 31 ～ 26 は将来の機能拡張のために予約されています（0 に固定）。

ビット 25 からビット 26 へのキャリーが発生しても無視します。また，ビット 0 は 0 に固定されており，奇数番地への分岐はできません。

図 4―5 プログラム・カウンタ【V850ES，V850E1】

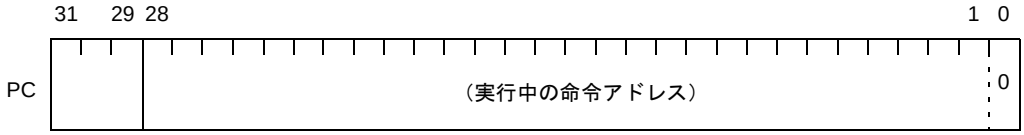


- V850E2

ビット 28 ～ 0 が有効で，ビット 31 ～ 29 は将来の機能拡張のために予約されています（0 に固定）。

ビット 28 からビット 29 へのキャリーが発生しも無視します。また，ビット 0 は 0 に固定されており，奇数番地への分岐はできません。

図 4―6 プログラム・カウンタ【V850E2】



4. 7. 3 アドレッシング

アドレス生成には，分岐を伴う命令が使用する命令アドレス，データをアクセスする命令が使用するオペランド・アドレスの 2 種類があります。

(1) 命令アドレス

命令アドレスは，プログラム・カウンタ（PC）の内容によって決定され，実行した命令のバイト数に応じて自動的にインクリメント（+2）されます。また，分岐命令を実行する際には，次に示すアドレッシングにより，分岐先アドレスを PC にセットします。

(a) レラティブ・アドレッシング（PC 相対）

プログラム・カウンタ（PC）に，命令コードの符号付き 9，22，または 32 ビット・データ（ディスプレースメント：disp）を加算します。このとき，ディスプレースメントは，2 の補数データとして扱われ，それぞれ，ビット 8，21，31 が符号ビット（S）となります。

JR disp22 命令，JARL disp22, reg2 命令，JR disp32 命令，JARL disp32, reg1 命令，Bcnd disp9 命令が本アドレッシングの対象となります。

図 4―7 レラティブ・アドレッシング（JR disp22/JARL disp22, reg2）【V850ES，V850E1】

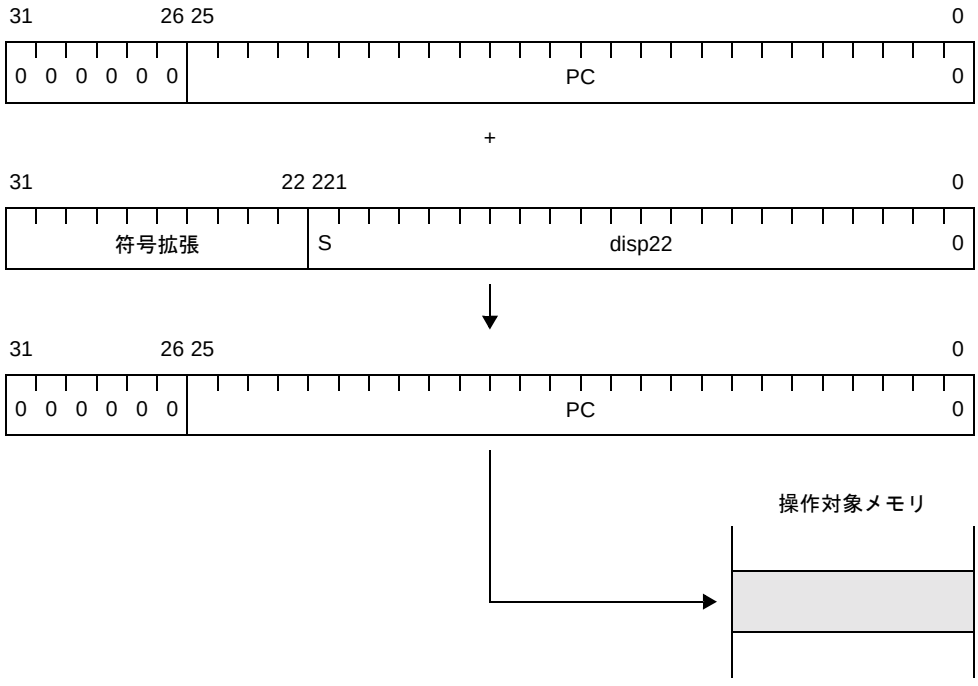


図 4―8 レラティブ・アドレッシング（JR disp22/JARL disp22, reg2）【V850E2】

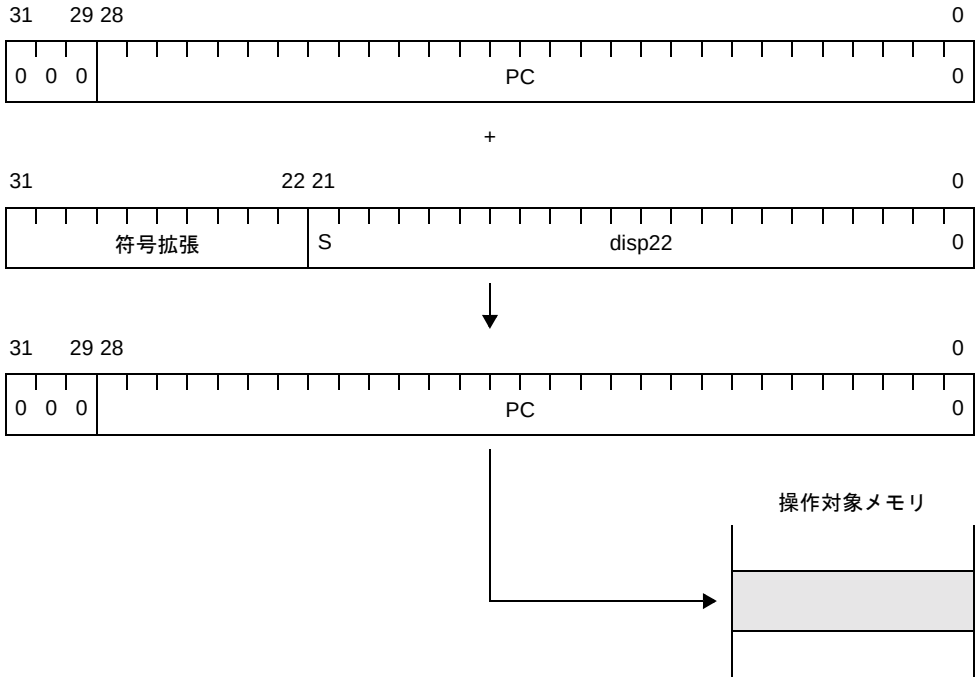


図 4―9 レラティブ・アドレッシング（JR disp32/JARL disp32, reg2）【V850E2】

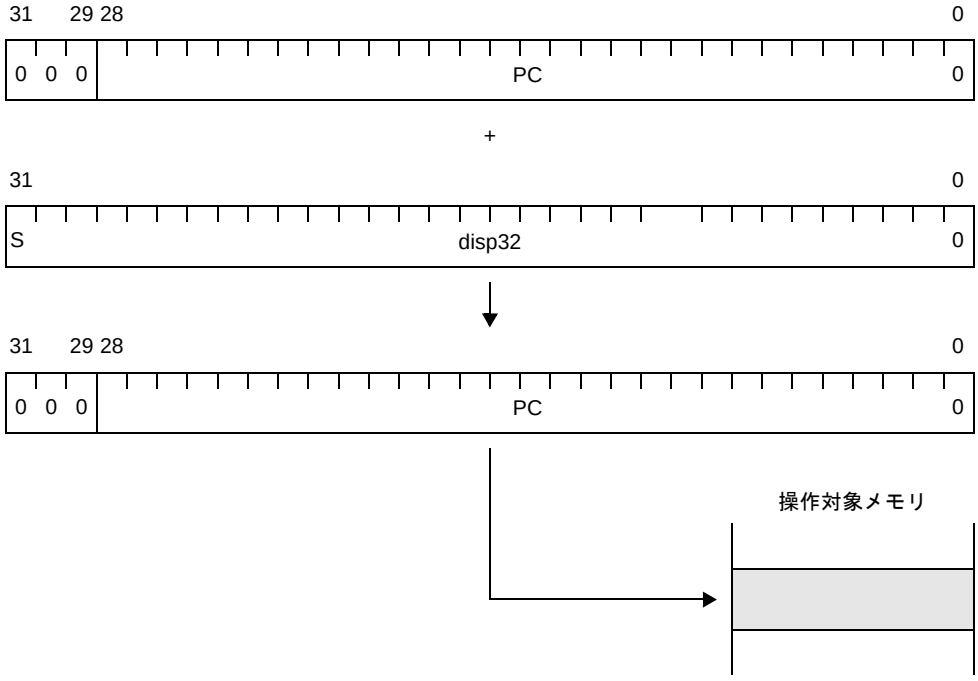


図 4―10 レラティブ・アドレッシング（Bcnd disp9）【V850ES，V850E1】

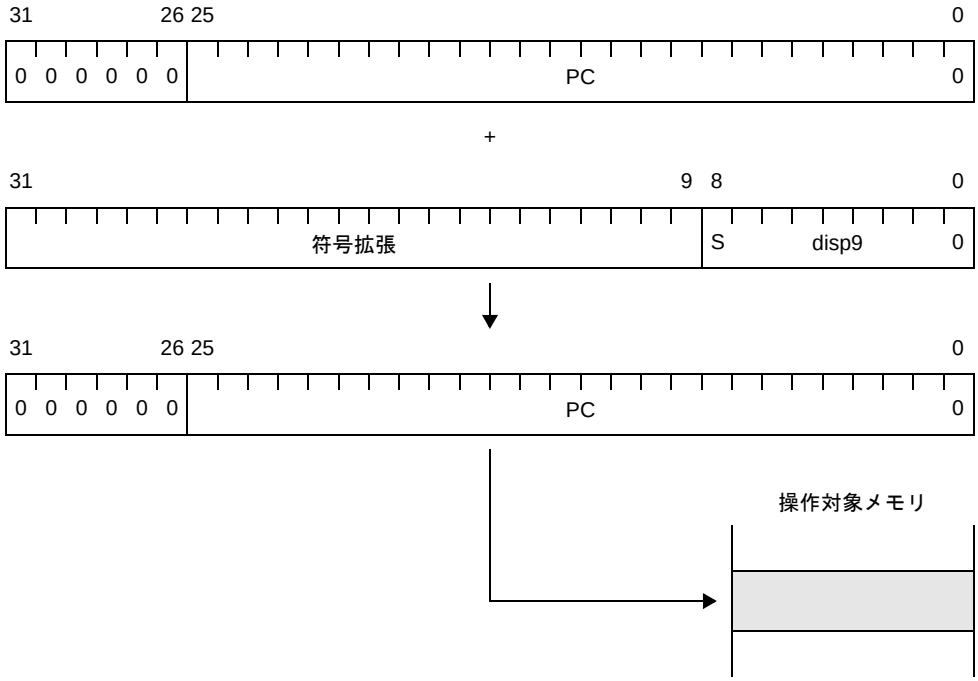
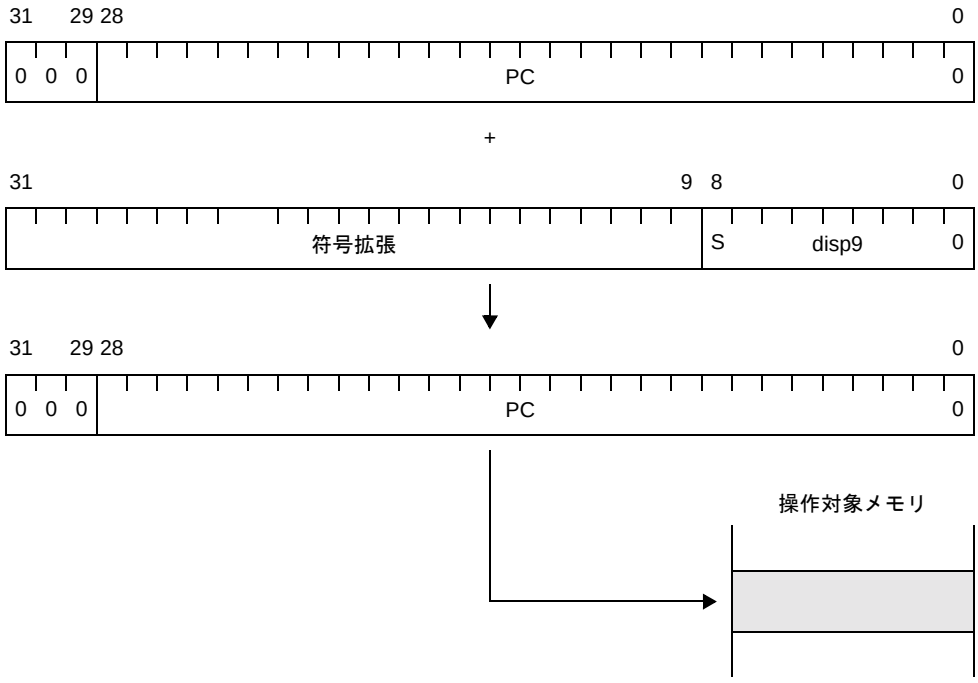


図 4―11 レラティブ・アドレッシング（Bcnd disp9）【V850E2】



(b) レジスタ・アドレッシング（レジスタ間接）

命令によって指定される汎用レジスタ（reg1）の内容をプログラム・カウンタ（PC）に転送します。

JMP [reg1] 命令が本アドレッシングの対象となります。

図 4―12 レジスタ・アドレッシング（JMP [reg1]）【V850ES，V850E1】

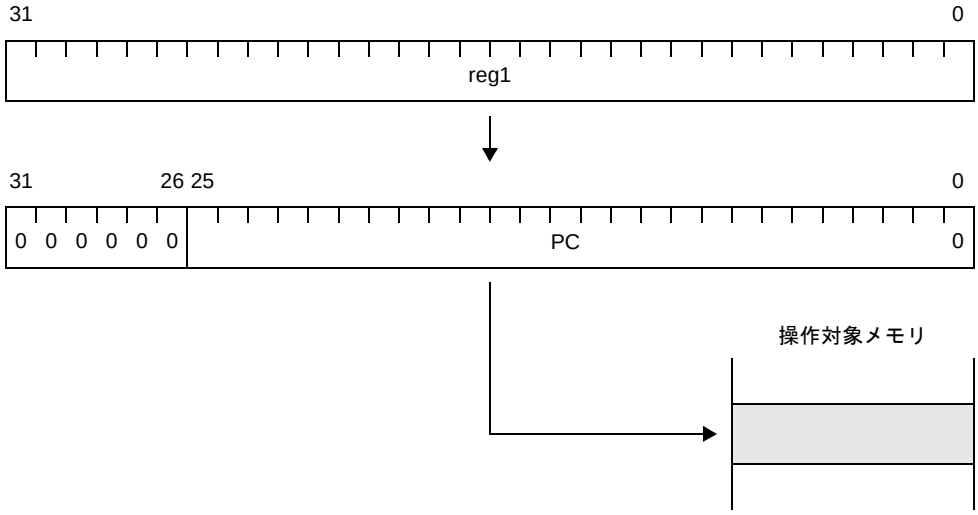
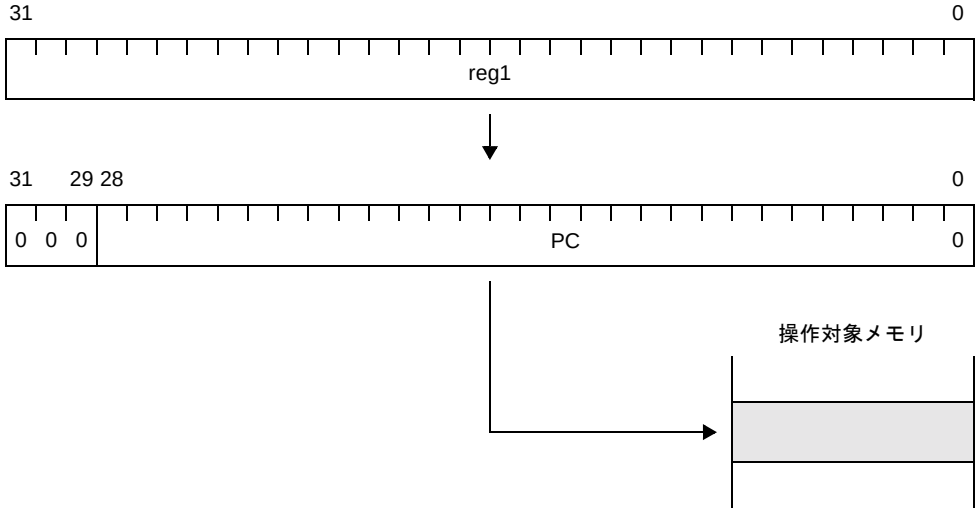


図 4―13 レジスタ・アドレッシング（JMP [reg1]）【V850E2】

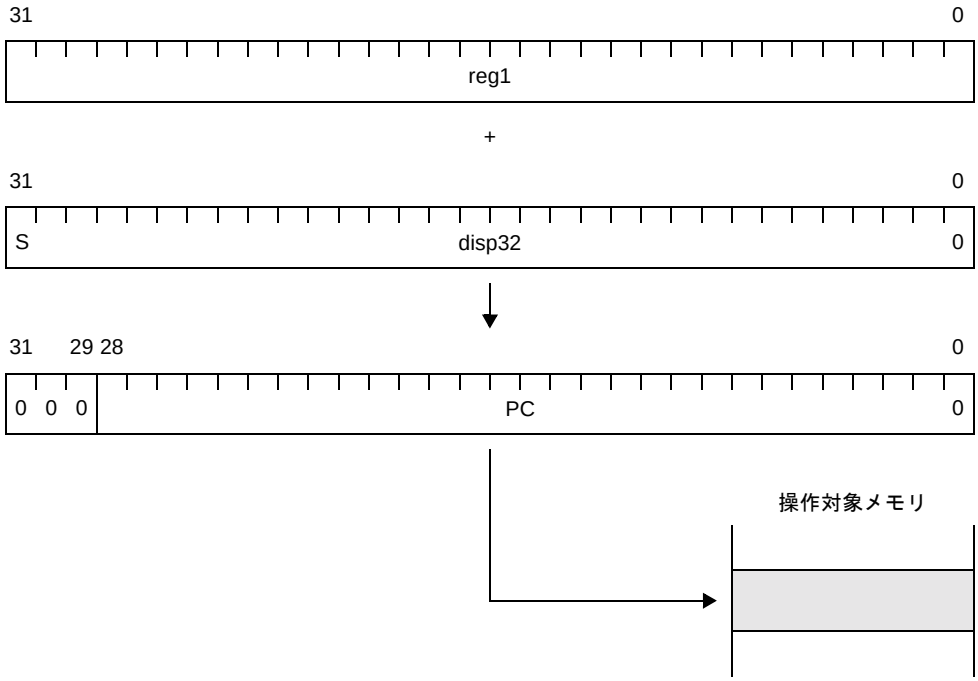


(c) ペースト・アドレッシング

命令によって指定される汎用レジスタ（reg1）に，32 ビット・データ（ディスプレースメント：disp）を加算した内容をプログラム・カウンタ（PC）に転送します。

JMP disp32[reg1] 命令が本アドレッシングの対象となります。

図 4―14 レジスタ・アドレッシング（JMP disp32[reg1]）【V850E2】



(2) オペランド・アドレス

命令を実行する際に対象となるレジスタやメモリなどをアクセスするために，次に示す方法があります。

(a) レジスタ・アドレッシング

汎用レジスタ指定フィールドにより指定される汎用レジスタ，またはシステム・レジスタをオペランドとしてアクセスするアドレッシングです。

オペランドに reg1，reg2, , reg3，または regID を含む命令が本アドレッシングの対象となります。

(b) イミーディエト・アドレッシング

命令コード中に操作対象となる 5 ビット・データ，16 ビット・データを持つアドレッシングです。

オペランドに imm5，imm16，vector，または cccc を含む命令が本アドレッシングの対象となります。

- vector

トラップ・ベクタ（00H ～ 1FH）を指定する 5 ビット・イミーディエトであり，TRAP 命令で使用されるオペランドです。

- cccc

条件コード指定用の 4 ビット・データであり，CMOV，SASF，SETF 命令などで使用されるオペランドです。0 の 1 ビットを上位に付加し，5 ビット・イミーディエトとして命令コード中に割り当てられます。

（c）ペースト・アドレッシング

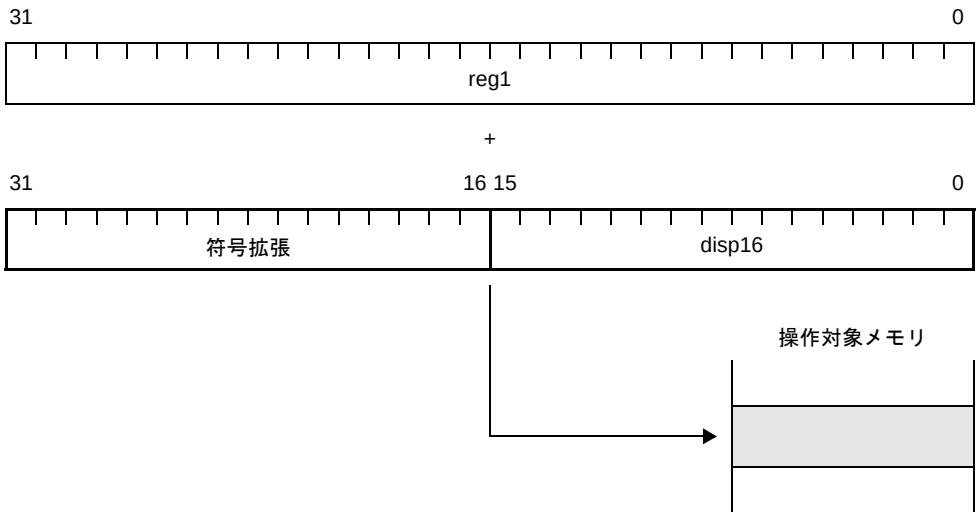
ペースト・アドレッシングには，次に示す 2 種類があります。

- タイプ 1

命令コード中のアドレッシング指定フィールドで指定される汎用レジスタ（reg1）の内容と 16 ビット・ディスプレースメント（disp16）の和がオペランド・アドレスとなって，操作対象となるメモリへのアクセスを行うアドレッシングです。

オペランドに disp16[reg1] を含む命令が本アドレッシングの対象となります。

図 4―15 ペースト・アドレッシング（タイプ 1）【V850ES，V850E1，V850E2】

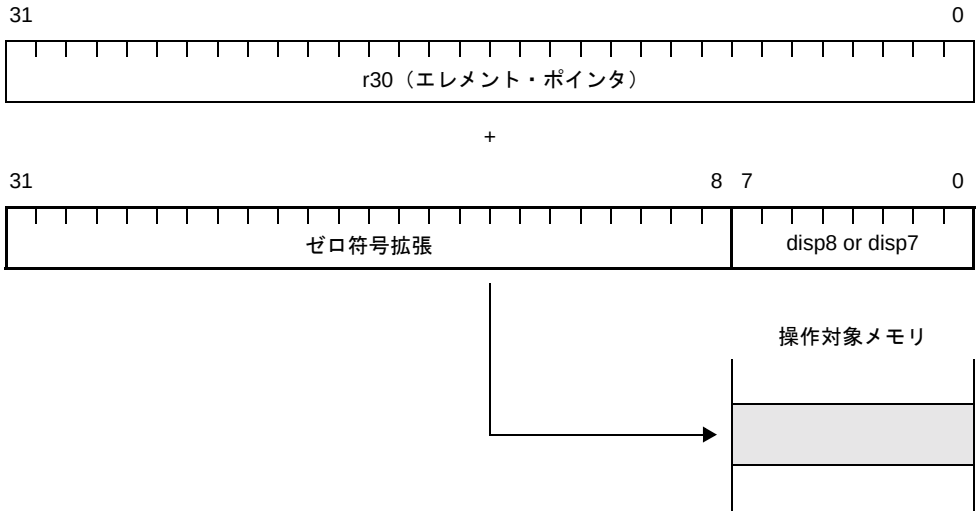


- タイプ 2

エレメント・ポインタ（r30）の内容と，7，または 8 ビット・ディスプレースメント・データ（disp7，disp8）の和がオペランド・アドレスとなって，操作対象となるメモリへのアクセスを行うアドレッシングです。

SLD 命令，SST 命令が本アドレッシングの対象となります。

図 4―16 ベースト・アドレッシング（タイプ 2）【V850ES，V850E1，V850E2】



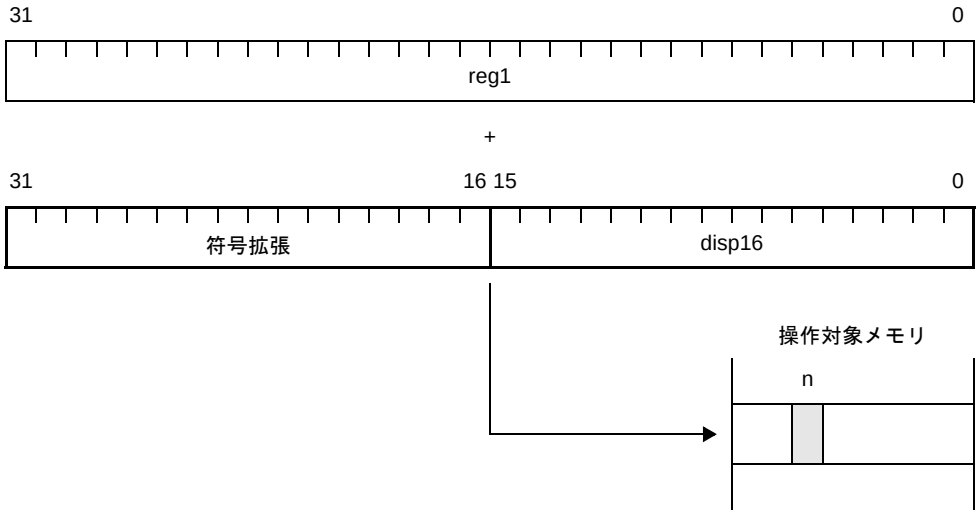
備考 バイト・アクセス = disp7
ハーフワード・アクセス，またはワード・アクセス = disp8

(d) ビット・アドレッシング

汎用レジスタ（reg1）の内容とワード長まで符号拡張した 16 ビット・ディスプレースメント（disp16）の和をオペランド・アドレスとして，操作対象となるメモリ空間の 1 バイト中の 1 ビット（3 ビット・データ「bit#3」で指定）をアクセスするアドレッシングです。

ビット操作命令が本アドレッシングの対象となります。

図 4―17 ビット・アドレッシング【V850ES，V850E1，V850E2】



備考 n：3 ビット・データ（bit#3）で指定されるビット位置（n = 0 ～ 7）

4.7.4 命令セット

この項では，CX がサポートする命令セットについて説明します。

(1) 記号の説明

次表に，以降で用いる記号の意味を示します。

表 4―23 記号の意味

記号	意味
CMD	命令
CMDi	命令（andi，ori，または xori）
reg，reg1，reg2	レジスタ
r0，R0	ゼロ・レジスタ
R1	アセンブラ予約レジスタ（r1）
gp	グローバル・ポインタ（r4）
ep	エレメント・ポインタ（r30）
[reg]	ベース・レジスタ
disp	ディスプレースメント（アドレスからの偏位） 特に記述のない場合 32 ビット幅を持ちます。
disp*n*	*n* ビットのディスプレースメント
imm	イミーディエト（即値） 特に記述のない場合 32 ビット幅を持ちます。
imm*n*	*n* ビットのイミーディエト
bit#3	ビット・ナンバ指定用 3 ビット・データ
cc#3	浮動小数点システム・レジスタ FPSR の CC0 ～ CC7(24 ～ 31 ビット) 指定用 3 ビット・データ
#label	ラベルの絶対アドレス参照
label	ラベルのセクション内オフセット参照，または PC オフセット参照 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，セクション内オフセットの代わりに tp シンボルからのオフセット参照
$label	ラベルの gp オフセット参照
!label	ラベルの絶対アドレス参照（命令展開なし）
%label	ep オフセット参照
HIGHW(*value*)	*value* の上位 16 ビット
LOWW(*value*)	*value* の下位 16 ビット
HIGHW1(*value*)	*value* の上位 16 ビット ＋ *value* のビット番号 15 のビット値[注]
HIGH(*value*)	*value* の下位 16 ビット中の上位 8 ビット
LOW(*value*)	*value* の下位 8 ビット
addr	アドレス

記号	意味
PC	プログラム・カウンタ
PSW	プログラム・ステータス・ワード
regID	システム・レジスタ番号（0 ～ 31）
vector	トラップ・ベクタ（0 ～ 31）
BITIO	周辺 I/O レジスタ（1 ビット操作専用）

注 LSB（Least Significant Bit）はビット番号 0 です。

(2) オペランド

以下に，アセンブラにおけるオペランドの記述形式について説明します。アセンブラでは，命令，および疑似命令に対するオペランドとして，レジスタ，定数，シンボル，ラベル参照，および定数，シンボル，ラベル参照，演算子，かっこで構成した式を指定できます。

(a) レジスタ

アセンブラにおいて指定できるレジスタを次に示します。**注**

r0，zero，r1，r2，hp，r3，sp，r4，gp，r5，tp，r6，r7，r8，r9，r10，r11，r12，r13，r14，r15，r16，r17，r18，r19，r20，r21，r22，r23，r24，r25，r26，r27，r28，r29，r30，ep，r31，lp

注 PSW，およびシステム・レジスタは，ldsr/stsr 命令において，番号で指定します。なお，アセンブラでは，PC をオペランドに指定する方法はありません。

r0 と zero（ゼロ・レジスタ），r2 と hp（ハンドラ・スタック・ポインタ），r3 と sp（スタック・ポインタ），r4 と gp（グローバル・ポインタ），r5 と tp（テキスト・ポインタ），r30 と ep（エレメント・ポインタ），r31 と lp（リンク・ポインタ）は同じレジスタを示します。

(b) r0

r0 は，常に 0 の値を持つレジスタです。したがって，デスティネーション・レジスタとして指定した場合にも，結果の代入は行われません。なお，r0 をデスティネーション・レジスタとして指定した場合，次のメッセージが出力され**注**，アセンブルが続行されます。

注 このメッセージの出力は，アセンブラの起動時に警告メッセージ抑止オプション（-Xwarning_level）を指定することにより抑止できます。

```
mov     0x10, r0
    ↓
W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーションレジスタとしてオペランドに指定されています。
```

(c) r1

アセンブラ予約レジスタ（r1）は，アセンブラにおいて，命令展開を行う際のテンポラリ・レジスタとして用いられるレジスタです。なお，r1 をソース・レジスタ，またはデスティネーション・レジスタとして指定した場合，次のメッセージが出力され[注]，アセンブルが続行されます。

注 このメッセージの出力は，アセンブラの起動時に警告メッセージ抑止オプション（-Xwarning_level）を指定することにより抑止できます。

```
mov     0x10, r1
    ↓
W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーションレジスタとしてオペランドに指定されています。
```

```
mov     r1, r10
    ↓
W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ソースレジスタとしてオペランドに指定されています。
```

(d) 定数

アセンブラでは，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，整定数，および文字定数を用いることができます。また，ld/st 命令，およびビット操作命令のオペランド指定には，各デバイス・ファイルで定義されている「周辺 I/O レジスタ名」も指定できます。これにより，ポート・アドレスに対する入出力を行うことができます。また，.float 疑似命令のオペランド指定には浮動小数点定数を，.set 疑似命令のオペランド指定には文字列定数を用いることができます。

(e) シンボル

アセンブラでは，命令，および疑似命令のオペランド指定で使用可能な絶対値式，または相対値式の構成要素として，シンボルを用いることができます。

(f) ラベル参照

アセンブラでは，次に示した命令／疑似命令のオペランド指定で，使用可能な相対値式の構成要素として，ラベル参照を用いることができます。

- メモリ参照命令（ロード／ストア命令，およびビット操作命令）
- 演算命令（算術演算命令，飽和演算命令，および論理演算命令）
- 分岐命令
- 領域確保疑似命令

アセンブラでは，ラベル参照は参照方法の違い，およびそのラベル参照を用いている命令／疑似命令の違いにより次に示すように異なる意味を持ちます。

表 4―24 ラベル参照

参照方法	用いている命令	意味
#label	メモリ参照命令，演算命令，jmp 命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 32 ビットのアドレスを持ち，mov 命令以外は，必ず 2 命令に展開。
	領域確保疑似命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値
!label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 16 ビットのアドレスを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
	領域確保疑似命令	ラベル label 定義の存在する位置の絶対アドレス（アドレス 0 からのオフセット**注 1**）。 ただし，32 ビットのアドレスを，確保した領域の大きさに準じてマスクした値。
label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 32 ビットのオフセットを持ち，mov 命令以外は，必ず 2 命令に展開。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセット参照。
	jmp 命令を除く分岐命令	ラベル label 定義の存在する位置の PC オフセット（ラベル label 参照を用いている命令の先頭アドレスからのオフセット**注 2**）。
	領域確保疑似命令	ラベル label 定義の存在する位置のセクション内オフセット（ラベル label 定義の存在するセクションの先頭アドレスからのオフセット**注 2**）。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。

参照方法	用いている命令	意味
%label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置の ep オフセット (エレメント・ポインタの示すアドレスからのオフセット)。 16 ビットのオフセットを持ち，16 ビット・ディスプレースメント，またはイミーディエトをもつ命令に指定した場合，命令展開は行われません。 その他の命令に指定した場合，適切な 1 命令に展開されます。 ラベル label の定義したアドレスが，16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
	領域確保疑似命令	ラベル label 定義の存在する位置の ep オフセット (エレメント・ポインタの示すアドレスからのオフセット)。 ただし，32 ビットのオフセットを，確保した領域の大きさに準じてマスクした値。
$label	メモリ参照命令，演算命令	ラベル label 定義の存在する位置の gp オフセット（グローバル・ポインタの指すアドレスからのオフセット**注 2**）。

注 1. リンク後のオブジェクト・モジュール・ファイルにおけるアドレス 0 からのオフセットです。

2. 上記出力セクションが割り当てられたセグメントに対する，テキスト・ポインタ・シンボルの値＋グローバル・ポインタ・シンボルの値の指すアドレスからのオフセットです。

次に，メモリ参照命令，演算命令，分岐命令，および領域確保疑似命令におけるラベル参照の意味を示します。

表 4―25 メモリ参照命令

参照方法	意味
#label[reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。#label[r0] とすることにより絶対アドレスによる参照を指定できます。 [reg] の部分が省略でき，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
label[reg]	ラベル label のセクション内オフセットがディスプレースメントして扱われます。32 ビットの値を持ち，必ず 2 命令に展開されます。reg に対象とするセクションの先頭アドレスを指すレジスタを指定し，label[reg] とすることにより一般的なレジスタ相対の参照が指定できます。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットをディスプレースメントとして扱います。

参照方法	意味
$label[reg]	ラベル label の gp オフセットがディスプレースメントとして扱われます。ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令展開のパターンが変化します注。16 ビットの値を持つ命令展開が行われた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。 $label[gp] とすることにより gp レジスタ相対の参照（gp オフセット参照と呼ぶ）が指定できます。[reg] の部分が省略でき，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
!label[reg]	ラベル label の絶対アドレスがディスプレースメントとして扱われます。16 ビットの値を持ち，命令展開は行われません。ラベル label に定義したアドレスが 16 ビットで表現できない場合，リンク時にエラーになります。!label[r0] とすることにより絶対アドレスによる参照が指定できます。 [reg] の部分が省略でき，省略した場合は [r0] が指定されたものとみなされます。 ただし，#label[reg] 参照とは異なり，命令展開は行われません。
%label[reg]	ラベル label 定義の存在する位置の ep シンボルからのオフセットがディスプレースメントとして扱われます。 16 ビット，または命令によってはそれ以下の値を持ち，その範囲で表現できる値でない場合，リンク時にエラーになります。 [reg] の部分が省略でき，省略した場合，アセンブラでは，[ep] が指定されたものとみなされます。

注 「(h) gp オフセット参照」を参照してください。

表 4―26 演算命令

参照方法	意味
#label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 32 ビットの値を持ち，mov 命令以外は，必ず 2 命令に展開されます。
label	ラベル label のセクション内オフセットがイミーディエトとして扱われます。 32 ビットの値を持ち，mov 命令以外は，必ず 2 命令に展開されます。 ただし，tp シンボルの生成において対象となっているセグメントに割り当てられたセクションに対しては，tp シンボルからのオフセットがイミーディエトとして扱われます。
$label	ラベル label の gp オフセットがイミーディエトとして扱われます。 ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令のパターンが変化します注 1。16 ビットの値を持つ展開をされた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。

参照方法	意味
!label	ラベル label の絶対アドレスがイミーディエトとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令[注 2] に指定した場合，命令展開は行われません。add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開されます。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります。
%label	ラベル label 定義の存在する位置の ep シンボルからのオフセットがイミーディエトとして扱われます。 16 ビットの値を持ち，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令[注 2] に指定した場合，命令展開は行われません。また，この参照方法は，イミーディエトとして 16 ビットの値を指定できるアーキテクチャの演算命令と，add，mov，および mulh 命令のみで指定できます。add，mov，および mulh 命令に指定した場合，適切な 1 命令に展開します。それ以外の命令に指定することはできません。16 ビットで表現できる範囲でない場合，リンク時にエラーになります

注 1. 「(h) gp オフセット参照」を参照してください。

2. イミーディエトとして 16 ビットの値を指定できる命令は，addi，andi，movea，mulhi，ori，satsubi，および xori です。

表 4―27 分岐命令

参照方法	意味
#label	jmp 命令において，ラベル label の絶対アドレスが飛び先アドレスとして扱われます。 32 ビットの値を持ち，必ず 2 命令に展開されます。
label	jmp 命令以外の分岐命令において，ラベル label の PC オフセットがディスプレースメントとして扱われます。 22 ビットの値を持ち，表現できない範囲である場合，リンク時にエラーになります。

表 4―28 領域確保疑似命令

参照方法	意味
#label !label	.db4/.db2/.db 疑似命令において，ラベル label の絶対アドレスを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。

参照方法	意味
label %label	.db4/.db2/.db 疑似命令において，ラベル label 定義の存在する位置の ep シンボルからのオフセットがイミーディエトとして扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。
$label	.db4/.db2/.db 疑似命令において，ラベル label の gp オフセットを値として扱われます。 32 ビットの値を持ちますが，各疑似命令のビット幅に応じてマスクされます。

（g）ep オフセット参照

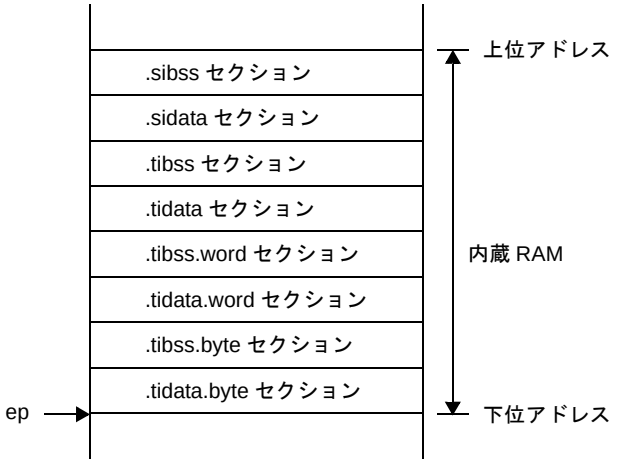
ここでは，ep オフセット参照について説明します。CX では，明示的に内蔵 RAM に置かれるデータについては，基本的に，次のことが想定されています。

エレメント・ポインタ（ep）の指すアドレスからのオフセットによって参照する

なお，内蔵 RAM に置かれるデータは，次の 2 つに分けられます。

- .tidata/.tibss/.tidata.byte/.tibss.byte/.tidata.word/.tibss.word セクション（コード・サイズが小さいメモリ参照命令（sld/sst）で参照するデータ）
- .sidata/.sibss セクション（コード・サイズが大きいメモリ参照命令（ld/st）で参照するデータ）

図 4―18 内蔵 RAM のメモリ配置イメージ



- データの割り当て

内蔵 RAM に置くセクションへのデータの割り当ては，次の方法で行います。

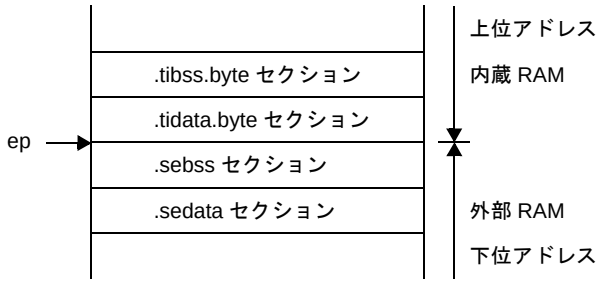
- C 言語を用いてプログラムを作成する場合

“#pragma section”指令により，セクション種別“tidata”，“tidata.byte”，“tidata.word”，または“sidata”を指定してデータを割り当てます。

- アセンブリ言語を用いてプログラムを作成する場合

セクション定義疑似命令により，セクション種別 .tidata，.tibss，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，または .sibss のセクションへデータを割り当てます。なお，上記と同様の方法により，.sedata，または .sebss セクションへデータを割り当てることで，外部 RAM に置かれる一定範囲のデータに対しても，ep オフセット参照を行うことができます。

図 4―19 外部 RAM（.sedata/.sebss セクション）のメモリ配置イメージ



- データの参照

「データの割り当て」に従い，アセンブラでは，次のように機械語命令列が生成されます。

- .tidata，.tibss，.tidata.byte，.tibss.byte，.tidata.word，.tibss.word，.sidata，.sibss，.sedata，または .sebss セクションに割り当てるデータの %label による参照に対しては，ep オフセット参照を行う機械語命令が生成されます。
- 上記以外のセクションに割り当てるデータの %label による参照に対しては，セクション内オフセット参照を行う機械語命令列が生成されます。

例

```
        .dseg   SIDATA
sidata: .db2    0xFFF0
        .dseg   DATA
data:   .db2    0xFFF0
        .cseg   TEXT
        ld.h    %sidata, r20    ; (1)
        ld.h    %data, r20      ; (2)
```

アセンブラでは，%label による参照に対して，（1）の場合，定義したデータが .sidata セクションに配置されているため ep オフセット参照とみなされ，（2）の場合，セクション内オフセット参照とみなさ

れて，機械語の命令列が生成されます。なお，アセンブラでは，データが配置されたセクションが正しいものとして処理が行われます。このため，データの配置に誤りがある場合でも，検出できません。

例

```
.dseg   TEXT
ld.h    %label[ep], r20
```

label を .sidata セクションに配置し，ep オフセット参照を行う命令を記述したが，配置誤りにより .data セクションに配置された場合，ベース・レジスタである ep シンボル値 ＋ label の .data セクション内オフセット値にあるデータがロードされます。

（h）gp オフセット参照

ここでは，gp オフセット参照について説明します。CX では，（.sedata/.sebss セクション以外の）外部 RAM に置かれるデータについては，基本的に次のことが想定されています。

```
グローバル・ポインタ（gp）の指すアドレスからのオフセットによって参照する
```

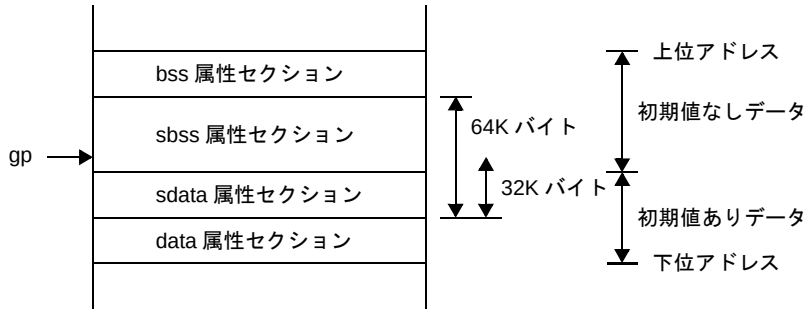
また，C 言語における“#pragma section”指令や，アセンブリ言語におけるセクション定義疑似命令による，内蔵 ROM/ 内蔵 RAM などの r0 相対のメモリ割り当てを行わない場合，すべてのデータが gp オフセット参照となります。

- データの割り当て

  V850 マイクロコントローラの機械語命令におけるメモリ参照命令（ld/st）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，CX ではデータを以下に示した 2 つに分け，前者のデータが sdata 属性セクション，または sbss 属性セクションに，後者のデータが data 属性セクション，または bss 属性セクションに割り当てられます。なお，初期値ありデータは sdata/data 属性セクションに，初期値なしデータは sbss/bss 属性セクションに割り当てられます。CX では，デフォルトで，data/sdata/sbss/bss 属性セクションの順に下位アドレスから割り当てられます。また，グローバル・ポインタ（gp）は sdata 属性セクションの先頭アドレスに 32K バイトを加算したアドレスを指すよう，スタートアップ・ルーチンなどにおいて設定することを想定しています。

  - グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ
  - グローバル・ポインタ（gp）と（複数の命令によって構成される）32 ビットのディスプレースメントを用いて参照することのできるメモリ範囲に割り当てるデータ

図 4―20　gp オフセット参照セクションのメモリ配置イメージ



備考　sdata 属性セクションと sbss 属性セクションをあわせて 64K バイトです。gp は sdata 属性セクションの先頭から 32K バイトの位置です。

したがって，sdata/sbss 属性セクションのデータを参照する場合は，1 命令で行えますが，data/bss 属性セクションのデータを参照する場合は，複数の命令が必要となります。つまり，より多くのデータを sdata/sbss 属性セクションに割り当てたほうが，生成された機械語命令の実行効率，およびオブジェクト効率は向上します。しかし，16 ビットのディスプレースメントで参照できるメモリ範囲の大きさは限られています。

そこで，すべてのデータを sdata/sbss 属性セクションに割り当てられない場合，どのデータを sdata/sbss 属性セクションに割り当てるかを定めなければなりません。

CX では，“できるだけ多くのデータを sdata/sbss 属性セクションに割り当てる”という方針がとられています。デフォルトの処理では，すべてのデータが sdata/sbss 属性セクションに割り当てられますが，割り当てられるデータを選別する場合，次の方法により行います。

- -Xsdata オプションを指定する場合

  C コンパイラ，またはアセンブラ起動時に -Xsdata=*num* オプションを指定することにより，sdata/sbss 属性セクションへ *num* バイト以下のデータが割り当てられます。

- プログラムでデータを割り当てるセクションを指定する場合

  参照頻度の高いデータを，明示的に sdata/sbss 属性セクションへ割り当てます。アセンブリ言語の場合，セクション定義疑似命令で，C 言語の場合，#pragma section 指令で割り当てられます。

- データの参照

前述のデータの割り付けに従い，アセンブラでは，次のようになります。

- sdata/sbss 属性セクションに割り当てるデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
- data/bss 属性セクションに割り当てるデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

例

```
        .dseg   DATA
data:   .db4    0xFFF00010      ; (1)
        .cseg   TEXT
        ld.w    $data[gp], r20  ; (2)
```

アセンブラでは，（1）で定義したデータを gp オフセット参照している（2）の ld.w 命令に対して，次の命令列に等価な機械語の命令列が生成されます。**注**

```
movhi   HIGHW1($data), gp, r1
ld.w    LOWW($data)[r1], r20
```

注 HIGHW1/LOWW に関しては，「(j) HIGH/LOW/HIGHW/LOWW/HIGHW1 演算子について」を参照してください。

なお，アセンブラでは，1 ファイルずつ処理が行われます。このため，指定したファイル内に定義を持つデータに対しては，そのデータがどの属性セクションに割り当てられるデータであるかを判断することができますが，指定したファイル内の定義を持たないデータに対しては判定できません。そこで，アセンブラでは，“前述の割り当ての方針（一定サイズ以下のデータを sdata/sbss 属性セクションに割り当てる）が守られている”ことを想定し，起動時に -Xsdata=*num* オプションが指定された場合，次のように機械語命令が生成されます。

- 指定したファイル内に定義を持たない，*num* バイト以下のデータの gp オフセット参照に対しては，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
- 指定したファイル内に定義を持たない，*num* バイトより大きいデータの gp オフセット参照に対しては，（複数の機械語命令によって構成される）32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成生成されます。

しかし，この判定を行うには，指定したファイル内に定義を持たない gp オフセット参照されているデータのサイズが，判定できなければなりません。そこで，アセンブリ言語を用いてプログラムを作成する場合，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対し，.extern 疑似命令を用いて，サイズを指定してください。

```
.extern data, 4          ; (1)
.cseg   TEXT
ld.w    $data[gp], r20   ; (2)
```

アセンブラでは，起動時に -Xsdata=2 を指定した場合，（1）で宣言したデータを gp オフセット参照している（2）の ld.w 命令に対しては，次の命令列に等価な機械語の命令列が生成されます。**注**

```
movhi   HIGHW1($data), gp, r1
ld.w    LOWW($data)[r1], r20
```

注 HIGHW1/LOWW に関しては，「(j) HIGH/LOW/HIGHW/LOWW/HIGHW1 演算子について」を参照してください。

また，C 言語を用いてプログラム作成する場合，CX に含まれる C コンパイラが，指定したファイル内に定義を持たないデータ（実際には指定したファイル内に定義を持たず gp オフセット参照されているラベル）に対して，自動的に .extern 疑似命令を生成し，サイズを指定するコードを出力します。

備考 アセンブラにおけるデータの gp オフセット参照（具体的には，ラベルの gp オフセット参照を持つ相対値式をディスプレースメントとするメモリ参照命令）の扱いをまとめると，次のようになります。

- そのデータが，指定したファイル内に定義を持つ場合
  - sdata/sbss 属性セクションに割り付けるデータである場合**注**
    16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
  - sdata/sbss 属性セクションに割り付けるデータでない場合
    32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

注 “ラベル±定数式”の形式の相対値式で，定数式の値が 16 ビットの範囲を越える場合は，アセンブラでは，32 ビットのディスプレースメントを用いた参照を行う機械語命令列が生成されます。

- そのデータが，指定したファイル内に定義を持たない場合
  - 起動時に -Xsdata=*num* オプションを指定した場合
    データ（gp オフセット参照されているラベル）に対し，.comm/.extern/.public 疑似命令により，
    【0 以外かつ *num* バイト以下を指定した場合】
    sdata/sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。
    【上記以外の場合】
    sdata/sbss 属性セクションに割り当てるデータでないとみなされ，32 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。

- 起動時に -Xsdata オプションを指定しなかった場合
  sdata/sbss 属性セクションに割り当てるデータであるとみなされ，16 ビットのディスプレースメントを用いた参照を行う機械語命令が生成されます。

(i) マルチコアにおけるラベル参照について

マルチコアの場合では，シングルコアの場合とラベル参照について違いがあるので以下に説明します。

- -Xmulti=pe*n*(n：PE 番号) オプションを指定する場合
  - データおよびコードには，シングルコアと同様の参照でアクセスできます。

- -Xmulti=cmn オプションを指定する場合
  - データおよびコードには gp/ep/tp シンボルからのオフセット参照ではなく，絶対アドレス (アドレス 0 からのオフセット) 参照を使ってアクセスします。
  - gp/ep/tp シンボルからのオフセット参照を行うとエラーとなります。

(j) HIGH/LOW/HIGHW/LOWW/HIGHW1 演算子について

- 32 ビットのディスプレースメントを用いてメモリを参照する場合

V850 マイクロコントローラの機械語のメモリ参照命令（ロード／ストア命令）は，ディスプレースメントに 16 ビットのイミーディエトしかとれません。このため，アセンブラでは，32 ビットのディスプレースメントを用いてメモリを参照する場合，命令展開が行われ，movhi 命令とメモリ参照命令が用いられて，32 ビットのディスプレースメントの上位 16 ビットと，下位 16 ビットから，32 ビットのディスプレースメントが構成されて参照を行う命令列が生成されます。

例

`ld.w 0x18000[r11], r12`	`movhi HIGHW1(0x18000), r11, r1` `ld.w LOWW(0x18000)[r1], r12`

この際，下位 16 ビットをディスプレースメントとして用いる機械語のメモリ参照命令は，指定された 16 ビットのディスプレースメントを符号拡張し，32 ビットの値として扱います。この符号拡張された部分を補正するために，アセンブラでは，movhi 命令を用いて上位 16 ビットのディスプレースメントを構成する際，ただ単に上位 16 ビットのディスプレースメントを構成するのではなく，次のディスプレースメントを構成します。

上位 16 ビット ＋ 下位 16 ビットの最上位ビット（ビット番号 15 のビット）

- HIGHW/LOWW/HIGHW1/HIGH/LOW

次表のようにアセンブラでは，HIGHW，LOWW，HIGHW1，HIGH，および LOW 演算子を用いることにより，32 ビットの値の上位 16 ビット，32 ビットの値の下位 16 ビット，および 32 ビットの値の上位 16 ビット ＋ ビット番号 15 のビット値，16 ビットの値の上位 8 ビット値，16 ビットの値の下位 8 ビット値を指定できます。**注**

注 アセンブラ内部では解決できない場合，この情報はリロケーション情報に反映されリンク・エディタにおいて解決されます。

表 4―29 領域確保疑似命令

HIGHW/LOWW/ HIGHW1/HIGH/LOW	意味
HIGHW (*value*)	*value* の上位 16 ビット
LOWW (*value*)	*value* の下位 16 ビット
HIGHW1 (*value*)	*value* の上位 16 ビット ＋ *value* のビット番号 15 のビット値
HIGH (*value*)	*value* の下位 16 ビット中の上位 8 ビット
LOW (*value*)	*value* の下位 8 ビット

例

```
        .dseg   DATA
L1:
            :
        .cseg   TEXT
        movhi   HIGHW $L1, r0, r10      ; L1 の gp オフセットの値の上位 16 ビットを
                                        ; r10 の上位 16 ビットに格納し，
                                        ; 下位 16 ビットに 0 を格納する。
        movea   LOWW $L1, r0, r10       ; L1 の gp オフセットの値の下位 16 ビットを
                                        ; 符号拡張し r10 に格納する。
            :
        movhi   HIGHW1 $L1, r0, r1      ; L1 の gp オフセットの値を r10 に格納する。
        movea   LOWW $L1, r1, r10
```

4.7.5　命令の説明

アセンブラがサポートするアセンブリ言語命令について，次の形式で説明します。

なお，アセンブラが生成する機械語命令についての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

命令

命令の意味を日本語と英語で示します。

[指定形式]

命令の形式を示します。

[機能]

命令の機能を示します。

[詳細説明]

命令の動作の仕方を示します。

[フラグ]

命令実行によるフラグ（PSW）の動きを示します。

ただし，ビット操作命令（set1，clr1，not1）では，実行前のフラグの値を示しています。

なお，表中の“—”は，フラグの値が変化しないことを示します。

[注意事項]

命令における注意事項を示します。

4. 7. 6 ロード／ストア命令

この項では，ロード／ストア命令について説明します。次に，この項において説明する命令を示します。

表 4―30 ロード／ストア命令

命令		意味
ld	ld.b	バイト・データのロード
	ld.h	ハーフワード・データのロード
	ld.w	ワード・データのロード
	ld.bu	符号なしバイト・データのロード
	ld.hu	符号なしハーフワード・データのロード
sld	sld.b	バイト・データのロード（ショート・フォーマット）
	sld.h	ハーフワード・データのロード（ショート・フォーマット）
	sld.w	ワード・データのロード（ショート・フォーマット）
	sld.bu	符号なしバイト・データのロード（ショート・フォーマット）
	sld.hu	符号なしハーフワード・データのロード（ショート・フォーマット）
ld23	ld23.b	バイト・データのロード
	ld23.h	ハーフワード・データのロード
	ld23.w	ワード・データのロード
	ld23.bu	符号なしバイト・データのロード
	ld23.hu	符号なしハーフワード・データのロード
st	st.b	バイト・データのストア
	st.h	ハーフワード・データのストア
	st.w	ワード・データのストア
sst	sst.b	バイト・データのストア（ショート・フォーマット）
	sst.h	ハーフワード・データのストア（ショート・フォーマット）
	sst.w	ワード・データのストア（ショート・フォーマット）
st23	st.b	バイト・データのストア
	st.h	ハーフワード・データのストア
	st.w	ワード・データのストア

ld

データのロードを行います。（Load）

［指定形式］

- ld.b disp[reg1], reg2
- ld.h disp[reg1], reg2
- ld.w disp[reg1], reg2
- ld.bu disp[reg1], reg2
- ld.hu disp[reg1], reg2

disp に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

ld.b，ld.h，ld.w，ld.bu，ld.hu 命令は，第 1 オペランドに指定したアドレスから 1 バイト分，1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したレジスタにロードします。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の ld 命令[注]が 1 つ生成されます。なお，例中の“ld”は ld.b，ld.h，ld.w，ld.bu，ld.hu のいずれかになります。

（a）-32768 ～ +32767 の範囲の絶対値式

ld disp16[reg1], reg2	ld disp16[reg1], reg2

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

ld $label[reg1], reg2	ld $label[reg1], reg2

（c）!label，または %label を持つ相対値式

ld !label[reg1], reg2	ld !label[reg1], reg2
ld %label[reg1], reg2	ld %label[reg1], reg2

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

ld disp16[reg1], reg2	ld disp16[reg1], reg2

（e）デバイス・ファイルで定義された内部レジスタ名

ld *レジスタ名*[reg1], reg2	ld *レジスタ名*[reg1], reg2

注 機械語命令の ld 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

ld disp[reg1], reg2	movhi HIGHW1(disp), reg1, r1 ld LOWW(disp)[r1], reg2

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

ld #label[reg1], reg2	movhi HIGHW1(#label), reg1, r1 ld LOWW(#label)[r1], reg2
ld label[reg1], reg2	movhi HIGHW1(label), reg1, r1 ld LOWW(label)[r1], reg2
ld $label[reg1], reg2	movhi HIGHW1($label), reg1, r1 ld LOWW($label)[r1], reg2

- disp を省略した場合，アセンブラでは，0 を指定したものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラは [r0] が指定されたものとみなされます。

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- ld.b，および ld.h は，1 バイト分，および 1 ハーフワード分のデータを符号拡張し，1 ワード分のデータとしてレジスタにロードされます。
- ld.h，ld.w，ld.hu の disp に 2 の倍数でない値を指定した場合，アセンブラ，またはリンク・エディタでは，disp に対して，2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

- ld.bu，および ld.hu 命令に対し，第 2 オペランドに r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550240 : V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

sld

データのロードを行います。（Short format Load）

［指定形式］

- sld.b disp7[ep], reg2
- sld.h disp8[ep], reg2
- sld.w disp8[ep], reg2
- sld.bu disp4[ep], reg2
- sld.hu disp5[ep], reg2

disp4/5/7/8 に指定できるものを次に示します。

- sld.b では 7 ビット，sld.h，および sld.w では 8 ビット，sld.bu では 4 ビット，sld.hu では 5 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

sld.b，sld.h，sld.w，sld.bu，sld.hu 命令は，第 1 オペランドに指定したディスプレースメントとレジスタ ep の内容を加算して得たアドレスから，1 バイト分，1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したレジスタにロードします。

［詳細説明］

アセンブラでは，機械語命令の sld 命令が 1 つ生成されます。ベース・レジスタの指定“[ep]”は省略できます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- sld.b，および sld.h は，1 バイト分，および 1 ハーフワード分のデータを符号拡張し，1 ワードのデータとしてレジスタに格納されます。
- sld.h/sld.hu の disp8/disp5 に 2 の倍数でない値を指定した場合，および sld.w の disp8 に 4 の倍数でない値を指定した場合，アセンブラでは，disp8/disp5 に対して，それぞれ 2 の倍数，4 の倍数にアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

- sld.b の disp7 に 127 を越える値を指定した場合，sld.h，sld.w の disp8 に 255 を越える値を指定した場合，sld.bu の disp4 に 16 を越える値を指定した場合，および sld.hu の disp5 に 32 を越える値を指定した場合，次のメッセージが出力され，disp7，disp8，disp4，disp5 をそれぞれ 0x7F，0xFF，0xF，0x1F でマスクしたコードが生成されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。

- sld.bu，および sld.hu の第 2 オペランド（reg2）に r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550240 : V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

ld23

データのロードを行います。（Load）

［指定形式］

- ld23.b disp23[reg1], reg2
- ld23.h disp23[reg1], reg2
- ld23.w disp23[reg1], reg2
- ld23.bu disp23[reg1], reg2
- ld23.hu disp23[reg1], reg2

disp23 に指定可能なものを以下に示します。

- 23 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

ld23.b，ld23.h，ld23.w，ld23.bu，ld23.hu 命令は，第 1 オペランドに指定したアドレスから 1 バイト分，1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したレジスタにロードします。

［詳細説明］

- disp23 に次のものを指定した場合，アセンブラでは，機械語命令の ld23 命令[注]が 1 つ生成されます。なお，例中の“ld23”は ld23.b，ld23.h，ld23.w，ld23.bu，ld23.hu のいずれかになります。

（a）-4194304 ～ +4194303 の範囲の絶対値式

ld23 disp23[reg1], reg2	ld23 disp23[reg1], reg2

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

ld23 $label[reg1], reg2	ld23 $label[reg1], reg2

（c）!label，または %label を持つ相対値式

ld23 !label[reg1], reg2	ld23 !label[reg1], reg2
ld23 %label[reg1], reg2	ld23 %label[reg1], reg2

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

ld23 disp23[reg1], reg2	ld23 disp23[reg1], reg2

(e) デバイス・ファイルで定義された内部レジスタ名

ld23 *レジスタ名*[reg1], reg2	ld23 *レジスタ名*[reg1], reg2

(f) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

ld23 #label[reg1], reg2	ld23 #label[reg1], reg2
ld23 label[reg1], reg2	ld23 label[reg1], reg2
ld23 $label[reg1], reg2	ld23 $label[reg1], reg2

注 機械語命令の ld23 命令は，ディスプレースメントに -4194304 ～ +4194303（0xFFC00000 ～ 0x3FFFFF）の範囲のイミーディエトをとります。

- disp23 を省略した場合，アセンブラでは，0 を指定したものとみなされます。
- disp23 に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp23 に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合アセンブラでは，[gp] が指定されたものとみなされます。
- disp23 にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラは [r0] が指定されたものとみなされます。

[フラグ]

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- ld23.b，および ld23.h は，1 バイト分，および 1 ハーフワード分のデータを符号拡張し，1 ワード分のデータとしてレジスタにロードされます。
- ld23.h，ld23.w，ld23.hu の disp23 に 2 の倍数でない値を指定した場合，アセンブラ，またはリンク・エディタでは，disp23 に対して，2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

- ld23.bu，および ld23.hu 命令に対し，第 2 オペランドに r0 を指定した場合，メッセージが出力され，アセンブルが中止されます。

E0550240 : V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

st

データのストアを行います。（Store）

［指定形式］

- st.b reg2, disp[reg1]
- st.h reg2, disp[reg1]
- st.w reg2, disp[reg1]

disp に指定できるものを，次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

st.b，st.h，st.w 命令は，第 1 オペランドに指定したレジスタの下位 1 バイト分，下位 1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したアドレスにストアします。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の st 命令[注]が 1 つ生成されます。なお，例中の“st”は st.b，st.h，st.w のいずれかになります。

（a）-32768 ～ +32767 の範囲の絶対値式

st reg2, disp16[reg1]	st reg2, disp16[reg1]

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

st reg2, $label[reg1]	st reg2, $label[reg1]

（c）!label，または %label を持つ相対値式

st reg2, !label[reg1]	st reg2, !label[reg1]
st reg2, %label[reg1]	st reg2, %label[reg1]

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

st reg2, disp16[reg1]	st reg2, disp16[reg1]

（e）デバイス・ファイルで定義された内部レジスタ名

st reg2, *レジスタ名* [reg1]	st reg2, *レジスタ名* [reg1]

注 機械語命令の st 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

st reg2, disp[reg1], reg2	movhi HIGHW1(disp), reg1, r1 st reg2, LOWW(disp)[r1], reg2

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

st reg2, #label[reg1]	movhi HIGHW1(#label), reg1, r1 st reg2, LOWW(#label)[r1]
st reg2, label[reg1]	movhi HIGHW1(label), reg1, r1 st reg2, LOWW(label)[r1]
st reg2, $label[reg1]	movhi HIGHW1($label), reg1, r1 st reg2, LOWW($label)[r1]

- disp を省略した場合，アセンブラでは，0 を指定したものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－

［注意事項］

- st.h，および st.w の disp に 2 の倍数でない値を指定した場合，アセンブラでは，disp に対して 2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

sst

データのストアを行います。（Short format Store）

［指定形式］

- sst.b reg2, disp7[ep]
- sst.h reg2, disp8[ep]
- sst.w reg2, disp8[ep]

disp7/8 に指定できるものを次に示します。

- sst.b では 7 ビット，sst.h，および sst.w では 8 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

sst.b，sst.h，sst.w 命令は，第 1 オペランドに指定したレジスタの下位 1 バイト分，下位 1 ハーフワード分，および 1 ワード分のデータを，第 2 オペランドに指定したディスプレースメントとしてレジスタ ep の内容を加算して得たアドレスに格納します。

［詳細説明］

アセンブラでは，機械語命令の sst 命令が 1 つ生成されます。ベース・レジスタの指定“[ep]”は省略できます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- sst.h の disp8 に 2 の倍数でない値を指定した場合，および sst.w の disp8 に 4 の倍数でない値を指定した場合，アセンブラでは，disp8 に対して，それぞれ 2 の倍数，4 の倍数にアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

- sst.b の disp7 に 127 を越える値を指定した場合，および sst.h，sst.w の disp8 に 255 を越える値を指定した場合，次のメッセージが出力され，disp7，disp8 をそれぞれ 0x7F，0xFF でマスクしたコードが生成されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。

st23

データのストアを行います。（Store）

［指定形式］

- st23.b reg2, disp23[reg1]
- st23.h reg2, disp23[reg1]
- st23.w reg2, disp23[reg1]

disp23 に指定できるものを，次に示します。

- 23 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

st23.b，st23.h，st23.w 命令は，第 1 オペランドに指定したレジスタの下位 1 バイト分，下位 1 ハーフワード分，および 1 ワード分のデータを取り込み，第 2 オペランドに指定したアドレスにストアします。

［詳細説明］

- disp23 に次のものを指定した場合，アセンブラでは，機械語命令の st23 命令[注]が 1 つ生成されます。なお，例中の“st”は st23.b，st23.h，st23.w のいずれかになります。

（a）-4194304 ～ +4194303 の範囲の絶対値式

st23 reg2, disp23[reg1]	s23 reg2, disp23[reg1]

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

st23 reg2, $label[reg1]	st23 reg2, $label[reg1]

（c）!label，または %label を持つ相対値式

st23 reg2, !label[reg1]	st23 reg2, !label[reg1]
st23 reg2, %label[reg1]	st23 reg2, %label[reg1]

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

st23 reg2, disp23[reg1]	st23 reg2, disp23[reg1]

(e) デバイス・ファイルで定義された内部レジスタ名

st23 reg2, *レジスタ名* [reg1]	st23 reg2, *レジスタ名* [reg1]

(f) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

st23 reg2, #label[reg1]	st23 reg2, #label[reg1]
st23 reg2, label[reg1]	st23 reg2, label[reg1]
st23 reg2, $label[reg1]	st23 reg2, $label[reg1]

注 機械語命令の st 命令は，ディスプレースメントに -4194304 ～ +4194303（0xFFC00000 ～ 0x3FFFFF）の範囲のイミーディエトをとります。

- disp23 を省略した場合，アセンブラでは，0 を指定したものとみなされます。
- disp23 に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp23 に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp23 にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

[フラグ]

CY	–
OV	–
S	–
Z	–
SAT	–

[注意事項]

- st23.h，および st23.w の disp23 に 2 の倍数でない値を指定した場合，アセンブラでは，disp23 に対して 2 でアライメントしたコードが生成され，次のいずれかのメッセージが出力されます。

W0550010 : ディスプレースメントの値が指定可能な値の範囲を超えています。
W0560413 : ロード／ストア系のリロケーション・エントリ（ファイル :*file*, セクション :*section*, オフセット :*offset*, リロケーション・タイプ :*relocation type*）によってリロケートされた値 (*value*) が奇数になっています。

4. 7. 7 算術演算命令

この項では，算術演算命令について説明します。次に，この項において説明する命令を示します。

表 4―31 算術演算命令

命令	意味
add	加算
addi	加算（イミーディエト）
adf	条件付き加算【V850E2】
sub	減算
subr	逆減算
sbf	条件付き減算【V850E2】
mulh	符号付き乗算（ハーフワード）
mulhi	符号付き乗算（ハーフワード・イミーディエト）
mul	符号付き乗算（ワード）
mulu	符号なし乗算
mac	符号付きワード・データの加算付き乗算【V850E2】
macu	符号なしワード・データの加算付き乗算【V850E2】
divh	符号付き除算（ハーフワード）
div	符号付き除算（ワード）
divhu	符号なし除算（ハーフワード）
divu	符号なし除算（ワード）
divq	符号付きワード・データの除算（可変ステップ）【V850E2V3】
divqu	符号なしワード・データの除算（可変ステップ）【V850E2V3】
cmp	比較
mov	データの転送
movea	実行アドレスの転送
movhi	上位ハーフワードの転送
mov32	32 ビット・データの転送
cmov	フラグ条件付きデータの転送
setf	フラグ条件の設定
sasf	論理左シフト付きフラグ条件の設定

V850E2V3 のインストラクション・セットをもつデバイスの命令の詳細については，V850E2V3 のインストラクション・セットをもつデバイス製品のユーザーズ・マニュアル，およびアーキテクチャ編を参照してください。

add

加算を行います。（Add）

［指定形式］

- add reg1, reg2
- add imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“add reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。

-“add imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の値と第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“add reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の add 命令が 1 つ生成されます。

-“add imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の add 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`add imm5, reg`	`add imm5, reg`

注 機械語命令の add 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

-“add imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`add imm16, reg`	`addi imm16, reg, reg`

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

add imm, reg	movhi HIGHW(imm), r0, r1 add r1, reg

上記以外の場合

add imm, reg	mov imm, r1 add r1, reg

（c）!label，または %label を持つ相対値式，および sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

add !label, reg	addi !label, reg, reg
add %label, reg	addi %label, reg, reg
add $label, reg	addi $label, reg, reg

（d）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

add #label, reg	mov #label, r1 add r1, reg
add label, reg	mov label, r1 add r1, reg
add $label, reg	mov $label, r1 add r1, reg

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

addi

加算を行います。（Add Immediate）

［指定形式］

- addi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値と第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の addi 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

addi imm16, reg1, reg2	addi imm16, reg1, reg2

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

addi $label, reg1, reg2	addi $label, reg1, reg2

（c）!label，または %label を持つ相対値式

addi !label, reg1, reg2	addi !label, reg1, reg2
addi %label, reg1, reg2	addi %label, reg1, reg2

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

addi imm16, reg1, reg2	addi imm16, reg1, reg2

注 機械語命令の addi 命令は，第 1 オペランドに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

addi imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 add reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

addi imm, reg1, r0	movhi HIGHW(imm), r0, r1 add reg1, r1

上記以外の場合

addi imm, reg1, reg2	mov imm, reg2 add reg1, reg2

上記以外，ただし reg2 が r0 の場合

addi imm, reg1, r0	mov imm, r1 add reg1, r1

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

addi #label, reg1, r0	mov #label, r1 add reg1, r1
addi label, reg1, r0	mov label, r1 add reg1, r1
addi $label, reg1, r0	mov $label, r1 add reg1, r1

上記以外の場合

addi #label, reg1, reg2	mov #label, reg2 add reg1, reg2
addi label, reg1, reg2	mov label, reg2 add reg1, reg2
addi $label, reg1, reg2	mov $label, reg2 add reg1, reg2

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

adf

条件付き加算を行います。（Add on Condition Flag）【V850E2】

［指定形式］

- adf　 imm4, reg1, reg2, reg3
- adf*cnd*　reg1, reg2, reg3

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式（0xD は指定できません）

［機能］

-“adf imm4, reg1, reg2, reg3”の形式

　第 1 オペランドに指定した絶対値式の下位 4 ビットの値で示されるフラグ状態（「表 4―32　adfcnd 命令」を参照）と現在のフラグ状態を比較します。

　値が一致した場合には，第 3 オペランドに指定したレジスタのワード・データに，第 2 オペランドに指定したレジスタのワード・データを加算し，さらに 1 を加算します。結果を第 4 オペランドに指定したレジスタに格納します。

　値が一致しなかった場合には，第 3 オペランドに指定したレジスタのワード・データに，第 2 オペランドに指定したレジスタのワード・データを加算します。その結果を第 4 オペランドに指定したレジスタに格納します。

-“adf*cnd* reg1, reg2, reg3”の形式

　cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。

　値が一致した場合に第 2 オペランドに指定したレジスタのワード・データに，第 1 オペランドに指定したレジスタのワード・データを加算し，さらに 1 を加算します。その結果を第 3 オペランドに指定したレジスタに格納します。

　値が一致しなかった場合には，第 2 オペランドに指定したレジスタのワード・データに，第 1 オペランドに指定したレジスタのワード・データを加算します。その結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- adf 命令に対し，アセンブラでは，機械語命令の adf 命令が 1 つ生成されます。
- adf*cnd* 命令に対し，アセンブラでは，対応する adf 命令が生成され（「表 4―32　adfcnd 命令」を参照），“adf imm4, reg1, reg2, reg3”の形式に展開されます。

表 4―32　adf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
adfgt	((S xor OV) or Z) = 0	Greater than（signed）	adf 0xF

命令	フラグ状態	フラグ状態の意味	命令展開
adfge	（S xor OV）= 0	Greater than or equal（signed）	adf 0xE
adflt	（S xor OV）= 1	Less than（signed）	adf 0x6
adfle	((S xor OV）or Z）= 1	Less than or equal（signed）	adf 0x7
adfh	（CY or Z）= 0	Higher（Greater than）	adf 0xB
adfnl	CY = 0	Not lower（Greater than or equal）	adf 0x9
adfl	CY = 1	Lower（Less than）	adf 0x1
adfnh	（CY or Z）= 1	Not higher（Less than or equal）	adf 0x3
adfe	Z = 1	Equal	adf 0x2
adfne	Z = 0	Not equal	adf 0xA
adfv	OV = 1	Overflow	adf 0x0
adfnv	OV = 0	No overflow	adf 0x8
adfn	S = 1	Negative	adf 0x4
adfp	S = 0	Positive	adf 0xC
adfc	CY = 1	Carry	adf 0x1
adfnc	CY = 0	No carry	adf 0x9
adfz	Z = 1	Zero	adf 0x2
adfnz	Z = 0	Not zero	adf 0xA
adft	always 1	Always 1	adf 0x5

[フラグ]

CY	MSB（Most Significant Bit）からのキャリーがあれば 1，そうでない場合 0
OV	オーバーフローが起こった場合 1，そうでない場合 0
S	演算結果が負の場合 1，そうでない場合 0
Z	演算結果が 0 の場合 1，そうでない場合 0
SAT	—

[注意事項]

- adf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

```
W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。
```

- adf 命令の imm4 に 0xD を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550261 : 指定された条件コードが不正です。
```

sub

減算を行います。（Subtract）

［指定形式］

- sub reg1, reg2
- sub imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“sub reg1, reg2”の形式

第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

-“sub imm, reg2”の形式

第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定した絶対値式，または相対値式の値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“sub reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の sub 命令が 1 つ生成されます。

-“sub imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`sub 0, reg`	`sub r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`sub imm5, reg`	`mov imm5, r1` `sub r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`sub imm16, reg`	`movea imm16, r0, r1` `sub r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

sub imm, reg	movhi HIGHW(imm), r0, r1 sub r1, reg

上記以外の場合

sub imm, reg	mov imm, r1 sub r1, reg

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

sub $label, reg	movea $label, r0, r1 sub r1, reg

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

sub #label, reg	mov #label, r1 sub r1, reg
sub label, reg	mov label, r1 sub r1, reg
sub $label, reg	mov $label, r1 sub r1, reg

注 機械語命令の sub 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

subr

逆減算を行います。（Subtract Reverse）

［指定形式］

- subr reg1, reg2
- subr imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “subr reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

- “subr imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “subr reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の subr 命令が 1 つ生成されます。
- “subr imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`subr 0, reg`	`subr r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`subr imm5, reg`	`mov imm5, r1` `subr r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`subr imm16, reg`	`movea imm16, r0, r1` `subr r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

subr imm, reg	movhi HIGHW(imm), r0, r1 subr r1, reg

上記以外の場合

subr imm, reg	mov imm, r1 subr r1, reg

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

subr $label, reg	movea $label, r0, r1 subr r1, reg

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

subr #label, reg	mov #label, r1 subr r1, reg
subr label, reg	mov label, r1 subr r1, reg
subr $label, reg	mov $label, r1 subr r1, reg

注 機械語命令の subr 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

sbf

条件付き減算を行います。（Subtract on Condition Flag）【V850E2】

［指定形式］

- sbf imm4, reg1, reg2, reg3
- sbf*cnd* reg1, reg2, reg3

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式（0xD は指定できません）

［機能］

- “sbf imm4, reg1, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式の下位 4 ビットの値で示されるフラグ状態（「表 4―33 sbfcnd 命令」を参照）と現在のフラグ状態を比較します。

  値が一致した場合には，第 3 オペランドに指定したレジスタのワード・データから，第 2 オペランドに指定したレジスタのワード・データを減算し，さらに 1 を減算します。その結果を第 4 オペランドに指定したレジスタに格納します。

  値が一致しなかった場合には，第 3 オペランドに指定したレジスタのワード・データから，第 2 オペランドに指定したレジスタのワード・データを減算します。その結果を第 4 オペランドに指定したレジスタに格納します。

- “sbf*cnd* reg1, reg2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。

  値が一致した場合には，第 2 オペランドに指定したレジスタのワード・データから，第 1 オペランドに指定したレジスタのワード・データを減算し，さらに 1 を減算します。その結果を第 3 オペランドに指定したレジスタに格納します。

  値が一致しなかった場合には，第 2 オペランドに指定したレジスタのワード・データから，第 1 オペランドに指定したレジスタのワード・データを減算します。その結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- sbf 命令に対し，アセンブラでは，機械語命令の sbf 命令が 1 つ生成されます。
- sbf*cnd* 命令に対し，アセンブラでは，対応する sbf 命令が生成され（「表 4―33 sbfcnd 命令」を参照），“sbf imm4, reg1, reg2, reg3”の形式に展開されます。

表 4―33 sbf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
sbfgt	((S xor OV) or Z) = 0	Greater than（signed）	sbf 0xF

命令	フラグ状態	フラグ状態の意味	命令展開
sbfge	（S xor OV）= 0	Greater than or equal（signed）	sbf 0xE
sbflt	（S xor OV）= 1	Less than（signed）	sbf 0x6
sbfle	（（S xor OV）or Z）= 1	Less than or equal（signed）	sbf 0x7
sbfh	（CY or Z）= 0	Higher（Greater than）	sbf 0xB
sbfnl	CY = 0	Not lower（Greater than or equal）	sbf 0x9
sbfl	CY = 1	Lower（Less than）	sbf 0x1
sbfnh	（CY or Z）= 1	Not higher（Less than or equal）	sbf 0x3
sbfe	Z = 1	Equal	sbf 0x2
sbfne	Z = 0	Not equal	sbf 0xA
sbfv	OV = 1	Overflow	sbf 0x0
sbfnv	OV = 0	No overflow	sbf 0x8
sbfn	S = 1	Negative	sbf 0x4
sbfp	S = 0	Positive	sbf 0xC
sbfc	CY = 1	Carry	sbf 0x1
sbfnc	CY = 0	No carry	sbf 0x9
sbfz	Z = 1	Zero	sbf 0x2
sbfnz	Z = 0	Not zero	sbf 0xA
sbft	always 1	Always 1	sbf 0x5

［フラグ］

CY	MSB（Most Significant Bit）からのボローがあれば 1，そうでない場合 0
OV	オーバーフローが起こった場合 1，そうでない場合 0
S	演算結果が負の場合 1，そうでない場合 0
Z	演算結果が 0 の場合 1，そうでない場合 0
SAT	―

［注意事項］

- sbf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

- sbf 命令の imm4 に 0xD を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550261 : 指定された条件コードが不正です。

mulh

符号付き乗算（ハーフワード）を行います。（Multiply Half-word）

［指定形式］

- mulh reg1, reg2
- mulh imm, reg2

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式

注 アセンブラでは，16 ビットを越えるかどうかチェックは行われません。生成された機械語命令 mulh 命令では，下位 16 ビットを用いて演算が行われます。

［機能］

-“mulh reg1, reg2”の形式

第 1 オペランドに指定したレジスタの下位ハーフワード・データの値と，第 2 オペランドに指定したレジスタの下位ハーフワード・データの値を，符号付きの値として乗算し，結果を第 2 オペランドに指定したレジスタに格納します。

-“mulh imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値と，第 2 オペランドに指定したレジスタの下位ハーフワード・データの値を，符号付きの値として乗算し，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“mulh reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の mulh 命令が 1 つ生成されます。
-“mulh imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の add 命令注が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`mulh imm5, reg`	`mulh imm5, reg`

注 機械語命令の add 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

- “mulh imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

mulh imm16, reg	mulhi imm16, reg, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

mulh imm, reg	movhi HIGHW(imm), r0, r1 mulh r1, reg

上記以外の場合

mulh imm, reg	mov imm, r1 mulh r1, reg

（c）!label，または %label を持つ相対値式，および sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

mulh !label, reg	mulhi !label, reg, reg
mulh %label, reg	mulhi %label, reg, reg
mulh $label, reg	mulhi $label, reg, reg

（d）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

mulh #label, reg	mov #label, r1 mulh r1, reg
mulh label, reg	mov label, r1 mulh r1, reg
mulh $label, reg	mov $label, r1 mulh r1, reg

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- 第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

mulhi

符号付き乗算（ハーフワード）を行います。（Multiply Half-word）

［指定形式］

- mulhi imm, reg1, reg2

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

注 アセンブラでは，16 ビットを越えるかどうかチェックは行われません。生成された機械語命令 mulhi 命令では，下位 16 ビットを用いて演算が行われます。

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値と，第 2 オペランドに指定したレジスタ値を符号付きの値として乗算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の mulhi 命令注が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mulhi $label, reg1, reg2`	`mulhi $label, reg1, reg2`

（c）!label，または %label を持つ相対値式

`mulhi !label, reg1, reg2`	`mulhi !label, reg1, reg2`
`mulhi %label, reg1, reg2`	`mulhi %label, reg1, reg2`

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

`mulhi imm16, reg1, reg2`	`mulhi imm16, reg1, reg2`

注 機械語命令の mulhi 命令は，第 1 オペランドに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mulhi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `mulh reg1, reg2`

上記以外の場合

`mulhi imm, reg1, reg2`	`mov imm, reg2` `mulh reg1, reg2`

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`mulhi #label, reg1, reg2`	`mov #label, reg2` `mulhi reg1, reg2`
`mulhi label, reg1, reg2`	`mov label, reg2` `mulh reg1, reg2`
`mulhi $label, reg1, reg2`	`mov $label, reg2` `mulh reg1, reg2`

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- 第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

mul

符号付き乗算（ワード）を行います。（Multiply Word）

［指定形式］

- mul reg1, reg2, reg3
- mul imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “mul reg1, reg2, reg3”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を，符号付きの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには乗算結果の上位 32 ビットを格納します。

- “mul imm, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を，符号付きの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには乗算結果の上位 32 ビットを格納します。

［詳細説明］

- “mul reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の mul 命令が 1 つ生成されます。
- “mul imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）0

`mul 0, reg2, reg3`	`mul r0, reg2, reg3`

（b）0 以外の -256 ～ +255 の範囲の絶対値式

`mul imm9, reg2, reg3`	`mul imm9, reg2, reg3`

（c）-256 ～ +255 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`mul imm16, reg2, reg3`	`movea imm16, r0, r1` `mul r1, reg2, reg3`

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mul imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mul r1, reg2, reg3`

上記以外の場合

`mul imm, reg2, reg3`	`mov imm, r1` `mul r1, reg2, reg3`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mul $label, reg2, reg3`	`movea $label, r0, r1` `mul r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`mul #label, reg2, reg3`	`mov #label, r1` `mul r1, reg2, reg3`
`mul label, reg2, reg3`	`mov label, r1` `mul r1, reg2, reg3`
`mul $label, reg2, reg3`	`mov $label, r1` `mul r1, reg2, reg3`

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- “mul reg1, reg2, reg3”の形式の命令に対して“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1，reg3 と異なるレジスタである”，“reg1，reg3 が r0，または r1 ではない”の条件をすべて満たす場合，アセンブラは命令展開を行い，複数個の機械語命令を生成します。

```
mov     reg1, r1
mul     r1, reg2, reg3
```

- “mul reg1, reg2, reg3”の形式の命令に対し，“reg1，reg3 が r1 である”，“reg2 は reg1，reg3 と異なるレジスタである”の条件をすべて満たす場合，アセンブラは次のメッセージを出力し，アセンブルを中止します。

W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ソース レジスタとしてオペランドに指定されています。
W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーション レジスタとしてオペランドに指定されています。
E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

- “mul imm, reg2, reg3”の形式の命令に対し，“reg3 が r1 である”，“reg2 は reg3 と異なるレジスタである”の条件をすべて満たす場合で，命令展開により複数個の機械語命令を生成する可能性がある場合，アセンブラは次のメッセージを出力し，アセンブルを中止します。

W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーション レジスタとしてオペランドに指定されています。
E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

- 警告メッセージ抑止オプション -Xno_warning=W0550013 を指定した場合，次のメッセージを出力し，アセンブルを中止します。

E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

mulu

符号なし乗算（ワード）を行います。（Multiply Word Unsigned）

［指定形式］

- mulu reg1, reg2, reg3
- mulu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “mulu reg1, reg2, reg3”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を，符号なしの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには結果の上位 32 ビットを格納します。
- “mulu imm, reg2, reg3”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を，符号なしの値として乗算し，結果の下位 32 ビットを第 2 オペランドに指定したレジスタに，上位 32 ビットを第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには結果の上位 32 ビットを格納します。

［詳細説明］

- “mulu reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の mulu 命令が 1 つ生成されます。
- “mulu imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）0

`mulu 0, reg2, reg3`	`mulu r0, reg2, reg3`

（b）0 以外の -256 ～＋ 255 の範囲

`mulu imm9, reg2, reg3`	`mulu imm9, reg2, reg3`

（c）-256 ～ +255 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`mulu imm16, reg2, reg3`	`movea imm16, r0, r1` `mulu r1, reg2, reg3`

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mulu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mulu r1, reg2, reg3`

上記以外の場合

`mulu imm, reg2, reg3`	`mov imm, r1` `mulu r1, reg2, reg3`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mulu $label, reg2, reg3`	`movea $label, r0, r1` `mulu r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`mulu #label, reg2, reg3`	`mov #label, r1` `mulu r1, reg2, reg3`
`mulu label, reg2, reg3`	`mov label, r1` `mulu r1, reg2, reg3`
`mulu $label, reg2, reg3`	`mov $label, r1` `mulu r1, reg2, reg3`

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- “mulu reg1, reg2, reg3”の形式の命令に対して“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1，reg3 と異なるレジスタである”，“reg1，reg3 が r0，または r1 ではない”の条件をすべて満たす場合，アセンブラは命令展開を行い，複数個の機械語命令を生成します。

```
mov     reg1, r1
mulu    r1, reg2, reg3
```

- “mulu reg1, reg2, reg3”の形式の命令に対し，“reg1 と reg3 は同一のレジスタである”，“reg2 は reg1，reg3 と異なるレジスタである”，“reg1，reg3 が r1 である”の条件をすべて満たす場合，アセンブラは次のメッセージを出力し，アセンブルを中止します。

W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ソース レジスタとしてオペランドに指定されています。
W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーション レジスタとしてオペランドに指定されています。
E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

- “mulu imm, reg2, reg3”の形式の命令に対し，“reg2 は reg3 と異なるレジスタである”，“reg3 が r1 である”の条件をすべて満たす場合で，命令展開により複数個の機械語命令を生成する可能性がある場合，アセンブラは次のいずれかメッセージを出力し，アセンブルを中止します。

W0550013 : レジスタ (r0)、アセンブラ予約レジスタ (r1) が、ディスティネーション レジスタとしてオペランドに指定されています。
E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

- 警告メッセージ抑止オプション -Xno_warning=W0550013 を指定した場合，メ次のメッセージが出力され，アセンブルが中止されます。

E0550259 : mul/mulu 命令のディスティネーション・レジスタにはアセンブラ予約レジスタ (r1) を指定できません。

mac

符号付きワード・データの加算付き乗算を行います。（Multiply Word and Add）【V850E2】

［指定形式］

- mac reg1, reg2, reg3, reg4

［機能］

汎用レジスタ reg2 のワード・データに，汎用レジスタ reg1 のワード・データを乗算した結果（64 ビット・データ）と，汎用レジスタ reg3 を下位 32 ビットとして，汎用レジスタ reg3+1（たとえば，reg3 が r6 の場合，「reg3+1」は r7 となります）を上位 32 ビットとして結合した 64 ビット・データを加算し，その結果（64 ビット・データ）の上位 32 ビットを汎用レジスタ reg4+1 に，下位 32 ビットを汎用レジスタ reg4 に格納します。

汎用レジスタ reg1，reg2 の内容を 32 ビットの符号付き整数として扱います。

汎用レジスタ reg1，reg2，reg3，reg3+1 は影響を受けません。

［詳細説明］

アセンブラでは，機械語命令の mac 命令が 1 つ生成されます。

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- reg3，または reg4 に指定できる汎用レジスタは，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）だけです。奇数番号の付いたレジスタ（r1，r3，…，r31）を指定した場合は，次のメッセージが出力され，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）を指定したとして，アセンブルが続行されます。

> W0550026 : 奇数番号の付いたレジスタ（r*XX*）が指定されています。
> 偶数番号の付いたレジスタ（r*YY*）を指定したとして、アセンブルを続行します。

macu

符号なしワード・データの加算付き乗算を行います。（Multiply Word Unsigned and Add）【V850E2】

［指定形式］

- macu reg1, reg2, reg3, reg4

［機能］

汎用レジスタ reg2 のワード・データに，汎用レジスタ reg1 のワード・データを乗算した結果（64 ビット・データ）と，汎用レジスタ reg3 を下位 32 ビットとして，汎用レジスタ reg3+1（たとえば，reg3 が r6 の場合，「reg3+1」は r7 となります）を上位 32 ビットとして結合した 64 ビット・データを加算し，その結果（64 ビット・データ）の上位 32 ビットを汎用レジスタ reg4+1 に，下位 32 ビットを汎用レジスタ reg4 に格納します。

汎用レジスタ reg1，reg2 の内容を 32 ビットの符号なし整数として扱います。

汎用レジスタ reg1，reg2，reg3，reg3+1 は影響を受けません。

［詳細説明］

アセンブラでは，機械語命令の macu 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- reg3，または reg4 に指定できる汎用レジスタは，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）だけです。奇数番号の付いたレジスタ（r1，r3，…，r31）を指定した場合は，次のメッセージが出力され，偶数番号の付いたレジスタ（r0，r2，r4，…，r30）を指定したとして，アセンブルが続行されます。

W0550026 : 奇数番号の付いたレジスタ（r*XX*）が指定されています。
偶数番号の付いたレジスタ（r*YY*）を指定したとして、アセンブルを続行します。

divh

符号付き除算（ハーフワード）を行います。（Divide Half-word）

［指定形式］

- divh reg1, reg2
- divh imm, reg2
- divh reg1, reg2, reg3
- divh imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式注
- 相対値式

注 アセンブラでは，16 ビットを越えるかどうかのチェックは行われません。生成された機械語命令では，下位 16 ビットを用いて演算が行われます。

［機能］

- “divh reg1, reg2”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに格納します。

- “divh imm, reg2”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに格納します。

- “divh reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divh imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divh reg1, reg2”，および“divh reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の divh 命令が 1 つ生成されます。
- “divh imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`divh 0, reg`	`divh r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`divh imm5, reg`	`mov imm5, r1` `divh r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`divh imm16, reg`	`movea imm16, r0, r1` `divh r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`divh imm, reg`	`movhi HIGHW(imm), r0, r1` `divh r1, reg`

上記以外の場合

`divh imm, reg`	`mov imm, r1` `divh r1, reg`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`divh $label, reg`	`movea $label, r0, r1` `divh r1, reg`

(f) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式【V850E】

divh #label, reg	mov #label, r1 divh r1, reg
divh label, reg	mov label, r1 divh r1, reg
divh $label, reg	mov $label, r1 divh r1, reg

注 機械語命令の divh 命令は，オペランドにイミーディエトをとりません。

- “divh imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

(a) 0

divh 0, reg2, reg3	divh r0, reg2, reg3

(b) 0 以外の -16 ～ +15 の範囲の絶対値式

divh imm5, reg2, reg3	mov imm5, r1 divh r1, reg2, reg3

(c) -16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

divh imm16, reg2, reg3	movea imm16, r0, r1 divh r1, reg2, reg3

(d) -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

divh imm, reg2, reg3	movhi HIGHW(imm), r0, r1 divh r1, reg2, reg3

上記以外の場合

divh imm, reg2, reg3	mov imm, r1 divh r1, reg2, reg3

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`divh $label, reg2, reg3`	`movea $label, r0, r1` `divh r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`divh #label, reg2, reg3`	`mov #label, r1` `divh r1, reg2, reg3`
`divh label, reg2, reg3`	`mov label, r1` `divh r1, reg2, reg3`
`divh $label, reg2, reg3`	`mov $label, r1` `divh r1, reg2, reg3`

［フラグ］

CY	―
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

［注意事項］

- “divh reg1, reg2”の第 1 オペランド（reg1）に r0 を指定すると，次のいずれかメッセージが出力され，アセンブルが中止されます。

E0550239:V850Ex コア指定時には、ソース・オペランドに r0 を指定することはできません。
E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

- “divh imm, reg2”の第 2 オペランド（reg2）に r0 を指定すると，次のいずれかメッセージが出力され，アセンブルが中止されます。

E0550239:V850Ex コア指定時には、ソース・オペランドに r0 を指定することはできません。
E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

- “divh imm, reg2”の第 2 オペランド（imm）に 0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550239:V850Ex コア指定時には、ソース・オペランドに r0 を指定することはできません。

div

符号付き除算（ワード）を行います。（Divide Word）

［指定形式］

- div reg1, reg2, reg3
- div imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “div reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタ値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに，それぞれ格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “div imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の値で符号付きの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “div reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の div 命令が 1 つ生成されます。
- “div imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`div 0, reg2, reg3`	`div r0, reg2, reg3`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`div imm5, reg2, reg3`	`mov imm5, r1` `div r1, reg2, reg3`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`div imm16, reg2, reg3`	`movea imm16, r0, r1` `div r1, reg2, reg3`

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`div imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `div r1, reg2, reg3`

上記以外の場合

`div imm, reg2, reg3`	`mov imm, r1` `div r1, reg2, reg3`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`div $label, reg2, reg3`	`movea $label, r0, r1` `div r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`div #label, reg2, reg3`	`mov #label, r1` `div r1, reg2, reg3`
`div label, reg2, reg3`	`mov label, r1` `div r1, reg2, reg3`
`div $label, reg2, reg3`	`mov $label, r1` `div r1, reg2, reg3`

注 機械語命令の div 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

divhu

符号なし除算（ハーフワード）を行います。（Divide Half-word Unsigned）

［指定形式］

- divhu reg1, reg2, reg3
- divhu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式**注**
- 相対値式

注 アセンブラでは，16 ビットを越えるかどうかのチェックは行われません。生成された機械語命令では，下位 16 ビットを用いて演算が行われます。

［機能］

- “divhu reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタの下位ハーフワード・データの値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divhu imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の下位ハーフワード・データの値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divhu reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の divhu 命令が 1 つ生成されます。
- “divhu imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`divhu 0, reg2, reg3`	`divhu r0, reg2, reg3`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

divhu imm5, reg2, reg3	mov imm5, r1 divhu r1, reg2, reg3

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

divhu imm16, reg2, reg3	movea imm16, r0, r1 divhu r1, reg2, reg3

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

divhu imm, reg2, reg3	movhi HIGHW(imm), r0, r1 divhu r1, reg2, reg3

上記以外の場合

divhu imm, reg2, reg3	mov imm, r1 divhu r1, reg2, reg3

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

divhu $label, reg2, reg3	movea $label, r0, r1 divhu r1, reg2, reg3

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

divhu #label, reg2, reg3	mov #label, r1 divhu r1, reg2, reg3
divhu label, reg2, reg3	mov label, r1 divhu r1, reg2, reg3
divhu $label, reg2, reg3	mov $label, r1 divhu r1, reg2, reg3

注 機械語命令の divhu 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	―
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	―

divu

符号なし除算（ワード）を行います。（Divide Word Unsigned）

［指定形式］

- divu reg1, reg2, reg3
- divu imm, reg2, reg3

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “divu reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定したレジスタ値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

- “divu imm, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値を，第 1 オペランドに指定した絶対値式，または相対値式の値で符号なしの値として除算し，商を第 2 オペランドに指定したレジスタに，剰余を第 3 オペランドに指定したレジスタに格納します。第 2 オペランドと第 3 オペランドが同じレジスタの場合，レジスタには剰余を格納します。

［詳細説明］

- “divu reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の divu 命令が 1 つ生成されます。
- “divu imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`divu 0, reg2, reg3`	`divu r0, reg2, reg3`

（b）0 以外の -16 ～＋ 15 の範囲の絶対値式

`divu imm5, reg2, reg3`	`mov imm5, r1` `divu r1, reg2, reg3`

（c）-16 ～＋ 15 の範囲を越え，-32,768 ～ +32,767 の範囲の絶対値式

`divu imm16, reg2, reg3`	`movea imm16, r0, r1` `divu r1, reg2, reg3`

（d）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`divu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `divu r1, reg2, reg3`

上記以外の場合

`divu imm, reg2, reg3`	`mov imm, r1` `divu r1, reg2, reg3`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`divu $label, reg2, reg3`	`movea $label, r0, r1` `divu r1, reg2, reg3`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義に持たないラベルの $label を持つ相対値式

`divu #label, reg2, reg3`	`mov #label, r1` `divu r1, reg2, reg3`
`divu label, reg2, reg3`	`mov label, r1` `divu r1, reg2, reg3`
`divu $label, reg2, reg3`	`mov $label, r1` `divu r1, reg2, reg3`

注 機械語命令の divu 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	－
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

cmp

比較を行います。（Compare）

［指定形式］

- cmp reg1, reg2
- cmp imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“cmp reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値と第 2 オペランドに指定したレジスタ値を比較し，結果をフラグに示します。なお，比較は，第 2 オペランドに指定したレジスタ値から第 1 オペランドに指定したレジスタ値を減算することにより行われます。

-“cmp imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の値と第 2 オペランドに指定したレジスタ値を比較し，結果をフラグに示します。なお，比較は，第 2 オペランドに指定したレジスタ値から第 1 オペランドに指定した値を減算することにより行われます。

［詳細説明］

-“cmp reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の cmp 命令が 1 つ生成されます。

-“cmp imm, reg2”の形式の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の cmp 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`cmp imm5, reg`	`cmp imm5, reg`

注 機械語命令の cmp 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

-“cmp imm, reg2”の形式の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`cmp imm16, reg`	`movea imm16, r0, r1` `cmp r1, reg`

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`cmp imm, reg`	`movhi HIGHW(imm), r0, r1` `cmp r1, reg`

上記以外の場合

`cmp imm, reg`	`mov imm, r1` `cmp r1, reg`

（c）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`cmp $label, reg`	`movea $label, r0, r1` `cmp r1, reg`

（d）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`cmp #label, reg`	`mov #label, r1` `cmp r1, reg`
`cmp label, reg`	`mov label, r1` `cmp r1, reg`
`cmp $label, reg`	`mov $label, r1` `cmp r1, reg`

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	－

mov

データの転送を行います。（Move）

［指定形式］

- mov reg1, reg2
- mov imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“mov reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値を，第 2 オペランドに指定したレジスタに格納します。

-“mov imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の値を，第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“mov reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の mov 命令が 1 つ生成されます。

-“mov imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の mov 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

`mov imm5, reg`	`mov imm5, reg`

注 機械語命令の mov 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

-“mov imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え -32768 ～ +32767 の範囲の絶対値式

`mov imm16, reg`	`movea imm16, r0, reg`

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`mov imm, reg`	`movhi HIGHW(imm), r0, reg`

上記以外の場合[注]

`mov imm, reg`	`mov imm, reg`

注 16 ビット長の mov 命令を，48 ビット長の mov 命令に置き換えます。

（c）!label，または %label を持つ相対値式，および sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`mov !label, reg`	`movea !label, r0, reg`
`mov %label, reg`	`movea %label, r0, reg`
`mov $label, reg`	`movea $label, r0, reg`

（d）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式[注]

`mov #label, reg`	`mov #label, reg`
`mov label, reg`	`mov label, reg`
`mov $label, reg`	`mov $label, reg`

注 16 ビット長の mov 命令を，48 ビット長の mov 命令に置き換えます。

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- “mov imm, reg2”の命令に対し，第 1 オペランドに -16 ～ +15 の範囲の絶対値式を指定し，かつ第 2 オペランドに r0 を指定した場合，または“mov reg1, reg2”の第 2 オペランドに r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

movea

実効アドレスの転送を行います。（Move Effective Address）

［指定形式］

- movea imm, reg1, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値を，第 2 オペランドに指定したレジスタ値と加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の movea 命令[注]が 1 つ生成されます。
- reg1 に r0 を指定した場合，アセンブラでは，mov imm, reg2 を指定したものとして扱います。

（a）-32768 ～ +32767 の範囲の絶対値式

`movea imm16, reg1, reg2`	`movea imm16, reg1, reg2`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`movea $label, reg1, reg2`	`movea $label, reg1, reg2`

（c）!label，または %label を持つ相対値式

`movea !label, reg1, reg2`	`movea !label, reg1, reg2`
`movea %label, reg1, reg2`	`movea %label, reg1, reg2`

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

`movea imm16, reg1, reg2`	`movea imm16, reg1, reg2`

注 機械語命令の movea 命令は，第 1 オペランドに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`movea imm, reg1, reg2`	`movhi HIGHW(imm), reg1, reg2`

上記以外の場合

`movea imm, reg1, reg2`	`movhi HIGHW1(imm), reg1, r1` `movea LOWW(imm), r1, reg2`

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`movea #label, reg1, reg2`	`movhi HIGHW1(#label), reg1, r1` `movea LOWW(#label), r1, reg2`
`movea label, reg1, reg2`	`movhi HIGHW1(label), reg1, r1` `movea LOWW(label), r1, reg2`
`movea $label, reg1, reg2`	`movhi HIGHW1($label), reg1, r1` `movea LOWW($label), r1, reg2`

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- 第 3 オペランドに r0 を指定すると，メッセージが出力され，アセンブルが中止されます。

movhi

上位ハーフワードの転送を行います。（Move High half-word）

［指定形式］

- movhi imm16, reg1, reg2

imm16 に指定可能なものを以下に示します。

- 16 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した値を上位 16 ビットの値，0 を下位 16 ビットの値とするワード・データと，第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の movhi 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- imm16 に 0 ～ 65535 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550231 : イミーディエトとして指定された値が指定可能な値の範囲を越えています。
```

- 第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。
```

mov32

32 ビット・データの転送を行います。（32bit Move）

［指定形式］

- mov32 imm, reg2

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値を，第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の 48 ビット長 mov 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

cmov

フラグ条件付きデータの転送を行います。（Conditional Move）

［指定形式］

- cmov imm4, reg1, reg2, reg3
- cmov imm4, imm, reg2, reg3
- cmov*cnd* reg1, reg2, reg3
- cmov*cnd* imm, reg2, reg3

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式**注**

注 機械語命令の cmov 命令は，第 1 オペランドに 0 ～ 15（0x0 ～ 0xF）の範囲のイミーディエトをとります。

imm に指定可能なものを以下に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “cmov imm4, reg1, reg2, reg3”の形式

  第 1 オペランドに指定した定数式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は，第 2 オペランドに指定したレジスタ値を，一致しなかった場合は第 3 オペランドに指定したレジスタ値を，第 4 オペランドに指定したレジスタに格納します。
- “cmov imm4, imm, reg2, reg3”の形式

  第 1 オペランドに指定した定数式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は，第 2 オペランドに指定した絶対値式の値を，一致しなかった場合は第 3 オペランドに指定したレジスタ値を，第 4 オペランドに指定したレジスタに格納します。
- “cmov*cnd* reg1, reg2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，値が一致した場合は，第 1 オペランドに指定したレジスタ値を，一致しなかった場合は第 2 オペランドに指定したレジスタ値を，第 3 オペランドに指定したレジスタに格納します。
- “cmov*cnd* imm, reg2, reg3”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，値が一致した場合は，第 1 オペランドに指定した絶対値式の値を，一致しなかった場合は第 2 オペランドに指定したレジスタ値を，第 3 オペランドに指定したレジスタに格納します。

表 4―34 cmov*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
cmovgt	((S xor OV) or Z) = 0	Greater than（signed）	cmov 0xF
cmovge	(S xor OV) = 0	Greater than or equal（signed）	cmov 0xE
cmovlt	(S xor OV) = 1	Less than（signed）	cmov 0x6
cmovle	((S xor OV) or Z) = 1	Less than or equal（signed）	cmov 0x7
cmovh	(CY or Z) = 0	Higher（Greater than）	cmov 0xB
cmovnl	CY = 0	Not lower（Greater than or equal）	cmov 0x9
cmovl	CY = 1	Lower（Less than）	cmov 0x1
cmovnh	(CY or Z) = 1	Not higher（Less than or equal）	cmov 0x3
cmove	Z = 1	Equal	cmov 0x2
cmovne	Z = 0	Not equal	cmov 0xA
cmovv	OV = 1	Overflow	cmov 0x0
cmovnv	OV = 0	No overflow	cmov 0x8
cmovn	S = 1	Negative	cmov 0x4
cmovp	S = 0	Positive	cmov 0xC
cmovc	CY = 1	Carry	cmov 0x1
cmovnc	CY = 0	No carry	cmov 0x9
cmovz	Z = 1	Zero	cmov 0x2
cmovnz	Z = 0	Not zero	cmov 0xA
cmovt	always 1	Always 1	cmov 0x5
cmovsa	SAT = 1	Saturated	cmov 0xD

［詳細説明］

- “cmov imm4, reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の cmov 命令[注]が 1 つ生成されます。

注 機械語命令の cmov 命令は，第 2 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

- “cmov imm4, imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の cmov 命令が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`cmov imm4, imm5, reg2, reg3`	`cmov imm4, imm5, reg2, reg3`

- “cmov imm4, imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

cmov imm4, imm16, reg2, reg3	movea imm16, r0, r1 cmov imm4, r1, reg2, reg3

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

cmov imm4, imm, reg2, reg3	movhi HIGHW(imm), r0, r1 cmov imm4, r1, reg2, reg3

上記以外の場合

cmov imm4, imm, reg2, reg3	mov imm, r1 cmov imm4, r1, reg2, reg3

（c）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

cmov imm4, #label, reg2, reg3	mov #label, r1 cmov imm4, r1, reg2, reg3
cmov imm4, label, reg2, reg3	mov label, r1 cmov imm4, r1, reg2, reg3
cmov imm4, $label, reg2, reg3	mov $label, r1 cmov imm4, r1, reg2, reg3

（d）!label，または %label を持つ相対値式，および sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

cmov imm4, !label, reg2, reg3	movea !label, r0, r1 cmov imm4, r1, reg2, reg3
cmov imm4, %label, reg2, reg3	movea %label, r0, r1 cmov imm4, r1, reg2, reg3
cmov imm4, $label, reg2, reg3	movea $label, r0, r1 cmov imm4, r1, reg2, reg3

- “cmov*cnd* reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，対応する cmov 命令が生成され（「表 4―34 cmovcnd 命令」を参照），“cmov imm4, reg1, reg2, reg3”の形式に展開されます。

- “cmov*cnd* imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，対応する cmov 命令が生成され（「表 4―34 cmovcnd 命令」を参照），“cmov imm4, imm, reg2, reg3”の形式に展開されます。

（a）-16 ～ +15 の範囲の絶対値式

- “cmov*cnd* imm, reg2, reg3”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

cmov*cnd* imm16, reg2, reg3	movea imm16, r0, r1 cmov*cnd* r1, reg2, reg3

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

cmov*cnd* imm, reg2, reg3	movhi HIGHW(imm), r0, r1 cmov*cnd* r1, reg2, reg3

上記以外の場合

cmov*cnd* imm, reg2, reg3	mov imm, r1 cmov*cnd* r1, reg2, reg3

（c）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

cmov*cnd* #label, reg2, reg3	mov #label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* label, reg2, reg3	mov label, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* $label, reg2, reg3	mov $label, r1 cmov*cnd* r1, reg2, reg3

（d）!label，または %label を持つ相対値式，および sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

cmov*cnd* !label, reg2, reg3	movea !label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* %label, reg2, reg3	movea %label, r0, r1 cmov*cnd* r1, reg2, reg3
cmov*cnd* $label, reg2, reg3	movea $label, r0, r1 cmov*cnd* r1, reg2, reg3

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- cmov 命令の imm4 に 4 ビットの範囲を越える定数式を指定した場合，次のメッセージが出力されます。
  なお，4 ビット幅を越える場合は，0xF でマスクした値を用いてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

setf

フラグ条件の設定を行います。（Set Flag Condition）

［指定形式］

- setf imm4, reg
- setf*cnd* reg

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

- “setf imm4, reg”の形式

  第 1 オペランドに指定した絶対値式の値の下位 4 ビットの値で示されるフラグ状態と，現在のフラグ状態を比較し，値が一致した場合は 1 を，一致しなかった場合は 0 を第 2 オペランドに指定したレジスタに格納します。

- “setf*cnd* reg”の形式

  cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較し，一致する場合は 1，一致しなかった場合は 0 を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- setf 命令に対し，アセンブラでは，機械語命令の setf 命令が 1 つ生成されます。
- setf*cnd* 命令に対し，アセンブラでは，対応する setf 命令が生成され（「表 4―35 setfcnd 命令」を参照），“setf imm4, reg”の形式に展開されます。

表 4―35 setf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
setfgt	((S xor OV) or Z) = 0	Greater than（signed）	setf 0xF
setfge	(S xor OV) = 0	Greater than or equal（signed）	setf 0xE
setflt	(S xor OV) = 1	Less than（signed）	setf 0x6
setfle	((S xor OV) or Z) = 1	Less than or equal（signed）	setf 0x7
setfh	(CY or Z) = 0	Higher（Greater than）	setf 0xB
setfnl	CY = 0	Not lower（Greater than or equal）	setf 0x9
setfl	CY = 1	Lower（Less than）	setf 0x1
setfnh	(CY or Z) = 1	Not higher（Less than or equal）	setf 0x3
setfe	Z = 1	Equal	setf 0x2
setfne	Z = 0	Not equal	setf 0xA

命令	フラグ状態	フラグ状態の意味	命令展開
setfv	OV = 1	Overflow	setf 0x0
setfnv	OV = 0	No overflow	setf 0x8
setfn	S = 1	Negative	setf 0x4
setfp	S = 0	Positive	setf 0xC
setfc	CY = 1	Carry	setf 0x1
setfnc	CY = 0	No carry	setf 0x9
setfz	Z = 1	Zero	setf 0x2
setfnz	Z = 0	Not zero	setf 0xA
setft	always 1	Always 1	setf 0x5
setfsa	SAT = 1	Saturated	setf 0xD

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- setf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

sasf

論理左シフト付きフラグ条件の設定を行います。（Shift And Set Flag Condition）

［指定形式］

- sasf　imm4, reg
- sasf*cnd* reg

imm4 に指定可能なものを以下に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

-“sasf imm4, reg”の形式

　第 1 オペランドに指定した絶対値式の値の下位 4 ビットの値で示されるフラグ状態（「表 4―36　sasfcnd 命令」を参照）と，現在のフラグ状態を比較します。値が一致した場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトした値と 1 とを論理和して，第 2 オペランドに指定したレジスタに格納し，一致しなかった場合は，第 2 オペランドで指定したレジスタの内容を左 1 ビット論理シフトして第 2 オペランドで指定したレジスタに格納します。

-“sasf*cnd* reg”の形式

　cnd 部分の文字列で示されるフラグ状態と現在のフラグの状態を比較します。値が一致した場合は，第 1 オペランドで指定したレジスタの内容を左 1 ビット論理シフトした値と 1 とを論理和して第 1 オペランドで指定したレジスタに格納し，一致しなかった場合は，第 1 オペランドで指定したレジスタの内容を左 1 ビット論理シフトして第 1 オペランドで指定したレジスタに格納します。

［詳細説明］

- sasf 命令に対し，アセンブラでは，機械語命令の sasf 命令が 1 つ生成されます。
- sasf*cnd* 命令に対し，アセンブラでは，対応する sasf 命令が生成され（「表 4―36　sasfcnd 命令」を参照），“sasf imm4, reg”の形式に展開されます。

表 4―36　sasf*cnd* 命令

命令	フラグ状態	フラグ状態の意味	命令展開
sasfgt	((S xor OV) or Z) = 0	Greater than（signed）	sasf 0xF
sasfge	(S xor OV) = 0	Greater than or equal（signed）	sasf 0xE
sasflt	(S xor OV) = 1	Less than（signed）	sasf 0x6
sasfle	((S xor OV) or Z) = 1	Less than or equal（signed）	sasf 0x7
sasfh	(CY or Z) = 0	Higher（Greater than）	sasf 0xB
sasfnl	CY = 0	Not lower（Greater than or equal）	sasf 0x9
sasfl	CY = 1	Lower（Less than）	sasf 0x1

命令	フラグ状態	フラグ状態の意味	命令展開
sasfnh	（CY or Z）= 1	Not higher（Less than or equal）	sasf 0x3
sasfe	Z = 1	Equal	sasf 0x2
sasfne	Z = 0	Not equal	sasf 0xA
sasfv	OV = 1	Overflow	sasf 0x0
sasfnv	OV = 0	No overflow	sasf 0x8
sasfn	S = 1	Negative	sasf 0x4
sasfp	S = 0	Positive	sasf 0xC
sasfc	CY = 1	Carry	sasf 0x1
sasfnc	CY = 0	No carry	sasf 0x9
sasfz	Z = 1	Zero	sasf 0x2
sasfnz	Z = 0	Not zero	sasf 0xA
sasft	always 1	Always 1	sasf 0x5
sasfsa	SAT = 1	Saturated	sasf 0xD

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

［注意事項］

- sasf 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定された値の下位 4 ビットが用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

4. 7. 8　飽和演算命令

この項では，飽和演算命令について説明します。次に，この項において説明する命令を示します。

表 4―37　飽和演算命令

命令	意味
satadd	飽和加算
satsub	飽和減算
satsubi	飽和減算（イミーディエト）
satsubr	飽和逆減算

satadd

飽和加算を行います。（Saturated Add）

［指定形式］

- satadd reg1, reg2
- satadd imm, reg2
- satadd reg1, reg2, reg3【V850E2】

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“satadd reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

-“satadd imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタの値を加算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

-“satadd reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値を加算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

-“satadd reg1, reg2”，および“satadd reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の satadd 命令が 1 つ生成されます。

-“satadd imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，機械語命令の satadd 命令[注]が 1 つ生成されます。

（a）-16 ～ +15 の範囲の絶対値式

satadd imm5, reg	satadd imm5, reg

注 機械語命令の satadd 命令は，第 1 オペランドにレジスタ，または -16 ～ +15（0xFFFFFFF0 ～ 0xF）の範囲のイミーディエトをとります。

- “satadd imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

satadd imm16, reg	movea imm16, r0, r1 satadd r1, reg

（b）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satadd imm, reg	movhi HIGHW(imm), r0, r1 satadd r1, reg

上記以外の場合

satadd imm, reg	mov imm, r1 satadd r1, reg

（c）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satadd $label, reg	movea $label, r0, r1 satadd r1, reg

（d）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satadd #label, reg	mov #label, r1 satadd r1, reg
satadd label, reg	mov label, r1 satadd r1, reg
satadd $label, reg	mov $label, r1 satadd r1, reg

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- “satadd reg1, reg2”，および“satadd imm, reg2”の形式の命令に対し，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

satsub

飽和減算を行います。（Saturated Subtract）

［指定形式］

- satsub reg1, reg2
- satsub imm, reg2
- satsub reg1, reg2, reg3【V850E2】

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “satsub reg1, reg2”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。
- “satsub imm, reg2”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定した絶対値式，または相対値式の値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。
- “satsub reg1, reg2, reg3”の形式

  第 2 オペランドに指定したレジスタ値から，第 1 オペランドに指定したレジスタ値を減算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- “satsub reg1, reg2”，および“satsub reg1, reg2, reg3”の形式の命令に対し，アセンブラでは，機械語命令の satsub 命令が 1 つ生成されます。
- “satsub imm, reg2”の形式で imm に次のもの，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`satsub  0, reg`	`satsub  r0, reg`

（b）-32768 ～ +32767 の範囲の絶対値式

satsub imm16, reg	satsubi imm16, reg, reg

（c）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satsub imm, reg	movhi HIGHW(imm), r0, r1 satsub r1, reg

上記以外の場合

satsub imm, reg	mov imm, r1 satsub r1, reg

（d）data/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsub $label, reg	satsubi $label, reg, reg

（e）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satsub #label, reg	mov #label, r1 satsub r1, reg
satsub label, reg	mov label, r1 satsub r1, reg
satsub $label, reg	mov $label, r1 satsub r1, reg

注 機械語命令の satsub 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- “satsub reg1, reg2”，および“satsub imm, reg2”の形式の命令に対し，第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

satsubi

飽和減算を行います。（Saturated Subtract Immediate）

［指定形式］

- satsubi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用したものの値を，第 2 オペランドに指定したレジスタ値から減算し，結果を第 3 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 3 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の satsubi 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

satsubi imm16, reg1, reg2	satsubi imm16, reg1, reg2

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsubi $label, reg1, reg2	satsubi $label, reg1, reg2

（c）!label，または %label を持つ相対値式

satsubi !label, reg1, reg2	satsubi !label, reg1, reg2
satsubi %label, reg1, reg2	satsubi %label, reg1, reg2

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

satsubi imm16, reg1, reg2	satsubi imm16, reg1, reg2

注 機械語命令の satsubi 命令は，第 1 オペランドに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

satsubi imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 satsubr reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

satsubi imm, reg1, r0	movhi HIGHW(imm), r0, r1 satsubr reg1, r1

上記以外の場合

satsubi imm, reg1, reg2	mov imm, reg2 satsubr reg1, reg2

上記以外，ただし reg2 が r0 の場合

satsubi imm, reg1, r0	mov imm, r1 satsubr reg1, r1

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

satsubi #label, reg1, r0	movhi #label, r1 satsubr reg1, r1
satsubi label, reg1, r0	mov label, r1 satsubr reg1, r1
satsubi $label, reg1, r0	mov $label, r1 satsubr reg1, r1

上記以外の場合

satsubi #label, reg1, reg2	movhi #label, reg2 satsubr reg1, reg2
satsubi label, reg1, reg2	mov label, reg2 satsubr reg1, reg2
satsubi $label, reg1, reg2	mov $label, reg2 satsubr reg1, reg2

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- 第 3 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

satsubr

飽和逆減算を行います。（Saturated Subtract Reverse）

［指定形式］

- satsubr reg1, reg2
- satsubr imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “satsubr reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値から，第 2 オペランドに指定したレジスタ値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

- “satsubr imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値から，第 2 オペランドに指定したレジスタの値を減算し，結果を第 2 オペランドに指定したレジスタに格納します。ただし，結果が正の最大値 0x7FFFFFFF を越えた場合は 0x7FFFFFFF を，負の最大値 0x80000000 を越えた場合は 0x80000000 を第 2 オペランドに指定したレジスタに格納し，SAT フラグに 1 を設定します。

［詳細説明］

- “satsubr reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の satsubr 命令が 1 つ生成されます。
- “satsubr imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`satsubr 0, reg`	`satsubr r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`satsubr imm5, reg`	`mov imm5, r1` `satsubr r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

satsubr imm16, reg	movea imm16, r0, r1 satsubr r1, reg

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式【V850E】

imm の値の下位 16 ビットがすべて 0 の場合

satsubr imm, reg	movhi HIGHW(imm), r0, r1 satsubr r1, reg

上記以外の場合

satsubr imm, reg	mov imm, r1 satsubr r1, reg

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

satsubr $label, reg	movea $label, r0, r1 satsubr r1, reg

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

satsubr #label, reg	mov #label, r1 satsubr r1, reg
satsubr label, reg	mov label, r1 satsubr r1, reg
satsubr $label, reg	mov $label, r1 satsubr r1, reg

注 機械語命令の satsubr 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	MSB（Most Significant Bit）へのボローが生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	OV = 1 になった場合 1，そうでない場合 －

［注意事項］

- 第 2 オペランドに r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

4. 7. 9 論理演算命令

この項では，論理演算命令について説明します。次に，この項において説明する命令を示します。

表 4―38 論理演算命令

命令	意味
or	論理和
ori	論理和（イミディエト）
xor	排他的論理和
xori	排他的論理和（イミーディエト）
and	論理積
andi	論理積（イミーディエト）
not	論理否定（1 の補数をとる）
shr	論理右シフト
sar	算術右シフト
shl	論理左シフト
sxb	バイト・データの符号拡張
sxh	2 バイト・データの符号拡張
zxb	バイト・データのゼロ拡張
zxh	2 バイト・データのゼロ拡張
bsh	ハーフワード・データのバイト・スワップ
bsw	ワード・データのバイト・スワップ
hsh	ハーフワード・データのハーフワード・スワップ【V850E2】
hsw	ワード・データのハーフワード・スワップ
tst	テスト
sch0l	MSB 側からのビット（0）検索【V850E2】
sch0r	LSB 側からのビット（0）検索【V850E2】
sch1l	MSB 側からのビット（1）検索【V850E2】
sch1r	LSB 側からのビット（1）検索【V850E2】

or

論理和を行います。（Or）

［指定形式］

- or reg1, reg2
- or imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“or reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

-“or imm, reg2”の形式

第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

-“or reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の or 命令が 1 つ生成されます。

-“or imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`or 0, reg`	`or r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`or imm16, reg`	`ori imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`or imm5, reg`	`mov imm5, r1` `or r1, reg`

(d) -32768 ～ -17 の範囲の絶対値式

or imm16, reg	movea imm16, r0, r1 or r1, reg

(e) 上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

or imm, reg	movhi HIGHW(imm), r0, r1 or r1, reg

上記以外の場合

or imm, reg	mov imm, r1 or r1, reg

(f) sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

or $label, reg	movea $label, r0, r1 or r1, reg

(g) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

or #label, reg	mov #label, r1 or r1, reg
or label, reg	mov label, r1 or r1, reg
or $label, reg	mov $label, r1 or r1, reg

注 機械語命令の or 命令は，オペランドにイミーディエトをとりません。

[フラグ]

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

ori

論理和を行います。（Or Immediate）

［指定形式］

- ori imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値と，第 2 オペランドに指定したレジスタ値の論理和をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の ori 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

（b）!label，または %label を持つ相対値式

`ori !label, reg1, reg2`	`ori !label, reg1, reg2`
`ori %label, reg1, reg2`	`ori %label, reg1, reg2`

（c）HIGHW，LOWW，または HIGHW1 を適用したもの

`ori imm16, reg1, reg2`	`ori imm16, reg1, reg2`

注 機械語命令の ori 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xFFFF）のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

(a) -16 ～ -1 の範囲の絶対値式

reg2 が r0 の場合

ori imm5, reg1, r0	mov imm5, r1 or reg1, r1

上記以外の場合

ori imm5, reg1, reg2	mov imm5, reg2 or reg1, reg2

(b) -32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

ori imm16, reg1, r0	movea imm16, r0, r1 or reg1, r1

上記以外の場合

ori imm16, reg1, reg2	movea imm16, r0, reg2 or reg1, reg2

(c) 上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

ori imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 or reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

ori imm, reg1, r0	movhi HIGHW(imm), r0, r1 or reg1, r1

上記以外の場合

ori imm, reg1, reg2	mov imm, reg2 or reg1, reg2

上記以外，ただし reg2 が r0 の場合

ori imm, reg1, r0	mov imm, r1 or reg1, r1

(d) sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

ori $label, reg1, r0	movea $label, r0, r1 or reg1, r1

上記以外の場合

ori $label, reg1, reg2	movea $label, r0, reg2 or reg1, reg2

(e) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

ori #label, reg1, r0	mov #label, r1 or reg1, r1
ori label, reg1, r0	mov label, r1 or reg1, r1
ori $label, reg1, r0	mov $label, r1 or reg1, r1

上記以外の場合

ori #label, reg1, reg2	mov #label, reg2 or reg1, reg2
ori label, reg1, reg2	mov label, reg2 or reg1, reg2
ori $label, reg1, reg2	mov $label, reg2 or reg1, reg2

[フラグ]

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

xor

排他的論理和を行います。（Exclusive Or）

［指定形式］

- xor reg1, reg2
- xor imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “xor reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “xor imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “xor reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の xor 命令が 1 つ生成されます。
- “xor imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`xor 0, reg`	`xor r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`xor imm16, reg`	`xori imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`xor imm5, reg`	`mov imm5, r1` `xor r1, reg`

(d) -32768 ～ -17 の範囲の絶対値式

xor imm16, reg	movea imm16, r0, r1 xor r1, reg

(e) 上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

xor imm, reg	movhi HIGHW(imm), r0, r1 xor r1, reg

上記以外の場合

xor imm, reg	mov imm, r1 xor r1, reg

(f) sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

xor $label, reg	movea $label, r0, r1 xor r1, reg

(g) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

xor #label, reg	mov #label, r1 xor r1, reg
xor label, reg	mov label, r1 xor r1, reg
xor $label, reg	mov $label, r1 xor r1, reg

注 機械語命令の xor 命令は，オペランドにイミーディエトをとりません。

[フラグ]

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

xori

排他的論理和を行います。（Exclusive Or Immediate）

［指定形式］

- xori imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値と，第 2 オペランドに指定したレジスタ値の排他的論理和をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の xori 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

xori imm16, reg1, reg2	xori imm16, reg1, reg2

（b）!label，または %label を持つ相対値式

xori !label, reg1, reg2	xori !label, reg1, reg2
xori %label, reg1, reg2	xori %label, reg1, reg2

（c）HIGHW，LOWW，または HIGHW1 を適用したもの

xori imm16, reg1, reg2	xori imm16, reg1, reg2

注 機械語命令の xori 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xFFFF）のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

(a) -16 ～ -1 の範囲の絶対値式

reg2 が r0 の場合

xori imm5, reg1, r0	mov imm5, r1 xor reg1, r1

上記以外の場合

xori imm5, reg1, reg2	mov imm5, reg2 xor reg1, reg2

(b) -32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

xori imm16, reg1, r0	movea imm16, r0, r1 xor reg1, r1

上記以外の場合

xori imm16, reg1, reg2	movea imm16, r0, reg2 xor reg1, reg2

(c) 上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

xori imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 xor reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

xori imm, reg1, r0	movhi HIGHW(imm), r0, r1 xor reg1, r1

上記以外の場合

xori imm, reg1, reg2	mov imm, reg2 xor reg1, reg2

上記以外，ただし reg2 が r0 の場合

xori imm, reg1, r0	mov imm, r1 xor reg1, r1

(d) sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

xori $label, reg1, r0	movea $label, r0, r1 xor reg1, r1

上記以外の場合

xori $label, reg1, reg2	movea $label, r0, reg2 xor reg1, reg2

(e) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

xori #label, reg1, r0	mov #label, r1 xor reg1, r1
xori label, reg1, r0	mov label, r1 xor reg1, r1
xori $label, reg1, r0	mov $label, r1 xor reg1, r1

上記以外の場合

xori #label, reg1, reg2	mov #label, reg2 xor reg1, reg2
xori label, reg1, reg2	mov label, reg2 xor reg1, reg2
xori $label, reg1, reg2	mov $label, reg2 xor reg1, reg2

[フラグ]

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

and

論理積を行います。（And）

［指定形式］

- and reg1, reg2
- and imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “and reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “and imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “and reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の and 命令が 1 つ生成されます。
- “and imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`and 0, reg`	`and r0, reg`

（b）1 ～ 65535 の範囲の絶対値式

`and imm16, reg`	`andi imm16, reg, reg`

（c）-16 ～ -1 の範囲の絶対値式

`and imm5, reg`	`mov imm5, r1` `and r1, reg`

（d）-32768 ～ -17 の範囲の絶対値式

and imm16, reg	movea imm16, r0, r1 and r1, reg

（e）上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

and imm, reg	movhi HIGHW(imm), r0, r1 and r1, reg

上記以外の場合

and imm, reg	mov imm, r1 and r1, reg

（f）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

and $label, reg	movea $label, r0, r1 and r1, reg

（g）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

and #label, reg	mov #label, r1 and r1, reg
and label, reg	mov label, r1 and r1, reg
and $label, reg	mov $label, r1 and r1, reg

注 機械語命令の and 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

andi

論理積を行います。（And Immediate）

［指定形式］

- andi imm, reg1, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

［機能］

第 1 オペランドに指定した絶対値式，相対値式，あるいは，HIGHW，LOWW，または HIGHW1 を適用した値と，第 2 オペランドに指定したレジスタ値の論理積をとり，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

- imm に次のものを指定した場合，アセンブラでは，機械語命令の andi 命令[注]が 1 つ生成されます。

（a）0 ～ 65535 の範囲の絶対値式

andi imm16, reg1, reg2	andi imm16, reg1, reg2

（b）!label，または %label を持つ相対値式

andi !label, reg1, reg2	andi !label, reg1, reg2
andi %label, reg1, reg2	andi %label, reg1, reg2

（c）HIGHW，LOWW，または HIGHW1 を適用したもの

andi imm16, reg1, reg2	andi imm16, reg1, reg2

注 機械語命令の andi 命令は，第 1 オペランドに 0 ～ 65535（0 ～ 0xFFFF）のイミーディエトをとります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。

(a) -16 ～ -1 の範囲の絶対値式

reg2 が r0 の場合

andi imm5, reg1, r0	mov imm5, r1 and reg1, r1

上記以外の場合

andi imm5, reg1, reg2	mov imm5, reg2 and reg1, reg2

(b) -32768 ～ -17 の範囲の絶対値式

reg2 が r0 の場合

andi imm16, reg1, r0	movea imm16, r0, r1 and reg1, r1

上記以外の場合

andi imm16, reg1, reg2	movea imm16, r0, reg2 and reg1, reg2

(c) 上記の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

andi imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 and reg1, reg2

imm の値の下位 16 ビットがすべて 0，ただし reg2 が r0 の場合

andi imm, reg1, r0	movhi HIGHW(imm), r0, r1 and reg1, r1

上記以外の場合

andi imm, reg1, reg2	mov imm, reg2 and reg1, reg2

上記以外，ただし reg2 が r0 の場合

andi imm, reg1, reg2	mov imm, r1 and reg1, r1

(d) sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

reg2 が r0 の場合

andi $label, reg1, r0	movea $label, r0, r1 and reg1, r1

上記以外の場合

andi $label, reg1, reg2	movea $label, r0, reg2 and reg1, reg2

(e) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

reg2 が r0 の場合

andi #label, reg1, r0	mov #label, r1 and reg1, r1
andi label, reg1, r0	mov label, r1 and reg1, r1
andi $label, reg1, r0	mov $label, r1 and reg1, r1

上記以外の場合

andi #label, reg1, reg2	mov #label, reg2 and reg1, reg2
andi label, reg1, reg2	mov label, reg2 and reg1, reg2
andi $label, reg1, reg2	mov $label, reg2 and reg1, reg2

[フラグ]

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

not

論理否定（1 の補数をとる）を行います。（Not）

［指定形式］

- not reg1, reg2
- not imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

- “not reg1, reg2”の形式

  第 1 オペランドに指定したレジスタ値の論理否定（1 の補数）をとり，結果を第 2 オペランドに指定したレジスタに格納します。

- “not imm, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値の論理否定（1 の補数）をとり，結果を第 2 オペランドに指定したレジスタに格納します。

［詳細説明］

- “not reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の not 命令が 1 つ生成されます。
- “not imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`not 0, reg`	`not r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`not imm5, reg`	`mov imm5, r1` `not r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`not imm16, reg`	`movea imm16, r0, r1` `not r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

not imm, reg	movhi HIGHW(imm), r0, r1 not r1, reg

上記以外の場合

not imm, reg	mov imm, r1 not r1, reg

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

not $label, reg	movea $label, r0, r1 not r1, reg

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

not #label, reg	mov #label, r1 not r1, reg
not label, reg	mov label, r1 not r1, reg
not $label, reg	mov $label, r1 not r1, reg

注 機械語命令の not 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

shr

論理右シフトを行います。（Shift Logical Right）

［指定形式］

- shr reg1, reg2
- shr imm5, reg2
- shr reg1, reg2, reg3【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“shr reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shr imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shr reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に論理シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の shr 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

［注意事項］

- “shr imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注が用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

注　機械語命令の shr 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1F）のイミーディエトをとります。

sar

算術右シフトを行います。（Shift Arithmetic Right）

［指定形式］

- sar reg1, reg2
- sar imm5, reg2
- sar reg1, reg2, reg3【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“sar reg1, reg2”の形式

第 1 オペランドに指定したレジスタの値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“sar imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“sar reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を右に算術シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の sar 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

［注意事項］

- “sar imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注が用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

注　機械語命令の sar 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1F）のイミーディエトをとります。

shl

論理左シフトを行います。（Shift Logical Left）

［指定形式］

- shl reg1, reg2
- shl imm5, reg2
- shl reg1, reg2, reg3【V850E2】

imm5 に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

-“shl reg1, reg2”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shl imm5, reg2”の形式

第 1 オペランドに指定した絶対値式の値で示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 2 オペランドに指定したレジスタに格納します。

-“shl reg1, reg2, reg3”の形式

第 1 オペランドに指定したレジスタ値の下位 5 ビットで示されるビット数分，第 2 オペランドに指定したレジスタ値を左に論理シフトし，結果を第 3 オペランドに指定したレジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の shl 命令が 1 つ生成されます。

［フラグ］

CY	最後にシフト・アウトしたビットの値が 1 の場合 1，そうでない場合 0 （指定したビット数が 0 の場合 0）
OV	0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

［注意事項］

- “shl imm5, reg2”の形式で imm5 に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット[注]が用いられてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

注　機械語命令の shl 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1F）のイミーディエトをとります。

sxb

バイト・データの符号拡張を行います。（Sign Extend Byte）

［指定形式］

- sxb　reg

［機能］

第 1 オペランドに指定したレジスタの下位 1 バイトのデータを，ワード長に符号拡張します。

［詳細説明］

アセンブラでは，機械語命令の sxb 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

sxh

2 バイト・データの符号拡張を行います。（Sign Extend Half-word）

［指定形式］

- sxh reg

［機能］

第 1 オペランドに指定したレジスタの下位 2 バイトのデータを，ワード長に符号拡張します。

［詳細説明］

アセンブラでは，機械語命令の sxh 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

zxb

バイト・データのゼロ拡張を行います。（Zero Extend Byte）

［指定形式］

- zxb reg

［機能］

第 1 オペランドに指定したレジスタの下位 1 バイトのデータを，ワード長にゼロ拡張します。

［詳細説明］

アセンブラでは，機械語命令の zxb 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

zxh

2 バイト・データのゼロ拡張を行います。（Zero Extend Half-word）

［指定形式］

- zxh reg

［機能］

第 1 オペランドに指定したレジスタの下位 2 バイトのデータを，ワード長にゼロ拡張します。

［詳細説明］

アセンブラでは，機械語命令の zxh 命令が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

bsh

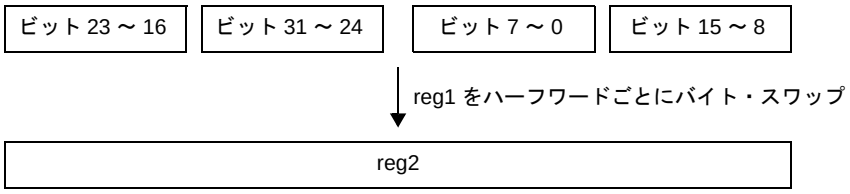
ハーフワード・データのバイト・スワップを行います。（Byte Swap Half-word）

［指定形式］

- bsh reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，ハーフワード単位でバイト・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

アセンブラでは，機械語命令の bsh 命令が 1 つ生成されます。

［フラグ］

CY	レジスタの下位ハーフワード中の 1 つ以上のバイトが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果の下位ハーフ・ワード・データが 0 になった場合 1，そうでない場合 0
SAT	−

bsw

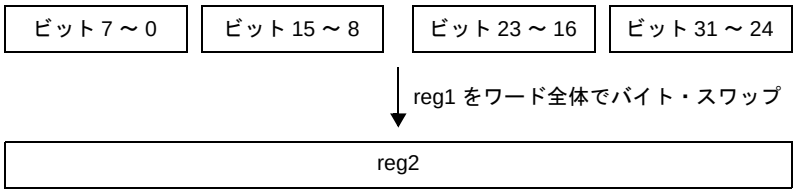
ワード・データのバイト・スワップを行います。（Byte Swap Word）

［指定形式］

- bsw reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，バイト・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

アセンブラでは，機械語命令の bsw 命令が 1 つ生成されます。

［フラグ］

CY	レジスタのワード中の 1 つ以上のバイトが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果のワード・データが 0 になった場合 1，そうでない場合 0
SAT	−

hsh

ハーフワード・データのハーフワード・スワップを行います。（Half-word Swap Half-word）【V850E2】

［指定形式］

- hsh　reg2, reg3

［機能］

第 1 オペランドに指定したレジスタ値を第 2 オペランドに指定したレジスタに格納し，フラグの判定結果を PSW レジスタに格納します。

［詳細説明］

アセンブラでは，機械語命令の hsh 命令が 1 つ生成されます。

［フラグ］

CY	演算結果の下位ハーフワードが 0 の場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果の下位ハーフワードが 0 の場合 1，そうでない場合 0
SAT	－

hsw

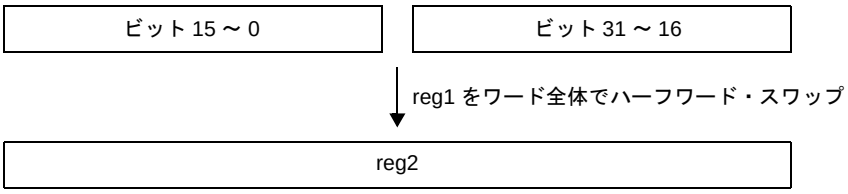
ワード・データのハーフワード・スワップを行います。（Half-word Swap Word）

［指定形式］

- hsw reg1, reg2

［機能］

第 1 オペランドに指定したレジスタ値を，ハーフワード・スワップして，第 2 オペランドに指定したレジスタに格納します。



［詳細説明］

アセンブラでは，機械語命令の hsw 命令が 1 つ生成されます。

［フラグ］

CY	演算結果のワード・データ中に 0 のハーフワードが 1 つ以上含まれる場合 1，そうでない場合 0
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	演算結果のワード・データが 0 になった場合 1，そうでない場合 0
SAT	—

tst

テストを行います。（Test）

［指定形式］

- tst reg1, reg2
- tst imm, reg2

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式

［機能］

-“tst reg1, reg2”の形式

第 2 オペランドに指定したレジスタ値と，第 1 オペランドに指定したレジスタ値の論理積をとり，結果は格納せず，フラグのみを設定します。

-“tst imm, reg2”の形式

第 2 オペランドに指定したレジスタ値と，第 1 オペランドに指定した絶対値式，または相対値式の値の論理積をとり，結果は格納せず，フラグのみを設定します。

［詳細説明］

-“tst reg1, reg2”の形式の命令に対し，アセンブラでは，機械語命令の tst 命令が 1 つ生成されます。

-“tst imm, reg2”の形式で imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，1 つ，または複数個の機械語命令が生成されます。**注**

（a）0

`tst 0, reg`	`tst r0, reg`

（b）0 以外の -16 ～ +15 の範囲の絶対値式

`tst imm5, reg`	`mov imm5, r1` `tst r1, reg`

（c）-16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

`tst imm16, reg`	`movea imm16, r0, r1` `tst r1, reg`

（d）imm に -32768 ～ +32767 の範囲を越える絶対値式

imm の値の下位 16 ビットがすべて 0 の場合

`tst imm, reg`	`movhi HIGHW(imm), r0, r1` `tst r1, reg`

上記以外の場合

`tst imm, reg`	`mov imm, r1` `tst r1, reg`

（e）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`tst $label, reg`	`movea $label, r0, r1` `tst r1, reg`

（f）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`tst #label, reg`	`mov #label, r1` `tst r1, reg`
`tst label, reg`	`mov label, r1` `tst r1, reg`
`tst $label, reg`	`mov $label, r1` `tst r1, reg`

注 機械語命令の tst 命令は，オペランドにイミーディエトをとりません。

［フラグ］

CY	—
OV	0
S	演算結果のワード・データの MSB が 1 の場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

sch0l

MSB 側からのビット（0）検索を行います。（SearchZero from Left）【V850E2】

［指定形式］

- sch0l reg1, reg2

［機能］

第 1 オペランドに指定したレジスタのワード・データを左側（MSB 側）から検索し，最初にビット（0）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します（たとえば，第 1 オペランドに指定したレジスタのビット 31 が 0 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（0）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（0）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

アセンブラでは，機械語命令の sch0l 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（0）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（0）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch0r

LSB 側からのビット（0）検索を行います。（Search Zero from Right）【V850E2】

［指定形式］

- sch0r reg1, reg2

［機能］

第 1 オペランドに指定したレジスタのワード・データを右側（LSB 側）から検索し，最初にビット（0）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します（たとえば，第 1 オペランドに指定したレジスタのビット 0 が 0 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（0）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（0）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

アセンブラでは，機械語命令の sch0r 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（0）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（0）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch1l

MSB 側からのビット（1）検索を行います。（Search One from Left）【V850E2】

［指定形式］

- sch1l reg1, reg2

［機能］

第 1 オペランドに指定したレジスタのワード・データを左側（MSB 側）から検索し，最初にビット（1）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で書き込みます（たとえば，第 1 オペランドに指定したレジスタのビット 31 が 1 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（1）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（1）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

アセンブラでは，機械語命令の sch1l 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（1）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（1）が見つからなかった場合 1，そうでない場合 0
SAT	—

sch1r

LSB 側からのビット（1）検索を行います。（Search One from Right）【V850E2】

［指定形式］

- sch1r reg2, reg3

［機能］

第 1 オペランドに指定したレジスタのワード・データを右側（LSB 側）から検索し，最初にビット（1）が見つかった位置を第 2 オペランドに指定したレジスタに 16 進数で格納します（たとえば，第 1 オペランドに指定したレジスタのビット 0 が 1 の場合は，第 2 オペランドに指定したレジスタに 01H を格納します）。

ビット（1）が見つからなかった場合は，第 2 オペランドに指定したレジスタに 0 を書き込み，同時に Z フラグをセット（1）します。最後にビット（1）が見つかった場合は，CY フラグをセット（1）します。

［詳細説明］

アセンブラでは，機械語命令の sch1r 命令が 1 つ生成されます。

［フラグ］

CY	最後にビット（1）が見つかった場合 1，そうでない場合 0
OV	0
S	0
Z	ビット（1）が見つからなかった場合 1，そうでない場合 0
SAT	—

4. 7. 10　分岐命令

この項では，分岐命令について説明します。次に，この項において説明する命令を示します。

表 4―39　分岐命令

命令	意味
jmp	無条件分岐
jmp32	無条件分岐【V850E2】
jr	無条件分岐（PC 相対）
jr22	無条件分岐（PC 相対）【V850E2】
jr32	無条件分岐（PC 相対）【V850E2】
jcnd	条件分岐
jarl	ジャンプ・アンド・レジスタ・リンク
jarl22	ジャンプ・アンド・レジスタ・リンク【V850E2】
jarl32	ジャンプ・アンド・レジスタ・リンク【V850E2】

jmp

無条件分岐を行います。（Jump）

［指定形式］

- jmp [reg]
- jmp addr
- jmp disp32[reg]【V850E2】

addr に指定できるものを次に示します。

- ラベルの絶対アドレス参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

［機能］

- “jmp [reg]”の形式

  オペランドに指定したレジスタ値が示すアドレスに制御を移します。
- “jmp addr”の形式

  オペランドに指定した相対値式の値が示すアドレスに制御を移します。
- “jmp disp32[reg]”の形式

  オペランドに指定したディスプレースメントとレジスタの内容を加算して得たアドレスに制御を移します。

［詳細説明］

- “jmp [reg]”の形式の命令に対し，アセンブラでは，機械語命令の jmp 命令が 1 つ生成されます。
- “jmp addr”の形式の命令に対し，V850E1 動作時には，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

`jmp #label`	`mov #label, r1` `jmp [r1]`

- “jmp addr”の形式の命令に対し，V850E2 動作時には，アセンブラでは，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。
- “jmp disp32[reg]”の形式の命令に対し，アセンブラでは，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- “jmp addr”の形式において，addr にラベルの絶対アドレス参照を持つ相対値式以外のものを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550224 : jmp 命令に対し、絶対アドレス参照（string）以外のものが指定されています。

jmp32

無条件分岐を行います。（Jump）【V850E2】

［指定形式］

- jmp32 disp32[reg]
- jmp32 addr

addr に指定できるものを次に示します。

- ラベルの絶対アドレス参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

［機能］

- “jmp32 disp32[reg]”の形式

  オペランドに指定したディスプレースメントとレジスタの内容を加算して得たアドレスに制御を移します。

- “jmp32 addr”の形式

  オペランドに指定した相対値式の値が示すアドレスに制御を移します。

［詳細説明］

アセンブラでは，機械語命令の jmp 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- “jmp32 addr”の形式において，addr にラベルの絶対アドレス参照を持つ相対値式以外のものを指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550224 : jmp 命令に対し、絶対アドレス参照（string）以外のものが指定されています。

jr

無条件分岐（PC 相対）を行います。（Jump Relative）

［指定形式］

- jr disp22
- jr disp32【V850E2】

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

- “jr disp22”の形式

  オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

- “jr disp32”の形式

  オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

- “jr disp22”の形式の命令に対し，disp22 に次のものを指定した場合，アセンブラで機械語命令の jr 命令[注]が 1 つ生成されます。

  （a）-2097152 ～ +2097151 の範囲の絶対値式

  （b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

  （c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jr は，ディスプレースメントに -2097152 ～ +2097151（0xFE00000 ～ 0x1FFFFF）の範囲のイミーディエトをとります。

- “jr disp32”の形式の命令に対し，アセンブラでは，機械語命令の jr 命令（6 バイト長命令）が 1 つ生成されます。

[フラグ]

CY	–
OV	–
S	–
Z	–
SAT	–

[注意事項]

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。
```

- disp22/disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550226 : 奇数のディスプレースメントが指定されています。
```

- アセンブラオプション -Xfar_jump を指定しない場合に，disp32 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。
```

jr22

無条件分岐（PC 相対）を行います。（Jump Relative）【V850E2】

［指定形式］

- jr22 disp22

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

- disp22 に次のものを指定した場合，アセンブラでは，機械語命令の jr 命令[注]が 1 つ生成されます。

（a）-2097152 ～ +2097151 の範囲の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jr は，ディスプレースメントに -2097152 ～ +2097151（0xFE00000 ～ 0x1FFFFF）の範囲のイミーディエトをとります。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550226 : 奇数のディスプレースメントが指定されています。

jr32

無条件分岐（PC 相対）を行います。（Jump Relative）【V850E2】

［指定形式］

- jr32 disp32

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。

［詳細説明］

アセンブラでは，機械語命令の jr 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550226 : 奇数のディスプレースメントが指定されています。

jcnd

条件分岐を行います。（Jump on Condition）

［指定形式］

- j*cnd* disp22

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

cnd 部分の文字列で示されるフラグ状態（「表 4―40 jcnd 命令」を参照）と，現在のフラグ状態を比較し，一致した場合は，オペランドに指定した絶対値式，または相対値式の値と現在のプログラム・カウンタ（PC）の値を加算したアドレスに制御を移します。**注**

注 jbr 以外の j*cnd* 命令に対しては，b*cnd* というニモニックを，jbr 命令に対しては br という機械語命令を用いることもできます（機能に違いはありません）。

表 4―40 j*cnd* 命令

命令	フラグ状態	フラグ状態の意味
jgt	((S xor OV) or Z) = 0	Greater than（signed）
jge	(S xor OV) = 0	Greater than or equal（signed）
jlt	(S xor OV) = 1	Less than（signed）
jle	((S xor OV) or Z) = 1	Less than or equal（signed）
jh	(CY or Z) = 0	Higher（Greater than）
jnl	CY = 0	Not lower（Greater than or equal）
jl	CY = 1	Lower（Less than）
jnh	(CY or Z) = 1	Not higher（Less than or equal）
je	Z = 1	Equal
jne	Z = 0	Not equal
jv	OV = 1	Overflow
jnv	OV = 0	No overflow
jn	S = 1	Negative
jp	S = 0	Positive
jc	CY = 1	Carry
jnc	CY = 0	No carry
jz	Z = 1	Zero

命令	フラグ状態	フラグ状態の意味
jnz	Z = 0	Not zero
jbr	―	Always（無条件）
jsa	SAT = 1	Saturated

［詳細説明］

- disp22 に次のものを指定した場合，アセンブラでは，機械語命令の b*cnd* 命令[注]が 1 つ生成されます。

（a）-256 ～ +255 の範囲の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-256 ～ +255 の範囲の相対値式

j*cnd* disp9	b*cnd* disp9

注 機械語命令の b*cnd* は，ディスプレースメントに -256 ～ +255（0xFFFFFF00 ～ 0xFF）の範囲のイミーディエトをとります。

- disp22 に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数の機械語命令が生成されます。

（a）-256 ～ +255 の範囲を越え -2097150 ～ +2097153 の範囲[注 1] の絶対値式

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-256 ～ +255 の範囲を越え -2097150 ～ +2097153 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

jbr disp22	jr disp22
jsa disp22	bsa Label1 br Label2 Label1: jr disp22 - 4 Label2:
j*cnd* disp22	bn*cnd* Label[注 2] jr disp22 - 2 Label:

注 1. -2097150 ～ +2097153 の範囲は，jbr，および jsa 命令以外の場合の範囲で，jbr 命令の場合は -2097152 ～ +2097151，jsa 命令の場合は -2097148 ～ +2097155 となります。

2. bn*cnd* は，たとえば，bz に対する bnz.bgt に対する ble というように，逆の条件で分岐を行う命令を示しています。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097150 ～ +2097153 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097150 ～ +2097153 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。
```

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550226 : 奇数のディスプレースメントが指定されています。
```

jarl

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）

［指定形式］

- jarl disp22, reg2
- jarl disp32, reg1【V850E2】

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

- “jarl disp22, reg2”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

- “jarl disp32, reg1”の形式

  第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

- “jarl disp22, reg2”の形式の命令に対し，disp22 に次のものを指定した場合，アセンブラでは，機械語命令の jarl 命令[注]が 1 つ生成されます。

  （a）-2097152 ～ +2097151 の範囲の絶対値

  （b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

  （c）本命令と同じファイル内に定義を持たないか，同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jarl は，オペランドに -2097152 ～ +2097151（0xFE00000 ～ 0x1FFFFF）の範囲のイミーディエトをとります。

- “jarl disp32, reg1”の形式の命令に対し，アセンブラでは，機械語命令の jarl 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。
```

- disp22/disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550226 : 奇数のディスプレースメントが指定されています。
```

- アセンブラオプション -Xfar_jump を指定しない場合に，disp32 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。
```

- reg1/reg2 に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。
```

jarl22

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）【V850E2】

［指定形式］

- jarl22 disp22, reg1

disp22 に指定できるものを次に示します。

- 22 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

- disp22 に次のものを指定した場合，アセンブラでは，機械語命令の jarl 命令[注]が 1 つ生成されます。

（a）-2097152 ～ +2097151 の範囲の絶対値

（b）本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち，-2097152 ～ +2097151 の範囲の相対値式

（c）本命令と同じファイル内に定義を持たないか，同じセクションに定義を持たないラベルの PC オフセット参照を持つ相対値式

注 機械語命令の jarl は，オペランドに -2097152 ～ +2097151（0xFE00000 ～ 0x1FFFFF）の範囲のイミーディエトをとります。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp22 に，-2097152 ～ +2097151 の範囲を越える絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち -2097152 ～ +2097151 の範囲を越える相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550230 : ディスプレースメントとして指定された値が指定可能な値の範囲を越えています。

- disp22 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550226 : 奇数のディスプレースメントが指定されています。

- reg2 に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

jarl32

ジャンプ・アンド・レジスタ・リンクを行います。（Jump and Register Link）【V850E2】

［指定形式］

- jarl32 disp32, reg1

disp32 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- ラベルの PC オフセット参照を持つ相対値式

［機能］

第 1 オペランドに指定した絶対値式，または相対値式の値と，現在のプログラム・カウンタ（PC）値を加算したアドレスに制御を移します。なお，戻りアドレスは，第 2 オペランドに指定したレジスタに格納されます。

［詳細説明］

アセンブラでは，機械語命令の jarl 命令（6 バイト長命令）が 1 つ生成されます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- disp32 に，奇数値を持つ絶対値式，または本命令と同じファイル内の同じセクションに定義を持つラベルの PC オフセット参照を持ち奇数値を持つ相対値式を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

```
E0550226 : 奇数のディスプレースメントが指定されています。
```

- reg1 に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

```
E0550240:V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。
```

4. 7. 11　ビット操作命令

この項では，ビット操作命令について説明します。次に，この項において説明する命令を示します。

表 4―41　ビット操作命令

命令	意味
set1	ビット・セット
clr1	ビット・クリア
not1	ビット・ノット
tst1	ビット・テスト

set1

ビット・セットを行います。（Set Bit）

［指定形式］

- set1 bit#3, disp[reg1]
- set1 reg2, [reg1]
- set1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

注意 “set1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

-“set1 bit#3, disp[reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットをセットします。指定したビット以外は影響を受けません。

-“set1 reg2, [reg1]”の形式

第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットをセットします。指定したビット以外は影響を受けません。

-“set1 BITIO”の形式

第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットをセットします。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の set1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`set1 #bit3, $label[reg1]`	`set1 #bit3, $label[reg1]`

(c) !label，または %label を持つ相対値式

`set1 #bit3, !label[reg1]`	`set1 #bit3, !label[reg1]`
`set1 #bit3, %label[reg1]`	`set1 #bit3, %label[reg1]`

(d) HIGHW，LOWW，または HIGHW1 を適用したもの

`set1 #bit3, disp16[reg1]`	`set1 #bit3, disp16[reg1]`

(e) デバイス・ファイルで定義された内部レジスタ名

`set1 reg2, `*レジスタ名*`[reg1]`	`set1 reg2, `*レジスタ名*`[reg1]`

注 機械語命令の set1 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

(a) -32768 ～ +32767 の範囲を越える絶対値式

`set1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `set1 #bit3, LOWW(disp)[r1]`

(b) #label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`set1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `set1 #bit3, LOWW(#label)[r1]`
`set1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `set1 #bit3, LOWW(label)[r1]`
`set1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `set1 #bit3, LOWW($label)[r1]`

- disp を省略した場合，アセンブラでは，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	–

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

clr1

ビット・クリアを行います。（Clear Bit）

［指定形式］

- clr1 bit#3, disp[reg1]
- clr1 reg2, [reg1]
- clr1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

注意 “clr1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

-“clr1 bit#3, disp[reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットをクリアします。指定したビット以外は影響を受けません。

-“clr1 reg2, [reg1]”の形式

第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットをクリアします。指定したビット以外は影響を受けません。

-“clr1 BITIO”の形式

第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットをクリアします。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の clr1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`clr1 #bit3, disp16[reg1]`	`clr1 #bit3, disp16[reg1]`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`clr1 #bit3, $label[reg1]`	`clr1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

clr1 #bit3, !label[reg1]	clr1 #bit3, !label[reg1]
clr1 #bit3, %label[reg1]	clr1 #bit3, %label[reg1]

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

clr1 #bit3, disp16[reg1]	clr1 #bit3, disp16[reg1]

（e）デバイス・ファイルで定義された内部レジスタ名

clr1 reg2, *レジスタ名*[reg1]	clr1 reg2, *レジスタ名*[reg1]

注 機械語命令の clr1 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

clr1 #bit3, disp[reg1]	movhi HIGHW1(disp), reg1, r1 clr1 #bit3, LOWW(disp)[r1]

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

clr1 #bit3, #label[reg1]	movhi HIGHW1(#label), reg1, r1 clr1 #bit3, LOWW(#label)[r1]
clr1 #bit3, label[reg1]	movhi HIGHW1(label), reg1, r1 clr1 #bit3, LOWW(label)[r1]
clr1 #bit3, $label[reg1]	movhi HIGHW1($label), reg1, r1 clr1 #bit3, LOWW($label)[r1]

- disp を省略した場合，アセンブラでは，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分を省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	–

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

not1

ビット・ノットを行います。（Not Bit）

［指定形式］

- not1 bit#3, disp[reg1]
- not1 reg2, [reg1]
- not1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

注意 “not1 reg2, [reg1]”の形式では disp は指定できません。

［機能］

- “not1 bit#3, disp[reg1]”の形式

  第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットを反転（0→1，1→0）します。指定したビット以外は影響を受けません。

- “not1 reg2, [reg1]”の形式

  第 2 オペランドのレジスタ値で指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットを反転（0→1，1→0）します。指定したビット以外は影響を受けません。

- “not1 BITIO”の形式

  第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットを反転（0→1，1→0）します。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の not1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`not1 #bit3, disp16[reg1]`	`not1 #bit3, disp16[reg1]`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`not1 #bit3, $label[reg1]`	`not1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

not1 #bit3, !label[reg1]	not1 #bit3, !label[reg1]
not1 #bit3, %label[reg1]	not1 #bit3, %label[reg1]

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

not1 #bit3, disp16[reg1]	not1 #bit3, disp16[reg1]

（e）デバイス・ファイルで定義された内部レジスタ名

not1 reg2, *レジスタ名*[reg1]	not1 reg2, *レジスタ名*[reg1]

注 機械語命令の not1 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

not1 #bit3, disp[reg1]	movhi HIGHW1(disp), reg1, r1 not1 #bit3, LOWW(disp)[r1]

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

not1 #bit3, #label[reg1]	movhi HIGHW1(#label), reg1, r1 not1 #bit3, LOWW(#label)[r1]
not1 #bit3, label[reg1]	movhi HIGHW1(label), reg1, r1 not1 #bit3, LOWW(label)[r1]
not1 #bit3, $label[reg1]	movhi HIGHW1($label), reg1, r1 not1 #bit3, LOWW($label)[r1]

- disp を省略した場合，アセンブラでは，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0 **注**
SAT	–

注　Z フラグの値は，この命令実行前の該当ビットの値を示しています。この命令実行後を示すものではありません。

tst1

ビット・テストを行います。（Test Bit）

［指定形式］

- tst1 bit#3, disp[reg1]
- tst1 reg2, [reg1]
- tst1 BITIO

disp に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式
- 相対値式
- 上記のものに HIGHW，LOWW，または HIGHW1 を適用したもの

注意 “tst1 bit#3, disp[reg1]”の形式では disp は指定できません。

［機能］

-“tst1 bit#3, disp[reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドに指定したビットの値に従い，フラグのみを設定します。第 2 オペランドの値，および指定したビットは変更されません。

-“tst1 reg2, [reg1]”の形式

第 2 オペランドで指定したアドレスが示すデータの，第 1 オペランドで指定したレジスタ値の下位 3 ビットが示すビットの値に従い，フラグのみを設定します。第 2 オペランドの値，および指定したビットは変更されません。

-“tst1 BITIO”の形式

第 1 オペランドで指定したアドレスが示すデータにおいて，周辺 I/O レジスタのビット名（デバイス・ファイルで定義されている予約語のみ）で指定したビットの値に従い，フラグのみを設定します。周辺 I/O レジスタのビットの値は変更されません。

［詳細説明］

- disp に次のものを指定した場合，アセンブラでは，機械語命令の tst1 命令[注]が 1 つ生成されます。

（a）-32768 ～ +32767 の範囲の絶対値式

`tst1 #bit3, disp16[reg1]`	`tst1 #bit3, disp16[reg1]`

（b）sdata/sbss 属性セクションに定義を持つラベルの $label を持つ相対値式

`tst1 #bit3, $label[reg1]`	`tst1 #bit3, $label[reg1]`

（c）!label，または %label を持つ相対値式

`tst1 #bit3, !label[reg1]`	`tst1 #bit3, !label[reg1]`
`tst1 #bit3, %label[reg1]`	`tst1 #bit3, %label[reg1]`

（d）HIGHW，LOWW，または HIGHW1 を適用したもの

`tst1 #bit3, disp16[reg1]`	`tst1 #bit3, disp16[reg1]`

（e）デバイス・ファイルで定義された内部レジスタ名

`tst1 reg2, `*`レジスタ名`*`[reg1]`	`tst1 reg2, `*`レジスタ名`*`[reg1]`

注 機械語命令の tst1 命令は，ディスプレースメントに -32768 ～ +32767（0xFFFF8000 ～ 0x7FFF）の範囲のイミーディエトをとります。

- disp に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）-32768 ～ +32767 の範囲を越える絶対値式

`tst1 #bit3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `tst1 #bit3, LOWW(disp)[r1]`

（b）#label，または label を持つ相対値式，および sdata/sbss 属性セクションに定義を持たないラベルの $label を持つ相対値式

`tst1 #bit3, #label[reg1]`	`movhi HIGHW1(#label), reg1, r1` `tst1 #bit3, LOWW(#label)[r1]`
`tst1 #bit3, label[reg1]`	`movhi HIGHW1(label), reg1, r1` `tst1 #bit3, LOWW(label)[r1]`
`tst1 #bit3, $label[reg1]`	`movhi HIGHW1($label), reg1, r1` `tst1 #bit3, LOWW($label)[r1]`

- disp を省略した場合，アセンブラでは，0 が指定されたものとみなされます。
- disp に #label を持つ相対値式，または #label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。
- disp に $label を持つ相対値式，または $label を持つ相対値式に HIGHW，LOWW，または HIGHW1 を適用したものを指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[gp] が指定されたものとみなされます。
- disp にデバイス・ファイルで定義されている周辺 I/O レジスタ名を指定した場合，その後ろの [reg1] の部分が省略できます。ただし，省略した場合，アセンブラでは，[r0] が指定されたものとみなされます。

［フラグ］

CY	–
OV	–
S	–
Z	指定したビットが 0 の場合 1，1 の場合 0
SAT	–

4. 7. 12　スタック操作命令

この項では，スタック操作命令について説明します。次に，この項において説明する命令を示します。

表 4―42　スタック操作命令

命令	意味
push	スタック領域へのプッシュ（単一レジスタ）
pushm	スタック領域へのプッシュ（複数レジスタ）
pop	スタック領域からのポップ（単一レジスタ）
popm	スタック領域からのポップ（複数レジスタ）

push

スタック領域へのプッシュを行います。（Push）

［指定形式］

push　reg

［機能］

オペランドに指定したレジスタ値を，スタック領域にプッシュします。

［詳細説明］

- push 命令に対し，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

`push　reg`	`add　-4, sp` `st.w　reg, [sp]`

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意　命令展開が行われ，add 命令により設定されます。

pushm

スタック領域へのプッシュ（複数レジスタ）を行います。（Push Multiple）

［指定形式］

pushm reg1, reg2, …, regN

［機能］

オペランドに指定したレジスタ値を，スタック領域へプッシュします。なお，オペランドに指定可能なレジスタ数は，最大 32 個です。

［詳細説明］

- pushm 命令に対し，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

レジスタが 4 個以下の場合

pushm reg1, reg2, …, regN	add -4 * N, sp st.w regN, 4 * (N - 1)[sp] : st.w reg2, 4 * 1[sp] st.w reg1, 4 * 0[sp]

レジスタが 5 個以上の場合

pushm reg1, reg2, …, regN	addi -4 * N, sp, sp st.w regN, 4 * (N - 1)[sp] : st.w reg2, 4 * 1[sp] st.w reg1, 4 * 0[sp]

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	—

注意 命令展開が行われ，add/addi 命令により設定されます。

pop

スタック領域からのポップを行います。（Pop）

［指定形式］

pop reg

［機能］

オペランドに指定したレジスタ値を，スタック領域からポップします。

［詳細説明］

- pop 命令に対し，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

pop reg	ld.w [sp], reg add 4, sp

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意 命令展開が行われ，add 命令により設定されます。

popm

スタック領域からのポップ（複数レジスタ）を行います。（Pop Multiple）

［指定形式］

popm　reg1, reg2, …, regN

［機能］

オペランドに指定したレジスタ値を，指定した順にスタック領域からポップします。なお，オペランドに指定可能なレジスタ数は，最大 32 個です。

［詳細説明］

- popm 命令に対し，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

レジスタが 3 個以下の場合

popm　reg1, …, regN	ld.w　4 * 0[sp], reg1 : ld.w　4 * (N - 1)[sp], regN add　4 * N, sp

レジスタが 4 個以上の場合

popm　reg1, reg2, …, regN	ld.w　4 * 0[sp], reg1 ld.w　4 * 1[sp], reg2 : ld.w　4 * (N - 1)[sp], regN addi　4 * N, sp, sp

［フラグ］

CY	MSB（Most Significant Bit）からのキャリーを生じた場合 1，そうでない場合 0
OV	Integer-Overflow を生じた場合 1，そうでない場合 0
S	結果が負になった場合 1，そうでない場合 0
Z	結果が 0 になった場合 1，そうでない場合 0
SAT	−

注意　命令展開が行われ，add/addi 命令により設定されます。

4. 7. 13 特殊命令

この項では，特殊命令について説明します。次に，この項において説明する命令を示します。

表 4―43 特殊命令

命令	意味
ldsr	システム・レジスタへのロード
stsr	システム・レジスタの内容のストア
di	マスカブル割り込みの禁止
ei	マスカブル割り込みの許可
reti	トラップ，または割り込みルーチンからの復帰
eiret	EI レベル例外からの復帰【V850E2V3】
feret	FE レベル例外からの復帰【V850E2V3】
halt	プロセッサの停止
trap	ソフトウエア・トラップ
rmtrap	ランタイム・モニタ・トラップ【V850E2V3】
fetrap	FE レベル・ソフトウエア例外命令【V850E2V3】
nop	ノー・オペレーション
switch	テーブル参照分岐
callt	テーブル参照コール
ctret	callt からの復帰
caxi	比較と交換【V850E2V3】
rie	予約命令例外【V850E2V3】
syncm	メモリ同期化命令【V850E2V3】
syncp	パイプライン同期化命令【V850E2V3】
dbtrap	デバッグ・トラップ
dbret	デバッグ・トラップからの復帰
prepare	スタック・フレームの生成（関数の前処理）
dispose	スタック・フレームの削除（関数の後処理）
synce	例外同期化命令【V850E2V3】
syscall	システム・コール例外【V850E2V3】

V850E2V3 のインストラクション・セットをもつデバイスの命令の詳細については，V850E2V3 のインストラクション・セットをもつデバイス製品のユーザーズ・マニュアル，およびアーキテクチャ編を参照してください。

ldsr

システム・レジスタへのロードを行います。（Load System Register）

［指定形式］

- ldsr reg, regID

regID に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

第 1 オペランドに指定したレジスタ値を，第 2 オペランドに指定したシステム・レジスタ番号で示されるシステム・レジスタ注に格納します。

注 システム・レジスタに関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

注意 システム・レジスタにプログラム・ステータス・ワード（PSW）を指定した場合，各フラグには reg に対応したビットを設定します。

［注意事項］

- regID に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値を下位 5 ビット注を用いてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

注 機械語命令の ldsr 命令は，第 2 オペランドに 0 ～ 31（0x0 ～ 0x1F）の範囲のイミーディエトをとります。

- regID に予約レジスタ番号やアクセス禁止のレジスタ番号（ECR など）を指定した場合，またはデバッグ・モード時だけアクセス可能のレジスタ番号を指定した場合，次のメッセージが出力され，そのままアセンブルが続行されます。

W0550018 : ldsr 命令に指定した番号のシステム・レジスタはアクセス禁止です。

stsr

システム・レジスタの内容のストアを行います。（Store System Register）

［指定形式］

- stsr　regID, reg

regID に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

第 1 オペランドに指定したシステム・レジスタ番号で示されるシステム・レジスタ注の値を，第 2 オペランドに指定したレジスタに格納します。

注　システム・レジスタに関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－

［注意事項］

- regID に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット注を用いてアセンブルが続行されます。

```
W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。
```

注　機械語命令の stsr 命令は，第 1 オペランドに 0 ～ 31（0x0 ～ 0x1F）の範囲のイミーディエトをとります。

- regID に予約レジスタ番号を指定した場合，またはデバッグ・モード時だけアクセス可能のレジスタ番号を指定した場合，次のメッセージが出力され，そのままアセンブルが続行されます。

```
W0550018 : ldsr 命令に指定した番号のシステム・レジスタはアクセス禁止です。
```

di

マスカブル割り込みの禁止を行います。（Disable Interrupt）

［指定形式］

- di

［機能］

PSW 中の ID ビットに 1 を設定し，本命令実行中からマスカブル割り込みの受け付けを禁止します。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－
ID	1

ei

マスカブル割り込みの許可を行います。（Enable Interrupt）

［指定形式］

- ei

［機能］

PSW 中の ID ビットに 0 を設定し，次の命令よりマスカブル割り込みの受け付けを許可します。

［フラグ］

CY	－
OV	－
S	－
Z	－
SAT	－
ID	0

reti

トラップ，または割り込みルーチンからの復帰を行います。（Return from Trap or Interrupt）

［指定形式］

- reti

［機能］

トラップ，または割り込みルーチンから復帰します。**注**

注　機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

halt

停止します。（Halt）

［指定形式］

- halt

［機能］

プロセッサを停止し，HALT 状態に遷移します。なお，HALT 状態からの処理再開は，マスカブル割り込み，NMI，リセットにより行われます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

trap

ソフトウエア・トラップを発生させます。（Trap）

［指定形式］

- trap vector

vector に指定できるものを次に示します。

- 5 ビット幅までの値を持つ絶対値式

［機能］

ソフトウエア・トラップを発生させます。**注**

注 機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- vector に 0 ～ 31 の範囲を越える絶対値式を指定した場合，次のメッセージが出力され，指定した値の下位 5 ビット**注**を用いてアセンブルが続行されます。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

注 機械語命令の trap 命令は，オペランドに 0 ～ 31（0x0 ～ 0x1F）の範囲のイミーディエトをとります。

nop

ノー・オペレーションです。（No Operation）

［指定形式］

- nop

［機能］

何も行いません。命令シーケンス内に領域を確保したり，命令実行に遅延を挿入したりするために用いることができます。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

switch

テーブル参照分岐を行います。（Jump With Table Look Up）

［指定形式］

switch reg

［機能］

次の順に処理を行います。

(1) オペランドで指定した値を 1 ビット論理左シフトした値とテーブルの先頭アドレス（switch 命令の次のアドレス）とを加算し，テーブル・エントリ・アドレスを生成します。

(2) 生成したテーブル・エントリ・アドレスから符号付きハーフワード・データをロードします。

(3) ロードした値を 1 ビット論理左シフトし，ワード長に符号拡張したあと，テーブルの先頭アドレスを加算し，アドレスを生成します。

(4) 生成したアドレスへ分岐します。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

［注意事項］

- reg に r0 を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550239 : V850Ex コア指定時には、ソース・オペランドに r0 を指定することはできません。

callt

テーブル参照コールを行います。（Call With Table Look Up）

［指定形式］

- callt　imm6

imm6 に指定できるものを次に示します。

- 6 ビット幅までの値を持つ絶対値式

［機能］

次の順に処理を行います。**注**

（1）復帰 PC と PSW の値を CTPC と CTPSW に退避します。

（2）オペランドで指定された値を 1 ビット左シフトして CTBP（CALLT Base Pointer）からのオフセット値とし，CTBP 値と加算してテーブル・エントリ・アドレスを生成します。

（3）生成したテーブル・エントリ・アドレスから符号なしハーフワード・データをロードします。

（4）ロードした値と CTBP 値を加算してアドレスを生成します。

（5）生成したアドレスへ分岐します。

注　システム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

ctret

callt から復帰します。（Return from Callt）

［指定形式］

- ctret

［機能］

callt による分岐から復帰します。次の順に処理を行います。**注**

（1）復帰 PC と PSW を，CTPC と CTPSW から取り出します。

（2）取り出した値を PC と PSW に設定し，制御を移します。

注　システム・レジスタについては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

dbtrap

デバッグ・トラップを起こします。（Debug Trap）

［指定形式］

- dbtrap

［機能］

デバッグ・トラップを起こします。注

注　機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

dbret

デバッグ・トラップから復帰します。（Return from Debug Trap）

［指定形式］

- dbret

［機能］

デバッグ・トラップから復帰します。**注**

注　機能の詳細に関しては，各デバイスのユーザーズ・マニュアルを参照してください。

［フラグ］

CY	取り出した値
OV	取り出した値
S	取り出した値
Z	取り出した値
SAT	取り出した値

prepare

スタック・フレームの生成（関数の前処理）を行います。（Function Prepare）

［指定形式］

- prepare list, imm1
- prepare list, imm1, imm2
- prepare list, imm1, sp

imm1/imm2 に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

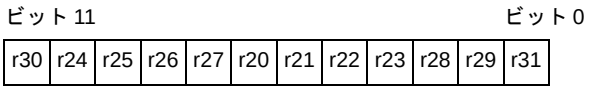
list は，prepare 命令でプッシュ可能な 12 本のレジスタを指定するものです。list に指定できるものを次に示します。

- レジスタ

  プッシュの対象となるレジスタ（r20 ～ r31）をカンマで区切って指定します。

- 12 ビット幅までの値を持つ定数式

  12 ビットと 12 本のレジスタとの対応は次のとおりです。



次の 2 つの指定は等価です。

`prepare r26, r29, r31, 0x10`	`prepare 0x103, 0x10`

［機能］

prepare 命令は，関数の前処理をする命令です。

- “prepare list, imm1”の形式

（a）第 1 オペランドで指定したレジスタを 1 つプッシュし，スタック・ポインタ（sp）から 4 を減算します。

（b）第 1 オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

（c）第 2 オペランドで指定した絶対値式の値を sp から減算[注]し，sp をレジスタ退避領域に設定します。

- “prepare list, imm1, imm2”の形式

(a) 第1オペランドで指定したレジスタを1つプッシュし，spから4を減算します。

(b) 第1オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

(c) 第2オペランドで指定した絶対値式の値をspから減算[注]し，spをレジスタ退避領域に設定します。

(d) 第3オペランドで指定した絶対値式の値をepに設定します。

- “prepare list, imm1, sp”の形式

(a) 第1オペランドで指定したレジスタを1つプッシュし，spから4を減算します。

(b) 第1オペランドで指定したレジスタをすべてプッシュし終わるまで（a）を繰り返します。

(c) 第2オペランドで指定した絶対値式の値をspから減算[注]し，spをレジスタ退避領域に設定します。

(d) 第3オペランドで指定したspの値をepに設定します。

注 機械語命令で実際にspに減算される値は，imm1を左へ2ビット・シフトした値となります。このためアセンブラは，指定したimm1をあらかじめ右へ2ビット・シフトしてコードに反映します。

[詳細説明]

- imm1に次のものを指定した場合，アセンブラでは，機械語命令のprepare命令が1つ生成されます。

(a) 0～127の範囲の絶対値式

`prepare list, imm1`	`prepare list, imm1`
`prepare list, imm1, imm2`	`prepare list, imm1, imm2`
`prepare list, imm1, sp`	`prepare list, imm1, sp`

- listに定数式以外[注]を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550249 : 構成に誤りがあります。

注 未定義シンボルやラベルの参照です。

- imm1 に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）0 ～ 127 の範囲を越え，0 ～ 32767 の範囲の絶対値式

`prepare list, imm1`	`prepare list, 0` `movea -imm1, sp, sp`
`prepare list, imm1, imm2`	`prepare list, 0, imm2` `movea -imm1, sp, sp`
`prepare list, imm1, sp`	`prepare list, 0, sp` `movea -imm1, sp, sp`

（b）0 ～ 32767 の範囲を越える絶対値式

`prepare list, imm1`	`prepare list, 0`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, imm2`	`prepare list, 0, imm2`
	`mov imm1, r1`
	`sub r1, sp`

`prepare list, imm1, sp`	`prepare list, 0, sp`
	`mov imm1, r1`
	`sub r1, sp`

［フラグ］

CY	–
OV	–
S	–
Z	–
SAT	–

注意　命令展開により sub 命令が生じた場合，フラグ値は変化する可能性があります。

［注意事項］

- sp で指定された下位 2 ビットのアドレスは，ミス・アライン・アクセスがイネーブルであっても 0 にマスクされます。そのため sp の値は 4 バイト・アライメントした値を設定してください。

dispose

スタック・フレームの削除（関数の後処理）を行います。（Function Dispose）

［指定形式］

- dispose imm, list
- dispose imm, list, [reg]

imm に指定できるものを次に示します。

- 32 ビット幅までの値を持つ絶対値式

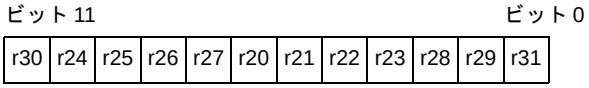
list は，dispose 命令でポップ可能な 12 本のレジスタを指定するものです。list に指定できるものを，次に示します。

- レジスタ

  プッシュの対象となるレジスタ（r20 ～ r31）をカンマで区切って指定します。

- 12 ビット幅までの値を持つ定数式

  12 ビットと 12 本のレジスタとの対応は次のとおりです。



次の 2 つの指定は等価です。

`dispose 0x10, r26, r29, r31`	`dispose 0x10, 0x103`

［機能］

dispose 命令は，関数の後処理をする命令です。

- “dispose imm, list”の形式

（a）第 1 オペランドで指定した絶対値式の値をスタック・ポインタ（sp）に加算[注]し，sp をレジスタ退避領域に設定します。

（b）第 2 オペランドで指定したレジスタを 1 つポップし，sp に 4 を加算します。

（c）第 2 オペランドで指定したレジスタをすべてポップし終わるまで（b）を繰り返します。

- “dispose imm, list, [reg]”の形式

（a）第 1 オペランドで指定した絶対値式の値をスタック・ポインタ（sp）に加算[注]し，sp をレジスタ退避領域に設定します。

（b）第 2 オペランドで指定したレジスタを 1 つポップし，sp に 4 を加算します。

（c）第 2 オペランドで指定したレジスタをすべてポップし終わるまで（b）を繰り返します。

（d）第 3 オペランドで指定したレジスタ値をプログラム・カウンタ（PC）に設定します。

注 機械語命令で実際に sp に加算される値は，imm を左へ 2 ビット・シフトした値となります。したがって，アセンブラは，指定した imm をあらかじめ右へ 2 ビット・シフトしてコードに反映します。

[詳細説明]

- imm に次のものを指定した場合，アセンブラでは，機械語命令の dispose 命令を 1 つ生成します。

（a）0 ～ 127 の範囲の絶対値式

`dispose imm, list`	`dispose imm, list`
`dispose imm, list, [reg]`	`dispose imm, list, [reg]`

- list に定数式以外を指定した場合，次のメッセージが出力され，アセンブルが中止されます。

E0550249 : 構成に誤りがあります。

- imm に次のものを指定した場合，アセンブラでは，命令展開が行われ，複数個の機械語命令が生成されます。

（a）0 ～ 127 の範囲を越え，0 ～ 32767 の範囲の絶対値式

`dispose imm, list`	`movea imm, sp, sp` `dispose 0, list`
`dispose imm, list, [reg]`	`movea imm, sp, sp` `dispose 0, list, [reg]`

（b）0 ～ 32767 の範囲を越える絶対値式

dispose imm, list	mov imm, r1
	add r1, sp
	dispose 0, list

dispose imm, list, [reg]	mov imm, r1
	add r1, sp
	dispose 0, list, [reg]

［フラグ］

CY	—
OV	—
S	—
Z	—
SAT	—

注意 命令展開により add 命令が生じた場合，フラグ値は変化する可能性があります。

［注意事項］

- sp で指定された下位 2 ビットのアドレスは，ミス・アライン・アクセスがイネーブルであっても 0 にマスクされます。そのため sp の値は 4 バイト・アライメントした値を設定してください。
- 形式“dispose imm, list, [reg]”の [reg] に r0 を指定すると，次のメッセージが出力され，アセンブルが中止されます。

E0550240 : V850Ex コア指定時には、デスティネーション・オペランドに r0 を指定することはできません。

4. 7. 14 浮動小数点演算命令【V850E2V3】

次に，浮動小数点演算命令の一覧を示します。

詳細については，V850E2V3 のインストラクション・セットをもつデバイス製品のユーザーズ・マニュアル，およびアーキテクチャ編を参照してください。

表 4―44 浮動小数点演算命令（基本演算命令）

命令	意味
absf.d	浮動小数点絶対値（倍精度）
absf.s	浮動小数点絶対値（単精度）
addf.d	浮動小数点加算（倍精度）
addf.s	浮動小数点加算（単精度）
divf.d	浮動小数点除算（倍精度）
divf.s	浮動小数点除算（単精度）
maxf.d	浮動小数点最大値（倍精度）
maxf.s	浮動小数点最大値（単精度）
minf.d	浮動小数点最小値（倍精度）
minf.s	浮動小数点最小値（単精度）
mulf.d	浮動小数点乗算（倍精度）
mulf.s	浮動小数点乗算（単精度）
negf.d	浮動小数点符号反転（倍精度）
negf.s	浮動小数点符号反転（単精度）
recipf.d	逆数（倍精度）
recipf.s	逆数（単精度）
rsqrtf.d	平方根の逆数（倍精度）
rsqrtf.s	平方根の逆数（単精度）
sqrtf.d	平方根（倍精度）
sqrtf.s	平方根（単精度）
subf.d	浮動小数点減算（倍精度）
subf.s	浮動小数点減算（単精度）

表 4―45 浮動小数点演算命令（拡張基本演算命令）

命令	意味
maddf.s	浮動小数点積和算（単精度）
msubf.s	浮動小数点積和算（単精度）
nmaddf.s	浮動小数点積和算（単精度）
nmsubf.s	浮動小数点積和算（単精度）

表 4―46 浮動小数点演算命令（変換命令）

命令	意味
ceilf.dl	整数形式への変換（倍精度）
ceilf.dw	整数形式への変換（倍精度）
ceilf.dul	符号なし整数形式への変換（倍精度）
ceilf.duw	符号なし整数形式への変換（倍精度）
ceilf.sl	整数形式への変換（単精度）
ceilf.sw	整数形式への変換（単精度）
ceilf.sul	符号なし整数形式への変換（単精度）
ceilf.suw	符号なし整数形式への変換（単精度）
cvtf.dl	整数形式への変換（倍精度）
cvtf.ds	浮動小数点形式への変換（倍精度）
cvtf.dul	符号なし整数形式への変換（倍精度）
cvtf.duw	符号なし整数形式への変換（倍精度）
cvtf.dw	整数形式への変換（倍精度）
cvtf.ld	浮動小数点形式への変換（倍精度）
cvtf.ls	浮動小数点形式への変換（単精度）
cvtf.sd	浮動小数点形式への変換（倍精度）
cvtf.sl	整数形式への変換（単精度）
cvtf.sul	符号なし整数形式への変換（単精度）
cvtf.suw	符号なし整数形式への変換（単精度）
cvtf.sw	整数形式への変換（単精度）
cvtf.uld	浮動小数点形式への変換（倍精度）
cvtf.uls	浮動小数点形式への変換（単精度）
cvtf.uwd	浮動小数点形式への変換（倍精度）
cvtf.uws	浮動小数点形式への変換（単精度）
cvtf.wd	浮動小数点形式への変換（倍精度）
cvtf.ws	浮動小数点形式への変換（単精度）
floorf.d	整数形式への変換（倍精度）
floorf.dw	整数形式への変換（倍精度）
floorf.dul	符号なし整数形式への変換（倍精度）
floorf.duw	符号なし整数形式への変換（倍精度）
floorf.sl	整数形式への変換（単精度）
floorf.sw	整数形式への変換（単精度）
floorf.sul	符号なし整数形式への変換（単精度）
floorf.suw	符号なし整数形式への変換（単精度）
trncf.dl	整数形式への変換（倍精度）
trncf.dul	符号なし整数形式への変換（倍精度）

命令	意味
trncf.duw	符号なし整数形式への変換（倍精度）
trncf.dw	整数形式への変換（倍精度）
trncf.sl	整数形式への変換（単精度）
trncf.sul	符号なし整数形式への変換（単精度）
trncf.suw	符号なし整数形式への変換（単精度）
trncf.sw	整数形式への変換（単精度）

表 4―47　浮動小数点演算命令（比較命令）

命令	意味
cmpf.s	浮動小数点比較（単精度）
cmpf.d	浮動小数点比較（倍精度）

表 4―48　浮動小数点演算命令（条件付き転送命令）

命令	意味
cmovf.s	条件付き転送（単精度）
cmovf.d	条件付き転送（倍精度）

表 4―49　浮動小数点演算命令（条件ビット転送命令）

命令	意味
trfsr	フラグ転送

cmpf.s

浮動小数点比較（単精度）を行います。（Floating-point Compare (Single)）

［指定形式］

- cmpf.s imm4, reg1, reg2, cc#3
- cmpf*cnd*.s reg1, reg2

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

- “cmpf.s imm4, reg1, reg2, cc#3”の形式

  reg2 で指定されるレジスタ・ペアにある単精度浮動小数点形式の内容を，比較条件 imm4 により，reg1 で指定されるレジスタ・ペアにある単精度浮動小数点形式の内容と比較します。結果（真ならば 1，偽ならば 0）を cc#3 で指定される FPSR レジスタのコンディション・ビット（CC（7:0）ビット：ビット 31 ～ 24）にセットします。cc#3 が省略された場合は CC0 ビット（ビット 24）にセットします。
- “cmpf*cnd*.s reg1, reg2”の形式

  cmpf*cnd*.s により対応する cmpf.s 命令が生成され（「表 4―50 cmpfcnd.s 命令一覧」を参照），“cmpf.s imm4, reg1, reg2, cc#3”の形式に展開されます。reg2 で指定されるレジスタ・ペアにある単精度浮動小数点形式の内容を，比較条件により，reg1 で指定されるレジスタ・ペアにある単精度浮動小数点形式の内容と比較します。結果（真ならば 1，偽ならば 0）を cc#3 で指定される FPSR レジスタのコンディション・ビット（CC（7:0）ビット：ビット 31 ～ 24）にセットします。cc#3 が省略された場合は CC0 ビット（ビット 24）にセットします。

［詳細説明］

- “cmpf.s imm4, reg1, reg2, cc#3”の形式の命令に対し，アセンブラでは，機械語命令の cmpf.s 命令が 1 つ生成されます。
- “cmpf*cnd*.s reg1, reg2”の形式の命令に対し，アセンブラでは，対応する cmpf.s 命令が生成され（「表 4―50 cmpfcnd.s 命令一覧」参照），“cmpf.s imm4, reg1, reg2, cc#3”の形式に展開されます。

表 4―50 cmpf*cnd*.s 命令一覧

命令	定義	説明	命令展開
cmpff.s	FALSE	常に偽	cmpf.s 0x0
cmpfun.s	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.s 0x1
cmpfeq.s	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.s 0x2
cmpfueq.s	reg2 ？＝ reg1	少なくとも一方が非数か，等しい	cmpf.s 0x3
cmpfolt.s	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.s 0x4

命令	定義	説明	命令展開
cmpfult.s	reg2 ? ＜ reg1	少なくとも一方が非数か，より小さい	cmpf.s 0x5
cmpfole.s	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.s 0x6
cmpfule.s	reg2 ? ≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.s 0x7
cmpfsf.s	FALSE	常に偽	cmpf.s 0x8
cmpfngle.s	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.s 0x9
cmpfseq.s	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.s 0xA
cmpfngl.s	reg2 ? ＝ reg1	少なくとも一方が非数か，等しい	cmpf.s 0xB
cmpflt.s	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.s 0xC
cmpfnge.s	reg2 ? ＜ reg1	少なくとも一方が非数か，より小さい	cmpf.s 0xD
cmpfle.s	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.s 0xE
cmpfngt.s	reg2 ? ≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.s 0xF

備考 ?：Unordered（比較不能）

［注意事項］

- cmpf.s 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージを出力し，指定した値の下位 4 ビットを用いてアセンブルを続行します。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

cmpf.d

浮動小数点比較（倍精度）を行います。（Floating-point Compare (Double)）

［指定形式］

- cmpf.d imm4, reg1, reg2, cc#3
- cmpf*cnd*.d reg1, reg2

imm4 に指定できるものを次に示します。

- 4 ビット幅までの値を持つ絶対値式

［機能］

- “cmpf.d imm4, reg1, reg2, cc#3”の形式

  reg2 で指定されるレジスタ・ペアにある倍精度浮動小数点形式の内容を，比較条件 imm4 により，reg1 で指定されるレジスタ・ペアにある倍精度浮動小数点形式の内容と比較します。結果（真ならば 1，偽ならば 0）を cc#3 で指定される FPSR レジスタのコンディション・ビット（CC（7:0）ビット：ビット 31 ～ 24）にセットします。cc#3 が省略された場合は CC0 ビット（ビット 24）にセットします。
- “cmpf*cnd*.d reg1, reg2”の形式

  cmpf*cnd*.d により対応する cmpf.d 命令が生成され（「表 4―51 cmpfcnd.d 命令一覧」を参照），“cmpf.d imm4, reg1, reg2, cc#3”の形式に展開されます。reg2 で指定されるレジスタ・ペアにある倍精度浮動小数点形式の内容を，比較条件により，reg1 で指定されるレジスタ・ペアにある倍精度浮動小数点形式の内容と比較します。結果（真ならば 1，偽ならば 0）を cc#3 で指定される FPSR レジスタのコンディション・ビット（CC（7:0）ビット：ビット 31 ～ 24）にセットします。cc#3 が省略された場合は CC0 ビット（ビット 24）にセットします。

［詳細説明］

- “cmpf.d imm4, reg1, reg2, cc#3”の形式の命令に対し，アセンブラでは，機械語命令の cmpf.d 命令が 1 つ生成されます。
- “cmpf*cnd*.d reg1, reg2”の形式の命令に対し，アセンブラでは，対応する cmpf.d 命令が生成され（「表 4―51 cmpfcnd.d 命令一覧」参照），“cmpf.d imm4, reg1, reg2, cc#3”の形式に展開されます。

表 4―51 cmpf*cnd*.d 命令一覧

命令	定義	説明	命令展開
cmpff.d	FALSE	常に偽	cmpf.d 0x0
cmpfun.d	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.d 0x1
cmpfeq.d	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.d 0x2
cmpfueq.d	reg2 ？＝ reg1	少なくとも一方が非数か，等しい	cmpf.d 0x3
cmpfolt.d	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.d 0x4

命令	定義	説明	命令展開
cmpfult.d	reg2 ? ＜ reg1	少なくとも一方が非数か，より小さい	cmpf.d 0x5
cmpfole.d	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.d 0x6
cmpfule.d	reg2 ? ≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.d 0x7
cmpfsf.d	FALSE	常に偽	cmpf.d 0x8
cmpfngle.d	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.d 0x9
cmpfseq.d	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.d 0xA
cmpfngl.d	reg2 ? ＝ reg1	少なくとも一方が非数か，等しい	cmpf.d 0xB
cmpflt.d	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.d 0xC
cmpfnge.d	reg2 ? ＜ reg1	少なくとも一方が非数か，より小さい	cmpf.d 0xD
cmpfle.d	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.d 0xE
cmpfngt.d	reg2 ? ≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.d 0xF

備考 ?：Unordered（比較不能）

［注意事項］

- cmpf.d 命令の imm4 に 4 ビットの範囲を越える絶対値式を指定した場合，次のメッセージを出力し，指定した値の下位 4 ビットを用いてアセンブルを続行します。

W0550011 : イミーディエトの値が指定可能な値の範囲を超えています。

第 5 章　リンク・ディレクティブ仕様

この章では，リンク・ディレクティブに必要となる項目や，リンク・ディレクティブ・ファイルの記述方法について説明します。

組み込み系のアプリケーションでは，プログラム・コードをある番地から配置したり，分割して配置するなど，メモリ配置に気を配る必要があります。

期待どおりのメモリ配置を実現するには，プログラム・コードやデータの配置情報を，リンカに指示する必要があります。この指示の情報を「リンク・ディレクティブ」と呼び，リンク・ディレクティブを記述するファイルを「リンク・ディレクティブ・ファイル」と呼びます。

リンカは，このリンク・ディレティブ・ファイルに従ってメモリ配置を決定し，ロード・モジュールを作成します。

5. 1　指定項目

リンク・ディレクティブで指定する主な項目は，大きく分けて次の 2 つになります。

- セグメント・ディレクティブとマッピング・ディレクティブ
- シンボル・ディレクティブ

5. 1. 1　セグメント・ディレクティブとマッピング・ディレクティブ

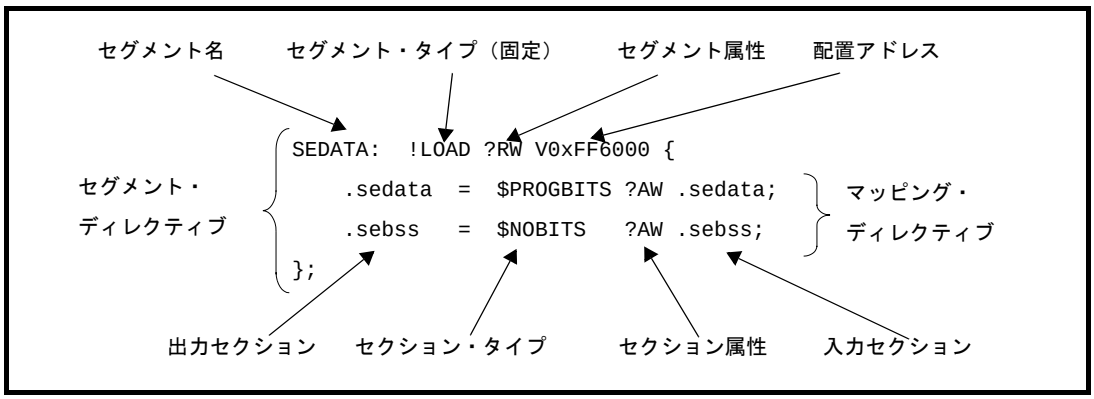
プログラムやデータが配置される領域である「セクション」の情報を，タイプと属性別に分けて「セグメント」の情報としてまとめ，その配置アドレスを決めます。

ここで，セクション情報の記述を「マッピング・ディレクティブ」，セグメント情報の記述を「セグメント・ディレクティブ」といいます。

次に，リンク・ディレクティブ・ファイルに記述する「セグメント・ディレクティブ」と「マッピング・ディレクティブ」の一例を示します。

なお，詳しい書式については，「5. 4　コーディング方法」を参照してください。

図 5―1　セグメント・ディレクティブとマッピング・ディレクティブ



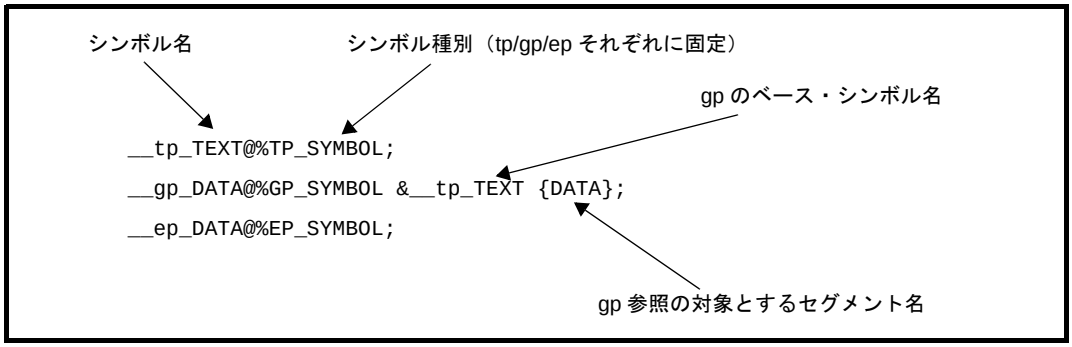
5. 1. 2　シンボル・ディレクティブ

tp（テキスト・ポインタ），gp（グローバル・ポインタ），ep（エレメント・ポインタ）を生成するための「シンボル」を作成します。これらのシンボルの情報を「シンボル・ディレクティブ」と呼びます。

次に，リンク・ディレクティブ・ファイルに記述する「シンボル・ディレクティブ」の一例を示します。

なお，詳しい書式については「5. 4　コーディング方法」を参照してください。

図 5―2　シンボル・ディレクティブ



5. 2　セクションとセグメント

ここでは，セクションとセグメントについて，それぞれ説明します。

5. 2. 1　セクション

セクションとは，プログラムを構成する基本的な単位（プログラムやデータが配置される領域）です。たとえば，プログラム・コードは text 属性セクションへ，初期値を持つ変数は data 属性セクションへ，というように，セクションごとに分けて配置することになります。

セクション名はアプリケーション内で指定できます。C 言語では“#pragma section 指令”や“#pragma text 指令”，アセンブリ言語では“セクション定義疑似命令”によって指定できます。

ただし #pragma 指令でセクションの指定をしない場合でも，コンパイラはプログラム・コードやデータ（変数）にデフォルトとして決められたセクションを割り当てるようにしています。

5. 2. 2　セグメント

セグメントとは，プログラムやデータをメモリにロードする際の単位です。属性が同じだったり，タイプが同じであるようなセクション群を 1 つに集め，1 セグメントとして扱います。つまり，「似たようなセクションの集まり = セグメント」というイメージになります。

リンク・ディレクティブでは，セグメント名，属性，配置するアドレスなどを自由に設定できます。

注意　セグメント名，属性として指定できない文字があります。詳細は，「5. 4. 3　セグメント・ディレクティブ」を参照してください。

読み出し可能（R）で実行可能（X）なセグメント「TEXT1」を 0x100000 番地へ配置する場合，リンク・ディレクティブ・ファイルでは次のように記述します。

```
TEXT1:!LOAD ?RX V0x100000 {
        :
（マッピング・ディレクティブ）
        :
};
```

セグメントはメモリにロードする際の単位であるため，プログラム・コードやデータを配置するときは，セグメント単位で行います。つまり，あるセクションを特定のメモリに配置したい場合，そのセクション情報をマッピング・ディレクティブにしたがって記述し，そのマッピング・ディレクティブを含むセグメントを作成します。そしてそのセグメントの配置アドレスを決定するという手順を踏みます。

注意 マッピング・ディレクティブにおいて，セクションに直接アドレスを指定することもできますが，通常はセグメント単位でアドレスを指定します。

例 変数 i を sdata 領域に配置し，関数 func1 を 0x120000 番地へ配置する場合

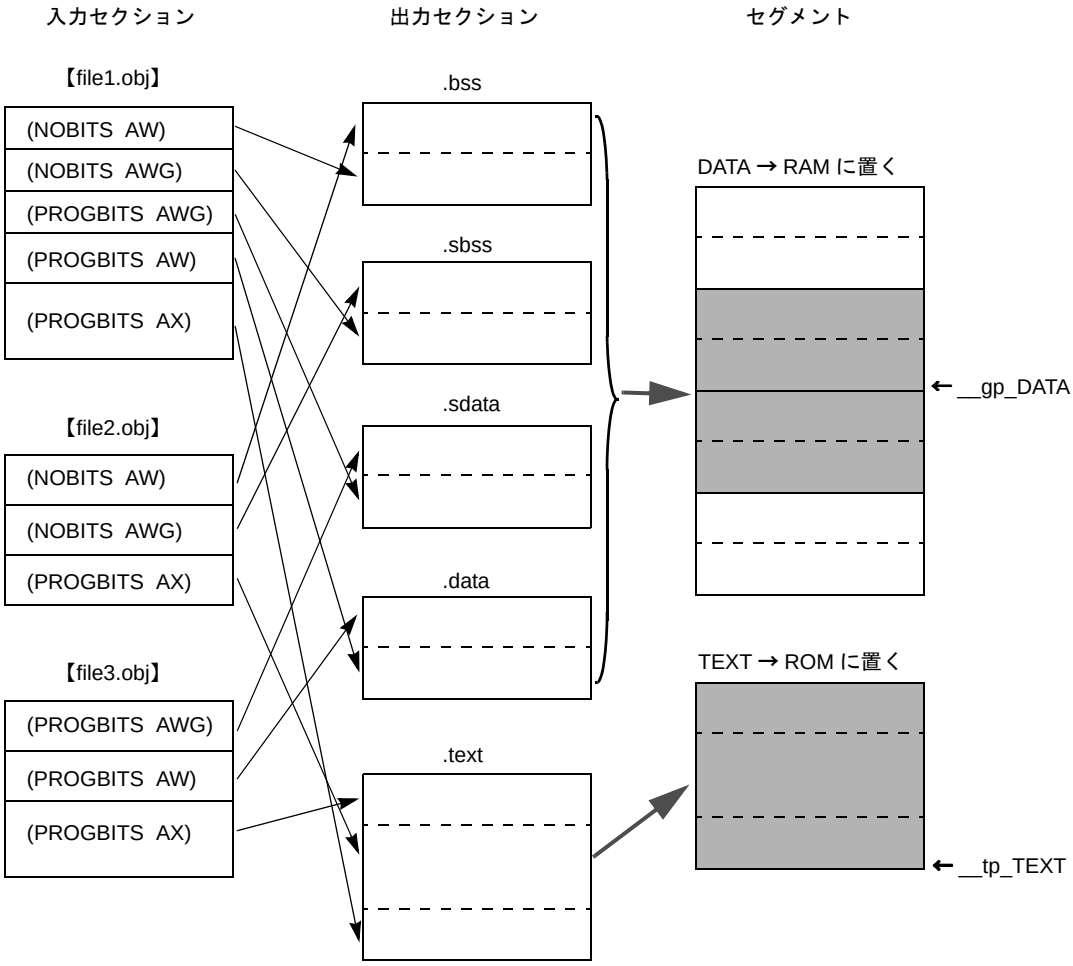
```
【test1.c】
#pragma section sdata
i = 10;
#pragma section default

#pragma text "f1" func1

void func1() {
            :
        return;
}
```

```
【リンク・ディレクティブ（一部）】
TEXT2 : !LOAD ?RX V0x120000 {
        text1= $PROGBITS ?AX f1.text;
};
DATA  : !LOAD ?R V0x200000 {
        .data   = $PROGBITS ?AW;
        .sdata  = $PROGBITS ?AWG;
        .sbss   = $NOBITS ?AWG;
        .bss    = $NOBITS ?AW;
};
            :
```

5. 2. 3 セグメントとセクションの関係

セグメントとセクションの関係を，マッピングのイメージで見ると次の例のようになります。

図 5―3 セグメントとセクションの関係



それぞれのオブジェクト（file1.obj，file2.obj，file3.obj）に含まれているセクションを「入力セクション」と呼びます。これらは同種のセクションごとにまとめられます。まとめられて出力されたセクションを「出力セクション」と呼びます。出力セクション群は，リンク・ディレクティブにおいて，それぞれセグメント（DATA セグメント，TEXT セグメント）にまとめられているので，これにしたがって適切な場所へ（アドレスが指定されていれば，そのアドレスへ）マッピングされます。

テキスト・ポインタ（tp）のシンボル“__tp_TEXT”とグローバル・ポインタ（gp）のシンボル“__gp_DATA”も規則に従ってセットされます。

5. 2. 4　セクションの種類

ここでは，CX で扱うことのできるセクションとその特長について説明します。

割り当てを指定できるセクション種別とその特徴は，「表 5―1　CX の配置セクション種別」のとおりです。

なお，この形式やセクション・ファイルによりセクションへの割り当てを指定しないデータは，CX のオプションで指定されたサイズに従い，CX によって，.sdata セクション，.data セクション，.sbss セクション，.bss セクションのいずれかに配置されます**注 1**。

型修飾子 const が指定されたデータ，および文字列定数は CX のオプションで指定されたサイズにしたがい，CX によって，.const セクション，.sconst セクションに配置されます**注 2**。

また，セクションへの割り当ては，シンボル情報ファイルによっても指定できます**注 3**。

注 1. デフォルトでは全データを .sdata セクション /.sbss セクションへ配置します。

2. 「CubeSuite ビルド編（CX コンパイラ）」の -Xsconst オプションに関する説明を参照してください。

3. 「CubeSuite ビルド編（CX コンパイラ）」のシンボル情報ファイルを参照してください。

表 5―1　CX の配置セクション種別

種別	特長	指定文字列
.tidata.byte セクション .tidata.word セクション .tibss.byte セクション .tibss.word セクション （tiny internal data/ tiny internal bss）	ep（エレメント・ポインタ）から 1 命令で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。 sidata 属性 /sibss 属性セクションと違うところは，同じ 1 命令アクセスでも，使用するアセンブル命令が違うことです。sidata 属性 /sibss 属性セクションは，格納／参照に 4 バイト長命令の "st/ld 命令 " を使用しますが，tidata 属性 /tibss 属性セクションは，2 バイト長命令の "sst/sld 命令 " を使用してアクセスします。つまり，sidata 属性 /sibss 属性セクションよりもコード効率がよくなります。ただし，sst/sld 命令が適用できる範囲は小さいので，多くの変数を配置することはできません。 初期値ありデータは，tidata（tidata.byte，tidata.word）属性セクションへ，初期値なしデータは，tibss（tibss.byte，tibss.word）属性セクションへ配置されます。 バイト・データを配置する場合は tidata.byte/tibss.byte 属性を，ワード・データを配置する場合は tidata.word/tibss.word 属性を指定しますが，CX に自動判別させたい場合は，tidata/tibss 属性を指定します。	tidata tidata_byte tidata_word
.data セクション .bss セクション （data/bss）	gp（グローバル・ポインタ）から 2 命令で参照可能なセクションです。 アドレス生成を行ってからアクセス（ld/st 命令）するため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。つまり，RAM 上であれば，どこにでも配置が可能なセクションです。 初期値ありデータは data 属性セクションへ，初期値なしデータは bss 属性セクションへ配置されます。	data

種別	特長	指定文字列
.sdata セクション .sbss セクション （sdata/sbss）	gp（グローバル・ポインタ）から 1 命令（ld/st 命令）で参照可能なセクションで，gp から± 32K バイト内に配置される必要があります（あわせて 64K バイト）。 初期値ありデータは sdata 属性セクションへ，初期値なしデータは sbss 属性セクションへ配置されます。 CX では，まずこのセクションに配置するコードを生成しようとします。ただし，この属性のセクションに収まりきらないような場合は，data 属性 /bss 属性セクションへ配置するコードを生成します。 なお，sdata 属性 /sbss 属性セクションへの配置データを少しでも多くする策として，CX のオプション "-Xsdata" で，配置されるデータのサイズの上限を指定し，それ以上のサイズは sdata 属性 /sbss 属性セクションに配置しないという指定ができます。	sdata
.sedata セクション .sebss セクション （small extended data/ small extended bss）	ep（エレメント・ポインタ）から 1 命令（ld/st 命令）で参照可能なセクションで，ep からマイナス方向へアクセスするセクションです。つまり，ep からマイナス方向 32K バイト内に配置されるセクションです。 初期値ありデータは sedata 属性セクションへ，初期値なしデータは sebss 属性セクションへ配置されます。 gp から 1 命令でアクセスできる sdata 属性 /sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。sidata 属性 /sibss 属性セクションは ep からプラス方向にアクセスするためのセクションですが，sedata 属性 /sebss 属性セクションは ep からマイナス方向にアクセスするためのセクションです。	sedata
.sidata セクション .sibss セクション （small internal data/ small internal bss）	ep（エレメント・ポインタ）から 1 命令（ld/st 命令）で参照可能なセクションで，ep からプラス方向へアクセスするセクションです。つまり，ep からプラス方向 32K バイト内に配置されるセクションです。 初期値ありデータは sidata 属性セクションへ，初期値なしデータは sibss 属性セクションへ配置されます。 gp から 1 命令でアクセスできる sdata 属性 /sbss 属性セクションに入りきらなくなったが，1 命令アクセスしたい変数がまだ存在する場合，ep を使って 1 命令でアクセスできる範囲に置くことができます。sidata 属性 /sibss 属性セクションは ep からプラス方向にアクセスするためのセクションですが，sedata 属性 /sebss 属性セクションは ep からマイナス方向にアクセスするためのセクションです。	sidata

種別	特長	指定文字列
.sconst セクション （small const data）	r0，つまり，0 番地から 1 命令（ld/st 命令）で参照可能なセクションで，0 番地から± 32K バイト内に配置される必要があります。基本的に "ROM に固定してもよいデータ " を配置するセクションです。 V850 で内蔵 ROM を持つ製品の場合，0 番地からプラス方向が内蔵 ROM である場合が多く，そこに 1 命令で参照したい，かつ ROM 固定してもよいデータを，sconst 属性セクションとして配置します。内蔵 ROM を持たないデバイス，および ROM レスモードを指定した場合には，外部メモリに位置します。 sconst 属性 /const 属性セクションに配置するデータは，const 修飾子をつけて宣言された変数／データが対象となります。この属性のセクションに収まりきらないような場合は，const 属性セクションへ配置することになります。 なお，sconst 属性セクションへの配置データを少しでも多くする策として，CX のオプション "-Xsconst" で，配置されるデータのサイズの上限を指定し，それ以上のサイズは sconst 属性セクションに配置しないという指定ができます（オプションの詳細は「CubeSuite ビルド編（CX コンパイラ）」を参照してください）。	sconst
.const セクション （const data）	r0，つまり，0 番地から 2 命令で参照可能なセクションです。アドレス生成を行ってからアクセス（ld/st 命令）するため，その分コードが多くなり，実行速度も落ちますが，32 ビット空間内すべてにアクセスが可能です。sconst 属性セクションに入りきらなかった "ROM 固定してもよいデータ " や，V850 の ROM レス品で，外部 ROM にデータを配置したい場合に，const 属性セクションに配置します。	const

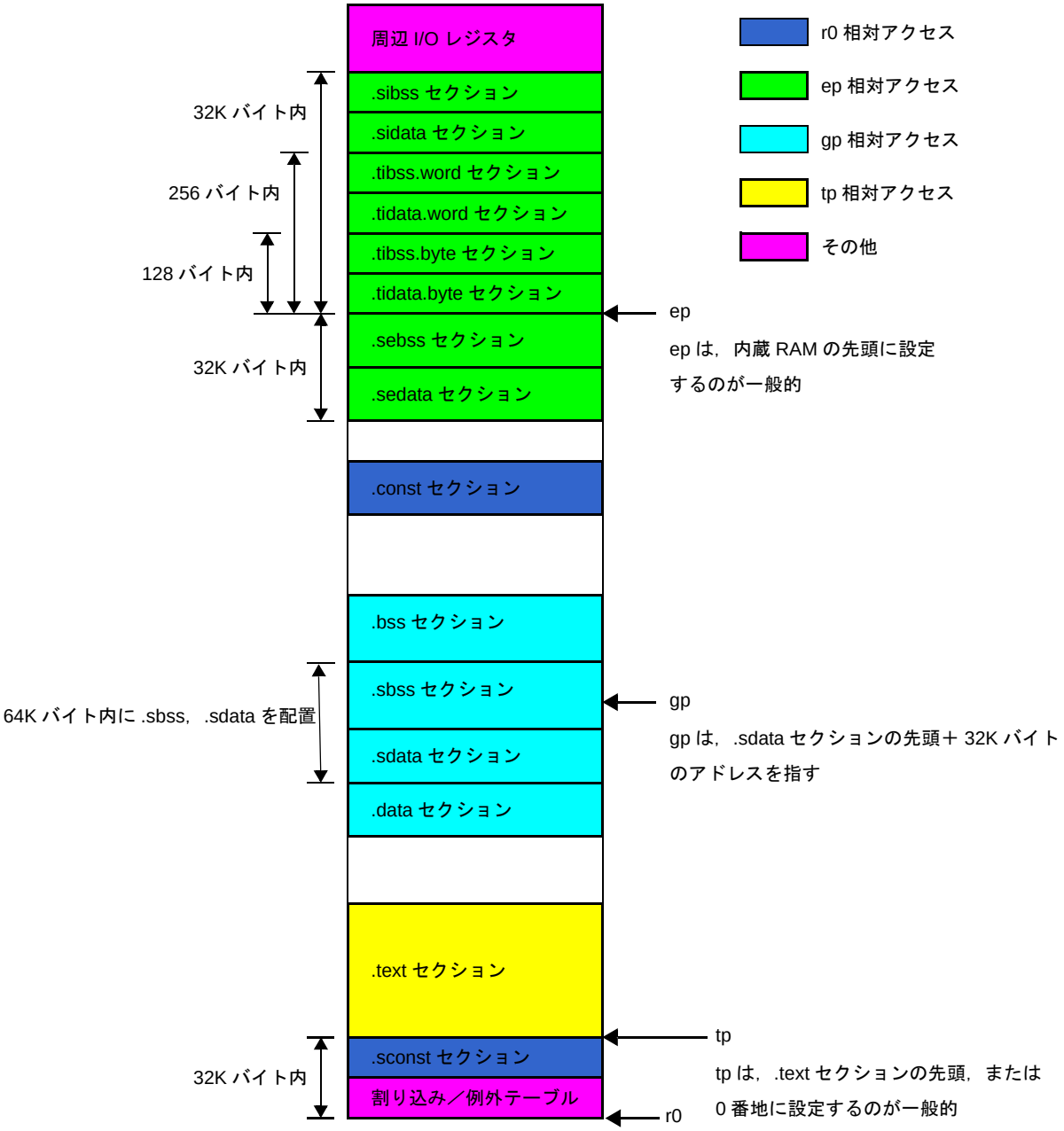
注意 1. “(2 命令)” は，アセンブラにより 2 命令に命令展開されるものです。

2. “gp 相対”，“r0” 相対というのは，コンパイラが gp 相対や r0 相対のコードを出力することを示します。

3. “外部メモリ” とあるセクション種別は，ターゲット・システムに外部メモリが実装されている場合に適用できます。

次に，各セクションのメモリ配置イメージの例を示します。

図 5―4　CX による各セクションのメモリ配置イメージの例（内蔵 ROM あり）



5. 2. 5　セクションのタイプと属性

セクションのタイプと属性について説明します。

これらは，マッピング・ディレクティブでセクションの情報を記述するときに必要となります。

セクションのタイプは，次のように分類できます。

表 5―2　セクションのタイプ

セクション・タイプ	意味
PROGBITS	オブジェクト・モジュール・ファイル内に実際の値を持っているセクション →テキスト，初期値ありデータ（変数）
NOBITS	オブジェクト・モジュール・ファイル内に実際の値を持っていないセクション →初期値なしデータ（変数）

セクションの属性は，次のように分類できます。

表 5―3　セクションの属性

セクション属性	意味
A	メモリを占有するセクション（すべてのセクションが該当）
W	書き込み可能なセクション（RAM 上に配置するセクション）
X	実行可能なセクション（主にテキスト・セクション）
G	グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲内に割り付けるセクション（.sdata, .sbss セクション）

セクションを，タイプと属性別に分けると，次の 6 種類に分類されます。

表 5―4　セクションの分類

セクション属性	セクション・タイプ／セクション属性		該当する予約セクション
bss 属性	セクション・タイプ	NOBITS	.bss .sebss .sibss .tibss.byte .tibss.word
	セクション属性	AW	
const 属性	セクション・タイプ	PROGBITS	.const .sconst
	セクション属性	A	
data 属性	セクション・タイプ	PROGBITS	.data .sedata .sidata .tidata.byte .tidata.word
	セクション属性	AW	

セクション属性	セクション・タイプ／セクション属性		該当する予約セクション
sbss 属性	セクション・タイプ	NOBITS	.sbss
	セクション属性	AWG	
sdata 属性	セクション・タイプ	PROGBITS	.sdata
	セクション属性	AWG	
text 属性	セクション・タイプ	PROGBITS	.pro_epi_runtime .text
	セクション属性	AX	

注意　アプリケーション内でセクション名を独自に作成する場合，“表 5―4　セクションの分類”のようにそのセクションがどの属性にあたるのかを明確にし，ユーザ自身がマッピング・ディレクティブにおいて，セクション・タイプ，セクション属性とともに指定する必要があります。
なお，セクション名として，リンク・ディレクティブの書式により“V/H/A/+ 数字”ではじまるセクション名は作成できません。

5. 3　シンボル

CX では，アプリケーションを実行する際に次のポインタを必要とします。

- テキスト・ポインタ（tp）
- グローバル・ポインタ（gp）
- エレメント・ポインタ（ep）

各ポインタの値は「セグメントの位置」に関係するため，リンク・ディレクティブでポインタの値を決定する仕組みが必要となります。

リンク・ディレクティブでは，ポインタの値を決定するために「シンボル」を定義します。定義したシンボルの値はリンカによって決定され，その値をアプリケーション内でポインタにコピーすることにより，ポインタの値を決定できます。このように，リンク・ディレクティブで各ポインタ用のシンボルを定義することから，「シンボル・ディレクティブ」と呼ばれます。

ここでは，各ポインタの役割と，ポインタ値の決め方について説明します。

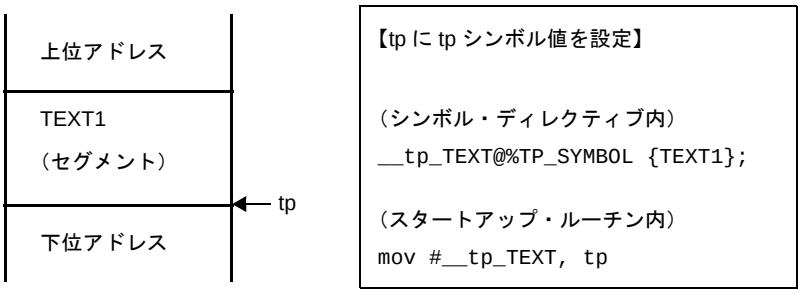
5. 3. 1　テキスト・ポインタ（tp）

アプリケーションのテキスト領域を参照するとき，配置される位置に依存することのない参照（Position Independent Code：PIC）を実現するために用意されるポインタが「テキスト・ポインタ（tp）」です。つまり，テキストは tp 相対で参照されます。コンパイラは tp が正しく設定（テキストの先頭）されていることを前提としたコードを出力するので，ポインタの値を正しく設定する必要があります。

tp はアプリケーションに 1 つだけではなく，セグメント単位で複数作成することもできます。

ただし，複数生成した場合，tp の切り替えはアプリケーション・プログラムで明示的に行う必要があります。

図 5―5　tp の設定例



この例では，リンク・ディレクティブで tp シンボル値が TEXT1 セグメントの先頭を指すように設定します。tp シンボル名は“__tp_TEXT”なので，リンク時に決定される TEXT1 セグメントの先頭アドレスが，シンボル“__tp_TEXT”にセットされます。

この値を tp に設定するために，スタートアップ・ルーチンなどで，変数“tp”にこの“__tp_TEXT”の値を代入する式（mov #__tp_TEXT, tp）を記述します。これにより tp に正しくテキスト・ポインタの値が設定されます。

5. 3. 2　グローバル・ポインタ（gp）

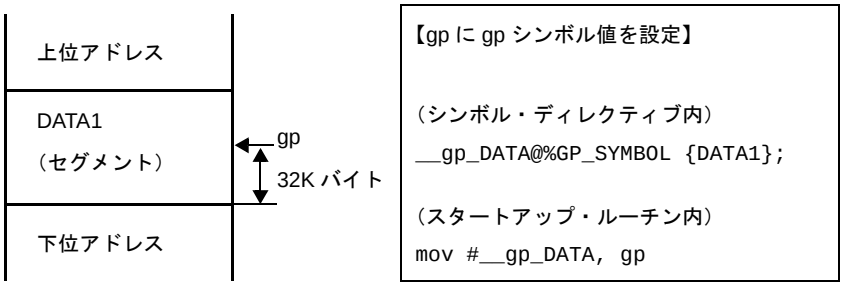
アプリケーション内でグローバル宣言したデータはメモリに配置されます。このメモリに配置されているデータを参照（ロード，ストア）する際，配置位置に依存することのない参照（Position Independent Data：PID）を実現するために用意されるポインタが「グローバル・ポインタ（gp）」です。

グローバル宣言したデータは gp 相対で参照されます。V850 コアでは「gp と 1 命令」か「gp と 2 命令」のどちらかでこれらのデータを参照できます。つまり，「gp と 1 命令」でアクセスできる方が，アプリケーションの速度の向上，コード・サイズの縮小が見込めます。

gp と 1 命令（ld/st 命令）で参照できるセクションは「sdata 属性，sbss 属性を持つセクション」，gp と 2 命令（movhi ＋ ld/st 命令）で参照できるセクションは「data 属性，bss 属性を持つセクション」です。つまり，gp 相対で参照できるセクション（の属性）はこの 4 つとなります。このうち「sdata 属性，sbss 属性をもつセクション」は gp から正方向，負方向にそれぞれ 32K バイト内に配置されているため，この範囲にデータ（変数）を置くと 1 命令でアクセス可能で，高速参照，コード・サイズ縮小につながります。

gp は 1 つだけではなく，セグメント単位で複数作成することもできます。ただし複数生成した場合，gp の切り替えはアプリケーション・プログラムで明示的に行う必要があります。

図 5―6　gp の設定例（セグメントを指定する場合）



この例では，リンク・ディレクティブで gp シンボル値が DATA1 セグメント内を参照できるように設定します。gp シンボル名が“__gp_DATA”なので，リンク時に決定される DATA1 セグメントの先頭から 32K バイト足したアドレスが，シンボル“__gp_DATA”にセットされます（「図 5―6　gp の設定例（セグメントを指定する場合）」参照）。

この値を gp に設定するために，スタートアップ・ルーチンなどで，変数“gp”にこの“__gp_DATA”の値を代入する式（mov #__gp_DATA, gp）を記述します。これにより gp に正しくグローバル・ポインタの値が設定されます。

なお，gp シンボルはアドレスだけではなく，tp シンボルからのオフセット値を指定することもできます。

次に，gp シンボルのオフセット指定について説明します。

（1）gp シンボル値のオフセット指定

gp シンボル値の指定方法は，上記で説明したように「gp のアクセス対象とするセグメントを指定する方法」が一般的です。

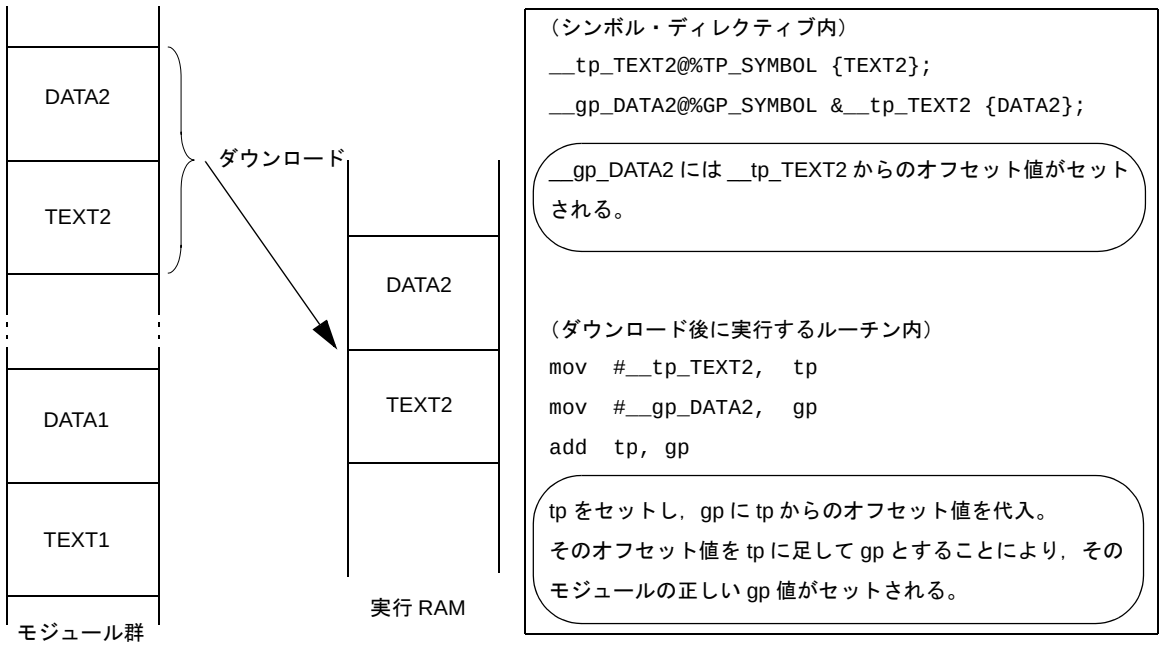
このほかに，「gp シンボルのアドレスを直接指定する方法」と「ベース・シンボルを決めて，そのシンボルからのオフセットを gp シンボル値とする方法」があります。ここでは後者について説明します（前者については，「（2）gp シンボル値の決定規則」を参照してください）。

gp シンボルのベース・シンボルとして指定するものは「tp シンボル」です。

gp シンボルを作成する際に，tp シンボルをベース・シンボルとして指定すると，リンク・ディレクティブで gp のシンボル値として決定される値は「tp シンボル値からのオフセット値」となります。

こうすることにより tp シンボル値から gp シンボル値を「tp シンボル値 ＋ tp シンボル値からのオフセット値」という計算によって容易に算出することができ，配置に依存しないアプリケーションの作成に有効となります。たとえば，複数の実行モジュールを持つアプリケーションで，その中の 1 つを RAM にコピーしてそこで実行したい場合に役立ちます。つまり，tp/gp の値を決定する際に，tp の値がわかれば，そのアドレスに gp のシンボル値（tp からのオフセット値）を足し，それを gp の値とすればよいことになります。

図 5―7　gp の設定例（tp からのオフセット指定の場合）



（2）gp シンボル値の決定規則

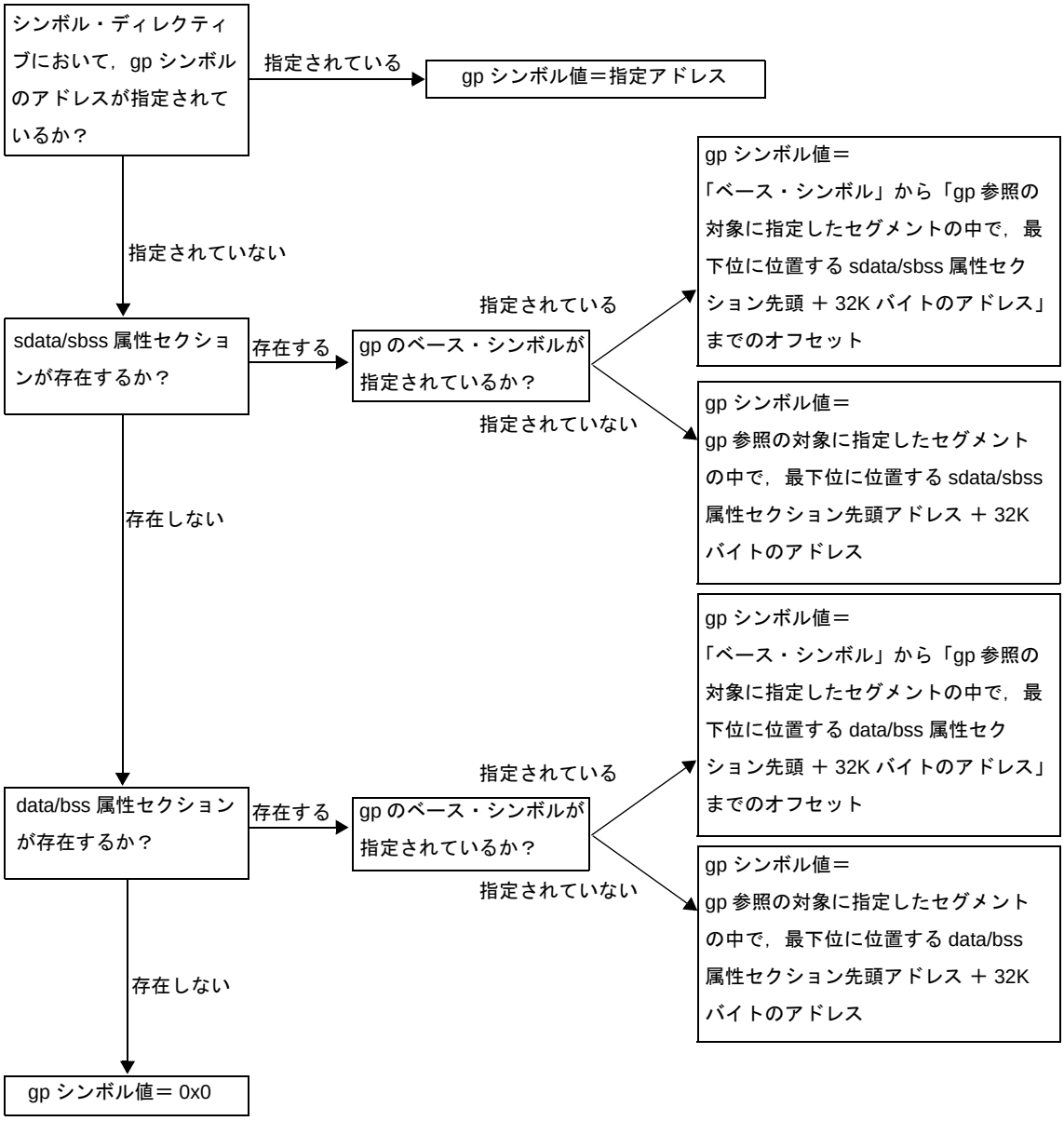
gp シンボル値の決定には，次のことが関係します。

- シンボル・ディレクティブにおけるアドレスの指定の有無
- sdata 属性 /sbss 属性 /data 属性 /bss 属性のセクションの有無
- ベース・シンボル指定の有無

リンカは，リンク・ディレクティブ・ファイルからこれらのことを調べ，gp シンボル値を決定します。

gp シンボル値の決定規則を図にまとめると，次のようになります。

図 5―8　グローバル・ポインタ値の決定規則



5. 3. 3 エレメント・ポインタ（ep）

アプリケーション内でグローバル宣言したデータ（変数）を，V850 コアに内蔵されている RAM 領域へ配置し，より高速な参照（ロード，ストア）を実現するために用意されているポインタが「エレメント・ポインタ（ep）」です。

グローバル宣言し，かつ，内蔵 RAM に配置するデータ（変数）は ep 相対で参照されます。

「ep と 1 命令」で参照することになりますが，その 1 命令が「sld/sst 命令」か「ld/st 命令」かによってセクションの属性が分かれます。

- 「ep と sld/sst 命令」で参照できるセクション
  tidata.byte 属性，tibss.byte 属性，tidata.word 属性，tibss.word 属性
- 「ep と ld/st 命令」で参照できるセクション
  sidata 属性，sibss 属性，sedata 属性，sebss 属性

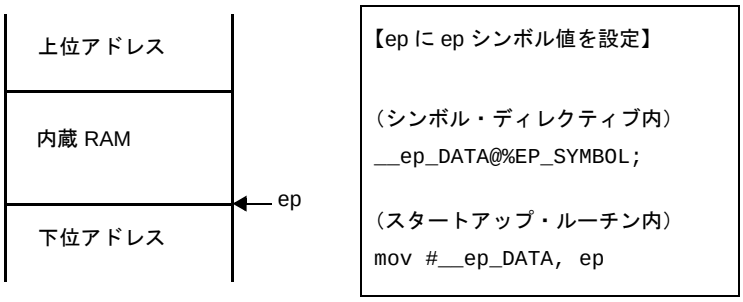
ただし，sedata 属性，sebss 属性は，内蔵 RAM ではなく，ep 相対でアクセス可能な外部 RAM のセクション属性になります。

一般的に内蔵 RAM の容量は少なく，多くのデータ（変数）を配置できませんが，特に参照を高速にしたいデータ（変数）を上記に示した領域に配置して「ep と 1 命令」でアクセスできるようにすると，アプリケーションの速度の向上やコード・サイズの縮小が見込めます。特に sld/sst 命令は，命令長が ld/st 命令の 4 バイトに対し，2 バイトとコード・サイズ縮小に役立ちます。

リンク・ディレクティブ・ファイルのシンボル・ディレクティブで ep シンボルの生成を指示した場合，リンカは使用するデバイスごとに用意されているデバイス・ファイルの情報を基に，自動的に内蔵 RAM 領域の先頭に設定されます。

なお ep はアプリケーション内で 1 つだけ生成できます。複数生成はできません。

図 5―9 ep の設定例



この例では，リンク・ディレクティブで ep シンボルを作成するように宣言します。ep シンボル名が“__ep_DATA”なので，リンカは内蔵 RAM の先頭アドレスを“__ep_DATA”にセットします。

この値を ep に設定するために，スタートアップ・ルーチンなどで，変数“ep”にこの“__ep_DATA”の値を代入する式（mov #__ep_DATA, ep）を記述します。これにより ep に正しくエレメント・ポインタの値が設定されます。

備考 アプリケーションが外部 RAM を一切使用せず，内蔵 RAM だけを使用するような場合は，gp シンボルを生成せず，ep シンボルだけを生成しても問題ありません。ただし，ランタイム・ライブラリを使用している場合，ランタイム関数が gp 相対でデータ（変数）参照をしているので，gp シンボルを生成する必要があります。

(1) ep シンボル値の決定規則

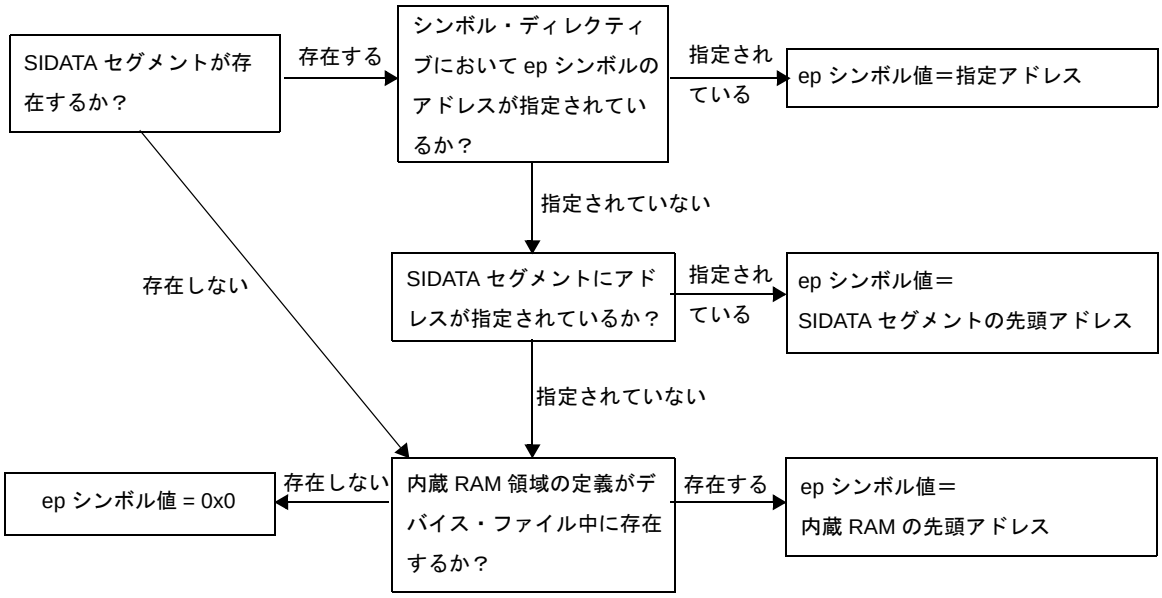
ep シンボル値の決定には次のことが関係します。

- シンボル・ディレクティブにおけるアドレス指定の有無
- SIDATA セグメントの有無
- デバイス・ファイル中の内蔵 RAM 領域の定義の有無

リンカは上記を調べ，ep シンボル値を決定します。

ep シンボル値の決定規則を図にまとめると，次のようになります。

図 5―10 エレメント・ポインタ値の決定規則



5.4 コーディング方法

ここではリンク・ディレクティブ・ファイルの書式について，次の項目ごとに説明します。

- セグメント・ディレクティブ
- マッピング・ディレクティブ
- シンボル・ディレクティブ

リンク・ディレクティブの書式の概要は，次に示すとおりで，これらをエディタなどにより，テキスト形式で記述します。

```
セグメント・ディレクティブ1 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ2 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ3 {
    マッピング・ディレクティブ;
};
セグメント・ディレクティブ4 {
    マッピング・ディレクティブ;
};
tp シンボル・ディレクティブ;
gp シンボル・ディレクティブ;
ep シンボル・ディレクティブ;
```

備考 セグメント・ディレクティブは，アドレスの若い順に記述することを推奨します。

5.4.1 使用できる文字

リンク・ディレクティブ・ファイルで使用できる文字は次のとおりです。

- 数字（0 ～ 9）
- 英大文字（A ～ Z）
- 英小文字（a ～ z）
- アンダスコア（_）
- ドット（.）
- スラッシュ（/）
- バック・スラッシュ，円マーク（\，\）
- コロン（:）（ファイル名のみ使用できます）
- S-JIS 文字（ファイル名のみ使用できます）
- 半角カナ文字（ファイル名のみ使用できます）
- #（コメント用）

リンク・ディレクティブ・ファイル中の“#”は，コメントの始まりを示します。“#”が現れた位置からその行の行末まではコメントとして扱われます。

5. 4. 2　ファイル名

リンク・ディレクティブ・ファイルにつける「ファイル名」には，ファイル名として指定できる文字を使用していれば，特に制限はありません。ただし，拡張子が必要です。推奨は“dir”です。CubeSuite を使用する際は，必ず“dir”，または“dr”としてください。また，文字数があまり長くなると，リンク時に扱える全体の文字数（OS 依存）を越えてしまい，リンクできなくなってしまうことがあるので注意が必要です。

リンク時には，コマンド・ラインやメイク・ファイルを使用する際は，“-Xlink_directive”オプションを使用してリンク・ディレクティブ・ファイルを指定します。

5. 4. 3　セグメント・ディレクティブ

ここでは，次の項目でセグメント・ディレクティブの書式について説明します。

- 指定項目
- セグメント・ディレクティブの指定例

(1) 指定項目

セグメント・ディレクティブで指定する項目は，次のとおりです。

表 5―5　セグメント・ディレクティブで指定できる項目

項目	指定形式	意味	省略
セグメント名	セグメント名	作成するセグメントの名前	不可
セグメント・タイプ	!LOAD	メモリにロードされるタイプ（固定）	一部不可[注]
セグメント属性	?[R][W][X]	作成するセグメントの属性が「読み出し可能（R）」「書き込み可能（W）」「実行可能（X）」であるかを指定（複数指定可能）	一部不可[注]
アドレス	V アドレス	作成するセグメントの先頭アドレス	可
最大メモリ・サイズ	L 最大メモリ・サイズ	作成するセグメントが占有するメモリの上限	可
ホール・サイズ	H ホール・サイズ	セグメントの後ろに作成するホールのサイズ（次のセグメントとの間に入れる余白）	可
充てん値	F 充てん値	ホール領域を埋めるのに用いる値	可
整列条件	A 整列条件	メモリ割り付けにおける整列条件（アライメント）	可

注　セグメント・タイプとセグメント属性は，一部省略不可となります。
そのセグメントに属するマッピング・ディレクティブの出力セクションとして予約セクションのみを指定した場合，タイプと属性は省略可能です。

出力セクションとして独自に作成したセクションを含む場合，タイプと属性は省略しないことを推奨します。タイプと属性を省略すると「!LOAD ?RW」を指定したものとして解釈します。

セグメント・ディレクティブの具体的な書式は次とおりです。

```
セグメント名：　!セグメント・タイプ　?セグメント属性　Vアドレス　L最大メモリ・サイズ　Hホール・サイズ　F充てん値　A整列条件{
　　：
　（マッピング・ディレクティブ）
　　：
};
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。

なお，「V アドレス」「L 最大メモリ・サイズ」「H ホール・サイズ」「F 充てん値」「A 整列条件」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―6　省略可能項目のデフォルト値（セグメント・ディレクティブ）

項目	デフォルト値
アドレス	先頭のセグメントである場合は 0x0 番地，以降のセグメントは，前のセグメントの終端に続いた値
最大メモリ・サイズ	0x100000（バイト），セグメントを配置するメモリサイズが 1M を越えるデバイスを品種指定した場合は，セグメントを配置するメモリのサイズ
ホール・サイズ	0x0（バイト）
充てん値	0x0000
整列条件	0x8（バイト）

注意　セグメント・ディレクティブは，アドレスの若い順に記述してください。

(a) セグメント名

作成するセグメントの名前を指定します。

セグメント作成においては，このセグメント名の指定は省略できません。

セグメント名として，任意の長さの文字列を指定できます。ただし，次表にある予約セクションを割り当てるセグメントは，セグメント名が固定されています。それ以外のセグメント名は使用できません。

表 5―7　セグメント名が固定されている予約セクション名

セクション名	セグメント名
.sidata .sibss .tidata .tibss .tidata.byte .tibss.byte .tidata.word .tibss.word	SIDATA

セクション名	セグメント名
.sedata .sebss	SEDATA
.sconst	SCONST

備考 .sconst は変更することはできますが，一部エラー・チェックが行われません。

(b) セグメント・タイプ

作成するセグメントのタイプを指定します。

セグメント作成においては，このセグメント・タイプの指定は一部省略不可となります。

そのセグメントに属するマッピング・ディレクティブの出力セクションとして予約セクションのみを指定した場合，セグメント・タイプの指定は省略可能です。

出力セクションとして独自に作成したセクションを含む場合，セグメント・タイプの指定は省略しないことを推奨します。省略すると「!LOAD」を指定したものとして解釈します。

ここで指定できるタイプは，現在のところ「メモリにロードされるセグメント・タイプ」で「LOAD」のみです。これ以外の値を指定すると，リンカはエラーを出力します。なお「LOAD」は小文字でも可です。

セグメント・タイプの指定は“!”を先頭とし，“!”の直後に空白を置かないでください。

(c) セグメント属性

作成するセグメントの属性を指定します。

セグメント作成においては，このセグメント属性の指定は一部省略不可となります。

そのセグメントに属するマッピング・ディレクティブの出力セクションとして予約セクションのみを指定した場合，セグメント属性の指定省略可能です。

出力セクションとして独自に作成したセクションを含む場合，セグメント属性の指定は省略しないことを推奨します。省略すると「?RW」を指定したものとして解釈します。

指定できるセグメント属性とその意味は次のとおりです。

なお，セグメント属性は，そのセグメントに属するマッピング・ディレクティブの属性に依存します。指定するときはマッピング・ディレクティブで指定するセクションの属性を考慮して設定する必要があります。

表 5―8 セグメント属性とその意味

セグメント属性	意味
R	読み出し可能なセグメント
W	書き込み可能なセグメント
X	実行可能なセグメント

セグメント属性は，複数個同時に指定することができ，R，W，X の順番は任意で，空白を置かずに続けて記述します。また，セグメント属性の指定は，“?”を先頭とし，“?”の直後に空白を置かないでください。

備考 1つのセグメント・ディレクティブにおいて，セグメント属性の指定が複数回行われた場合，リンカはエラーを出力しリンクを中止します。

例

```
SEG:    !LOAD   ?RX ?RW {};
```

(d) アドレス

作成するセグメントの開始アドレスを指定します。

セグメント作成においては，このアドレスの指定は省略できます。省略した場合，先頭のセグメントである場合は 0x0 番地，以降のセグメントは，前のセグメントの終端に続いた（アライメントを考慮した）値が開始アドレスとなります。

アドレス指定は，使用するマイコンのメモリ配置を考慮して指定する必要があります。

たとえば，V850E コアの場合，種類ごとに搭載しているメモリの大きさが違っているので，内蔵 ROM/RAM の開始・終了アドレスも違っています。そのため使用するマイコンに応じて，各セグメントの配置アドレスを考慮する必要があります。各 CPU のメモリ情報などに関しては，該当 CPU のユーザーズ・マニュアルを参照してください。

アドレスの値は偶数で指定してください。奇数を指定すると，リンカはメッセージを出力し，指定されたアドレスに 1 を加えたアドレスが指定されたものとみなしてリンクを続行します。

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。アドレスとして指定できる数値は 10 進数，16 進数のどちらかで，16 進数で指定する場合は数値の前に“0x”を記述してください。なお，アドレスの指定に式を用いることはできません。

備考 セグメント“DATA_CMN”の配置は，デフォルトの配置として RAM 領域の PE1 領域の先頭に配置されます。
他の場所に変更したい場合は，アドレス指定 (V) で指定してください。

(e) 最大メモリ・サイズ

作成するセグメントのメモリ・サイズの最大値を指定します。

これはセグメントが意図したサイズを越えないようにするための指定です。ですから，実際のサイズが最大メモリ・サイズとして指定した値よりも小さい場合は，次にくるセグメントは，そのセグメントの直後になります。

セグメント作成においては，この最大メモリ・サイズの指定は省略できます。省略した場合，0x100000（バイト），またはセグメントを配置するメモリ・サイズが 1M を越えるデバイスを品種指定した場合は，セグメントを配置するメモリのサイズがデフォルトの値として用いられます。

作成されたセグメントが最大メモリ・サイズで指定された値を越えた場合，リンカはエラーを出力し，リンクを中止します。

最大メモリ・サイズの指定は“L”（大文字・小文字どちらでも可）を先頭とし，“L”の直後に空白を置かないでください。なお，最大メモリ・サイズの指定に式を用いることはできません。

(f) ホール・サイズ

作成するセグメントのホールのサイズを指定します。

セグメントのホールとは，セグメントとセグメントの間の余白を意味します。ホール・サイズの指定を行った場合，そのセグメントの終わりにホールが作成されます。

セグメント作成においては，このホール・サイズの指定は省略できます。省略した場合，0x0（バイト）がデフォルトの値として用いられます（ホールは作成しません）。

ホール・サイズの指定は“H”（大文字・小文字どちらでも可）を先頭とし，“H”の直後に空白を置かないでください。

なお，ホール・サイズの指定に式を用いることはできません。

(g) 充てん値

セグメントの割り付けにおいて生じるホール，および“H”指定で明示的に作成したホールに対し，ホールの領域を埋めるのに用いる値（充てん値）を指定します。

充てん値の指定を用いる場合は，“-Xtwo_pass_link”オプションを指定し，2 パス・モードでリンクを行ってください。デフォルトの 1 パス・モードで充てん値の指定が用いられた場合，リンカはメッセージを出力し，この指定を無視してリンクを続行します。

セグメント作成においては，この充てん値の指定は省略できます。省略した場合，0x0000 がデフォルトの値として用いられます（0 で埋めます）。ただし，リンカのオプションで“-Xalign_fill”（充てん値指定オプション）が指定された場合，リンカはメッセージを出力し，この指定を無視してリンクを続行します。

充てん値の指定は“F”（大文字・小文字どちらでも可）を先頭とし，“F”の直後に空白を置かないでください。充てん値は 2 バイト 4 桁の 16 進数で指定してください。4 桁に満たない場合は，満たない桁分の 0 が指定されたものとします。また，ホールのサイズが 2 バイトに満たない場合は必要な桁数分だけ，指定された充てん値の下位から取り出します。なお，充てん値の指定に式を用いることはできません。

(h) 整列条件

作成するセグメントのメモリ割り付けにおいて，セグメントの整列条件（アライメント値）を指定します。

セグメント作成においては，この整列条件の指定は省略できます。省略した場合，0x8（バイト）がデフォルトの値として用いられます（8 バイトでアラインされます）。

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。整列条件の値は偶数で指定してください。奇数を指定すると，リンカはメッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

アドレスが指定された場合，アドレスの指定を優先し整列条件の指定は無視されます。

(2) セグメント・ディレクティブの指定例

次のようなセグメントを作成する場合を例とします。

表 5―9　セグメント例

項目	値
セグメント名	PROG1
セグメント・タイプ	読み出し可能，実行可能
配置アドレス	0x1000 番地
最大メモリ・サイズ	0x200000（バイト）
ホール・サイズ	0x20（バイト）
充てん値	0xFFFF
整列条件	0x16（バイト）

この場合，セグメント・ディレクティブの記述は次のようになります。

```
PROG1:  !LOAD   ?RX V0x1000 L0x200000   H0x20   F0xFFFF A0x16 {
      :
    （マッピング・ディレクティブ）
      :
};
```

備考　基本的に，セグメント・ディレクティブは，配置するアドレス順に記述しなくても問題ありません。ただし，.sedata セクション /.sebss セクションをもつセグメント（デフォルトでは "SEDATA セグメント "）に関しては，配置アドレスを省略したときにかぎり，この規則の例外となります。
CX では，SEDATA セグメントは内蔵 RAM 手前の領域を ep 相対の 1 命令参照するものとして定義しており，配置アドレスを省略すると，デバイス・ファイルに定義された内蔵 RAM の先頭アドレスから 0x8000 を引いたアドレスを指定したものとして扱います。

たとえば，リンク・ディレクティブ・ファイル中に次の記述があったとします。

```
SIDATA: !LOAD   ?RW V0xFFB000 {
    .tidata.byte    = $PROGBITS ?AW     .tidata.byte;
    .tibss.byte     = $NOBITS   ?AW     .tibss.byte;
    .tidata.word    = $PROGBITS ?AW     .tidata.word;
    .tibss.word     = $NOBITS   ?AW     .tibss.word;
    .sidata         = $PROGBITS ?AW     .sidata;
    .sibss          = $NOBITS   ?AW     .sibss;
};
SEDATA: !LOAD   ?RW {
    .sedata         = $PROGBITS ?AW     .sedata;
    .sebss          = $NOBITS   ?AW     .sebss;
};
DATA:   !LOAD   ?RW {
    .data           = $PROGBITS ?AW     .data;
    .sdata          = $PROGBITS ?AWG    .sdata;
    .sbss           = $NOBITS   ?AWG    .sbss;
    .bss            = $NOBITS   ?AW     .bss;
};
```

SEDATA のアドレスが省略されており，デバイス・ファイル情報から，この先頭アドレスは 0xFF2000（= 0xFFB000 ～ 0x8000）と判断します。SIDATA は 0xFFB000 番地に配置定義されているため，CX は内部的に SEDATA を SIDATA の前に移動してリンクをします。

また，その後に定義されている DATA セグメントもアドレスが省略されているため，SEDATA セグメントの直後に配置されることになります。

5. 4. 4 マッピング・ディレクティブ

ここでは，次の項目でマッピング・ディレクティブの書式について説明します。

- 指定項目
- マッピング・ディレクティブの指定例

(1) 指定項目

マッピング・ディレクティブで指定する項目は，次のとおりです。

表 5―10 マッピング・ディレクティブで指定できる項目

項目	指定形式	意味	省略
出力セクション名	出力セクション名	ロード・モジュールに出力されるセクションの名前	不可
セクション・タイプ	$PROGBITS $NOBITS	作成するセクションのタイプ	一部不可[注]

項目	指定形式	意味	省略
セクション属性	?[A][W][X][G]	作成するセクションの属性が「メモリを占有（A）」「書き込み可能（W）」「実行可能（X）」「gp と 16 ビット・ディスプレースメントで参照可（G）」であるかを指定（複数指定可能）	一部不可注
入力セクション名	入力セクション名	出力セクションに割り付ける入力セクションの名前	可
アドレス	V アドレス	作成するセクションの先頭アドレス	可
ホール・サイズ	H ホール・サイズ	セクションの後ろに作成するホールのサイズ（次のセクションとの間に入れる余白）	可
整列条件	A 整列条件	メモリ割り付けにおける整列条件（アライメント）	可
オブジェクト・モジュール・ファイル名	{ ファイル名 ファイル名 …}	入力セクションとして抽出したいセクションを含むオブジェクト・モジュール・ファイル（複数指定可，スペースで区切る）	可

注 セクション・タイプとセクション属性は，一部省略不可となります。
マッピング・ディレクティブの出力セクションとして予約セクションを指定した場合，タイプと属性は省略可能です。出力セクションとして独自に作成したセクションを指定した場合，タイプと属性は省略しないことを推奨します。タイプと属性を省略すると「$NOBITS ?AW」を指定したものとして解釈します。

マッピング・ディレクティブの具体的な書式は次とおりです。

```
出力セクション名 =
    $ セクション・タイプ
    ? セクション属性
    入力セクション名
    V アドレス
    H ホール・サイズ
    A 整列条件
    { オブジェクト・モジュール・ファイル名 [ オブジェクト・モジュール・ファイル名 ]...};
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。

「V アドレス」「H ホール・サイズ」「A 整列条件」「入力セクション名」「オブジェクト・モジュール・ファイル名」は省略できます。省略した場合はデフォルトの値が使われたり，決まった規則が用いられます。デフォルトの値，規則は次のとおりです。

表 5―11　マッピング・ディレクティブの指定項目で省略可能な値のデフォルト値／規則

項目	デフォルト値／規則
アドレス	セグメント・ディレクティブで指定されたアドレスに従う 複数セクションがあり，先頭のセクションでなければ，前のセクションの終端に続く値 先頭のセクションならば，セグメントの先頭に続く値
ホール・サイズ	0x0（バイト）
整列条件	.tidata.byte/. tibss.byte セクション：0x1（バイト） 上記セクション以外：0x4（バイト）
入力セクション	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出 オブジェクト・モジュール・ファイル名が指定されていれば，そのオブジェクトから抽出
オブジェクト・モジュール・ファイル名	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出 入力セクションが指定されていれば，作成する出力セクションと同じ属性を持つすべてのオブジェクトから抽出

次にそれぞれの指定項目について詳しく説明します。

（a）出力セクション名

ロード・モジュールに出力されるセクションの名前を指定します。セクション作成においては，この出力セクション名の指定は省略できません。

出力セクション名として，任意の長さの文字列を指定できます。

ただし，次表にあるセクションは，出力セクション名と入力セクション名の対応が固定されているので，別の名前のセクション名は指定できません。

表 5―12　セグメント名が固定されている予約セクション名

入力セクション名	出力セクション名
.tidata セクション	.tidata
.tibss セクション	.tibss
.tidata.byte セクション	.tidata.byte
.tibss.byte セクション	.tibss.byte
.tidata.word セクション	.tidata.word
.tibss.word セクション	.tibss.word
.sidata セクション	.sidata
.sibss セクション	.sibss
.sedata セクション	.sedata
.sebss セクション	.sebss
.sconst セクション	.sconst

入力セクション名	出力セクション名
.pro_epi_runtime セクション	.pro_epi_runtime

備考 同一セグメント・ディレクティブ内で，複数のマッピング・ディレクティブを記述することができますが，異なるセグメント・ディレクティブであっても，同じ出力セクション名を複数個指定することはできません。同じ出力セクション名が複数個指定された場合，リンカはエラーを出力し，リンクを中止します。

(b) セクション・タイプ

出力セクションのタイプを指定します。

セクション作成においては，この出力セクションのタイプ指定は一部省略不可となります。

出力セクションとして予約セクションを指定した場合，タイプ指定は省略可能です。

出力セクションとして独自に作成したセクションを指定した場合，タイプ指定は省略しないことを推奨します。タイプ指定を省略すると「$NOBITS」を指定したものとして解釈します。

指定できるセクション・タイプとその意味は次のとおりです。

表 5―13 セクション・タイプとその意味

セクション・タイプ	意味
PROGBITS	オブジェクト・モジュール・ファイル内に実際の値を持っているセクション → テキスト，初期値ありデータ（変数）
NOBITS	オブジェクト・モジュール・ファイル内に実際の値を持っていないセクション → 初期値なしデータ（変数）

セクション・タイプの指定は，“$”を先頭とし，“$”の直後に空白を置かないでください。

また，“$”のみが指定された場合，リンカはエラーを出力し，リンクを中止します。

(c) セクション属性

作成するセクションの属性を指定します。

セクション作成においては，このセクション属性の指定は一部省略不可となります。

出力セクションとして予約セクションを指定した場合，セクション属性の指定は省略可能です。

出力セクションとして独自に作成したセクションを指定した場合，セクション属性の指定は省略しないことを推奨します。セクション属性の指定を省略すると「?AW」を指定したものとして解釈します。

指定できるセクション属性とその意味は次のとおりです。

表 5―14 セクション属性とその意味

セクション属性	意味
A	メモリを占有するセクション（すべてのセクションが該当）

セクション属性	意味
W	書き込み可能なセクション（RAM 上に配置するセクション）
X	実行可能なセクション（主にテキスト・セクション）
G	グローバル・ポインタ（gp）と 16 ビットのディスプレースメントを用いて参照することのできるメモリ範囲内に割り付けるセクション（.sdata, .sbss セクション）

セクション属性は，複数個同時に指定でき，A，W，X，G の順番は任意で，空白を置かずに続けて記述します。また，セクション属性の指定は“?”を先頭とし，“?”の直後に空白を置かないでください。

マッピング・ディレクティブをセグメント・ディレクティブ内に指定する場合，指定するセクション属性は，そのセグメント・ディレクティブにおいて指定されているセグメント属性に一致するようにしてください。つまり，セクション属性 G は無視して，セクション属性 A，W，および X がそれぞれセグメント属性 R，W，および X に相当するものとして一致するようにしてください。

備考 1 つのマッピング・ディレクティブにおいて，セクション属性の指定が複数回行われた場合，リンカはエラーを出力しリンクを中止します。

例

```
sec      = $PROGBITS ?AX ?AW;
```

内蔵 ROM/ 内蔵命令 RAM に，書き込み可能な属性を持つセクションを配置した場合，メッセージを出力し，リンクを続行します。

(d) 入力セクション名

作成する出力セクションの基となる，入力セクション情報を指定します。

セクション作成においては，この入力セクション名の指定，オブジェクト・モジュール・ファイル名の指定は省略できます。省略した場合，次のような記述の組み合わせにより，出力セクションに出力される情報が変化します。

表 5―15 入力セクションとオブジェクト・モジュール・ファイルの組み合わせ別の出力

記述パターン		出力
(1)	入力セクション名＋オブジェクト・モジュール・ファイル名	指定した入力セクションを，指定したオブジェクトから抽出して出力
(2)	入力セクション名のみ	指定した入力セクションを，すべてのオブジェクトから抽出して出力
(3)	オブジェクト・モジュール・ファイル名のみ	作成する出力セクションと同じ属性を持つセクションを，指定されたオブジェクトから抽出して出力

記述パターン		出力
(4)	両方とも記述しなかった場合	作成する出力セクションと同じ属性を持つセクションを，すべてのオブジェクトから抽出して出力

具体的な例を示すと次のようになります。

表 5―16　入力セクションとオブジェクト・モジュール・ファイルの組み合わせの具体例

記述例	出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX usrsec1 {file1.obj};
}
	file1.obj にある，usrsec1 セクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX usrsec1;
}
	すべてのオブジェクトから，usrsec1 セクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX {file1.obj file2.obj};
}
	file1.obj と file2.obj にある，$PROGBITS タイプで，属性 A，X のセクションを抽出し，sec1 セクションとして出力

SEG1: !LOAD ?RX {
  sec1 = $PROGBITS ?AX;
}
	すべてのオブジェクトから，$PROGBITS タイプで，属性 A，X のセクションを抽出し，sec1 セクションとして出力

なお，セクションの配置時に候補が複数ある場合では，「表 5―15　入力セクションとオブジェクト・モジュール・ファイルの組み合わせ別の出力」内の [記述パターン] 項目で示した番号を優先順位として（同順位の場合は低位アドレス優先），セクションを配置します。

入力セクション名は，アプリケーションで設定したセクション名を指定してください。特にアプリケーション中で設定しなかった場合は，デフォルトのセクション名として定義されていますので，そのセクション名を指定してください。

なお，「(a) 出力セクション名」で説明したように，出力セクション名と入力セクション名の対応が固定されているものがあります。これに該当するものは，別の名前のセクション名を指定することはできません。

(e) アドレス

作成するセクションの開始アドレスを指定します。

セクション作成においては，このアドレスの指定は省略できます。省略した場合，セグメント・ディレクティブで指定されたアドレスに従います。複数セクションがあり，先頭のセクションでなければ，前のセクションの終端に続く値になります。

通常セクションのアドレス指定は，セグメントごとにまとめて行いますが，特定のセクションを特定のアドレスに割り付けたい場合に用いることができます。

アドレスの値は偶数で指定してください。奇数を指定すると，リンカはメッセージを出力し，指定されたアドレスに 1 を加えたアドレスが指定されたものとみなしてリンクを続行します。

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。アドレスとして指定できる数値は 10 進数，16 進数のどちらかで，16 進数で指定する場合は数値の前に“0x”を記述してください。なお，アドレスの指定に式を用いることはできません。

（f）ホール・サイズ

作成するセクションのホールのサイズを指定します。

セクションのホールとは，セクションとセクションの間の余白を意味します。ホール・サイズの指定を行った場合，そのセクションの終わりにホールが作成されます。

セクション作成においては，このホール・サイズの指定は省略できます。省略した場合，0x0（バイト）がデフォルトの値として用いられます（ホールは作成しません）。

ホール・サイズの指定は“H”（大文字・小文字どちらでも可）を先頭とし，“H”の直後に空白を置かないでください。なお，ホール・サイズの指定に式を用いることはできません。

（g）整列条件

作成するセクションのメモリ割り付けにおいて，セクションの整列条件（アライメント値）を指定します。

セクション作成においては，この整列条件の指定は省略できます。省略した場合はデフォルトの値が使用されますが，次に示すようにセクションの種類によってその値が違います。

表 5―17　セクションの種類とその整列条件のデフォルト値

セクション名	整列条件
.tidata.byte/.tibss.byte セクション	0x1（バイト）
内蔵命令 RAM 以外の TEXT 属性セクション	0x2（バイト）
上記以外のセクション	0x4（バイト）

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。

整列条件の値として指定できるものは，.tidata.byte，.tibss.byte セクションの場合は偶数か奇数，その他のセクションにおいては偶数のみです。.tidata.byte，.tibss.byte 以外のセクションで奇数を指定すると，リンカはメッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

注意　内蔵命令 RAM に配置するセクションの整列条件は，4（または 4 の倍数）とします。4（または 4 の倍数）以外の値を指定した場合，メッセージを出力して，指定された整列条件でリンクを続行します。
書き込み時には 4 バイト・アクセスが必要で，ミス・アライン・アクセスを許容しない内蔵命令 RAM の場合には，ROM 化したセクションを内蔵命令 RAM に転送する際に，コピーするセクションの整列条件を 4 としておく必要があります。

(h) オブジェクト・モジュール・ファイル名

オブジェクト・モジュール・ファイル名の指定は，マッピング・ディレクティブの最後に記述し，ファイル名を“{”と“}”で囲んでください。また，複数指定するときは「空白」で区切ります（ファイル名に「空白」が含まれている場合は，ファイル名を“”で囲みます）。

オブジェクト・モジュール・ファイルを複数指定した場合，指定した順で，下位アドレスから上位アドレスの方向に割り付けられます。ただし，リンカ起動時に指定する「リンクするオブジェクト」の記述順が，リンク・ディレクティブ・ファイルにおける順番と異なる場合，引数で指定したファイル名の順番が優先されます。

```
リンク・ディレクティブ
sec = $PROGBITS ?AX {filel.obj file2.obj file3.obj}
```

```
リンカ起動
cx file3.obj filel.obj file2.obj
        → file3.obj，file1.obj，file2.obj の順番で，下位から割り付く
```

マッピング・ディレクティブにおいて，オブジェクト・モジュール・ファイル名を指定する場合，その属性のセクションを含んでいるファイル名はすべて記述してください。

たとえば，file1.obj，file2.obj，file3.obj，file4.obj の 4 つのオブジェクトが存在し，これらすべてに text 属性のセクションが含まれていたとします。このとき

```
TEXT1:  !LOAD   ?RX {
    .text1  = $PROGBITS ?AX {filel.obj file2.obj};
};
TEXT2:  !LOAD   ?RX {
    .text2  = $PROGBITS ?AX {file3.obj};
};
```

というリンク・ディレクティブを記述し，file4.obj の text 属性の配置場所を特定しなかった場合，file4.obj 内の text 属性を，適当な text 属性のセクションを探して配置します。したがって，意図したとおりのマッピングにならない可能性がありますので注意が必要です（他のどのセクションにも配置されない場合，リンカはメッセージを出力します）。

また，異なるディレクトリに配置した同名ファイルを指定したい場合には，リンク・マップに表示されたパス付きファイル名で次のように指定することができます。

```
textsec1    = $PROGBITS ?AX {c:\work\dir1\file1.obj};
textsec2    = $PROGBITS ?AX {c:\work\dir2\file1.obj};
textsec3    = $PROGBITS ?AX {file1.obj};
```

上記の場合，指定したディレクトリに存在する file1.obj は，それぞれ textsec1/textsec2 に配置され，それ以外は textsec3 に配置されます。なお，この際のパスの指定方法は，リンク・マップに表示された形式のみであるため，記述の際に注意が必要です。

さらに，ライブラリ・ファイル中のオブジェクトを，入力オブジェクト名に指定することもできます。たとえば，libusr.lib というアーカイブ・ファイル中にある lib1.obj というオブジェクトを usrlib セクションに出力したい場合，次のように記述します。

```
usrlib  = $PROGBITS ?AX {lib1.obj(a:\usrlib\libusr.lib)};
```

指定したライブラリ内のオブジェクトすべてを配置指定したい場合は，次のように記述します。

```
usrlib  = $PROGBITS ?AX {libusr.lib};
```

上記の場合，libusr.lib 内のオブジェクトは usrlib セクションに配置されます。

オブジェクト・モジュール・ファイル名の指定は省略することができます。

例

```
sec     = $PROGBITS ?AX .text;
```

ファイル名の指定が省略された場合，CX リンカは他に指定されていないすべてのオブジェクト・モジュール・ファイルが指定されたものとみなします。

たとえば，下記の例で，オブジェクト・モジュール・ファイル file1.obj，file2.obj，file3.obj，および file4.obj を指定して起動されていた場合，

```
sec1    = $PROGBITS ?AX .text;
sec2    = $PROGBITS ?AX .text {file1.obj};
```

下記の例と同じになります。

```
sec1    = $PROGBITS ?AX .text {file2.obj file3.obj file4.obj};
sec2    = $PROGBITS ?AX .text {file1.obj};
```

(i) 同じ指定の場合

複数のセグメントに対して，同じセクション・タイプ／セクション属性／入力セクション名（省略可）／入力ファイル名（入力可）が指定された場合で，それに対応するセクションが存在した場合には，低位アドレスに配置されたセグメントに対し割り付けが行われます。

```
TEXT1:  !LOAD   ?RX V0x1000 {
    .text1  = $PROGBITS ?AX .text   {filel.obj file2.obj};
};
TEXT2:  !LOAD   ?RX V0x2000 {
    .text2  = $PROGBITS ?AX .text   {filel.obj file2.obj};
};
```

上記の場合，セクション・タイプ／セクション属性／入力セクション名／入力ファイル名が TEXT1 と TEXT2 で同じであるため，低位アドレスに配置されている TEXT1 に対して割り付けします。

(2) マッピング・ディレクティブの指定例

次のような出力セクションを作りたい場合を例とし，2 種類のセクションを作るとします。

表 5―18　マッピング・ディレクティブの指定例

項目	値 ― 1	値 ― 2
出力セクション名	.text	textsec1
セクション・タイプ	テキスト	テキスト
セクション属性	読み出し可能，実行可能	読み出し可能，実行可能
ホール・サイズ	0x10（バイト）	0x20（バイト）
充てん値	0xFFFF	0xFFFF
整列条件	0x10（バイト）	0x10（バイト）
入力セクション名	.text	usrsec1
オブジェクト・モジュール・ファイル名	main.obj	―

この場合，マッピング・ディレクティブの記述は次のようになります。

```
.text       = $PROGBITS ?AX H0x10   F0xFFFF A0x10   .text   {main.obj};
textsec1    = $PROGBITS ?AX H0x20   F0xFFFF A0x10   usrsec1;
```

5. 4. 5　シンボル・ディレクティブ

ここでは，次の項目でシンボル・ディレクティブの書式について説明します。

- 指定項目
- シンボル・ディレクティブの指定例

(1) 指定項目

シンボル・ディレクティブで指定する項目は，次のとおりです。

- tp シンボル

表 5―19　tp シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する tp シンボルの名前	不可
シンボル種別	%TP_SYMBOL	作成するシンボル種別（固定）	不可
アドレス	V アドレス	作成する tp シンボルのアドレス	可

項目	指定形式	意味	省略
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可
セグメント名	{ セグメント名 セグメント名 …}	作成する tp の参照対象としたいセグメント名（複数指定可，空白で区切る）	可

具体的な書式は次のようになります。

```
シンボル名 @%TP_SYMBOL　V アドレス　A 整列条件　{ セグメント名　セグメント名 };
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。
「V アドレス」「A 整列条件」「セグメント名」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―20　tp シンボルのデフォルト値

項目	デフォルト値
アドレス	セグメント名が指定されていた場合，そのセグメント内で最下位に割り付けられた text 属性のセクションの先頭アドレス セグメント名が指定されていない場合，ロード・モジュールに存在する text 属性セグメント内で，最下位に割り付けられた text 属性のセクションの先頭アドレス
整列条件	0x4（バイト）
セグメント名	オブジェクトに存在するすべての text 属性セグメントを対象とします。

- gp シンボル

表 5―21　gp シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する gp シンボルの名前	不可
シンボル種別	%GP_SYMBOL	作成するシンボル種別（固定）	不可
ベース・シンボル名	& ベース・シンボル名	gp シンボルを tp シンボルからのオフセット値として指定する際の tp シンボル名	可
アドレス	V アドレス	作成する gp シンボルのアドレス	可
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可
セグメント名	{ セグメント名 セグメント名 …}	作成する gp の参照対象としたいセグメント名（複数指定可，空白で区切る）	可

具体的な書式は次のようになります。

```
シンボル名 @%GP_SYMBOL  & ベース・シンボル名  V アドレス  A 整列条件  { セグメント名  セグメント名 };
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。
「& ベース・シンボル名」,「V アドレス」,「A 整列条件」,「セグメント名」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―22 gp シンボルのデフォルト値

項目	デフォルト値
ベース・シンボル名	tp シンボルからのオフセットではなく，gp シンボル値として決定されるアドレス
アドレス	リンカが下記事項から gp シンボル値を決定します。 - sdata 属性 /sbss 属性 /data 属性 /bss 属性のセクションの有無 - ベース・シンボル指定の有無
整列条件	0x4（バイト）
セグメント名	オブジェクトに存在するすべての sdata/data/sbss/bss 属性セクションを含むセグメントを対象とします。

- ep シンボル

表 5―23 ep シンボル作成で指定できる項目

項目	指定形式	意味	省略
シンボル名	シンボル名	作成する ep シンボルの名前	不可
シンボル種別	%EP_SYMBOL	作成するシンボル種別（固定）	不可
アドレス	V アドレス	作成する ep シンボルのアドレス	可
整列条件	A 整列条件	シンボル値の整列条件（アライメント）	可

具体的な書式は次のようになります。

```
シンボル名 @%EP_SYMBOL  V アドレス  A 整列条件 ;
```

それぞれの項目は，空白で区切って記述します。また，終わりには，必ず“;”を付けます。
「V アドレス, 」「A 整列条件」は省略できます。省略した場合はデフォルトの値が使われます。デフォルトの値は次のとおりです。

表 5―24　ep シンボルのデフォルト値

項目	デフォルト値
アドレス	リンカが下記事項から ep シンボル値を決定します。 - SIDATA セグメントの有無 - デバイス・ファイル中の内蔵 RAM 領域の定義の有無
整列条件	0x4（バイト）

次にそれぞれの指定項目について詳しく説明します。

(a) シンボル名【 該当シンボル：tp，gp，ep 】

生成するシンボルの名前を指定します。シンボル作成においては，このシンボル名の指定は省略できません。

シンボル名として，任意の長さの文字列を指定できます。

(b) シンボル種別【 該当シンボル：tp，gp，ep 】

tp シンボルを生成するか，gp シンボルを生成するか，ep シンボルを生成するかを指定します。シンボル作成においては，このシンボル種別の指定は省略できません。

ここで指定できる種別は「tp シンボル」，「gp シンボル」，「ep シンボル」のどれかで，それぞれ「TP_SYMBOL」，「GP_SYMBOL」，「EP_SYMBOL」です。これ以外の値を指定すると，リンカはエラーを出力します。

シンボル種別の指定は“%”を先頭とし，“%”の直後に空白を置かないでください。

(c) ベース・シンボル名【 該当シンボル：gp 】

gp シンボルの生成において，gp シンボル値を定める際に用いる tp シンボルを指定します。ベース・シンボル名を指定すると，tp シンボル値からのオフセット値が gp シンボル値となります。

gp シンボル作成においては，このベース・シンボル名の指定は省略できます。

ベース・シンボルの指定は“&”を先頭とし，“&”の直後に空白を置かないでください。“&”の後にベースとしたい tp シンボル名を記述します。

(d) アドレス【 該当シンボル：tp，gp，ep 】

tp シンボル値，gp シンボル値，つまり，アドレスを指定します。

シンボル作成においては，このアドレスの指定は省略できます。アドレスを省略した場合は，次のように決定されます。

表 5―25　tp シンボル，gp シンボル，ep シンボルのアドレス指定

シンボル値	決定規則
tp シンボル	- セグメント名が指定されていた場合 そのセグメント内で最下位に割り付けられた text 属性のセクションの先頭アドレス - セグメント名が指定されていない場合 ロード・モジュールに存在する text 属性セグメント内で，最下位に割り付けられた text 属性のセクションの先頭アドレス
gp シンボル	リンカが下記事項を調べ，gp シンボル値を決定します。 - sdata 属性 /sbss 属性 /data 属性 /bss 属性のセクションの有無 - ベース・シンボル指定の有無
ep シンボル	リンカが下記事項から ep シンボル値を決定します。 - SIDATA セグメントの有無 - デバイス・ファイル中の内蔵 RAM 領域の定義の有無

アドレスの指定は“V”（大文字・小文字どちらでも可）を先頭とし，“V”の直後に空白を置かないでください。

(e) 整列条件【 該当シンボル：tp，gp，ep 】

作成する tp シンボル値，gp シンボル値，ep シンボル値の設定における整列条件（アライメント値）を指定します。

シンボルの作成においては，この整列条件の指定は省略できます。省略した場合はデフォルトの値が使用されます。デフォルト値は 0x4（バイト）です。

整列条件の指定は“A”（大文字・小文字どちらでも可）を先頭とし，“A”の直後に空白を置かないでください。整列条件の値は偶数で指定してください。奇数を指定すると，リンカはメッセージを出力し，指定された値に 1 を加えた値が指定されたものとみなしてリンクを続行します。なお，整列条件の指定に式を用いることはできません。

(f) セグメント名【 該当シンボル：tp，gp 】

作成する tp シンボル値，gp シンボル値の参照対象とするセグメント名を指定します。

つまり，作成する tp シンボル，gp シンボルで参照したいセグメントを指定します。参照対象のセグメントは複数指定することができます。

シンボルの作成においては，このセグメント名の指定は省略できます。省略した場合はデフォルトとして，次のように指定されたものとみなされます。

表 5―26　tp シンボル，gp シンボルの参照対象セグメント名

シンボル値	決定規則
tp シンボル	オブジェクトに存在するすべての text 属性セグメントを対象とします。

シンボル値	決定規則
gp シンボル	オブジェクトに存在するすべての sdata/data/sbss/bss 属性セクションを含むセグメントを対象とします。

なお，gp シンボル参照の対象とするセグメント名には，gp 相対参照が前提となっているセグメント名を指定してください。

たとえば，ep 相対参照が前提となっている .sedata や .sebss セクションを含むセグメントは指定しないでください。

ただし，省略した場合，sdata/sbss 属性セクションを複数セグメントに配置している場合に低位アドレスに配置されたセグメントが優先され，gp シンボル値が意図した値とならない可能性があります。gp 相対以外のセクションを作成する場合，gp シンボルの対象セグメント名には，必ず適切なセグメントを指定するようにしてください。

セグメント名の指定は，シンボル・ディレクティブの最後に記述し，セグメント名“{”と“}”で囲んでください。また，複数指定するときは「空白」で区切ります。

(2) シンボル・ディレクティブの指定例

次のようなシンボルを作成する場合とします。

表 5―27 シンボル・ディレクティブの指定例

シンボル	指摘項目	指定値
tp シンボル	シンボル名	__tp_TEXT
	参照対象セグメント名	TEXT1
gp シンボル	シンボル名	__gp_DATA
	オフセット指定シンボル	__tp_TEXT
	参照対象セグメント名	DATA1，DATA2
ep シンボル	シンボル名	__ep_DATA
	アドレス	0xFFFFD000

この場合，シンボル・ディレクティブの記述は次のようになります。

```
__tp_TEXT@%TP_SYMBOL    {TEXT1};
__gp_DATA@%GP_SYMBOL    &__tp_TEXT  {DATA1 DATA2};
__ep_DATA@%EP_SYMBOL    V0xFFFFD000;
```

なお，シンボル・ディレクティブの指定がされていなかった場合，シンボルは作成されないので注意が必要です。

5.5　予約語

リンク・ディレティブ・ファイルには予約語が存在します。予約語を指定された用途以外に使用することはできません。

予約語は次のとおりです。

- セグメント名（SIDATA，SEDATA，SCONST）
- セグメント・タイプ（LOAD）
- 出力セクション名（.tidata，.tibss など）
- セクション・タイプ（PROGBITS，NOBITS）
- シンボル種別（TP_SYMBOL，GP_SYMBOL，EP_SYMBOL）

第 6 章　関数仕様

この章では，CX 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

CX で提供しているライブラリは，次のとおりです。

表 6―1　提供ライブラリ

提供ライブラリ	ライブラリ名	概要
標準ライブラリ	libc.lib libc22.lib libc26.lib libc32.lib libccn.lib	可変個引数関数 文字列関数 メモリ管理関数 文字変換関数 文字分類関数 標準入出力関数 標準ユーティリティ関数 非局所分岐関数
数学ライブラリ		数学関数
初期化処理ライブラリ		周辺装置の初期化関数
ROM 化用ライブラリ		コピー関数
マルチコア用ライブラリ		マルチコア用擬似 main 関数
ランタイム・ライブラリ		演算用ランタイム関数 関数前後処理ランタイム関数
V850E2V3-FPU 使用ライブラリ	libf32.lib libf64.lib	V850E2V3-FPU 使用関数
データ位置非依存ライブラリ	libp.lib	
データ位置依存ライブラリ	libnp.lib	

標準ライブラリや数学ライブラリを，アプリケーション内で使用するときは，関連するヘッダ・ファイルをインクルードして，ライブラリ関数を使用します。

また，リンク・オプション（-l）で，これらのライブラリを参照するようにします。

ただし，可変個引数関数，文字変換関数，文字分類関数だけを使用している場合，ライブラリの参照を行う必要はありません。

CubeSuite を使用する場合，これらのライブラリはデフォルトで参照する設定になっています。

演算用ランタイム関数は，浮動小数点演算や整数演算を行うときに，CX が自動的に呼び出すルーチンです。また，関数前後処理ランタイム関数も，関数のプロローグ／エピローグ処理で CX が自動的に呼び出すルーチンです。

したがって，“演算用ランタイム関数”，および“関数前後処理ランタイム関数”の 2 つは，他のライブラリ関数とは異なり，C ソースやアセンブラ・ソースで記述する関数ではありません。

ROM 化用ライブラリは，リンカが参照するライブラリです。パッキングされたデータをコピーする関数（_rcopy，_rcopy1，_rcopy2，_rcopy4）が格納されています。

以降に各ライブラリについて記します。
なお，表中の各要素は，以下の意味を持ちます。

関数／マクロ名	関数／マクロの名前です。
概要	関数／マクロの機能概要です。
#include	この関数／マクロを使用するうえで，C ソースでインクルードする必要あるヘッダ・ファイルです。#include 指令でインクルードしてください。 例外発生時の errno を使用する場合は“errno.h”もインクルードする必要があります。
ANSI	この関数が ANSI 規格で規定されている関数かどうかの区別です。 規定されていれば“○”，規定されていなければ“－”がついています。
const	この関数／マクロが，メモリ領域“const 領域”を使用するかどうかの区別です。 .const セクションを使用する場合“○”，使用しない場合“－”がついています。
sdata	この関数／マクロが，メモリ領域“sdata 領域”を使用するかどうかの区別です。 つまり，関数が初期値を持つデータを RAM に配置しているかどうかを区別します。セクション名は“.sdata”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“.sdata セクション”を生成してください。 .sdata セクションを使用する場合“○”，使用しない場合“－”がついています。“○”の場合，初期値ありデータを持つ必要があるため，初期値をプログラム実行前に RAM にコピーする必要があります。つまり，“コピー関数”を用いて ROM 化処理を行う必要があります。
sbss	この関数／マクロが，メモリ領域“sbss 領域”を使用するかどうかの区別です。 つまり，関数がテンポラリとして RAM を使用するかどうかを区別します。セクション名は“.sbss”でなくてはならないため，ユーザ・アプリケーションでこの領域を使用していなくても，“.sbss セクション”を生成してください。 .sbss セクションを使用する場合“○”，使用しない場合“－”がついています。.sbss セクションは，初期値を持たないデータが配置されるので，“.sdata 使用”のときと違って ROM 化処理を行う必要はありません。
Re-ent	“リエントラント性”を持つかどうかの区別です。 リエントラント性がある場合“○”，ない場合“－”がついています。 “リエントラント”とは“再入可能”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

6. 1. 1 標準ライブラリ

標準ライブラリに収録されている関数を示します。

(1) 可変個引数関数

表 6―2 可変個引数関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
va_start	stdarg.h	○	―	―	―	○
va_end	stdarg.h	○	―	―	―	○
va_arg	stdarg.h	○	―	―	―	○

(2) 文字列関数

表 6―3 文字列関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
index	string.h	―	―	―	―	○
strpbrk	string.h	○	―	―	―	○
rindex	string.h	―	―	―	―	○
strrchr	string.h	○	―	―	―	○
strchr	string.h	○	―	―	―	○
strstr	string.h	○	―	―	―	○
strspn	string.h	○	―	―	―	○
strcspn	string.h	○	―	―	―	○
strcmp	string.h	○	―	―	―	○
strncmp	string.h	○	―	―	―	○
strcpy	string.h	○	―	―	―	○
strncpy	string.h	○	―	―	―	○
strcat	string.h	○	―	―	―	○
strncat	string.h	○	―	―	―	○
strtok	string.h	○	―	―	○	―
strlen	string.h	○	―	―	―	○
strerror	string.h	○	○	○	―	―

(3) メモリ管理関数

表 6―4 メモリ管理関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
memchr	string.h	○	―	―	―	○
memcmp	string.h	○	―	―	―	○
bcmp	string.h	―	―	―	―	○
memcpy	string.h	○	―	―	―	○
bcopy	string.h	―	―	―	―	○
memmove	string.h	○	―	―	―	○
memset	string.h	○	―	―	―	○

(4) 文字変換関数

表 6―5 文字変換関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
toupper	ctype.h	○	○	―	―	○
_toupper	ctype.h	―	―	―	―	○
tolower	ctype.h	○	○	―	―	○
_tolower	ctype.h	―	―	―	―	○
toascii	ctype.h	―	―	―	―	○

(5) 文字分類関数

表 6―6 文字分類関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
isalnum	ctype.h	○	○	―	―	○
isalpha	ctype.h	○	○	―	―	○
isascii	ctype.h	○	○	―	―	○
isupper	ctype.h	○	○	―	―	○
islower	ctype.h	○	○	―	―	○
isdigit	ctype.h	○	○	―	―	○
isxdigit	ctype.h	○	○	―	―	○
iscntrl	ctype.h	○	○	―	―	○
ispunct	ctype.h	○	○	―	―	○
isspace	ctype.h	○	○	―	―	○
isprint	ctype.h	○	○	―	―	○

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
isgraph	ctype.h	○	○	－	－	○

（6）標準入出力関数

表 6―7 標準入出力関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
fread	stdio.h	○	－	－	－	○
getc	stdio.h	○	－	－	－	○
fgetc	stdio.h	○	－	－	－	○
fgets	stdio.h	○	－	－	－	○
fwrite	stdio.h	○	－	－	－	○
putc	stdio.h	○	－	－	－	○
fputc	stdio.h	○	－	－	－	○
fputs	stdio.h	○	－	－	－	○
getchar	stdio.h	○	－	○	－	－
gets	stdio.h	○	－	○	－	－
putchar	stdio.h	○	－	○	－	－
puts	stdio.h	○	－	○	－	－
sprintf	stdio.h	○	○	－	○	－注
fprintf	stdio.h	○	○	－	○	－注
vsprintf	stdio.h	○	○	－	○	－
printf	stdio.h	○	○	－	○	－
vfprintf	stdio.h	○	○	－	○	－注
vprintf	stdio.h	○	○	－	○	－
sscanf	stdio.h	○	○	－	－	○
fscanf	stdio.h	○	○	－	－	○
scanf	stdio.h	○	○	○	－	－
ungetc	stdio.h	○	－	－	－	○
rewind	stdio.h	○	－	－	－	○
perror	stdio.h	○	○	○	○	－

注 例外発生時の errno の更新と matherrf (matherr) ／ matherrd の呼び出しに関してのみリエントラント性を持ちません。

備考 例外発生時の errno を使用する場合は，errno.h をインクルードする必要があります。

(7) 標準ユーティリティ関数

表 6―8 標準ユーティリティ関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
abs	stdlib.h	○	―	―	―	○
labs	stdlib.h	○	―	―	―	○
llabs	stdlib.h	―	―	―	―	○
bsearch	stdlib.h	○	―	―	―	○
qsort	stdlib.h	○	―	―	―	○
div	stdlib.h	○	―	―	―	○
ldiv	stdlib.h	○	―	―	―	○
lldiv	stdlib.h	―	―	―	―	○
itoa	stdlib.h	―	―	―	―	○
ltoa	stdlib.h	―	―	―	―	○
ultoa	stdlib.h	―	―	―	―	○
lltoa	stdlib.h	―	―	―	―	○
ulltoa	stdlib.h	―	―	―	―	○
ecvt	stdlib.h	―	○	―	○	―
ecvtf	stdlib.h	―	○	―	○	―
fcvt	stdlib.h	―	○	―	○	―
fcvtf	stdlib.h	―	○	―	○	―
gcvt	stdlib.h	―	○	―	○	_注1
gcvtf	stdlib.h	―	○	―	○	_注1
atoi	stdlib.h	○	○	―	○	_注2
atol	stdlib.h	○	○	―	○	_注2
atoll	stdlib.h	―	○	―	○	_注2
strtol	stdlib.h	○	○	―	○	_注2
strtoul	stdlib.h	○	○	―	○	_注2
strtoll	stdlib.h	―	○	―	○	_注2
strtoull	stdlib.h	―	○	―	○	_注2
atoff	stdlib.h	○	○	―	○	_注2
atof	stdlib.h	○	○	―	―	―
strtodf	stdlib.h	○	○	―	○	_注2
strtod	stdlib.h	○	○	―	○	_注2
calloc	stdlib.h	○	―	○	○	―
malloc	stdlib.h	○	―	○	○	―
realloc	stdlib.h	○	―	○	○	―
free	stdlib.h	○	―	○	○	―

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
rand	stdlib.h	○	–	○	–	–
srand	stdlib.h	○	–	○	–	–

注 1. 例外発生時の errno の更新と matherrf (matherr) ／ matherrd の呼び出しに関してのみリエントラント性を持ちません。

2. 例外発生時に errno が更新されたときは，リエントラント性は持ちません。

備考 例外発生時の errno を使用する場合は，errno.h をインクルードする必要があります。

(8) 非局所分岐関数

表 6―9 非局所分岐関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
longjmp	setjmp.h	○	–	–	–	–
setjmp	setjmp.h	○	–	–	–	○

6. 1. 2 数学ライブラリ

数学ライブラリに収録されている関数を示します。

(1) 数学関数

表 6―10 数学関数

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
j0f	math.h	–	○	–	○	–注
j1f	math.h	–	○	–	○	–注
jnf	math.h	–	○	–	○	–注
y0f	math.h	–	○	–	○	–注
y1f	math.h	–	○	–	○	–注
ynf	math.h	–	○	–	○	–注
erff	math.h	–	○	–	○	–注
erfcf	math.h	–	○	–	○	–注
expf	math.h	○	○	–	○	–注
exp	math.h	○	○	–	○	–注
logf	math.h	○	○	–	○	–注
log	math.h	○	○	–	○	–注
log2f	math.h	–	○	–	○	–注
log10f	math.h	○	○	–	○	–注

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
log10	math.h	○	○	–	○	–注
powf	math.h	○	○	–	○	–注
pow	math.h	○	○	–	○	–注
sqrtf	math.h	○	○	–	○	–注
sqrt	math.h	○	○	–	○	–注
cbrtf	math.h	–	○	–	○	–注
cbrt	math.h	–	○	–	○	–注
ceilf	math.h	○	–	–	–	○
ceil	math.h	○	–	–	–	○
fabsf	math.h	○	–	–	–	○
fabs	math.h	○	–	–	–	○
floorf	math.h	○	–	–	–	○
floor	math.h	○	–	–	–	○
fmodf	math.h	○	○	–	○	–注
fmod	math.h	○	○	–	○	–注
frexpf	math.h	○	○	–	○	–注
frexp	math.h	○	○	–	○	–注
ldexpf	math.h	○	○	–	○	–注
ldexp	math.h	○	○	–	○	–注
modff	math.h	○	–	–	–	○
modf	math.h	○	–	–	–	○
gammaf	math.h	–	○	–	○	–注
hypotf	math.h	–	○	–	○	–注
matherrf (matherr)	math.h	–	–	–	–	○
matherrd	math.h	–	–	–	–	○
cosf	math.h	○	○	–	○	–注
cos	math.h	○	○	–	○	–注
sinf	math.h	○	○	–	○	–注
sin	math.h	○	○	–	○	–注
tanf	math.h	○	○	–	○	–注
tan	math.h	○	○	–	○	–注
acosf	math.h	○	○	–	○	–注
acos	math.h	○	○	–	○	–注
asinf	math.h	○	○	–	○	–注
asin	math.h	○	○	–	○	–注
atanf	math.h	○	○	–	○	–注

関数／マクロ名	#include	ANSI	const	sdata	sbss	Re-ent
atan	math.h	○	○	–	○	–注
atan2f	math.h	○	○	–	○	–注
atan2	math.h	○	○	–	○	–注
coshf	math.h	○	○	–	○	–注
cosh	math.h	○	○	–	○	–注
sinhf	math.h	○	○	–	○	–注
sinh	math.h	○	○	–	○	–注
tanhf	math.h	○	○	–	○	–注
tanh	math.h	○	○	–	○	–注
acoshf	math.h	–	○	–	○	–注
asinhf	math.h	–	○	–	○	–注
atanhf	math.h	–	○	–	○	–注

注 例外発生時の errno の更新と matherrf (matherr) ／ matherrd の呼び出しに関してのみリエントラント性を持ちません。

備考 例外発生時の errno を使用する場合は“errno.h”を，“汎整数型の各種限界値”をマクロ名で使用する場合は“limits.h”を，浮動小数点型の各種限界値を使用する場合は“float.h”をインクルードする必要があります。

6. 1. 3 初期化処理ライブラリ

初期化処理ライブラリに収録されている関数を示します。

(1) 周辺装置の初期化関数

表 6―11 周辺装置の初期化関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
hdwinit	–	–	–	–	○

6. 1. 4 ROM 化用ライブラリ

ROM 化用ライブラリに収録されている関数を示します。

(1) コピー関数

表 6―12 コピー関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
_rcopy	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
_rcopy1	–	–	–	–	○
_rcopy2	–	–	–	–	○
_rcopy4	–	–	–	–	○

6. 1. 5 マルチコア用ライブラリ

マルチコア用ライブラリに収録されている関数を示します。

（1）マルチコア用擬似 main 関数

表 6―13 マルチコア用擬似 main 関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
main_pe2	–	–	–	–	○
main_pe3	–	–	–	–	○
main_pe4	–	–	–	–	○
main_pe5	–	–	–	–	○
main_pe6	–	–	–	–	○
main_pe7	–	–	–	–	○
main_pe8	–	–	–	–	○
main_pe9	–	–	–	–	○
main_pe10	–	–	–	–	○
main_pe11	–	–	–	–	○
main_pe12	–	–	–	–	○
main_pe13	–	–	–	–	○
main_pe14	–	–	–	–	○
main_pe15	–	–	–	–	○
main_pe16	–	–	–	–	○
main_pe17	–	–	–	–	○
main_pe18	–	–	–	–	○
main_pe19	–	–	–	–	○
main_pe20	–	–	–	–	○
main_pe21	–	–	–	–	○
main_pe22	–	–	–	–	○
main_pe23	–	–	–	–	○
main_pe24	–	–	–	–	○
main_pe25	–	–	–	–	○
main_pe26	–	–	–	–	○
main_pe27	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
main_pe28	–	–	–	–	○
main_pe29	–	–	–	–	○
main_pe30	–	–	–	–	○
main_pe31	–	–	–	–	○

6. 1. 6 ランタイム・ライブラリ

ランタイム・ライブラリに収録されている関数を示します。

(1) 演算用ランタイム関数

表 6―14 演算用ランタイム関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___addf.s	–	○	–	–	–注
___subf.s	–	○	–	–	–注
___mulf.s	–	○	–	–	–注
___divf.s	–	○	–	–	–注
___cmpf.s	–	○	–	–	–注
___fcmp.s	–	○	–	–	–注
___negf.s	–	○	–	–	–注
___notf.s	–	○	–	–	–注
___addf.d	–	○	–	–	–注
___subf.d	–	○	–	–	–注
___mulf.d	–	○	–	–	–注
___divf.d	–	○	–	–	–注
___fcmp.d	–	○	–	–	–注
___negf.d	–	○	–	–	–注
___notf.d	–	○	–	–	–注
___add.l	–	–	–	–	○
___sub.l	–	–	–	–	○
___mul.l	–	–	–	–	○
___div.l	–	–	–	–	○
___div.ul	–	–	–	–	○
___mod.l	–	–	–	–	○
___mod.ul	–	–	–	–	○
___shl.l	–	–	–	–	○
___shr.l	–	–	–	–	○
___sar.l	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___inc.l	–	–	–	–	○
___dec.l	–	–	–	–	○
___not.l	–	–	–	–	○
___neg.l	–	–	–	–	○
___cmp.l	–	–	–	–	○
___cmp.ul	–	–	–	–	○
___bext.l	–	–	–	–	○
___bext.ul	–	–	–	–	○
___bins.l	–	–	–	–	○
___cvt.ws	–	–	–	–	○
___cvt.wd	–	–	–	–	○
___cvt.uws	–	–	–	–	○
___cvt.uwd	–	–	–	–	○
___cvt.ls	–	–	–	–	○
___cvt.ld	–	–	–	–	○
___cvt.uls	–	–	–	–	○
___cvt.uld	–	–	–	–	○
___trnc.sw	–	–	–	–	○
___trnc.dw	–	–	–	–	○
___trnc.suw	–	–	–	–	○
___trnc.duw	–	–	–	–	○
___trnc.sl	–	–	–	–	○
___trnc.dl	–	–	–	–	○
___trnc.sul	–	–	–	–	○
___trnc.dul	–	–	–	–	○
___cvt.sd	–	–	–	–	○
___cvt.ds	–	–	–	–	○
___mul	–	–	–	–	○
___mulu	–	–	–	–	○
___div	–	–	–	–	○
___divu	–	–	–	–	○
___mod	–	–	–	–	○
___modu	–	–	–	–	○

注 matherrf (matherr) ／ matherrd の呼び出しに関してリエントラント性を持ちません。

(2) 関数前後処理ランタイム関数

表 6―15 関数前後処理ランタイム関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___Epush250	―	―	―	―	○
___Epush251	―	―	―	―	○
___Epush252	―	―	―	―	○
___Epush253	―	―	―	―	○
___Epush254	―	―	―	―	○
___Epush260	―	―	―	―	○
___Epush261	―	―	―	―	○
___Epush262	―	―	―	―	○
___Epush263	―	―	―	―	○
___Epush264	―	―	―	―	○
___Epush270	―	―	―	―	○
___Epush271	―	―	―	―	○
___Epush272	―	―	―	―	○
___Epush273	―	―	―	―	○
___Epush274	―	―	―	―	○
___Epush280	―	―	―	―	○
___Epush281	―	―	―	―	○
___Epush282	―	―	―	―	○
___Epush283	―	―	―	―	○
___Epush284	―	―	―	―	○
___Epush290	―	―	―	―	○
___Epush291	―	―	―	―	○
___Epush292	―	―	―	―	○
___Epush293	―	―	―	―	○
___Epush294	―	―	―	―	○
___Epushlp0	―	―	―	―	○
___Epushlp1	―	―	―	―	○
___Epushlp2	―	―	―	―	○
___Epushlp3	―	―	―	―	○
___Epushlp4	―	―	―	―	○
___push2000	―	―	―	―	○
___push2001	―	―	―	―	○
___push2002	―	―	―	―	○
___push2003	―	―	―	―	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___push2004	–	–	–	–	○
___push2040	–	–	–	–	○
___push2100	–	–	–	–	○
___push2101	–	–	–	–	○
___push2102	–	–	–	–	○
___push2103	–	–	–	–	○
___push2104	–	–	–	–	○
___push2140	–	–	–	–	○
___push2200	–	–	–	–	○
___push2201	–	–	–	–	○
___push2202	–	–	–	–	○
___push2203	–	–	–	–	○
___push2204	–	–	–	–	○
___push2240	–	–	–	–	○
___push2300	–	–	–	–	○
___push2301	–	–	–	–	○
___push2302	–	–	–	–	○
___push2303	–	–	–	–	○
___push2304	–	–	–	–	○
___push2340	–	–	–	–	○
___push2400	–	–	–	–	○
___push2401	–	–	–	–	○
___push2402	–	–	–	–	○
___push2403	–	–	–	–	○
___push2404	–	–	–	–	○
___push2440	–	–	–	–	○
___push2500	–	–	–	–	○
___push2501	–	–	–	–	○
___push2502	–	–	–	–	○
___push2503	–	–	–	–	○
___push2504	–	–	–	–	○
___push2540	–	–	–	–	○
___push2600	–	–	–	–	○
___push2601	–	–	–	–	○
___push2602	–	–	–	–	○
___push2603	–	–	–	–	○
___push2604	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___push2640	–	–	–	–	○
___push2700	–	–	–	–	○
___push2701	–	–	–	–	○
___push2702	–	–	–	–	○
___push2703	–	–	–	–	○
___push2704	–	–	–	–	○
___push2740	–	–	–	–	○
___push2800	–	–	–	–	○
___push2801	–	–	–	–	○
___push2802	–	–	–	–	○
___push2803	–	–	–	–	○
___push2804	–	–	–	–	○
___push2840	–	–	–	–	○
___push2900	–	–	–	–	○
___push2901	–	–	–	–	○
___push2902	–	–	–	–	○
___push2903	–	–	–	–	○
___push2904	–	–	–	–	○
___push2940	–	–	–	–	○
___pushlp00	–	–	–	–	○
___pushlp01	–	–	–	–	○
___pushlp02	–	–	–	–	○
___pushlp03	–	–	–	–	○
___pushlp04	–	–	–	–	○
___pushlp40	–	–	–	–	○
___pop2000	–	–	–	–	○
___pop2001	–	–	–	–	○
___pop2002	–	–	–	–	○
___pop2003	–	–	–	–	○
___pop2004	–	–	–	–	○
___pop2040	–	–	–	–	○
___pop2100	–	–	–	–	○
___pop2101	–	–	–	–	○
___pop2102	–	–	–	–	○
___pop2103	–	–	–	–	○
___pop2104	–	–	–	–	○
___pop2140	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___pop2200	–	–	–	–	○
___pop2201	–	–	–	–	○
___pop2202	–	–	–	–	○
___pop2203	–	–	–	–	○
___pop2204	–	–	–	–	○
___pop2240	–	–	–	–	○
___pop2300	–	–	–	–	○
___pop2301	–	–	–	–	○
___pop2302	–	–	–	–	○
___pop2303	–	–	–	–	○
___pop2304	–	–	–	–	○
___pop2340	–	–	–	–	○
___pop2400	–	–	–	–	○
___pop2401	–	–	–	–	○
___pop2402	–	–	–	–	○
___pop2403	–	–	–	–	○
___pop2404	–	–	–	–	○
___pop2440	–	–	–	–	○
___pop2500	–	–	–	–	○
___pop2501	–	–	–	–	○
___pop2502	–	–	–	–	○
___pop2503	–	–	–	–	○
___pop2504	–	–	–	–	○
___pop2540	–	–	–	–	○
___pop2600	–	–	–	–	○
___pop2601	–	–	–	–	○
___pop2602	–	–	–	–	○
___pop2603	–	–	–	–	○
___pop2604	–	–	–	–	○
___pop2640	–	–	–	–	○
___pop2700	–	–	–	–	○
___pop2701	–	–	–	–	○
___pop2702	–	–	–	–	○
___pop2703	–	–	–	–	○
___pop2704	–	–	–	–	○
___pop2740	–	–	–	–	○
___pop2800	–	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
___pop2801	–	–	–	–	○
___pop2802	–	–	–	–	○
___pop2803	–	–	–	–	○
___pop2804	–	–	–	–	○
___pop2840	–	–	–	–	○
___pop2900	–	–	–	–	○
___pop2901	–	–	–	–	○
___pop2902	–	–	–	–	○
___pop2903	–	–	–	–	○
___pop2904	–	–	–	–	○
___pop2940	–	–	–	–	○
___poplp00	–	–	–	–	○
___poplp01	–	–	–	–	○
___poplp02	–	–	–	–	○
___poplp03	–	–	–	–	○
___poplp04	–	–	–	–	○
___poplp40	–	–	–	–	○

6. 1. 7 V850E2V3-FPU 使用ライブラリ

V850E2V3-FPU 使用ライブラリに収録されている関数を示します。

(1) V850E2V3-FPU 使用関数

表 6―16 V850E2V3-FPU 使用関数

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
expf	○	○	–	○	–注
exp	○	○	–	○	–注
logf	○	○	–	○	–注
log	○	○	–	○	–注
log10f	○	○	–	○	–注
log10	○	○	–	○	–注
powf	○	○	–	○	–注
pow	○	○	–	○	–注
sqrtf	○	○	–	○	–注
sqrt	○	○	–	○	–注
ceilf	○	–	–	–	○

関数／マクロ名	ANSI	const	sdata	sbss	Re-ent
ceil	○	–	–	–	○
floorf	○	–	–	–	○
floor	○	–	–	–	○
fmodf	○	○	–	○	–注
fmod	○	○	–	○	–注
frexpf	○	○	–	○	–注
frexp	○	○	–	○	–注
ldexpf	○	○	–	○	–注
ldexp	○	○	–	○	–注
modff	○	–	–	–	○
modf	○	–	–	–	○
cosf	○	○	–	○	–注
cos	○	○	–	○	–注
sinf	○	○	–	○	–注
sin	○	○	–	○	–注
tanf	○	○	–	○	–注
tan	○	○	–	○	–注
acosf	○	○	–	○	–注
acos	○	○	–	○	–注
asinf	○	○	–	○	–注
asin	○	○	–	○	–注
atanf	○	○	–	○	–注
atan	○	○	–	○	–注
atan2f	○	○	–	○	–注
atan2	○	○	–	○	–注
coshf	○	○	–	○	–注
cosh	○	○	–	○	–注
sinhf	○	○	–	○	–注
sinh	○	○	–	○	–注
tanhf	○	○	–	○	–注
tanh	○	○	–	○	–注

注 例外発生時の errno の更新と matherrf (matherr) ／ matherrd の呼び出しに関してのみリエントラント性を持ちません。

6.2 ヘッダ・ファイル

CX 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―17 ヘッダ・ファイル

ファイル名	概要
ctype.h	文字の変換，分類のためのヘッダ・ファイル
errno.h	エラー条件の報告のためのヘッダ・ファイル
float.h	浮動小数点表現，浮動小数点演算のためのヘッダ・ファイル
limits.h	整数の数量的限界のためのヘッダ・ファイル
math.h	数学計算のためのヘッダ・ファイル
setjmp.h	非局所分岐のためのヘッダ・ファイル
stdarg.h	可変個の引数を持つ関数をサポートするためのヘッダ・ファイル
stddef.h	共通の定義のためのヘッダ・ファイル
stdio.h	標準入出力のためのヘッダ・ファイル
stdlib.h	標準ユーティリティのためのヘッダ・ファイル
string.h	メモリ操作，文字列操作のためのヘッダ・ファイル

6.3 リエントラント性

“リエントラント”とは“再入可能”という意味です。リエントラント性を持つ関数は，その関数実行中に，他のプロセスでもその関数実行しようとした場合でも，正しく実行できます。たとえば，リアルタイム OS を用いたアプリケーションで，あるタスクがこの関数を実行している最中に，割り込みなどがトリガになって他のタスクへディスパッチし，そこでもこの関数を実行しても正しく実行されます。関数が RAM をテンポラリとして使う必要のある関数は，リエントラント性を持たないことがあります。

各関数のリエントラント性については，「表 6―2 可変個引数関数」から「表 6―15 関数前後処理ランタイム関数」を参照してください。

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　可変個引数関数

可変個引数関数として，以下のものがあります。

表 6―18　可変個引数関数

関数／マクロ名	概要
va_start	引数リスト走査用変数の初期化
va_end	引数リスト走査の終了
va_arg	引数リスト走査用変数の移動

va_start

引数リスト走査用変数の初期化を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdarg.h>

void va_start(va_list *ap*, *last-named-argument*);

［詳細説明］

変数 *ap* を，可変個引数リストの先頭（*last-named-argument* の次の引数）を指すように初期化します。

なお，可変個の引数を持つ関数 *func* を，移植性を持つ形で定義するには，次に示した形式を用います。

```
#include    <stdarg.h>
void func(arg-declarations, ...) {
    va_list ap;
    type    argN;
    va_start(ap, last-named-argument);
    argN = va_arg(ap, type);
    va_end(ap);
}
```

備考 *arg-declarations* は，引数リストで，最後に *last-named-argument* が宣言されているものとします。後ろに続く“, ...”は可変個引数リストを示します。va_list は，引数リストの走査に用いられる変数（この例の場合 *ap*）の型です。

［使用例］

```
#include    <stdarg.h>
void abc(int first, int second, ...) {
    va_list ap;
    int     i;
    char    c, *fmt;
    va_start(ap, second);
    i = va_arg(ap, int);
    c = va_arg(ap, int);    /*char 型は int 型に変換 */
    fmt = va_arg(ap, char *);
    va_end(ap);
}
```

va_end

引数リスト走査の終了を示します。

［所属］

標準ライブラリ

［指定形式］

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

［詳細説明］

リストの走査の終了を示します。va_arg を va_start と本関数とで囲むことにより，リストの走査を繰り返すことができます。

va_arg

引数リスト走査用変数の移動を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdarg.h>

type va_arg(va_list *ap*, *type*);

［詳細説明］

変数 *ap* の指している引数を返し，次の引数を指すように変数 *ap* を進めます。*type* には，引数が関数に渡される際に変換される型を指定します。コンパイラでは，singed char 型，および short 型の引数に対しては int 型を指定し，unsinged char 型，および unsigned short 型の引数に対しては unsigned int 型を指定してください。引数ごとに異なる型を指定することができますが，“どの型の引数が渡されてきているか”は，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。

また，“実際に引数がいくつ渡されてきているか”に関しても，呼び出された側と呼び出し側の関数との間の取り決めによって規定されるようにしてください。

6. 4. 2 文字列関数

文字列関数として，以下のものがあります。

表 6―19 文字列関数

関数／マクロ名	概要
index	文字列検索（最初の位置）
strpbrk	文字列検索（最初の位置）
rindex	文字列検索（最後の位置）
strrchr	文字列検索（最後の位置）
strchr	文字列検索（指定文字の最初の位置）
strstr	文字列検索（指定文字列の最初の位置）
strspn	文字列検索（指定文字を含む最大の長さ）
strcspn	文字列検索（指定文字を含まない最大の長さ）
strcmp	文字列比較
strncmp	文字列比較（文字数指定）
strcpy	文字列コピー
strncpy	文字列コピー（文字数指定）
strcat	文字列連結
strncat	文字列連結（文字数指定）
strtok	トークン分割
strlen	文字列の長さ
strerror	エラー番号の文字列変換

index

文字列検索（最初の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

見つかった文字を指すポインタを返します。*c* がこの文字列中に現れなかった場合は，null ポインタを返します。

［詳細説明］

char 型に変換された *c* と同じ文字が，*s* の指す文字列中に最初に現れる位置を求めます。終端を示す null 文字（\ 0）は，この文字列の一部であるとみなされます。

strpbrk

文字列検索（最初の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

```
#include <string.h>
char    *strpbrk(const char *s1, const char *s2);
```

［戻り値］

この文字を指すポインタを返します。*s2* からの文字がいずれも *s1* の中に現れなかつた場合は，null ポインタを返します。

［詳細説明］

s2 の指す文字列中のいずれかの文字（null 文字（\ 0）は除く）が *s1* の指す文字列中に最初に現れた位置を求めます。

rindex

文字列検索（最後の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

見つかった *c* を指すポインタを返します。*c* がこの文字列中に現れなかった場合，null ポインタを返します。

［詳細説明］

char 型に変換された *c* が *s* の指す文字列中に最後に現れた位置を求めます。終端を示す null 文字（\ 0）は，この文字列の一部であるとみなされます。

strrchr

文字列検索（最後の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

見つかった *c* を指すポインタを返します。*c* がこの文字列中に現れなかった場合，null ポインタを返します。

［詳細説明］

char 型に変換された *c* が *s* の指す文字列中に最後に現れた位置を求めます。終端を示す null 文字（\ 0）は，この文字列の一部であるとみなされます。

strchr

文字列検索（指定文字の最初の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

見つかった文字を指すポインタを返します。*c* がこの文字列中に現れなかった場合は，null ポインタを返します。

［詳細説明］

char 型に変換された *c* と同じ文字が，*s* の指す文字列中に最初に現れる位置を求めます。終端を示す null 文字（\ 0）は，この文字列の一部であるとみなされます。

strstr

文字列検索（指定文字列の最初の位置）を行います。

［所属］

標準ライブラリ

［指定形式］

```
#include <string.h>
char    *strstr(const char *s1, const char *s2);
```

［戻り値］

見つかった文字列を指すポインタを返します。文字列 *s2* が見つからなかった場合，null ポインタを返します。*s2* が長さゼロの文字列を指している場合，*s1* を返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列と最初に一致する部分（null 文字（\ 0）は除く）の位置を求めます。

strspn

文字列検索（指定文字を含む最大の長さ）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

先頭部分の長さを返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列中にある文字（null 文字（\ 0）は除く）のみで構成されている先頭部分の長さを求めます。

strcspn

文字列検索（指定文字を含まない最大の長さ）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

先頭部分の長さを返します。

［詳細説明］

s1 の指す文字列の中で，*s2* の指す文字列（終わりの null 文字（\ 0）は除く）の中にない文字のみで構成されている，先頭部分の長さを求めます。

strcmp

文字列比較を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

int strcmp(const char *s1, const char *s2);

［戻り値］

s1 の指す文字列が *s2* の指す文字列と比べて大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。

［詳細説明］

s1 の指す文字列と *s2* の指す文字列とを比較します。

strncmp

文字列比較（文字数指定）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

int strncmp(const char *s1, const char *s2, size_t *length*);

［戻り値］

s1 の指す配列が *s2* の指す配列より大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，0 より小さい整数を返します。

［詳細説明］

s1 の指す配列の文字と *s2* の指す配列の文字を最大で *length* 文字比較します。

strcpy

文字列コピーを行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

char *strcpy(char **dst*, const char **src*);

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列を *dst* の指す配列にコピーします。

［使用例］

```
#include    <string.h>
void func(char *str, const char *src) {
    strcpy(str, src);   /*src の指す文字列を str の指す配列にコピー */
      :
}
```

strncpy

文字列コピー（文字数指定）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

char *strncpy(char **dst*, const char **src*, size_t *length*);

［戻り値］

dst の値を返します。

［詳細説明］

src の指す配列から *dst* の指す配列に最大で *length* 文字（null 文字（\ 0）を含む）コピーします。*src* の指す配列が *length* 文字より短い文字列の場合，全部で *length* 文字分書き込まれるまで，*dst* の指す配列内のコピーに null 文字（\ 0）が付加されます。

strcat

文字列連結を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列のコピーを，null 文字（\ 0）を含めて，*dst* の指す文字列の末尾に連結します。*src* の最初の文字は *dst* の終わりの null 文字（\ 0）を上書きします。

strncat

文字列連結（文字数指定）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

char *strncat(char *dst, const char **src*, size_t *length*);

［戻り値］

dst の値を返します。

［詳細説明］

src の指す文字列の先頭から，最大で *length* 文字（*src* の null 文字（\ 0）を含む）を *dst* の指す文字列の末尾に連結します。*src* の最初の文字は *dst* の終わりの null 文字（\ 0）を上書きします。この結果には，終端を示す null 文字（\ 0）が常に付加されます。

［注意事項］

null 文字（\ 0）は常に付加されるので，コピーが *length* によって制限される場合，*dst* に付加される文字の個数は *length* + 1 になることに注意してください。

strtok

トークン分割を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

トークンへのポインタを返します。トークンが存在しない場合は，null ポインタを返します。

［詳細説明］

s の指す文字列を，*delimiters* の指す文字列中の文字で区切ることによって，トークンの列に分割します。これは最初に呼び出されると，最初の引数として *s* を持ち，その後は null ポインタを最初の引数とする呼び出しが続きます。*delimiters* の指す区切り文字列は，呼び出しごとに異なっていてもかまいません。最初の呼び出しでは，*delimiters* の指す区切り文字列中に含まれない最初の文字を求めて *s* の指す文字列中をサーチします。そのような文字が見つからなかった場合，null ポインタを返します。そのような文字が見つかった場合，その文字が最初のトークンの始まりとなります。その後，そのときの区切り文字列に含まれる文字を求めてそこからサーチを行います。

そのような文字が見つからなかった場合，そのときのトークンは *s* の指す文字列の終わりまで拡張され，あとに続くサーチは null ポインタを返します。そのような文字が見つかった場合，その文字はトークンの終端を示す null 文字（\ 0）で上書きされます。本関数は，あとに続く文字を指すポインタをセーブします。null ポインタを最初の引数の値としている場合，リエントラントでないコードになります。これは，最後の区切り文字のアドレスをアプリケーション・プログラムの中で保存しておき，これを使って，*s* を空でない引数として渡すようにすることで回避できます。

strlen

文字列の長さを求めます。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

終端を示す null 文字（\ 0）の前に存在する文字の数を返します。

［詳細説明］

s の指す文字列の長さを求めます。

strerror

エラー番号の文字列変換を行います。

[所属]

標準ライブラリ

[指定形式]

```
#include <string.h>
char    *strerror(int errnum);
```

[戻り値]

変換した文字列へのポインタを返します。

[詳細説明]

処理系定義の対応関係に従って，エラー番号 *errnum* を文字列に変換します。*errnum* の値は，通常，グローバル変数 errno がコピーされたものです。指されている配列は，アプリケーション・プログラム側で変更しないでください。

6. 4. 3 メモリ管理関数

メモリ管理関数として，以下のものがあります。

表 6―20 メモリ管理関数

関数／マクロ名	概要
memchr	メモリ検索
memcmp	メモリ比較
bcmp	メモリ比較（memcmp の char 引数版）
memcpy	メモリ・コピー
bcopy	メモリ・コピー（memcpy の char 引数版）
memmove	メモリ移動
memset	メモリ・セット

memchr

メモリ検索を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

void *memchr(const void *s, int c, size_t length);

［戻り値］

c が見つかった場合はこの文字を指すポインタを返し，*c* が見つからなかった場合は null ポインタを返します。

［詳細説明］

s の指す領域の最初の *length* 個の文字の中に（char 型に変換された）文字 *c* が最初に現れた位置を求めます。

memcmp

メモリ比較を行います。

[所属]

標準ライブラリ

[指定形式]

#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);

[戻り値]

s1 の指すオブジェクトが *s2* の指すオブジェクトより大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，または 0 より小さい整数を返します。

[詳細説明]

s1 の指すオブジェクトの最初の *n* 文字を *s2* の指すオブジェクトと比較します。

[使用例]

```
#include    <string.h>
int func(const void *s1, const void *s2) {
    int i;
    i = memcmp(s1, s2, 5);  /*s1 の指す文字列の最初の 5 文字を s2 の指す文字列の最初の 5 文字と比較 */
    return(i);
}
```

bcmp

メモリ比較（memcmp の char 引数版）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

int bcmp(const char *s1, const char *s2, size_t n);

［戻り値］

s1 の指すオブジェクトが *s2* の指すオブジェクトより大きい，等しい，または小さいかによって，0 より大きい，0 に等しい，または 0 より小さい整数を返します。

［詳細説明］

s1 の指すオブジェクトの最初の *n* 文字を *s2* の指すオブジェクトと比較します。

memcpy

メモリ・コピーを行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

void　*memcpy(void **out*, const void **in*, size_t *n*);

［戻り値］

out の値を返します。コピー元とコピー先の領域が重なっている場合，その動作は不定です。

［詳細説明］

n バイト分を *in* の指すオブジェクトから *out* の指すオブジェクトへコピーします。

bcopy

メモリ・コピー（memcpy の char 引数版）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

char*　bcopy(const char *_in_, char *_out_, size_t _n_);

［戻り値］

out の値を返します。コピー元とコピー先の領域が重なっている場合，その動作は不定です。

［詳細説明］

_n_ バイト分を _in_ の指すオブジェクトから _out_ の指すオブジェクトへコピーします。

memmove

メモリ移動を行います。

［所属］

標準ライブラリ

［指定形式］

#include <string.h>

void *memmove(void **dst*, void **src*, size_t *length*);

［戻り値］

コピー先の *dst* の値を返します。

［詳細説明］

length 個の文字を，*src* の指すメモリ領域から *dst* の指すメモリ領域へ移動します。コピー元とコピー先の 2 つの領域が重なり合っていても，文字を *dst* の指すメモリ領域に正しくコピーします。

memset

メモリ・セットを行います。

[所属]

標準ライブラリ

[指定形式]

#include <string.h>

void *memset(const void *s, int *c*, size_t *length*);

[戻り値]

s の値を返します。

[詳細説明]

s の指すオブジェクトの最初の *length* 文字に unsigned char 型に変換した *c* の値をコピーします。

6. 4. 4 文字変換関数

文字変換関数として，以下のものがあります。

表 6―21 文字変換関数

関数／マクロ名	概要
toupper	英小文字から英大文字変換（引数が英大文字以外のときそのまま）
_toupper	英小文字から英大文字変換（引数が英小文字のときだけ正しく変換）
tolower	英大文字から英小文字変換（引数が英大文字以外のときそのまま）
_tolower	英大文字から英小文字変換（引数が英大文字のときだけ正しく変換）
toascii	整数から ASCII 文字変換

toupper

英小文字から英大文字変換（引数が英大文字以外のときそのまま）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int toupper(int *c*);

［戻り値］

c に対して islower が真となる場合，それに対応して isupper が真となる文字を返します。そうでない場合，*c* を返します。

［詳細説明］

小文字の英字を対応する大文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c が EOF ～ 255 の範囲の整数の場合にのみ定義されています。“#undef toupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

［使用例］

```
#include    <ctype.h>
int chc = 'a';
int ret = func(chc);
int func(int c) {
    int i;
    i = toupper(c); /*c の英小文字 'a' を英大文字 'A' に変換 */
    return(i);
}
```

_toupper

英小文字から英大文字変換（引数が英小文字のときだけ正しく変換）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int _toupper(int *c*);

［戻り値］

c に対して islower が真となる場合，それに対応して isupper が真となる文字を返します。そうでない場合，*c* を返します。

なお，*c* に不正な値が指定された場合，その動作は不定です。

［詳細説明］

引数が小文字の英字の場合に toupper と同じ動作をするマクロです。

引数のチェックは行わないため，引数が小文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“#undef _toupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

tolower

英大文字から英小文字変換（引数が英大文字以外のときそのまま）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int tolower(int *c*);

［戻り値］

c に対して isupper が真となる場合，それに対応して islower が真となる文字を返します。そうでない場合，*c* を返します。

［詳細説明］

大文字の英字を対応する小文字の英字に変換し，他の文字はすべてそのままにするマクロです。

c が EOF ～ 255 の範囲の整数の場合にのみ定義されています。“#undef tolower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

_tolower

英大文字から英小文字変換（引数が英大文字のときだけ正しく変換）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int _tolower(int *c*);

［戻り値］

c に対して isupper が真となる場合，それに対応して islower が真となる文字を返します。そうでない場合，*c* を返します。

なお，*c* に不正な値が指定された場合，その動作は不定です。

［詳細説明］

引数が大文字の英字の場合に tolower と同じ動作をするマクロです。

引数のチェックは行わないため，引数が大文字の英字である場合にのみ正しい変換を行い，それ以外のものである場合，その動作は不定となります。“#undef _tolower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

toascii

整数から ASCII 文字変換を行います。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int toascii(int *c*);

［戻り値］

0 から 127 までの範囲の整数を返します。

［詳細説明］

引数の 8 ビット目以上のビットを 0 にすることで，整数を ASCII 文字（0 ～ 127）に強制変換するマクロです。

“#undef toascii”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンを使用できます。

6. 4. 5 文字分類関数

文字分類関数として，以下のものがあります。

表 6―22 文字分類関数

関数／マクロ名	概要
isalnum	ASCII 英字，または数字であるかを判定
isalpha	ASCII 英字であるかを判定
isascii	ASCII コードであるかを判定
isupper	英大文字であるかを判定
islower	英小文字であるかを判定
isdigit	10 進数であるかを判定
isxdigit	16 進数であるかを判定
iscntrl	制御文字であるかを判定
ispunct	区切り文字であるかを判定
isspace	スペース／タブ／復帰／改行／垂直タブ／改ページであるかを判定
isprint	表示文字であるかを判定
isgraph	スペース以外の表示文字であるかを判定

isalnum

ASCII 英字，または数字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isalnum(int *c*);

［戻り値］

引数 *c* の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII 英字，または数字であるかどうか調べるマクロです。c が isascii で真になるか，または c が EOF の場合にのみ，定義されています。“#undef isalnum”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isalpha

ASCII 英字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isalpha(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII 英字であるかどうか調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isalpha”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isascii

ASCII コードであるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isascii(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

ASCII コード（0x00 ～ 0x7F）であるかどうかを調べるマクロです。すべての整数に対して定義されています。“#undef isascii”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isupper

英大文字であるかを判定します。

[所属]

標準ライブラリ

[指定形式]

#include <ctype.h>

int isupper(int *c*);

[戻り値]

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

[詳細説明]

大文字の英字（A ～ Z）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isupper”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

islower

英小文字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int islower(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

小文字の英字（a ～ z）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef islower”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isdigit

10 進数であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isdigit(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

10 進数の数字であるかどうか調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isdigit”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isxdigit

16 進数であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

16 進数の数字（0 ～ 9，a ～ f，または A ～ F）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isxdigit”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

iscntrl

制御文字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int iscntrl(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

制御文字（0x00 ～ 0x1F，または 0x7F）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef iscntrl”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

ispunct

区切り文字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int ispunct(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

印字可能な区切り文字“isgraph (c) && ! isalnum (c)”であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef ispunct”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isspace

スペース／タブ／復帰／改行／垂直タブ／改ページであるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isspace(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

スペース，タブ，復帰，改行，垂直タブ，改ページ（0x09 ～ 0x0D，または 0x20）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isspace”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

isprint

表示文字であるかを判定します。

[所属]

標準ライブラリ

[指定形式]

#include <ctype.h>

int isprint(int *c*);

[戻り値]

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

[詳細説明]

表示文字（0x20 ～ 0x7E）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isprint”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

[使用例]

```
#include    <ctype.h>
void func(void) {
    int     i,j = 0;
    char    s[50];
        for(i =50; i <= 99; i++) {
            if(isprint(i)) {        /* コード 50 ～ 99 内の表示可能文字を配列 s に格納する */
                    s[j] = i;
                    j++;
            }
        }
    :
}
```

isgraph

スペース以外の表示文字であるかを判定します。

［所属］

標準ライブラリ

［指定形式］

#include <ctype.h>

int isgraph(int *c*);

［戻り値］

c の値がそれぞれの記述に合致した場合（真）に 0 以外を返します。結果が偽であった場合は 0 を返します。

［詳細説明］

スペース（0x20）以外の表示文字**注**（0x20 ～ 0x7E）であるかどうかを調べるマクロです。*c* が isascii で真になるか，または *c* が EOF の場合にのみ，定義されています。“#undef isgraph”を使ってマクロ定義を無効にし，マクロ定義の代わりにコンパイル済みサブルーチンが使用できます。

注 printing character のことです。

6. 4. 6 標準入出力関数

標準入出力関数として，以下のものがあります。

表 6―23 標準入出力関数

関数／マクロ名	概要
fread	ストリームからの読み込み
getc	ストリームからの文字読み込み（fgetc と同じ）
fgetc	ストリームからの文字読み込み（getc と同じ）
fgets	ストリームからの一行読み込み
fwrite	ストリームへの書き込み
putc	ストリームへの文字書き込み
fputc	ストリームへの文字書き込み
fputs	ストリームへの文字列出力
getchar	標準入力からの一文字読み込み
gets	標準入力からの文字列読み込み
putchar	標準出力ストリームへの文字書き込み
puts	標準出力ストリームへの文字列出力
sprintf	書式付き出力
fprintf	フォーマット指定したテキストをストリームへ出力
vsprintf	フォーマット指定したテキストを文字列へ書き込み
printf	フォーマット指定したテキストを標準出力ストリームへ出力
vfprintf	フォーマット指定したテキストをストリームへ書き込み
vprintf	フォーマット指定したテキストを標準出力ストリームへ書き込み
sscanf	書式付き入力
fscanf	ストリームからのデータ読み込みと解釈
scanf	標準入力ストリームからのテキストの読み込みと解釈
ungetc	入力ストリームへの文字押し戻し
rewind	ファイル位置指示子のリセット
perror	エラー処理

fread

ストリームからの読み込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

size_t fread(void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

[戻り値]

入力した要素数 _nmemb_ を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す入力ストリームから，_size_ の大きさの要素を _nmemb_ 個入力し，_ptr_ へ格納します。_stream_ に指定できるのは，標準入出力の stdin だけです。

[使用例]

```
#include    <stdio.h>
void func(void) {
    struct {
        int     c;
        double  d;
    } buf[10];
    fread(buf, sizeof(buf[0]), sizeof(buf) / sizeof(buf [0]), stdin);
}
```

getc

ストリームからの文字読み込みを行います。（fgetc と同じ）

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

［所属］

標準ライブラリ

［指定形式］

#include <stdio.h>

int getc(FILE *_stream_);

［戻り値］

入力文字を返します。

エラー・リターンはありません。

［詳細説明］

stream が指す入力ストリームから，1 文字を入力します。*stream* に指定できるのは，標準入出力の stdin だけです。

fgetc

ストリームからの文字読み込みを行います。（getc と同じ）

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

［所属］

標準ライブラリ

［指定形式］

#include <stdio.h>

int fgetc(FILE *_stream_);

［戻り値］

入力文字を返します。

エラー・リターンはありません。

［詳細説明］

stream が指す入力ストリームから，1 文字を入力します。_stream_ に指定できるのは，標準入出力の stdin だけです。

［使用例］

```
#include    <stdio.h>

int func(void) {
    int    c;
    c = fgetc(stdin);
    return(c);
}
```

fgets

ストリームからの一行読み込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

s を返します。
エラー・リターンはありません。

[詳細説明]

stream が指す入力ストリームから，最大 *n* - 1 文字を入力し，*s* へ格納します。文字の入力は，改行文字の検出によっても終了します。この場合，改行文字も *s* へ格納されます。最後に文字列の終結 null 文字が *s* へ格納されます。*stream* に指定できるのは，標準入出力の stdin だけです。

fwrite

ストリームへの書き込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

size_t fwrite(const void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

[戻り値]

出力した要素数 _nmemb_ を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，_ptr_ が指す配列から，_size_ の大きさの要素を _nmemb_ 個出力します。_stream_ に指定できるのは，標準入出力の stdout と stderr だけです。

putc

ストリームへの文字書き込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int putc(int *c*, FILE **stream*);

[戻り値]

文字 *c* を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，文字 *c* を出力します。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

fputc

ストリームへの文字書き込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int fputc(int *c*, FILE **stream*);

[戻り値]

文字 *c* を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，文字 *c* を出力します。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

[使用例]

```
#include    <stdio.h>
void func(void) {
    fputc('a', stdout);
}
```

fputs

ストリームへの文字列出力を行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int fputs(const char *s, FILE *stream);

[戻り値]

0 を返します。

エラー・リターンはありません。

[詳細説明]

stream が指す出力ストリームへ，文字列 *s* を出力します。文字列の終端 null 文字は出力しません。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。

getchar

標準入力からの一文字読み込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

入力文字を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdin から，1 文字を入力します。

gets

標準入力からの文字列読み込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

s を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdin から，改行文字を検出するまで文字を入力し，*s* へ格納します。入力した改行文字は捨て，最後に，文字列の終結 null 文字が *s* へ格納されます。

putchar

標準出力ストリームへの文字書き込みを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int putchar(int *c*);

[戻り値]

文字 *c* を返します。

エラー・リターンはありません。

[詳細説明]

標準入出力の stdout へ，文字 *c* を出力します。

puts

標準出力ストリームへの文字列出力を行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

0 を返します。
エラー・リターンはありません。

[詳細説明]

標準入出力の stdout へ，文字列 *s* を出力します。文字列の終端 null 文字は出力せず，代わりに改行文字を出力します。

sprintf

書式付き出力を行います。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int sprintf(char *s, const char *format[, arg, ...]);

[戻り値]

出力された文字（null 文字（\ 0）は除きます）の数を返します。

エラー・リターンはありません。

[詳細説明]

それぞれの *arg* に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを *s* の指す配列に書き出します。

書式に対して引数が十分にない場合，動作は不定です。書式文字列の終わりに到達するとリターンします。書式で必要としている以上に引数がある場合，余分の引数を無視します。また，*s* の領域が引数の 1 つと重なっていると動作は不定になります。

format は，“後ろに続く引数がどのような出力に変換されるか”を指定しています。書き込まれた文字の最後には null 文字（\ 0）が付加されます（null 文字（\ 0）は返り値におけるカウントの対象とはなりません）。

format は，次に示す 2 種類のディレクティブにより構成されます。

通常文字	変換されずにそのまま出力にコピーされるものです（“%”以外）。
変換指示	0 個以上の引数を取り込み，指示を与えるものです。

各変換指示は，文字“%”で始まります（出力中に“%”を入れたい場合は，書式文字列の中では“%%”とします）。“%”の後ろは，次のようになります。

%[フラグ][フィールド長][精度][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) フラグ

任意の順に置かれた，変換指示の意味を修飾する 0 個以上のフラグです。フラグ文字とその意味を次に示します。

-	変換された結果をフィールド中に左詰めにし，右側は空白で満たされます（このフラグが指定されない場合，変換された結果は右詰めにされます）。
+	符号付きの変換の結果を常に + 符号，または - 符号で始めます（このフラグが指定されない場合，変換された結果は，負の値が変換された場合にのみ符号で始められます）。
スペース	符号付きの変換の最初の文字が符号でない場合，または符号付きの変換が文字を生じない場合，その結果の前にスペース（“ ”）を付けます。スペース・フラグと + フラグとが両方現れる場合，スペース・フラグは無視されます。
#	結果を“別の形式[注 1]”に変換します。o 変換に対しては，その変換結果の最初の数字が 0 になるようにその精度を増やします。x，または X 変換に対しては，0 以外の変換結果の先頭に 0x，または 0X を付加します。e，f，g，E，G 変換に対しては，その変換結果に小数点以下の数字が存在しない場合であっても，小数点“.”を付加します[注 2]。g，G 変換に対しては，変換結果から後ろに続く 0 が削除されないようにします。これら以外の変換に対しては，その動作は不定となります。
0	d，e，f，g，i，o，u，x，E，G，X 変換に対し，フィールド長を埋めるために，符号，または基底の指示に続いて 0 を付加します。 0 フラグと - フラグの両方が指定された場合，0 フラグは無視されます。d，i，o，u，x，X 変換については，精度を指定している場合，ゼロ（0）フラグを無視します。 0 はフラグとして解釈され，フィールド幅の始まりとは解釈されないことに注意してください。 これら以外の変換に対してはその動作は不定となります。

注 1. alternate format のことです。

2. 通常，小数点は，その後ろに数字が続く場合にのみ現れます。

(2) フィールド長

オプショナルな最小フィールド長です。変換された値がこのフィールド長より小さい場合，左側にスペースが詰められます（前述の左詰めフラグが与えられた場合は右側にスペースが詰められます）。このフィールド長は“*”，または 10 進整数の形を取ります。“*”で指定した場合，int 型の引数をフィールド長として使用します。負のフィールド長は，サポートしていません。負のフィールド長を指定しようとすると，正のフィールド長の前にマイナス（-）フラグが付いたものと解釈されます。

(3) 精度[注]

これに与えられる値は，d，i，o，u，x，X 変換に対しては現れる数字の個数の最小値であり，e，f，E 変換に対しては“.”の後ろに現れる数字の個数であり，g，G 変換に対しては最大有効桁数です。精度は，“*”，または 10 進整数が後ろに続く“.”の形式を取ります。“*”を指定した場合，int 型の引数を精度として使用します。負の精度を指定した場合，精度を省略したものとみなされます。“.”のみが指定された場合，精度は 0 とされます。精度がこれら以外の変換指示とともに現れた場合，動作は不定となります。

注 precision のことです。

(4) サイズ

対応する引数のデータ型を解釈するためのデフォルトの方法を変更する，任意選択のサイズ文字 h，l，ll，および L です。

h を指定した場合，後ろに続く d，i，o，n，u，x，X の型指定を強制的に short，または unsigned short に適用します。

l を指定した場合，後ろに続く d，i，o，u，x，X の型指定を強制的に long，または unsigned long に適用します。l はさらに，後ろに続く n の型指定を強制的に long へのポインタに適用します。h，または l といっしょにこれと別の型指定文字を使用した場合，その動作は不定です。

ll を指定した場合，後ろに続く d，i，o，u，x，X の型指定を強制的に long long，または unsigned long long に適用します。ll はさらに，後ろに続く n の型指定を強制的に long long へのポインタに適用します。ll と一緒にこれ以外の型指定文字を使用した場合，その動作は不定です。

L を指定した場合，後ろに続く e，E，f，g，G の型指定を強制的に long double に適用します。L といっしょにこれ以外の型指定文字を使用した場合，その動作は不定です。

(5) 型指定文字

適用される変換の型を指定する文字です。

変換の型を指定する文字とその意味を次に示します。

%	文字“%”を出力します。引数は変換されません。変換指示は“%%”となります。
c	int 型の引数を unsigned char 型に変換し，変換結果の文字を出力します。
d	int 型の引数を符号付きの 10 進数に変換します。
e，E	double 型の引数を，小数点の前に（引数が 0 でない場合 0 でない）1 つの文字を持ち，小数点以下の数字の個数は精度に等しい [-]d.dddde ± dd の形式に変換します。E 変換指示は，指数部が“e”ではなく“E”で始まる数字を生成します。
f	double 型の引数を [-]dddd.dddd の形式の 10 進表記に変換します。
g，G	精度には仮数部の数字の個数を指定するものとし，double 型の引数を e（G 変換指示の場合 E），または f の形式に変換します。変換結果の末尾の 0 は結果の小数点部から除かれます。小数点は，後ろに数字が続く場合にのみ現れます。
i	d の変換と同じ変換をします。
n	同じオブジェクト内で出力された文字の個数を格納します。int 型へのポインタを引数とします。
p	処理系定義書式でポインタを出力します。CX では，ポインタを unsigned long として扱っています（lu の指定と同じです）。
o，u，x，X	unsigned int 型の引数を dddd の形式の 8 進表記（o），符号なしの 10 進表記（u），符号なしの 16 進表記（x，または X）に変換します。x 変換に対しては文字 abcdef が用いられ X 変換に対しては文字 ABCDEF が用いられます。
s	引数は文字型の配列を指すポインタでなければなりません。この配列からの文字を，終端を示す null 文字（\0）の前まで（null 文字（\0）自身は含まずに）出力します。精度が指定された場合，それ以上の個数の文字は出力されません。精度が指定されなかった，または精度がこの配列の大きさ以上の値であった場合，この配列は null 文字（\0）を含むようにしてください。

［使用例］

```
#include    <stdio.h>
void func(int val) {
    char    s[20];
    sprintf(s, "%-10.51x\n", val);  /*val の値に対し，左詰め，フィールド長 10，精度 5，サイズ long，
                                        16 進表記を指定し，改行文字を付加して s の指す配列へ出力 */
}
```

fprintf

フォーマット指定したテキストをストリームへ出力します。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int fprintf(FILE *_stream_, const char *_format_[, _arg_, ...]);

[戻り値]

出力された文字数を返します。

[詳細説明]

それぞれの *arg* に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを *stream* に出力します。*stream* に指定できるのは，標準入出力の stdout と stderr だけです。*format* の記述方法は sprintf と同様です。sprintf と違って，最後に null 文字（\ 0）は出力されません。

[注意事項]

stream に stdout（標準出力），stderr（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct {
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
```

```
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

［使用例］

```
#include    <stdio.h>
void func(int val) {
    fprintf(stdout, "%-10.5x\n", val);
}
/* 汎用のエラー報告ルーチンにおける使用例 */
void error(char *function_name, char *format, ...) {
    va_list arg;
    va_start(arg, format);
    fprintf(stderr, "ERROR in %s:", function_name); /* エラーが発生した関数名を出力 */
    vfprintf(stderr, format, arg);                  /* 残りのメッセージを出力 */
    va_end(arg);
}
```

vsprintf

フォーマット指定したテキストを文字列へ書き込みます。

［所属］

標準ライブラリ

［指定形式］

#include <stdio.h>

int vsprintf(char *s, const char *f*ormat*, va_list *arg*);

［戻り値］

出力された文字（null 文字（\ 0）は除きます）の数を返します。

エラー・リターンはありません。

［詳細説明］

arg の指す引数列に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを *s* が指す配列に出力します。本関数は，可変個数実引数並びを *arg* で置き換えた sprintf と等価です。本関数の呼び出しの前に，va_start で *arg* を初期化しておく必要があります。

printf

フォーマット指定したテキストを標準出力ストリームへ出力します。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int printf(const char *format[, arg, ...]);

[戻り値]

出力された文字数を返します。

[詳細説明]

それぞれの *arg* に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。*format* の記述方法は sprintf と同様です。sprintf と異なり，最後に null 文字（\ 0）は出力されません。

[注意事項]

stream に stdout（標準出力），stderr（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct {
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
```

```
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

vfprintf

フォーマット指定したテキストをストリームへ書き込みます。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int vfprintf(FILE *_stream_, const char *_format_, va_list _arg_);

[戻り値]

出力された文字数を返します。

[詳細説明]

arg の指す引数列に _format_ の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを _stream_ に出力します。_stream_ に指定できるのは，標準入出力の stdout と stderr だけです。_format_ の記述方法は sprintf と同様です。本関数は，可変個数実引数並びを _arg_ で置き換えた fprintf と等価です。本関数の呼び出しの前に，va_start で _arg_ を初期化しておく必要があります。

[注意事項]

stream に stdin（標準入力），stdout（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct {
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
```

```
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

［使用例］

```
#include    <stdio.h>
void func(int val) {
    fprintf(stdout, "%-10.5x\n", val);
}
/* 汎用のエラー報告ルーチンにおける使用例 */
void error(char *function_name, char *format, ...) {
    va_list arg;
    va_start(arg, format);
    fprintf(stderr, "ERROR in %s:", function_name); /* エラーが発生した関数名を出力 */
    vfprintf(stderr, format, arg);                  /* 残りのメッセージを出力 */
    va_end(arg);
}
```

vprintf

フォーマット指定したテキストを標準出力ストリームへ書き込みます。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int vprintf(const char **format*, va_list *arg*);

[戻り値]

出力された文字数を返します。

[詳細説明]

arg の指す引数列に *format* の指す文字列で指定された書式を適用し，それにより出力された書式付きデータを標準入出力の stdout に出力します。*format* の記述方法は sprintf と同様です。本関数は，可変個数実引数並びを *arg* で置き換えた printf と等価です。本関数の呼び出しの前に，va_start で *arg* を初期化しておく必要があります。

[注意事項]

stream に stdout（標準出力），stderr（標準エラー）を指定します。ストリームの入出力先は I/O アドレスなど 1 メモリ・アドレスを割り当てます。デバッガとの連携でこれらのストリームを使用するには，stdio.h ファイルで定義されている，ストリーム構造体の初期値設定が必要です。関数を呼び出す前に，初期値設定を行ってください。

【stdio.h におけるストリーム構造体の定義】

```
typedef struct {
    int         mode;   /*with error descriptions*/
    unsigend    handle;
    int         unget_c;
} FILE;
typedef int     fpos_t;
#pragma section sdata
extern  FILE    __struct_stdin;
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
```

```
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

構造体の第一メンバ mode は，入出力状態を示します。ACCSD_OUT/ACCSD_IN として内部定義されています。第三メンバ unget_c は，押し戻し文字（stdin のみ）を示し，-1 として内部定義されています。

-1 の場合，押し戻し文字“なし”を表します。第二メンバ handle は，入出力 I/O アドレスを示します。handle には，使用するデバッガで決められている値を設定してください。

【入出力 I/O アドレス設定例】

```
__struct_stdout.handle = 0xFFFFF000;
__struct_stderr.handle = 0x00FFF000;
__struct_stdin.handle = 0xFFFFF002;
#pragma section sdata
extern  FILE    __struct_stdout;
extern  FILE    __struct_stderr;
#pragma section default
#define stdin(&__struct_stdin)
#define stdout(&__struct_stdout)
#define stderr(&__struct_stderr)
```

sscanf

書式付き入力を行います。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int sscanf(const char *s, const char *format[, arg, ...]);

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトを指すポインタとして扱い，*s* の指す配列から変換する入力を読み込みます。

format には，認識されうる入力列，および“代入のためにどのように変換を行うか”ということを指定します。*format* に対し十分な引数が存在しない場合，その動作は不定となります。引数が残っているのに *format* が使い果たされた場合，残された引数は無視されます。

format は，次に示す 3 種類のディレクティブにより構成されます。

1 個以上の空白類	スペース（ ），タブ（\t），改行（\n）です。 本関数を実行して，文字列内に空白文字が見つかった場合，次の空白でない文字まで連続するすべての空白文字を読み込みます（格納はしません）。
通常の文字	“%”以外のすべての ASCII 文字です。 本関数を実行して，文字列内に通常の文字が見つかった場合，それを読み込みますが，格納はしません。変換指示により，本関数は，入力フィールドから文字列を読み込み，特定の型の値に変換し，引数で指定した位置に格納します。変換指示で明示されて一致しているのでなければ，後ろに続く空白は読み込まれません。
変換指示	0 個以上の引数を取り込み，変換の指示を与えます。

各変換指示は“%”で始まります。“%”の後ろは，次のようになります。

%[代入抑制文字][フィールド長][サイズ][型指定文字]

それぞれの変換指示について，次に説明します。

(1) 代入抑制文字

入力フィールドの解釈，および代入を抑制する“*”です。

(2) フィールド長

最大フィールド長を規定する 0 以外の 10 進整数です。入力フィールドを変換する前に読み込まれる最大文字数を指定します。入力フィールドがこのフィールド長より小さい場合，本関数はフィールド内のすべての文字を読み込み，次のフィールドとその変換指示へ進みます。また，フィールド長分を読み込む前に，空白文字，または変換できない文字が見つかった場合，その文字までの文字群を読み込み，変換し，格納します。その後，本関数は次の変換指示へ進みます。

(3) サイズ

対応する引数のデータ型を解釈するデフォルトの方法を変更する，任意選択のサイズ文字 h，l，ll，および L です。

h を指定した場合，後ろに続く d，i，n，o，u，x の型指定を強制的に short int 型に変換し，short 型で格納します。c，e，f，n，p，s，D，I，O，U，X では，何もしません。

l を指定した場合，後ろに続く d，i，n，o，u，x の型指定を強制的に long int 型に変換し，long 型で格納します。e，f，g では，強制的に double 型に変換し，double 型で格納します。c，n，p，s，D，I，O，U，X では，何もしません。

ll を指定した場合，後ろに続く d，i，o，u，x，X の型指定を強制的に long long 型に変換し，long long 型で格納します。他の型指定では，何もしません。

L を指定した場合，後ろに続く e，f，g の型指定を強制的に long double 型に変換し，long double 型で格納します。他の型指定では，何もしません。

これら以外の場合，その動作は不定です。

(4) 型指定文字

適用される変換の型を指定する文字です。変換の型を指定する文字とその意味を次に示します。

%	文字“%”にマッチします。変換も代入も行われません。変換指示は“%%”となります。
c	1 文字を走査します。対応する引数は“char *arg”にしてください。
d	10 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
e，f，g	浮動小数点数を対応する引数に読み込みます。対応する引数は“float *arg”に，サイズ指定 l を指定した場合は“double *arg”にしてください。
i	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
n	対応する引数に読み込んだ文字の個数を格納します。対応する引数は“int *arg”にしてください。
o	8 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
p	走査したポインタを格納します。これは処理系定義です。 CX では，%p を %U とまったく同じように処理しています。対応する引数は“void **arg”にしてください。
s	与えられた配列の中に文字列を読み込みます。対応する引数は“char arg[ ]”にしてください。

u	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は“unsigned int *arg”にしてください。
x，X	16 進整数を対応する引数に読み込みます。対応する引数は“int *arg”にしてください。
D	10 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
E，F，G	浮動小数点数を対応する引数に読み込みます。対応する引数は“float *arg”に，サイズ指定 l を指定した場合は“double *arg”にしてください。
I	10 進，8 進，または 16 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
O	8 進整数を対応する引数に読み込みます。対応する引数は“long *arg”にしてください。
U	符号なし 10 進整数を対応する引数に読み込みます。対応する引数は“unsigned long *arg”にしてください。
[ ]	空でない文字列を引数 arg で始まるメモリの中へ読み込みます。この領域には，文字列と，自動的に付加される，文字列の終わりを示す null 文字（\0）とを受け入れられる大きさが必要です。対応する引数は“char *arg”にしてください。 [ ] で囲まれた文字パターンを，型指定文字 s の代わりに使用することができます。文字パターンは，本関数の入力フィールドを構成する文字の検索セットを定義する文字集合です。[ ] 内の最初の文字が“^”の場合，検索セットは反転され，[ ] 内の文字以外のすべての ASCII 文字が含まれます。また，ショートカットとして使用できる範囲指定機能もあります。たとえば，%[0-9] は，すべての 10 進数字と一致します。この集合内では，“-”は最初，または最後の文字にはできません。“-”の前の文字は，その後ろの文字よりも辞書式順序で小さくなるようにしてください。 - %[abcd] a，b，c，d のみを含む文字列と一致します。 - %[^abcd] a，b，c，d 以外の任意の文字を含む文字列と一致します。 - %[A-DW-Z] A，B，C，D，W，X，Y，Z を含む文字列と一致します。 - %[z-a] z，-，a と一致します（範囲指定とはみなされません）。

浮動小数点数（型指定文字 e，f，g，E，F，G）の場合，次の一般形式に対応させてください。

[+ | -] ddddd [.] ddd [E | e [+ | -] ddd]

ただし，上記の一般形式のうち [ ] で囲まれた部分は任意選択であり，ddd は 10 進数字を表します。

［注意事項］

- 通常のフィールド終了文字に到達する前に，特定フィールドの走査を停止したり完全に終了したりする可能性があります。
- 次の状況では，その時点でのフィールドの走査，格納を停止し，次の入力フィールドに移動します。
  - 代入抑制文字（*）が書式指定の中で“%”の後ろに現れており，その時点の入力フィールドは走査されているが格納はされていない。
  - フィールド長（正の 10 進整数）指定文字を読み込んだ。
  - 読み込む次の文字がその変換指示では変換できない（たとえば，指示が 10 進のときに Z を読み込む場合）。
  - 入力フィールド内の次の文字が検索セット内に現れていない（または反転検索セット内に現れている）。

  以上の理由からその時点の入力フィールドの走査を停止すると，次の文字が未読であるとみなされ，次の入力フィールドの最初の文字，またはその入力のあとの読み込み操作の最初の文字として使用されます。
- 本関数は，次の状況では終了します。
  - 入力フィールド内の次の文字が変換する文字列内の対応する通常文字と一致していない。
  - 入力フィールド内の次の文字が EOF である。
  - 変換する文字列が終了した。
- 変換する文字列に変換指示の一部ではない文字の並びが含まれている場合，この同じ文字の並びは入力の中に現れないようにしてください。本関数は一致する文字を走査しますが，格納はしません。不一致があった場合，一致していない最初の文字は読み取られていなかったかのように入力の中に残っています。

［使用例］

```
#include    <stdio.h>
void func(void) {
    int         i, n;
    float       x;
    const char  *s;
    char        name[10];
    s = "23 11.1e-1 NAME";
    n = sscanf(s,"%d%f%s", &i, &x, name);    /*i に 23，x に 1.110000，name に“NAME”を格納，
                                                戻り値 n は 3*/
}
```

fscanf

ストリームからのデータ読み込みと解釈を行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトとして扱い，*stream* から変換する入力を読み込みます。*stream* に指定できるのは，標準入出力の stdin だけです。*format* の記述方法は sscanf と同様です。

scanf

標準出力ストリームからのテキストの読み込みと解釈を行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

#include <stdio.h>

int scanf(const char **format*[, *arg*, ...]);

[戻り値]

走査，変換，格納が正常に実行できた入力フィールドの個数を返します。返却値には，格納されなかった走査済みフィールドは含まれません。ファイルの終わりで読み込もうとした場合，返却値は EOF です。フィールドが格納されなかった場合は，返却値は 0 です。

[詳細説明]

format の指す文字列で指定された書式に従い，その後ろに続く引数 *arg* を，変換された入力を格納するオブジェクトとして扱い，標準入出力の stdin から変換する入力を読み込みます。*format* の記述方法は sscanf と同様です。

[使用例]

```
#include    <stdio.h>
void func(void) {
    int     i, n;
    double  x;
    char    name[10];
    n = scanf("%d%lf%s", &i, &x, name); /*“23 11.1e-1 NAME”の形式の stdin から入力を
                                          書式化入力 */
}
```

ungetc

入力ストリームへの文字押し戻しを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

[所属]

標準ライブラリ

[指定形式]

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

[戻り値]

文字 *c* を返します。
エラー・リターンはありません。

[詳細説明]

文字 *c* を *stream* が指す入力ストリームへ押し戻します。ただし，*c* が EOF の場合，押し戻しは行われません。

押し戻された文字 *c* は，次の文字入力の際，最初の文字として入力されることになります。本関数によって，押し戻すことができるのは 1 文字だけです。本関数を続けて実行した場合，効果があるのは最後だけです。*stream* に指定できるのは，標準入出力の stdin だけです。

rewind

ファイル位置指示子のリセットを行います。

備考 CubeSuite が提供するデバッグ機能では，サポートされていません。

［所属］

標準ライブラリ

［指定形式］

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

［詳細説明］

stream が指す入力ストリームのエラー表示子をクリアし，ファイル位置表示子をファイルの先頭に位置付けます。

ただし，*stream* に指定できるのは，標準入出力の stdin だけです。そのため，本関数は ungetc による押し戻し文字を破棄する効果だけを持ちます。

perror

エラー処理を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdio.h>

void perror(const char *s);

［詳細説明］

グローバル変数 errno に対応するエラー・メッセージを stderr へ出力します。

出力されるメッセージは，次のようになります。

s が NULL でない場合	`fprintf(stderr, "%s:%s\n", s, s_fix);`
s が NULL の場合	`fprintf(stderr, "%s\n", s_fix);`

s_fix は，次のようになります。

errno が EDOM の場合	"EDOM error"
errno が ERANGE の場合	"ERANGE error"
errno が 0 の場合	"no error"
その他の場合	"error *xxx*"（*xxx* は abs (errno) % 1000）

［使用例］

```
#include    <stdio.h>
void func(double x) {
    double  d;
    errno = 0;
    d = exp(x);
    if(errno)
        perror("func");     /*exp で演算例外が発生した場合，perror を呼び出す */
}
```

6. 4. 7 標準ユーティリティ関数

標準ユーティリティ関数として，以下のものがあります。

表 6―24 標準ユーティリティ関数

関数／マクロ名	概要
abs	絶対値（int 型）を出力
labs	絶対値（long 型）を出力
llabs	絶対値（long long 型）を出力
bsearch	バイナリ検索
qsort	整列
div	除算（int 型）
ldiv	除算（long 型）
lldiv	除算（long long 型）
itoa	整数（int 型）を文字列に変換
ltoa	整数（long 型）を文字列に変換
ultoa	整数（unsigned long 型）を文字列に変換
lltoa	整数（long long 型）を文字列に変換
ulltoa	整数（unsigned long long 型）を文字列に変換
ecvt	浮動小数点値を数字文字列へ変換（総文字数指定）
ecvtf	浮動小数点値を数字文字列へ変換（総文字数指定）
fcvt	浮動小数点値を数字文字列へ変換（総文字数指定）
fcvtf	浮動小数点値を数字文字列へ変換（小数点数字数指定）
gcvt	浮動小数点値を数字文字列へ変換（書式指定）
gcvtf	浮動小数点値を数字文字列へ変換（書式指定）
atoi	文字列を整数（int 型）へ変換
atol	文字列を整数（long 型）へ変換
atoll	文字列を整数（long long 型）へ変換
strtol	文字列を整数（long 型）へ変換し，最終文字列へのポインタを格納
strtoul	文字列を整数（unsigned long 型）へ変換し，最終文字列へのポインタを格納
strtoll	文字列を整数（long long 型）へ変換し，最終文字列へのポインタを格納
strtoull	文字列を整数（unsigned long long 型）へ変換し，最終文字列へのポインタを格納
atoff	文字列を浮動小数点数（float 型）へ変換
atof	文字列を浮動小数点数（double 型）へ変換
strtodf	文字列を浮動小数点数（float 型）へ変換（最終文字列へのポインタ格納）
strtod	文字列を浮動小数点数（double 型）へ変換（最終文字列へのポインタ格納）
calloc	メモリ割り当て（ゼロ初期化付き）
malloc	メモリ割り当て（ゼロ初期化なし）
realloc	メモリの再割り当て

関数／マクロ名	概要
free	メモリ開放
rand	疑似乱数列生成
srand	疑似乱数列の種類を設定

abs

絶対値（int 型）を出力します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

int abs(int *j*);

［戻り値］

j の絶対値（*j* の大きさ），|*j*| を返します。

［詳細説明］

j の絶対値（*j* の大きさ），|*j*| を求めます。つまり，*j* が負の数の場合，結果は *j* の反転であり，負でない場合，*j* となります。

［使用例］

```
#include    <stdlib.h>
void func(int l) {
    int val;
    val = -15;
    l = abs(val);   /*val の値の絶対値 15 を l に返す */
}
```

labs

絶対値（long 型）を出力します。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

j の絶対値（*j* の大きさ），| *j* | を返します。

［詳細説明］

j の絶対値（*j* の大きさ），| *j* | を求めます。つまり，*j* が負の数の場合，結果は *j* の反転であり，負でない場合，*j* となります。abs と同じですが，int 型の値の代わりに long 型を使用し，戻り値も long 型です。

llabs

絶対値（long long 型）を出力します。

［所属］

標準ライブラリ

［指定形式］

```
#include <stdlib.h>
long long      llabs(long long j);
```

［戻り値］

j の絶対値（*j* の大きさ），| *j* | を返します。

［詳細説明］

j の絶対値（*j* の大きさ），| *j* | を求めます。つまり，*j* が負の数の場合，結果は *j* の反転であり，負でない場合，*j* となります。abs と同じですが，int 型の値の代わりに long long 型を使用し，戻り値も long long 型です。

bsearch

バイナリ検索を行います。

[所属]

標準ライブラリ

[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*));

[戻り値]

key と一致する配列の要素へのポインタを返します。一致する要素が複数ある場合，結果はその中で最初に見つかった要素を指します。*key* と一致する要素が見つからなかった場合，null ポインタを返します。

[詳細説明]

バイナリ検索法により，*base* から始まる配列の中で，*key* と一致する要素を検索します。*nmemb* は，配列の要素数です。*size* は，各要素のサイズです。配列は，*compar*（最後の引数）が指す比較関数に関し昇順で整列するようにしてください。*compar* が指す比較関数は，2 つの引数を持つように定義してください。結果は，1 番目の引数が 2 番目の引数よりも小さい場合は負，2 つの引数が一致する場合はゼロ，1 番目の引数が 2 番目の引数よりも大きい場合は正の整数を返すようにしてください。

[使用例]

```
#include    <stdlib.h>
#include    <string.h>
int compar(char **x, char **y);

void func(void) {
    static  char    *base[] = {"a", "b", "c", "d", "e", "f"};
    char            *key = "c";     /* 検索キーは “c” */
    char            **ret;
                                    /*ret に “c” へのポインタを格納 */
    ret = (char **) bsearch((char *) &key, (char *) base, 6, sizeof(char *), compar);
}
int compar(char **x, char **y) {
    return(strcmp(*x, *y));          /* 引数を比較して正，ゼロ，または負の整数を返す */
}
```

qsort

整列します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

void　qsort(void *_base_, size_t _nmemb_, size_t _size_, int (*_compar_)(const void*, const void *));

［詳細説明］

base の指す配列を _compar_ が指す比較関数に関し昇順に整列します。_nmemb_ は配列の要素数，_size_ は各要素のサイズです。_compar_ が指す比較関数は bsearch と同様です。

div

除算（int 型）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

div_t div(int *n*, int *d*);

［戻り値］

除算の結果を格納した構造体を返します。

［詳細説明］

int 型の値を除算する場合に使用します。

分子 *n* を分母 *d* で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 div_t のメンバとして格納します。

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

quot は商で，rem は剰余です。*d* がゼロでない場合，“*r* = div (*n*, *d*);”であれば，*n* は“*r*.rem + *d* * *r*.quot”に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が *n* と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

［使用例］

```
#include    <stdlib.h>
void func(void) {
    div_t   r;
    r = div(110, 3);    /*r.quot には 36，r.rem には 2 を格納 */
}
```

ldiv

除算（long 型）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

ldiv_t ldiv(long *n*, long *d*);

［戻り値］

除算の結果を格納した構造体を返します。

［詳細説明］

long 型の値を除算する場合に使用します。

分子 *n* を分母 *d* で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 ldiv_t のメンバとして格納します。

```
typedef struct {
    long    quot;
    long    rem;
} ldiv_t;
```

quot は商で，rem は剰余です。*d* がゼロでない場合，“*r* = ldiv (*n*, *d*);”であれば，*n* は“*r*.rem + *d* * *r*.quot”に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が *n* と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

lldiv

除算（long long 型）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

lldiv_t lldiv(long long *n*, long long *d*);

［戻り値］

除算の結果を格納した構造体を返します。

［詳細説明］

long long 型の値を除算する場合に使用します。

分子 *n* を分母 *d* で割ったその商と剰余を算出し，その 2 つの整数を次に示す構造体 lldiv_t のメンバとして格納します。

```
typedef struct {
    long long   quot;
    long long   rem;
} lldiv_t;
```

quot は商で，rem は剰余です。*d* がゼロでない場合，“*r* = lldiv (*n*, *d*);”であれば，*n* は“*r*.rem + *d* * *r*.quot”に等しい値です。

d がゼロの場合，結果の quot メンバは，符号が *n* と同じで，大きさが表現可能な最大の大きさとなります。また，rem メンバは 0 です。

itoa

整数（int 型）を文字列に変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char *itoa(int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。文字列の終わりには終端を示す null 文字（\ 0）が常に付加されます。*radix* に指定できるのは，2 から 36 までの数値です。*radix* が 10 の場合，*value* は符号付き数値として扱われ，*value* < 0 の場合文字列の先頭に“-”文字が付きます。その他の場合，*value* は符号なし数値として扱われます。*radix* > 10 の場合，10 から 35 に英小文字の a から z が当てられます。

［使用例］

```
#include    <stdlib.h>
void func(void) {
    char    buf[128];
    itoa(12345, buf, 16);   /*12345 を 16 進数文字列に変換 */
}
```

ltoa

整数（long 型）を文字列に変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char *ltoa(long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

ultoa

整数（unsigned long 型）を文字列に変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char *ultoa(unsigned long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

unsigned long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

lltoa

整数（long long 型）を文字列に変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char　*lltoa(long long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

long long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

ulltoa

整数（unsigned long long 型）を文字列に変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char　*ulltoa(unsigned long long int *value*, char **string*, int *radix*);

［戻り値］

string を返します。

［詳細説明］

unsigned long long int 型数値 *value* を *radix* 進数の文字列に変換して，*string* の示す配列に格納します。*value* の型を除き，itoa と同じです。

ecvt

浮動小数点値を数字文字列へ変換（総文字数指定）します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char *ecvt(double *val*, int *chars*, int **decpt*, int **sgn*);

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

double 型数値 *val* を数字で表した（null 文字（\ 0）で終端する）文字列を生成します。2 番目の引数 *chars* には，書き込む総文字数を指定します（数字のみを書き込むので変換された文字列の中の有効数字の数でもあります）。常に，*val* の整数部の桁がすべて含まれます。

ecvtf

浮動小数点値を数字文字列へ変換（総文字数指定）します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char *ecvtf(float *val*, int *chars*, int **decpt*, int **sgn*);

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

float 型数値 *val* を数字で表した（null 文字（\ 0）で終端する）文字列を生成します。2 番目の引数 *chars* には，書き込む総文字数を指定します（数字のみを書き込むので変換された文字列の中の有効数字の数でもあります）。常に，*val* の整数部の桁がすべて含まれます。

［使用例］

```
#include    <stdlib.h>
void func(void) {
    float   val;
    int     dec, sgn;
    val = 111.11;
    ecvtf(val, 12, &dec, &sgn); /*val の値 111.11 を 12 文字の文字列へ変換
                                    dec には小数点の左側の桁数 3，sgn には符号（正のため 0）を記録 */
}
```

fcvt

浮動小数点値を数字文字列へ変換（総文字数指定）します。

［所属］

標準ライブラリ

［指定形式］

```
#include <stdlib.h>
char    *fcvt(double val, int decimals, int *decpt, int *sgn);
```

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

2 番目の引数の解釈以外は ecvt と同じです。2 番目の引数 *decimals* には，小数点後に書き込む文字の数を指定します。ecvt と本関数は出力文字列の中に数字だけを書き込むので，小数点の位置を **decpt* に，数値の符号を **sgn* に記録しておきます。数をフォーマットしたあと，**decpt* には小数点の左側の桁数が入ります。**sgn* には，数値が正の場合は 0 が，負の場合は 1 が入ります。

fcvtf

浮動小数点値を数字文字列へ変換（小数点数字数指定）します。

［所属］

標準ライブラリ

［指定形式］

```
#include <stdlib.h>
char    *fcvtf(float val, int decimals, int *decpt, int *sgn);
```

［戻り値］

val の文字列表現を含む新しい文字列を指すポインタを返します。

［詳細説明］

2 番目の引数の解釈以外は ecvtf と同じです。2 番目の引数 *decimals* には，小数点後に書き込む文字の数を指定します。ecvtf と本関数は出力文字列の中に数字だけを書き込むので，小数点の位置を **decpt* に，数値の符号を **sgn* に記録しておきます。数をフォーマットしたあと，**decpt* には小数点の左側の桁数が入ります。**sgn* には，数値が正の場合は 0 が，負の場合は 1 が入ります。

gcvt

浮動小数点値を数字文字列へ変換（書式指定）します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char　*gcvtf(double *val*, int *prec*, char **buf*);

［戻り値］

val の書式付き文字列表現へのポインタ（引数 *buf* と同じです）を返します。

［詳細説明］

数値を文字列に書式変換し，バッファ *buf* に格納します。本関数は，sprintf の書式“%.*prec*”（負数だけに符号が付く）と同じ規則を使用し，有効桁数（*prec* で指定）に応じて，指数形式か，または通常の小数形式を選択します。

gcvtf

浮動小数点値を数字文字列へ変換（書式指定）します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

char　*gcvtf(float *val*, int *prec*, char **buf*);

［戻り値］

val の書式付き文字列表現へのポインタ（引数 *buf* と同じです）を返します。

［詳細説明］

数値を文字列に書式変換し，バッファ *buf* に格納します。本関数は，sprintf の書式“%.*prec*”（負数だけに符号が付く）と同じ規則を使用し，有効桁数（*prec* で指定）に応じて，指数形式か，または通常の小数形式を選択します。

atoi

文字列を整数（int 型）へ変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

int atoi(const char *_str_);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

［詳細説明］

str の指す文字列の最初の部分を int 型の表現に変換します。本関数は，“(int) strtol (*str*, NULL, 10)”と同じです。

atol

文字列を整数（long 型）へ変換します。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

［詳細説明］

str の指す文字列の最初の部分を long int 型の表現に変換します。本関数は，“strtol (*str*, NULL, 10)”と同じです。

atoll

文字列を整数（long long 型）へ変換します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

long long atoll(const char *_str_);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

［詳細説明］

str の指す文字列の最初の部分を long long int 型の表現に変換します。本関数は，“strtoll (*str*, NULL, 10)”と同じです。

strtol

文字列を整数（long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

long strtol(const char **str*, char ***ptr*, int *base*);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（変換された値が大きすぎる）場合，LONG_MAX，または LONG_MIN を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を long 型の表現に変換します。まず，入力文字を次の 3 つの部分，“最初の空白類”，“*base* の値により定められる基数において表現され，整数にする対象となる列”，“（null 文字（\ 0）を含む）最後の 1 個以上の認識されない文字列”に分割します。その後，対象となる列を整数へ変換し，その結果を返します。

（1）引数 *base* は，0，または 2 ～ 36 を指定します。

（a）*base* が 0 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号，16 進数であることを示す“0x”を前に持つ整数の形式となります。

（b）*base* の値が 2 ～ 36 の場合

対象となる文字列の予期される形式は，オプショナルな + 符号，または - 符号を前に持ち，*base* によって基数が指定された整数を表す文字列，または数字列となります。“a”（または“A”）から“z”（または“Z”）までの英字は 10 から 35 までの値を示すものとみなされます。与えられた値が *base* よりも小さい英字しか使用できません。

（c）*base* の値が 16 の場合

“0x”が文字と数字の列の前（符号が存在する場合は符号の後ろ）に置かれます（省略可能）。

(2) 対象となる列は，空白類以外の最初の文字で始まり，予期される形式を持つ入力文字列の先頭部分の最長の部分列として定義されます。

(a) 入力の文字列が空である場合やすべて空白類で構成されている場合，または空白類でない最初の文字が符号でも許容されうる文字でも数字でもない場合，対象となる列は空となります。

(b) 対象となる列が予期される形式を持ち，かつ，*base* の値が 0 の場合，入力文字列から基数を判断します。0x が先行する文字列は，16 進数数値とみなされ，先行 0 が付いていて x が付いていない文字列は 8 進数としてみなされます。他の文字列はすべて 10 進数としてみなされます。

(c) *base* が 2 から 36 までの間の値の場合，上述のように，これを変換用基数として使用します。

(d) 対象となる列が - 符号で始まる場合，変換結果の値の符号は反転されます。

(3) 最初の文字列を指すポインタ

(a) *ptr* が null ポインタでない場合，*ptr* の指すオブジェクトの中に格納されます。

(b) 対象となる列が空である場合，または予期された形式を持たない場合，変換は行われません。*str* の値は，*ptr* が null ポインタでない場合，*ptr* の指すオブジェクトに格納されます。

備考 本関数は，リエントラントではありません。

[使用例]

```
#include    <stdlib.h>
void func(long ret) {
    char    *p;
    ret = strtol("10", &p, 0);      /*ret に 10 を返す */
    ret = strtol("0x10", &p, 0);    /*ret に 16 を返す */
    ret = strtol("10x", &p, 2);     /*ret に 2 を返し，p の領域には‘x’へのポインタを格納 */
    ret = strtol("2ax3", &p, 16);   /*ret に 42 を返し，p の領域には‘x’へのポインタを格納 */
      :
}
```

strtoul

文字列を整数（unsigned long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

unsigned long　strtoul(const char **str*, char ***ptr*, int *base*);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる場合，ULONG_MAX を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

返却値の型が unsigned long 型になること以外，strtol と同じです。

strtoll

文字列を整数（long long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

long long　strtoll(const char **str*, char ***ptr*, int *base*);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（変換された値が大きすぎる）場合，LLONG_MAX，または LLONG_MIN を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

返却値の型が lng long 型になること以外，strtol と同じです。

strtoull

文字列を整数（unsigned long long 型）へ変換し，最終文字列へのポインタを格納します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

unsigned long long strtoull(const char **str*, char ***ptr*, int *base*);

［戻り値］

部分文字列が変換できた場合，変換された値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる場合，ULLONG_MAX を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

返却値の型が unsigned long long 型になること以外，strtol と同じです。

atoff

文字列を浮動小数点数（float 型）への変換を行います。

［所属］

標準ライブラリ

［指定形式］

```
#include <stdlib.h>
float   atoff(const char *str);
```

［戻り値］

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を float 型の表現に変換します。本関数は，“strtodf (*str*, NULL)”と同じです。

atof

文字列を浮動小数点数（double 型）への変換を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

double atof(const char *_str_);

［戻り値］

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を double 型の表現に変換します。本関数は，“strtod (*str*, NULL)”と同じです。

strtodf

文字列を浮動小数点数（float 型）への変換（最終文字列へのポインタ格納）を行います。

［所属］

標準ライブラリ

［指定形式］

```
#include <stdlib.h>
float   strtodf(const char *str, char **ptr);
```

［戻り値］

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を float 型の表現に変換します。変換される部分文字列は，次の形式の，空白でない通常の文字から始まる，*str* の最長先頭部分文字列です。

[+ | -] digits [.] [digits] [(e | E) [+ | -] digits]

str が空か，または空白文字だけから成り立っている場合，および最初の通常文字が“+，“-”，“.”，または数字以外の場合，部分文字列には文字が含まれていません。部分文字列が空の場合，変換は行われず，str の値が *ptr* の指す領域に格納されます。空でない場合，部分文字列は変換され，最終文字列（少なくとも str の終端を示す null 文字（\ 0）を含む）へのポインタが *ptr* の指す領域に格納されます。

備考 本関数は，リエントラントではありません。

［使用例］

```
#include    <stdlib.h>
#include    <stdio.h>
void func(float ret) {
    char    *p, *str, s[30];
    str = "+5.32a4e";
    ret = strtodf(str, &p);          /*ret には 5.320000 を返し，p の領域には‘a’へのポインタを
                                       格納 */
    sprintf(s, "%lf\t%c", ret, *p); /*s の指す配列に“5.320000 a”を格納 */
}
```

strtod

文字列を浮動小数点数（double 型）への変換（最終文字列へのポインタ格納）を行います。

［所属］

標準ライブラリ

［指定形式］

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

［戻り値］

部分文字列が変換できた場合，その値を返します。変換できなかった場合，0 を返します。

オーバーフローが生じる（値が表現可能な値の範囲にない）場合，HUGE_VAL，または -HUGE_VAL を返し，ERANGE をグローバル変数 errno にセットします。アンダフローが生じる場合，0 を返し，マクロ ERANGE をグローバル変数 errno にセットします。

［詳細説明］

str の指す文字列の最初の部分を double 型の表現に変換します。変換される部分文字列は，次の形式の，空白でない通常の文字から始まる，*str* の最長先頭部分文字列です。

[+ | -] digits [.] [digits] [(e | E) [+ | -] digits]

str が空か，または空白文字だけから成り立っている場合，および最初の通常文字が“+，“-”，“.”，または数字以外の場合，部分文字列には文字が含まれていません。部分文字列が空の場合，変換は行われず，str の値が *ptr* の指す領域に格納されます。空でない場合，部分文字列は変換され，最終文字列（少なくとも str の終端を示す null 文字（\ 0）を含む）へのポインタが *ptr* の指す領域に格納されます。

備考 本関数は，リエントラントではありません。

calloc

メモリ割り当て（ゼロ初期化付き）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

void *calloc(size_t *nmemb*, size_t *size*);

［戻り値］

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

［詳細説明］

大きさが *size* の，要素数 *nmemb* 個の配列領域を割り付けます。割り付けられた領域は 0 で初期化されます。

［注意事項］

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，デフォルトのサイズは 0x1000 バイトなので，変更する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x2000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

［使用例］

```
#include    <stdlib.h>
typedef struct {
    double  d[3];
    int     i[2];
} s_data;
int func(void) {
    s_data   *buf;
    if((buf = calloc(40, sizeof(s_data))) == NULL)  /*s_data40 個のための領域を割り付け */
        return(1);
    /* 処理を記述 */
    free(buf);                                      /* 領域を開放 */
    return(0);
}
```

malloc

メモリ割り当て（ゼロ初期化なし）を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

void *malloc(size_t *size*);

［戻り値］

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

［詳細説明］

大きさ *size* の領域を割り付けます。領域は初期化されません。

［注意事項］

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，デフォルトのサイズは 0x1000 バイトなので，変更する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x2000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

realloc

メモリの再割り当てを行います。

[所属]

標準ライブラリ

[指定形式]

#include <stdlib.h>

void *realloc(void *ptr, size_t size);

[戻り値]

領域の割り付けに成功した場合，その領域へのポインタを返します。割り付けができなかった場合，null ポインタを返します。

[詳細説明]

ptr が指す領域の大きさを，*size* の大きさに変更します。以前の大きさと，*size* の小さい方までの領域の内容は変わりません。領域を拡張する場合の，以前の大きさ以降の領域内容は初期化されません。*ptr* が null ポインタのときは，“malloc (*size*)”と同じ動作をします。それ以外の場合，*ptr* には，calloc，malloc，および本関数で獲得した領域を指定しなければなりません。

[注意事項]

記憶域管理の関数は，ヒープ・メモリ領域から必要に応じて自動的にメモリ領域を確保します。

また，デフォルトのサイズは 0x1000 バイトなので，変更する場合は，ヒープ・メモリ領域を確保する必要があります。領域の確保は，アプリケーションの最初で行ってください。

【ヒープ・メモリ設定例】

```
#define SIZEOF_HEAP 0x2000
int __sysheap[SIZEOF_HEAP >> 2];
size_t  __sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数“__sysheap”（アンダースコア‘_’は 2 つ）のシンボル“___sysheap”（アンダースコア‘_’は 3 つ）は，ヒープ・メモリの先頭アドレスを指します。この値は，4 の倍数にしてください。

2. 変数“__sizeof_sysheap”（最初のアンダースコア‘_’は 2 つ）に，必要なヒープ・メモリのサイズ（バイト）を設定してください。アセンブラ命令で記述する場合，シンボル“___sizeof_sysheap”（最初のアンダースコア‘_’は 3 つ）に設定してください。

free

メモリ開放を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

void free(void *ptr);

［詳細説明］

ptr が指す領域を開放し，その後の割り付けに使用できるようにします。*ptr* には，calloc，malloc，および realloc で獲得した領域を指定しなければなりません。

［使用例］

```
#include    <stdlib.h>
typedef struct {
    double  d[3];
    int     i[2];
} s_data;
int func(void) {
    s_data   *buf;
s
    if((buf = calloc(40, sizeof(s_data))) == NULL)  /*s_data40 個のための領域を割り付け */
        return(1);
    /* 処理を記述 */
    free(buf);                                      /* 領域を開放 */
    return(0);
}
```

rand

疑似乱数列生成を行います。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

int rand(void);

［戻り値］

乱数を返します。

［詳細説明］

0 以上 RAND_MAX 以下の乱数を返します。

［使用例］

```
#include    <stdlib.h>
void func(void) {
    if(rand() & 0xF) < 4)
        func1();    /*25%の確率で func1 を実行 */
}
```

srand

疑似乱数列の種類を設定します。

［所属］

標準ライブラリ

［指定形式］

#include <stdlib.h>

void srand(unsigned int *seed*);

［詳細説明］

後続する rand の呼び出しで使用する新しい疑似乱数列の種として，*seed* を与えます。本関数を同じ *seed* の値で呼んだ場合，rand により得られる乱数は，同じ値が同じ順番で現れることになります。本関数を実行せずに rand を実行した場合，最初に“srand (1)”を実行した場合と同じ結果になります。

6. 4. 8 非局所分岐関数

非局所分岐関数として，以下のものがあります。

表 6―25 非局所分岐関数

関数／マクロ名	概要
longjmp	非局所分岐
setjmp	非局所分岐の分岐先をセット

longjmp

非局所分岐を行います。

[所属]

標準ライブラリ

[指定形式]

#include <setjmp.h>

void longjmp(jmp_buf *env*, int *val*);

[戻り値]

第二引数 *val* を返します。ただし，*val* が 0 の場合，1 を返します。

[詳細説明]

setjmp で保存された *env* を使い，setjmp 直後へ非局所分岐します。*val* は，setjmp の返却値となります。

[注意事項]

-Xreg_mode=common 指定時は，setjmp，longjmp は -Xreg_mode=32 指定時と同じ動作をします。このため，r20 ～ r24 の値を setjmp 呼び出し後に変更しても，longjmp 呼び出し後には setjmp 呼び出し前の値に戻ります。

[使用例]

```
#include    <setjmp.h>
#define ERR_XXX1    1
jmp_buf jmp_env;

void func(void) {
    for(;;) {
        switch(setjmp(jmp_env)) {
            case    ERR_XXX1:   /*error XXX1 の終結処理 */
                break;
            case    0:          /* 非局所分岐ではない */
            default:
                    break;
        }
    }
}
```

```
void func1(void) {
    longjmp(jmp_env, ERR_XXX1); /*error XXX1の発生により非局所分岐*/
}
```

setjmp

非局所分岐の分岐先をセットします。

［所属］

標準ライブラリ

［指定形式］

#include <setjmp.h>

int setjmp(jmp_buf *env*);

［戻り値］

0 を返します。

［詳細説明］

非局所分岐のための戻り先を *env* にセットします。*env* には，本関数が実行された時点の環境が保存されます。

［注意事項］

-Xreg_mode=common 指定時は，setjmp，longjmp は -Xreg_mode=32 指定時と同じ動作をします。このため，r20 ～ r24 の値を setjmp 呼び出し後に変更しても，longjmp 呼び出し後には setjmp 呼び出し前の値に戻ります。

6. 4. 9 数学関数

数学関数として，以下のものがあります。

表 6―26 数学関数

関数／マクロ名	概要
j0f	第一種ベッセル関数（0 次）
j1f	第一種ベッセル関数（1 次）
jnf	第一種ベッセル関数（*n* 次）
y0f	第二種ベッセル関数（0 次）
y1f	第二種ベッセル関数（1 次）
ynf	第二種ベッセル関数（*n* 次）
erff	誤差関数（近似値）
erfcf	誤差関数（相補確率）
expf	指数関数
exp	指数関数
logf	対数関数（自然対数）
log	対数関数（自然対数）
log2f	対数関数（底 = 2）
log10f	対数関数（底 = 10）
log10	対数関数（底 = 10）
powf	べき乗関数
pow	べき乗関数
sqrtf	平方根関数
sqrt	平方根関数
cbrtf	立方根関数
cbrt	立方根関数
ceilf	ceiling 関数
ceil	ceiling 関数
fabsf	絶対値関数
fabs	絶対値関数
floorf	floor 関数
floor	floor 関数
fmodf	剰余関数
fmod	剰余関数
frexpf	浮動小数点数を仮数部とべき乗に分割
frexp	浮動小数点数を仮数部とべき乗に分割
ldexpf	浮動小数点数をべき乗に変換
ldexp	浮動小数点数をべき乗に変換

関数／マクロ名	概要
modff	浮動小数点数を整数部と小数部に分割
modf	浮動小数点数を整数部と小数部に分割
gammaf	対数ガンマ関数
hypotf	ユークリッド距離関数
matherrf (matherr)	エラー処理関数
matherrd	エラー処理関数
cosf	余弦
cos	余弦
sinf	正弦
sin	正弦
tanf	正接
tan	正接
acosf	逆余弦
acos	逆余弦
asinf	逆正弦
asin	逆正弦
atanf	逆正接
atan	逆正接
atan2f	逆正接（y / x）
atan2	逆正接（y / x）
coshf	双曲線余弦
cosh	双曲線余弦
sinhf	双曲線正弦
sinh	双曲線正弦
tanhf	双曲線正接
tanh	双曲線正接
acoshf	逆双曲線余弦
asinhf	逆双曲線正弦
atanhf	逆双曲線正接

j0f

第一種ベッセル関数（0 次）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　j0f(float *x*);

［戻り値］

0 次の第一種ベッセル関数値を返します。

［詳細説明］

0 次の第一種ベッセル関数値を求めます。

j1f

第一種ベッセル関数（1 次）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　j1f(float *x*);

［戻り値］

1 次の第一種ベッセル関数値を求めます。

［詳細説明］

1 次の第一種ベッセル関数値を返します。

備考　解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(void) {
    float   ret, x;
    ret = j1f(x);   /*x の値に対して，1 次の第一種ベッセル関数値を求め，ret に返す */
      :
    return(ret);
}
```

jnf

第一種ベッセル関数（*n* 次）です。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   jnf(int n, float x);
```

［戻り値］

n 次の第一種ベッセル関数値を返します。

［詳細説明］

n 次の第一種ベッセル関数値を求めます。

備考　*n* の絶対値が 3000 より大きい場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

y0f

第二種ベッセル関数（0 次）です。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   y0f(float x);
```

［戻り値］

0 次の第二種ベッセル関数値を返します。

［詳細説明］

0 次の第二種ベッセル関数値を求めます。

備考　0 を入力した場合，－∞を返し，グローバル変数 errno にマクロ ERANGE を設定します。
負数を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

y1f

第二種ベッセル関数（1 次）です。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   y1f(float x);
```

［戻り値］

1 次の第二種ベッセル関数値を返します。

［詳細説明］

1 次の第二種ベッセル関数値を求めます。

備考 0 を入力した場合，＋∞を返し，グローバル変数 errno にマクロ ERANGE を設定します。
負数を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

ynf

第二種ベッセル関数（*n* 次）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　ynf(int *n*, float *x*);

［戻り値］

n 次の第二種ベッセル関数値を返します。

［詳細説明］

n 次の第二種ベッセル関数値を求めます。

備考　*x* が 0 の場合，－∞を返し，グローバル変数 errno にマクロ ERANGE を設定します。
x が負数の場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
n の絶対値が 3000 より大きい場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
オーバーフローが生じた場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

erff

誤差関数（近似値）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float erff(float *x*);

［戻り値］

“誤差関数”の近似値（0 と 1 の間の数値）を返します。

［詳細説明］

観測値が平均の *x* 標準偏差の範囲になる確率を推定する“誤差関数”の近似値（0 と 1 の間の数値）を求めます。

備考 解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(void) {
    float   ret, x;
    ret = erff(x);  /*x の値に対して，誤差関数の近似値を求め，ret に返す */
      :
    return(ret);
}
```

erfcf

誤差関数（相補確率）です。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   erfcf(float x);
```

［戻り値］

相補確率を返します。

［詳細説明］

“1.0 - erff (*x*)”をして相補確率を求めます。これは，値の大きな *x* について“erff (*x*)”が呼び出された場合，その結果を 1.0 から引かれると精度が損なわれるために用意されています。

備考 解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

expf

指数関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　expf(float *x*);

［戻り値］

e の *x* 乗を返します。

アンダフローが生じた場合（*x* が結果を表せない大きさの負の数の場合），非正規化数を返し，グローバル変数 errno にマクロ ERANGE をセットします。オーバーフローが生じた場合（*x* が大きすぎる数の場合），HUGE_VAL（表現可能な最大の double 型数値）を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

e の *x* 乗を求めます（e は自然対数の底で，約 2.71828 です）。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

exp

指数関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double exp(double *x*);

［戻り値］

e の *x* 乗を返します。

アンダフローが生じた場合（*x* が結果を表せない大きさの負の数の場合），非正規化数を返し，グローバル変数 errno にマクロ ERANGE をセットします。オーバーフローが生じた場合（*x* が大きすぎる数の場合），HUGE_VAL（表現可能な最大の double 型数値）を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

e の *x* 乗を求めます（e は自然対数の底で，約 2.71828 です）。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

logf

対数関数（自然対数）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　logf(float *x*);

［戻り値］

x の自然対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

x の自然対数，つまり，底を e としてその対数を求めます。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

log

対数関数（自然対数）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double log(double *x*);

［戻り値］

x の自然対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

x の自然対数，つまり，底を e としてその対数を求めます。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

log2f

対数関数（底 = 2）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　log2f(float *x*);

［戻り値］

2 を底とする *x* の対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

2 を底とする *x* の対数を求めます。これは“log (*x*) / log (2)”によって実現されています。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

log10f

対数関数（底 = 10）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　log10f(float *x*);

［戻り値］

10 を底とする *x* の対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

10 を底とする *x* の対数を求めます。これは，“log (*x*) / log (10)”によって実現されています。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

log10

対数関数（底 = 10）です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double　log10(double *x*);

［戻り値］

10 を底とする *x* の対数を返します。

x が負の場合非数を返し，グローバル変数 errno にマクロ EDOM をセットします。x が 0 の場合，- ∞を返し，グローバル変数 errno にマクロ ERANGE をセットします。

［詳細説明］

10 を底とする *x* の対数を求めます。これは，“log (*x*) / log (10)”によって実現されています。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

powf

べき乗関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　powf(float *x*, float *y*);

［戻り値］

x の *y* 乗を返します。

x < 0 かつ *y* が奇整数の場合にのみ負の解を返します。*x* < 0 で *y* が非整数の場合，または *x* = *y* = 0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* = 0 かつ *y* < 0 の場合，またはオーバーフローが発生した場合，± HUGE_VAL を返し，errno にマクロ ERANGE をセットします。解が 0 へ向かって消滅した場合，0 を返し，errno に ERANGE をセットします。解が非正規化数の場合，errno に ERANGE をセットします。

［詳細説明］

x の *y* 乗を求めます。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(void) {
    float   ret, x, y;
    ret = powf(x, y);    /*x を y 乗した値を ret に返す */
      :
    return(ret);
}
```

pow

べき乗関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double pow(double *x*, double *y*);

［戻り値］

x の *y* 乗を返します。

x < 0 かつ *y* が奇整数の場合にのみ負の解を返します。*x* < 0 で *y* が非整数の場合，または *x* = *y* = 0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。*x* = 0 かつ *y* < 0 の場合，またはオーバーフローが発生した場合，± HUGE_VAL を返し，errno にマクロ ERANGE をセットします。解が 0 へ向かって消滅した場合，0 を返し，errno に ERANGE をセットします。解が非正規化数の場合，errno に ERANGE をセットします。

［詳細説明］

x の *y* 乗を求めます。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

sqrtf

平方根関数です。

[所属]

数学ライブラリ

[指定形式]

```
#include <math.h>
float   sqrtf(float x);
```

[戻り値]

x の正の平方根を返します。

x が実数で負の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。

[詳細説明]

x の正の平方根を求めます。

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

[注意事項]

V850E2V3 の FPU を持つデバイスの場合，最適化を有効にするとライブラリ関数呼び出しの代わりに sqrtf.s 命令を生成します。その場合はグローバル変数 errno の設定，および matherrf (matherr) 関数によるエラー処理の変更はできません。

ライブラリ関数を呼び出したい場合には，-Xcall_lib オプションを指定してください。

sqrt

平方根関数です。

［所属］

数学ライブラリ

［指定形式］

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

［戻り値］

x の正の平方根を返します。

x が実数で負の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。

［詳細説明］

x の正の平方根を求めます。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

cbrtf

立方根関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　cbrtf(float *x*);

［戻り値］

x の立方根を返します。

［詳細説明］

x の立方根を求めます。

cbrt

立方根関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double cbrt(double *x*);

［戻り値］

x の立方根を返します。

［詳細説明］

x の立方根を求めます。

ceilf

ceiling 関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　ceilf(float *x*);

［戻り値］

x 以上の最小の整数値を返します。

［詳細説明］

x 以上の最小の整数値を求めます。

ceil

ceiling 関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double ceil(double *x*);

［戻り値］

x 以上の最小の整数値を返します。

［詳細説明］

x 以上の最小の整数値を求めます。

fabsf

絶対値関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　fabsf(float *x*);

［戻り値］

x の絶対値（大きさ）を返します。

［詳細説明］

x のビット表現を直接操作して，*x* の絶対値（大きさ）を求めます。

fabs

絶対値関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double fabs(double *x*);

［戻り値］

x の絶対値（大きさ）を返します。

［詳細説明］

x のビット表現を直接操作して，*x* の絶対値（大きさ）を求めます。

floorf

floor 関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float floorf(float *x*);

［戻り値］

x 以下の最大の整数値を返します。

［詳細説明］

x 以下の最大の整数値を求めます。

floor

floor 関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double floor(double *x*);

［戻り値］

x 以下の最大の整数値を返します。

［詳細説明］

x 以下の最大の整数値を求めます。

fmodf

剰余関数です。

[所属]

数学ライブラリ

[指定形式]

#include <math.h>

float fmodf(float *x*, float *y*);

[戻り値]

x を *y* で割った剰余である浮動小数点数値を返します。

“fmodf (*x*, 0)”は，*x* を返します。

[詳細説明]

x を *y* で割った剰余である浮動小数点数値を求めます。つまり，*y* が 0 でない場合に，その結果の符号が *x* と同じ符号で大きさが *y* よりも小さい最大整数 i に対し，値“*x* - i * *y*”を求めます。

備考 *x* が±∞の場合，または *y* が 0 の場合，非数を返し，グローバル変数 errno にマクロ ERANGE を設定します。

matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
void func(void) {
    float   ret, x, y;
    ret = fmodf(x, y);  /*x を y で割った剰余を ret に返す */
      :
}
```

fmod

剰余関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double fmod(double *x*, double *y*);

［戻り値］

x を *y* で割った剰余である浮動小数点数値を返します。

“fmod (*x*, 0)”は，*x* を返します。

［詳細説明］

x を *y* で割った剰余である浮動小数点数値を求めます。つまり，*y* が 0 でない場合に，その結果の符号が *x* と同じ符号で大きさが *y* よりも小さい最大整数 i に対し，値“*x* - i * *y*”を求めます。

備考　*x* が±∞の場合，または *y* が 0 の場合，非数を返し，グローバル変数 errno にマクロ ERANGE を設定します。

matherrd を使用して，本関数のエラー処理を変更できます。

frexpf

浮動小数点数を仮数部とべき乗に分割します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float frexpf(float *val*, int **exp*);

［戻り値］

仮数部 m を返します。

val が 0 の場合，**exp* に 0 をセットし，0 を返します。

［詳細説明］

float 型の *val* を仮数部 m と 2 の p 乗で表します。結果の仮数部 m は，*val* が 0 でないかぎり，0.5 <= | x | < 1.0 となります。p は **exp* に格納されます。m，および p は，*val* = m * 2^p となるように計算されます。

備考 *val* が±∞の場合，0 を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
void func(void) {
    float   ret, x;
    int     exp;
    x = 5.28;
    ret = frexpf(x, &exp);  /* 結果の仮数部 0.66 が ret に返り，exp には 3 を格納 */
      :
}
```

frexp

浮動小数点数を仮数部とべき乗に分割します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double frexp(double *val*, int **exp*);

［戻り値］

仮数部 m を返します。

val が 0 の場合，**exp* に 0 をセットし，0 を返します。

［詳細説明］

double 型の *val* を仮数部 m と 2 の p 乗で表します。結果の仮数部 m は，*val* が 0 でないかぎり，0.5 <= | x | < 1.0 となります。p は **exp* に格納されます。m，および p は，*val* = m * 2^{p} となるように計算されます。

備考 *val* が±∞の場合，0 を返し，グローバル変数 errno にマクロ EDOM を設定します。

matherrd を使用して，本関数のエラー処理を変更できます。

ldexpf

浮動小数点数をべき乗に変換します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float ldexpf(float *val*, int *exp*);

［戻り値］

val * 2 ^{exp} で求めた値を返します。

アンダフロー，またはオーバーフローが生じた場合，グローバル変数 errno にマクロ ERANGE がセットされます。アンダフローの場合，非正規化数を返します。オーバーフローの場合，HUGE_VAL を返します。

［詳細説明］

val * 2 ^{exp} を求めます。

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

ldexp

浮動小数点数をべき乗に変換します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double ldexp(double *val*, int *exp*);

［戻り値］

val * 2 ^{exp} で求めた値を返します。

アンダフロー，またはオーバーフローが生じた場合，グローバル変数 errno にマクロ ERANGE がセットされます。アンダフローの場合，非正規化数を返します。オーバーフローの場合，HUGE_VAL を返します。

［詳細説明］

val * 2 ^{exp} を求めます。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

modff

浮動小数点数を整数部と小数部に分割します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float modff(float *val*, float **ipart*);

［戻り値］

小数部を返します。結果の符号は *val* の符号と同じです。

［詳細説明］

float 型の *val* を整数部と小数部とに分割し，整数部を **ipart* に格納します。丸めは行いません。整数部と小数部の和は，正確に *val* と一致するように保証されています。たとえば，“*realpart* = modff (*val*, &*intpart*)”であるとき，“*realpart* + *intpart*”は *val* と一致します。

modf

浮動小数点数を整数部と小数部に分割します。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double modf(double *val*, double **ipart*);

［戻り値］

小数部を返します。結果の符号は *val* の符号と同じです。

［詳細説明］

double 型の *val* を整数部と小数部とに分割し，整数部を **ipart* に格納します。丸めは行いません。整数部と小数部の和は，正確に *val* と一致するように保証されています。たとえば，“*realpart* = modf (*val*, *&intpart*)”であるとき，“*realpart* + *intpart*”は *val* と一致します。

gammaf

対数ガンマ関数です。

[所属]

数学ライブラリ

[指定形式]

#include <math.h>

float gammaf(float *x*);

[戻り値]

x のガンマ関数の自然対数を返します。

x が 0 のとき，またはオーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にマクロ ERANGE をセットします。

[詳細説明]

“ln (Γ (*x*))”，つまり，*x* のガンマ関数の自然対数を求めます。ガンマ関数“expf (gammaf (*x*))”は階乗の一般化であり，Γ (N) ≡ N * Γ (N - 1) という関係式を持っています。したがって，ガンマ関数自体の結果は非常に早く大きくなります。そのため，本関数は，表現可能な結果の有効範囲を拡大するために，単なる“Γ (*x*)”ではなく“ln (Γ (*x*))”として定義されています。

備考 負数を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

[使用例]

```
#include    <math.h>
float func(float x) {
    float   ret;
    ret = gammaf(x);    /*x のガンマ関数の自然対数を ret に返す */
      :
    return(ret);
}
```

hypotf

ユークリッド距離関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float hypotf(float *x*, float *y*);

［戻り値］

原点（0, 0）とカーテシアン座標（*x*, *y*）で表される点との間のユークリッド距離“$\sqrt{(x^2 + y^2)}$”を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

原点（0, 0）とカーテシアン座標（*x*, *y*）で表される点との間のユークリッド距離“$\sqrt{(x^2 + y^2)}$”を求めます。

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
void func(float x) {
    float   ret, y;
    ret = hypotf(x, y); /* (0, 0) 座標と (x, y) 座標の間のユークリッド距離を ret に返す */
}
```

matherrf (matherr)

エラー処理関数です。

備考 matherr を matherrf として使用できます。

[所属]

数学ライブラリ

[指定形式]

#include <math.h>

int matherrf(struct exceptionf *e);

[戻り値]

e->retval の値を変更することにより，カスタマイズした本関数からの呼び出し関数の結果を変更できます。これは，呼び出し側の関数にも伝播します。本関数は，エラーを解決した場合，0 以外の値を返し，エラーを解決できなかった場合，0 を返します。本関数が 0 を返す場合，呼び出し側でグローバル変数 errno に適切な値をセットします。

[詳細説明]

数学ライブラリ関数内でエラーが発生した場合に呼ばれる関数です。

したがって，matherrf という名前の関数をユーザ・サブルーチンで用意することにより，エラー処理をカスタマイズできます。カスタマイズする matherrf は，エラーの解決に失敗した場合に 0 を返し，エラーを解決した場合に 0 以外の値を返すようにする必要があります。matherrf が 0 以外の値を返した場合，グローバル変数 errno の値は変更されません。

エラー処理のカスタマイズは，構造体 exceptionf へのポインタ *e で渡された情報を利用して行うことができます。構造体 exceptionf は“math.h”中で次のように定義されています。

```
#define __exception exceptionf
struct  __exceptionf {
    int             type;
    const char      *name;
    float           arg1, arg2, retval;
};
```

各メンバの意味は，次のとおりです。

type	発生した数学関数エラーのタイプです。 マクロ・エンコーディング・エラーのタイプも“math.h”の中で定義されています。
name	エラーが発生した数学ライブラリ関数の名前を保持し，空文字で終わっている文字列を指すポインタです。
arg1，arg2	エラーの原因となった引数です。
retval	呼び出し関数が返すエラー・リターン値です。

発生する可能性のある数学ライブラリ関数エラーのタイプは，次のとおりです。

DOMAIN	引数が関数の定義域の範囲にない 例 logf (-1);
OVERFLOW	オーバーフロー 例 exp (1000);
INEXACT	解の 0 へ向かっての消滅 例 exp (-1000);
UNDERFLOW	アンダフロー，非正規化数の解 解 < 1.1755e -38 かつ非 0 の数で，精度が通常の数値より落ちた状態
Z_DIVISION	ゼロ除算

備考 演算例外発生時の matherrf の呼び出しと，標準関数でのグローバル変数 errno の更新は，リエントラント性を持ちません。

［注意事項］

-Xreg_mode=common 指定時のランタイム関数は -Xreg_mode=32 指定時と同じ動作をします。このため，例外発生時に r15 ～ r19 の値を matherrf 内で更新しても，ランタイム関数を呼び出したプログラムには反映されません。

［使用例］

```
#include    <math.h>
#include    <stdio.h>
void func(void) {
    float   ret;
    ret = logf(-0.1);   /*retに3を返す*/
}
int matherrf(struct exceptionf *e) {
    char    s[30];
    switch(e->type) {
        case    DOMAIN:
            sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
            e->retval = 3;  /*エラー・リターン値を3に変更*/
            break;
        default:
            sprintf(s, "%s other error %e\n", e->name, e->arg1);
    }
    return(1);
}
```

matherrd

エラー処理関数です。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

int matherrd(struct exceptiond *e);

［戻り値］

e->retval の値を変更することにより，カスタマイズした本関数からの呼び出し関数の結果を変更できます。これは，呼び出し側の関数にも伝播します。本関数は，エラーを解決した場合，0 以外の値を返し，エラーを解決できなかった場合，0 を返します。本関数が 0 を返す場合，呼び出し側でグローバル変数 errno に適切な値をセットします。

［詳細説明］

数学ライブラリ関数内でエラーが発生した場合に呼ばれる関数です。

したがって，matherrd という名前の関数をユーザ・サブルーチンで用意することにより，エラー処理をカスタマイズできます。カスタマイズする matherrd は，エラーの解決に失敗した場合に 0 を返し，エラーを解決した場合に 0 以外の値を返すようにする必要があります。matherrd が 0 以外の値を返した場合，グローバル変数 errno の値は変更されません。

エラー処理のカスタマイズは，構造体 exceptiond へのポインタ *e で渡された情報を利用して行うことができます。構造体 exceptiond は“math.h”中で次のように定義されています。

```
#define __exceptiond exceptiond
struct  __exceptiond {
    int             type;
    const char      *name;
    double          arg1, arg2, retval;
};
```

各メンバの意味は，次のとおりです。

type	発生した数学関数エラーのタイプです。 マクロ・エンコーディング・エラーのタイプも“math.h”の中で定義されています。
name	エラーが発生した数学ライブラリ関数の名前を保持し，空文字で終わっている文字列を指すポインタです。
arg1，arg2	エラーの原因となった引数です。
retval	呼び出し関数が返すエラー・リターン値です。

発生する可能性のある数学ライブラリ関数エラーのタイプは，次のとおりです。

DOMAIN	引数が関数の定義域の範囲にない 例 logf (-1);
OVERFLOW	オーバーフロー 例 exp (1000);
INEXACT	解の 0 へ向かっての消滅 例 exp (-1000);
UNDERFLOW	アンダフロー，非正規化数の解 解 < 1.1755e -38 かつ非 0 の数で，精度が通常の数値より落ちた状態
Z_DIVISION	ゼロ除算

備考 演算例外発生時の matherrd の呼び出しと，標準関数でのグローバル変数 errno の更新は，リエントラント性を持ちません。

［注意事項］

-Xreg_mode=common 指定時のランタイム関数は -Xreg_mode=32 指定時と同じ動作をします。このため，例外発生時に r15 ～ r19 の値を matherrd 内で更新しても，ランタイム関数を呼び出したプログラムには反映されません。

［使用例］

```
#include    <math.h>
#include    <stdio.h>
void func(void) {
    float   ret;
    ret = logf(-0.1);   /*retに3を返す*/
}
int matherrd(struct exceptiond *e) {
    char    s[30];
    switch(e->type) {
        case    DOMAIN:
            sprintf(s, "%s DOMAIN error %e\n", e->name, e->arg1);
            e->retval = 3;  /*エラー・リターン値を3に変更*/
            break;
        default:
            sprintf(s, "%s other error %e\n", e->name, e->arg1);
    }
    return(1);
}
```

cosf

余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   cosf(float x);
```

［戻り値］

x の余弦を返します。

［詳細説明］

x の余弦を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

cos

余弦を求めます。

[所属]

数学ライブラリ

[指定形式]

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

[戻り値]

x の余弦を返します。

[詳細説明]

x の余弦を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
matherrd を使用して，本関数のエラー処理を変更できます。

sinf

正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float sinf(float *x*);

［戻り値］

x の正弦を返します。

［詳細説明］

x の正弦を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

sin

正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double sin(double *x*);

［戻り値］

x の正弦を返します。

［詳細説明］

x の正弦を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrd を使用して，本関数のエラー処理を変更できます。

tanf

正接を求めます。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   tanf(float x);
```

［戻り値］

x の正接を返します。

［詳細説明］

x の正接を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

tan

正接を求めます。

［所属］

数学ライブラリ

［指定形式］

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

［戻り値］

x の正接を返します。

［詳細説明］

x の正接を求めます。角度はラジアン単位で指定します。

備考 ±∞を入力した場合，非数を返し，グローバル変数 errno にマクロ EDOM を設定します。
解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrd を使用して，本関数のエラー処理を変更できます。

acosf

逆余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float acosf(float *x*);

［戻り値］

x の逆余弦（アークコサイン）を返します。返す値はラジアン単位で，0 から π までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットます。

［詳細説明］

x の逆余弦（アークコサイン）を求めます。*x* は，$-1 \leqq x \leqq 1$ で指定します。

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

acos

逆余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double acos(double *x*);

［戻り値］

x の逆余弦（アークコサイン）を返します。返す値はラジアン単位で，0 から π までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットます。

［詳細説明］

x の逆余弦（アークコサイン）を求めます。*x* は，$-1 \leqq x \leqq 1$ で指定します。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

asinf

逆正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　asinf(float *x*);

［戻り値］

x の逆正弦（アークサイン）を返します。返す値はラジアン単位で，-π / 2 から π / 2 までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットします。

［詳細説明］

x の逆正弦（アークサイン）を求めます。*x* は，-1 ≦ *x* ≦ 1 で指定します。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

asin

逆正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double asin(double *x*);

［戻り値］

x の逆正弦（アークサイン）を返します。返す値はラジアン単位で，-π / 2 から π / 2 までの範囲です。

x が -1 と 1 の間にない場合，非数を返します。また，グローバル変数 errno にマクロ EDOM をセットします。

［詳細説明］

x の逆正弦（アークサイン）を求めます。*x* は，-1 ≦ *x* ≦ 1 で指定します。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

atanf

逆正接を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　atanf(float *x*);

［戻り値］

x の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-**π** / 2 から **π** / 2 の範囲です。

［詳細説明］

x の逆正接（アークタンジェント）を求めます。*x* は，-1 ≦ *x* ≦ 1 で指定します。

備考　解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(float x) {
    float   ret;
    ret = atanf(x); /*x の逆正接の値を ret に返す */
      :
    return(ret);
}
```

atan

逆正接を求めます。

［所属］

数学ライブラリ

［指定形式］

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

［戻り値］

x の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-**π** / 2 から **π** / 2 の範囲です。

［詳細説明］

x の逆正接（アークタンジェント）を求めます。*x* は，-1 ≦ *x* ≦ 1 で指定します。

備考 解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrd を使用して，本関数のエラー処理を変更できます。

atan2f

逆正接（y / x）を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　atan2f(float *y*, float *x*);

［戻り値］

y / *x* の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π から π までの範囲です。

x と *y* がともに 0.0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，グローバル変数 errno にマクロ ERANGE をセットします。解が非正規化数の場合，グローバル変数 errno に ERANGE をセットします。

［詳細説明］

y / *x* の逆正接（アークタンジェント）を求めます。本関数は，π / 2，または -π / 2 付近（*x* が 0 に近い場合）の角度の場合も正しい結果を求めます。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

atan2

逆正接（y / x）を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double atan2(double *y*, double *x*);

［戻り値］

y / *x* の逆正接（アークタンジェント）を返します。返す値はラジアン単位で，-π から π までの範囲です。

x と *y* がともに 0.0 の場合，非数を返し，グローバル変数 errno にマクロ EDOM をセットします。解が 0 へ向かって消滅した場合，± 0 を返し，グローバル変数 errno にマクロ ERANGE をセットします。解が非正規化数の場合，グローバル変数 errno に ERANGE をセットします。

［詳細説明］

y / *x* の逆正接（アークタンジェント）を求めます。本関数は，π / 2，または -π / 2 付近（*x* が 0 に近い場合）の角度の場合も正しい結果を求めます。

備考　matherrd を使用して，本関数のエラー処理を変更できます。

coshf

双曲線余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float coshf(float *x*);

［戻り値］

x の双曲線余弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線余弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} + e^{-x}) / 2$$

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

cosh

双曲線余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double cosh(double *x*);

［戻り値］

x の双曲線余弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線余弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} + e^{-x}) / 2$$

備考 matherrd を使用して，本関数のエラー処理を変更できます。

sinhf

双曲線正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float sinhf(float *x*);

［戻り値］

x の双曲線正弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線正弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} - e^{-x}) / 2$$

備考 matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

sinh

双曲線正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double sinh(double *x*);

［戻り値］

x の双曲線正弦を返します。

オーバーフローが生じた場合，HUGE_VAL を返し，グローバル変数 errno にはマクロ ERANGE をセットします。

［詳細説明］

x の双曲線正弦を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

$$(e^{x} - e^{-x}) / 2$$

備考 matherrd を使用して，本関数のエラー処理を変更できます。

tanhf

双曲線正接を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float tanhf(float *x*);

［戻り値］

x の双曲線正接を返します。

［詳細説明］

x の双曲線正接を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

sinh (*x*) / cosh (*x*)

備考 解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

tanh

双曲線正接を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

double tanh(double *x*);

［戻り値］

x の双曲線正接を返します。

［詳細説明］

x の双曲線正接を求めます。角度はラジアン単位で指定します。定義式は次のとおりです。

sinh (*x*) / cosh (*x*)

備考　解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrd を使用して，本関数のエラー処理を変更できます。

acoshf

逆双曲線余弦を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　acoshf(float *x*);

［戻り値］

x（*x* は 1 以上の数値）の逆双曲線余弦を返します。

x が 1 より小さい場合，非数を返します。また，グローバル変数 errno にはマクロ EDOM をセットします。

［詳細説明］

x（*x* は 1 以上の数値）の逆双曲線余弦を求めます。定義式は次のとおりです。

$$\ln\left(x + \sqrt{\left(x^2 - 1\right)}\right)$$

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

［使用例］

```
#include    <math.h>
float func(float x) {
    float   ret;
    ret = acoshf(x);    /*x の逆双曲線余弦の値を ret に返す */
      :
    return(ret);
}
```

asinhf

逆双曲線正弦を求めます。

［所属］

数学ライブラリ

［指定形式］

```
#include <math.h>
float   asinhf(float x);
```

［戻り値］

x の逆双曲線正弦を返します。

［詳細説明］

x の逆双曲線正弦を求めます。定義式は次のとおりです。

$$\mathrm{sign}\,(x) * \ln\,(|\,x\,| + \sqrt{\ }\,(1 + x^{2}))$$

備考 解が非正規化数の場合，グローバル変数 errno にマクロ ERANGE を設定します。
matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

atanhf

逆双曲線正接を求めます。

［所属］

数学ライブラリ

［指定形式］

#include <math.h>

float　atanhf(float *x*);

［戻り値］

x の逆双曲線正接を返します。

x の絶対値が 1 以上の場合，非数を返し，グローバル変数 errno にはマクロ EDOM をセットします。

［詳細説明］

x の逆双曲線正接を求めます。

備考　matherrf (matherr) を使用して，本関数のエラー処理を変更できます。

6. 4. 10 周辺装置の初期化関数

周辺装置の初期化関数として，以下のものがあります。

表 6―27 周辺装置の初期化関数

関数／マクロ名	概要
hdwinit	CPU リセット直後の周辺装置の初期化処理

hdwinit

CPU リセット直後に周辺装置の初期化処理を行います。

[所属]

初期化処理ライブラリ

[指定形式]

void hdwinit(void);

[詳細説明]

周辺装置の初期化関数は，CPU リセット直後に周辺装置の初期化処理を行う関数です。

スタートアップ・ルーチン内から呼び出されます。

ライブラリに含まれる関数は，実動作は何もしないダミー・ルーチンですので，システムに合わせて，記述してください。

6. 4. 11 コピー関数

初期値ありデータや，プログラム・コードを RAM にコピーするためのルーチンです。

- ROM 化用の関数自体は sdata 領域，sbss 領域は使用しません。sdata 領域にデータを書き込む動作は行います。
- ROM 化用の関数は，通常メイン・プログラムを実行する前に一度だけ呼び出します。そのため，リエントラント性に関しては考慮しません。
- インサーキット・エミュレータ（ICE）にロード・モジュールをダウンロードした場合，data 領域や sdata 領域に置かれる初期値ありデータは，ダウンロードした時点でセットされます。
  したがって，コピー関数を呼び出さなくても，デバッグが可能になります。しかし，ROM 化用ロード・モジュールを作成して実機で動作させる場合，コピー関数で初期値ありデータ・コピー等を行わないと，初期値が設定されずに期待した動作をしません。「インサーキット・エミュレータで動作したが，実機でうまく動作しない」という場合，その原因のほとんどは，このコピー関数で初期値を設定していないためです。また，初期化時に，RAM をゼロクリアする処理を行っているような場合は，そのルーチンよりも後にコピー関数を呼び出してください。初期値もゼロクリアされてしまうためです。

コピー関数として，以下のものがあります。

表 6―28 コピー関数

関数／マクロ名	概要
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

備考 1. _rcopy と _rcopy1 はどちらもまったく同じ動作をします。
内蔵命令 RAM を持っている V850（V850E/ME2 など）で，プログラム・コードを内蔵命令 RAM にコピーする場合，ハードウエアの仕様上，4 バイト単位でコピーする必要があります。その場合，_rcopy4 を使ってコピーします。ハードウエア的な制限がなければ，どの関数を使っても問題ありませんが，2 バイト，4 バイト単位でコピーする場合は，コピーの必要がある領域を越える（パッキングされた領域のサイズが 4 の倍数でなかった場合，パッキング領域外の領域も同時にコピーされる）可能性があるため，注意が必要です。

2. コピー関数についての詳細は，「8. 4 コピー関数」を参照してください。

6. 4. 12 マルチコア用擬似 main 関数

マルチコア用擬似 main 関数として，以下のものがあります。

表 6―29 マルチコア用擬似 main 関数

関数／マクロ名	概要
main_pen	制御を呼び出し側に戻さない

main_pe*n*

無限にループし，制御を呼び出し側に戻さないようにします。

［所属］

マルチコア用ライブラリ

［指定形式］

int main_pe*n* (void);

備考 便宜上の宣言であり，ユーザがスタートアップ・ルーチンで引数／戻り値を制御できます。

［詳細説明］

何もしない関数です。

マルチコア・デバイス使用時に，ユーザが PE1 以外用の main( ) 関数を用意しない場合に，マルチコア用のスタートアップルーチンからリンクされます。

内部では無限にループし，制御を呼び出し側に戻さないようにします。

main_pe2 ～ main_pe31 まであります。

［使用例］

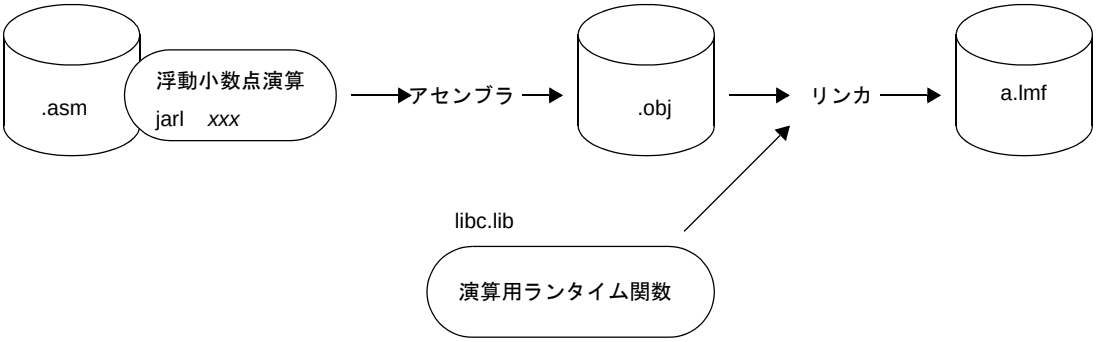
```
        ld.hu   PEID, r10
        cmp     1, r10
        be      .L1
        # PE1 以外の処理
        jarl    main_pe2, lp ;        /* 無限ループになっているので戻ってこない */
.L1:
        # PE1 用の処理が続く
```

6. 4. 13 演算用ランタイム関数

演算用ランタイム関数は，64 ビット・データの演算や浮動小数点演算を行うために CX が自動的に使用するルーチンです。標準ライブラリとともに，libc.lib ファイルに含まれています。ヘッダ・ファイルのインクルードは必要ありません。演算用ランタイム関数は，“関数前後処理ランタイム関数”と同様に，C ソースやアセンブラ・ソースで記述する関数ではありません。

なお，演算用ランタイム関数をアプリケーション・プログラムで使用する場合，実行可能なオブジェクト・モジュール・ファイル作成時に，リンカで libc.lib を参照する必要があります

図 6―1 演算用ランタイム関数使用イメージ



演算用ランタイム関数として，以下のものがあります。

表 6―30 演算用ランタイム関数

分類	関数名	概要
float 型演算関数	___addf.s	単精度浮動小数点の加算
	___subf.s	単精度浮動小数点の減算
	___mulf.s	単精度浮動小数点の乗算
	___divf.s	単精度浮動小数点の除算
	___cmpf.s	単精度浮動小数点の比較演算
	___fcmp.s	単精度浮動小数点の比較演算
	___negf.s	単精度浮動小数点の符号反転
	___notf.s	単精度浮動小数点の論理否定
double 型演算関数	___addf.d	倍精度浮動小数点型の加算
	___subf.d	倍精度浮動小数点型の減算
	___mulf.d	倍精度浮動小数点型の乗算
	___divf.d	倍精度浮動小数点型の除算
	___fcmp.d	倍精度浮動小数点型の比較演算
	___negf.d	倍精度浮動小数点型の符号反転
	___notf.d	倍精度浮動小数点型の論理否定

分類	関数名	概要
long long 型演算関数	___add.l	64 ビット整数の加算
	___sub.l	64 ビット整数の減算
	___mul.l	64 ビット整数の乗算
	___div.l	符号付き 64 ビット整数の除算
	___div.ul	符号なし 64 ビット整数の除算
	___mod.l	符号付き 64 ビット整数の剰余算
	___mod.ul	符号なし 64 ビット整数の剰余算
	___shl.l	64 ビット整数の論理左シフト演算
	___shr.l	64 ビット整数の論理右シフト演算
	___sar.l	64 ビット整数の算術右シフト演算
	___inc.l	64 ビット整数の増分
	___dec.l	64 ビット整数の減分
	___not.l	64 ビット整数の論理否定
	___neg.l	64 ビット整数の符号反転
	___cmp.l	符号付き 64 ビット整数の比較演算
	___cmp.ul	符号なし 64 ビット整数の比較演算
	___bext.l	符号付き 64 ビット整数のビット抽出
	___bext.ul	符号なし 64 ビット整数のビット抽出
	___bins.l	64 ビット整数のビット挿入
型変換関数	___cvt.ws	32 ビット整数から単精度浮動小数点への変換
	___cvt.wd	32 ビット整数から倍精度浮動小数点への変換
	___cvt.uws	符号なし 32 ビット整数から単精度浮動小数点への変換
	___cvt.uwd	符号なし 32 ビット整数から倍精度浮動小数点への変換
	___cvt.ls	64 ビット整数から単精度浮動小数点への変換
	___cvt.ld	64 ビット整数から倍精度浮動小数点への変換
	___cvt.uls	符号なし 64 ビット整数から単精度浮動小数点への変換
	___cvt.uld	符号なし 64 ビット整数から倍精度浮動小数点への変換
	___trnc.sw	単精度浮動小数点数から 32 ビット整数への変換
	___trnc.dw	倍精度浮動小数点数から 32 ビット整数への変換
	___trnc.suw	単精度浮動小数点数から符号なし 32 ビット整数への変換
	___trnc.duw	倍精度浮動小数点数から符号なし 32 ビット整数への変換
	___trnc.sl	単精度浮動小数点数から 64 ビット整数への変換
	___trnc.dl	倍精度浮動小数点数から 64 ビット整数への変換
	___trnc.sul	単精度浮動小数点数から符号なし 64 ビット整数への変換
	___trnc.dul	倍精度浮動小数点数から符号なし 64 ビット整数への変換
	___cvt.sd	単精度浮動小数点数から倍精度浮動小数点への変換
	___cvt.ds	倍精度浮動小数点数から単精度浮動小数点への変換

分類	関数名	概要
int 型演算関数	___mul	符号付き乗算
	___mulu	符号なし乗算
	___div	符号付き除算
	___divu	符号なし除算
	___mod	符号付き剰余算
	___modu	符号なし剰余算

備考 1. 演算用ランタイム関数は，本来，コード生成部が使用するものであり，単体で使用することを前提としていません。したがって，アセンブラ・ソースで使用する場合，演算用ランタイム関数を呼び出すための前処理が必要です。

2. 演算用ランタイム関数は，C ソース・プログラムでは使用できません。

___addf.s

float 型の加算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	加算の左項
r6	加算の右項

[戻り値]

r6	加算の結果

[詳細説明]

float 型の加算を行います。

___subf.s

float 型の減算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	減算の左項
r6	減算の右項

[戻り値]

r6	減算の結果

[詳細説明]

float 型の減算を行います。

___mulf.s

float 型の乗算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	乗算の左項
r6	乗算の右項

[戻り値]

r6	乗算の結果

[詳細説明]

float 型の乗算を行います。

___divf.s

float 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	除算の左項
r6	除算の右項

[戻り値]

r6	除算の結果

[詳細説明]

float 型の除算を行います。

___cmpf.s

float 型の比較演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r8	比較直前の PSW の値
r7	比較演算の左項
r6	比較演算の右項

[戻り値]

PSW	比較演算の結果
r6	比較後の PSW と同じ値

[詳細説明]

float 型の比較演算を行います。

以下の組み合わせの結果を返します。

	Z フラグ	CY フラグ	S フラグ
左項，または右項が非数	不定	不定	不定
左項＝右項＝＋∞	不定	不定	不定
左項＝右項＝－∞	不定	不定	不定
左項＞右項	0	0	0
左項＝右項	1	0	0
左項＜右項	0	1	1

___fcmp.s

float 型の比較演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	比較演算の左項
r6	比較演算の右項

[戻り値]

r6	比較結果を表す int 型の値

[詳細説明]

float 型の比較演算を行います。

以下の組み合わせの結果を返します。

	戻り値
左項，または右項が非数	1
左項＞右項	1
左項＝右項	0
左項＜右項	-1

___negf.s

float 型の符号反転を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	符号反転する値

[戻り値]

r6	符号反転した値

[詳細説明]

float 型の符号反転を行います。

___notf.s

float 型の論理否定を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	論理否定する値

[戻り値]

r6	論理否定した整数値

[詳細説明]

float 型の論理否定を行います。

戻り値は int 型です。

___addf.d

double 型の加算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	加算の左項
r7:r6	加算の右項

[戻り値]

r7:r6	加算の結果

[詳細説明]

double 型の加算を行います。

___subf.d

double 型の減算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	減算の左項
r7:r6	減算の右項

[戻り値]

r7:r6	減算の結果

[詳細説明]

double 型の減算を行います。

___mulf.d

double 型の乗算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	乗算の左項
r7:r6	乗算の右項

[戻り値]

r7:r6	乗算の結果

[詳細説明]

double 型の乗算を行います。

___divf.d

double 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	除算の左項
r7:r6	除算の右項

[戻り値]

r7:r6	除算の結果

[詳細説明]

double 型の除算を行います。

___fcmp.d

double 型の比較演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	比較演算の左項
r7:r6	比較演算の右項

[戻り値]

r6	比較結果を表す int 型の値

[詳細説明]

double 型の比較演算を行います。

以下の組み合わせの結果を返します。

	戻り値
左項，または右項が非数	1
左項＞右項	1
左項＝右項	0
左項＜右項	-1

___negf.d

double 型の符号反転を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	符号反転する値

[戻り値]

r7:r6	符号反転した値

[詳細説明]

double 型の符号反転を行います。

___notf.d

double 型の論理否定を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	論理否定する値

[戻り値]

r6	論理否定した整数値

[詳細説明]

double 型の論理否定を行います。

戻り値は int 型です。

___add.l

long long 型の加算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	加算の左項
r7:r6	加算の右項

[戻り値]

r7:r6	加算の結果

[詳細説明]

long long 型の加算を行います。

___sub.l

long long 型の減算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	減算の左項
r7:r6	減算の右項

[戻り値]

r7:r6	減算の結果

[詳細説明]

long long 型の減算を行います。

___mul.l

long long 型の乗算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	乗算の左項
r7:r6	乗算の右項

[戻り値]

r7:r6	乗算の結果

[詳細説明]

long long 型の乗算を行います。

___div.l

long long 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	除算の左項
r7:r6	除算の右項

[戻り値]

r7:r6	除算の結果

[詳細説明]

long long 型の除算を行います。

除算の除数が 0 の場合は，結果に 0 を返します。

___div.ul

unsigned long long 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	除算の左項
r7:r6	除算の右項

[戻り値]

r7:r6	除算の結果

[詳細説明]

unsigned long long 型の除算を行います。

除算の除数が 0 の場合は，結果に 0 を返します。

___mod.l

long long 型の剰余算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	剰余算の左項
r7:r6	剰余算の右項

[戻り値]

r7:r6	剰余算の結果

[詳細説明]

long long 型の剰余算を行います。

除算の除数が 0 の場合は，結果に 0 を返します。

___mod.ul

unsigned long long 型の剰余算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	剰余算の左項
r7:r6	剰余算の右項

[戻り値]

r7:r6	剰余算の結果

[詳細説明]

unsigned long long 型の剰余算を行います。

除算の除数が 0 の場合は，結果に 0 を返します。

___shl.l

long long 型の左シフト演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	シフト演算の左項
r8	シフト演算の右項

[戻り値]

r7:r6	シフト演算の結果

[詳細説明]

long long 型の論理左シフト演算を行います。

右項は符号の有無によらず 0x3F でマスクしてから演算を行います。

___shr.l

unsigned long long 型の右シフト演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	シフト演算の左項
r8	シフト演算の右項

[戻り値]

r7:r6	シフト演算の結果

[詳細説明]

unsigned long long 型の論理右シフト演算を行います。

右項は符号の有無によらず 0x3F でマスクしてから演算を行います。

___sar.l

long long 型の右シフト演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	シフト演算の左項
r8	シフト演算の右項

[戻り値]

r7:r6	シフト演算の結果

[詳細説明]

long long 型の算術右シフト演算を行います。

右項は符号の有無によらず 0x3F でマスクしてから演算を行います。

___inc.l

long long 型の増分演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	増分演算する値

[戻り値]

r7:r6	増分演算の結果

[詳細説明]

long long 型の増分演算を行います。

___dec.l

long long 型の減分演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	減分演算する値

[戻り値]

r7:r6	減分演算の結果

[詳細説明]

long long 型の減分演算を行います。

___not.l

long long 型の論理否定を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	論理否定する値

[戻り値]

r7:r6	論理否定の結果

[詳細説明]

long long 型の論理否定を行います。

___neg.l

long long 型の符号反転を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	符号反転する値

[戻り値]

r7:r6	符号反転の結果

[詳細説明]

long long 型の符号反転を行います。

___cmp.l

long long 型の比較演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	比較演算の左項
r7:r6	比較演算の右項

[戻り値]

r6	比較結果を表す int 型の値

[詳細説明]

long long 型の比較演算を行います。

以下の組み合わせの結果を返します。

	戻り値
左項＞右項	1
左項＝右項	0
左項＜右項	-1

___cmp.ul

unsigned long long 型の比較演算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r9:r8	比較演算の左項
r7:r6	比較演算の右項

[戻り値]

r6	比較結果を表す int 型の値

[詳細説明]

unsigned long long 型の比較演算を行います。

以下の組み合わせの結果を返します。

	戻り値
左項＞右項	1
左項＝右項	0
左項＜右項	-1

___bext.l

long long 型の符号付きビットフィールド抽出を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	ビットフィールド抽出元
r8	上位 16 ビット：ビットフィールド抽出幅 下位 16 ビット：ビットフィールド抽出位置

[戻り値]

r7:r6	抽出したビットフィールド値

[詳細説明]

long long 型の符号付きビットフィールド抽出を行います。

r8 の下位 16 ビットの値を 0x3F でマスクした値を抽出する最下位ビット位置とします。

r8 の上位 16 ビットの値を 0xFFFF でマスクした値を抽出するビット幅とします。ただし，抽出ビット位置と合わせて 64 ビットを越える場合は 64 ビット幅に収まるまで抽出ビット幅を縮小します。

抽出したビットフィールド値の最上位ビットを符号ビットとして，符号拡張した long long 型として返します。

抽出ビット幅が 0 の場合は 0 を返します。

___bext.ul

long long 型の符号なしビットフィールド抽出を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	ビットフィールド抽出元
r8	上位 16 ビット：ビットフィールド抽出幅 下位 16 ビット：ビットフィールド抽出位置

[戻り値]

r7:r6	抽出したビットフィールド値

[詳細説明]

long long 型の符号なしビットフィールド抽出を行います。

r8 の下位 16 ビットの値を 0x3F でマスクした値を抽出する最下位ビット位置とします。

r8 の上位 16 ビットの値を 0xFFFF でマスクした値を抽出するビット幅とします。ただし，抽出ビット位置と合わせて 64 ビットを越える場合は 64 ビット幅に収まるまで抽出ビット幅を縮小します。

抽出したビットフィールド値をゼロ拡張して返します。

抽出ビット幅が 0 の場合は 0 を返します。

___bins.l

long long 型のビットフィールド挿入を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	ビットフィールド挿入先
r9:r8	ビットフィールド挿入データ
0[sp]	上位 16 ビット：ビットフィールド挿入幅 下位 16 ビット：ビットフィールド挿入位置

[戻り値]

r7:r6	挿入後の long long 値

[詳細説明]

long long 型のビットフィールド挿入を行います。

0[sp] に格納されている値の下位 16 ビットを 0x3F でマスクした値をビットフィールド挿入位置とします。

0[sp] に格納されている値の上位 16 ビットを 0xFFFF でマスクした値をビットフィールド挿入幅とします。ただし，挿入位置と合わせて 64 ビットを越える場合は，64 ビット幅に収まるまで挿入幅を縮小します。

挿入ビット幅が 0 の場合は r7:r6 の値をそのまま返します。

___cvt.ws

long 型から float 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

long 型から float 型への変換を行います。

___cvt.wd

long 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

long 型から double 型への変換を行います。

___cvt.uws

unsigned long 型から float 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

unsigned long 型から float 型への変換を行います。

___cvt.uwd

unsigned long 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

unsigned long 型から double 型への変換を行います。

___cvt.ls

long long 型から float 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

long long 型から float 型への変換を行います。

___cvt.ld

long long 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

long long 型から double 型への変換を行います。

___cvt.uls

unsigned long long 型から float 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

unsigned long long 型から float 型への変換を行います。

___cvt.uld

unsigned long long 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

unsigned long long 型から double 型への変換を行います。

___trnc.sw

float 型から long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

float 型から long 型への変換を行います。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x80000000 より小さい	0
+0xFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.dw

double 型から long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

double 型から long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x80000000 より小さい	0
+0xFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.suw

float 型から unsigned long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

float 型から unsigned long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x80000000 より小さい	0
+0xFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.duw

double 型から unsigned long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

double 型から unsigned long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x80000000 より小さい	0
+0xFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.sl

float 型から long long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

float 型から long long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x8000000000000000 より小さい	0
+0xFFFFFFFFFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.dl

double 型から long long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

double 型から long long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x8000000000000000 より小さい	0
+0xFFFFFFFFFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.sul

float 型から unsigned long long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

float 型から unsigned long long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x8000000000000000 より小さい	0
+0xFFFFFFFFFFFFFFFF より大きい	0
その他	変換後の整数値

___trnc.dul

double 型から unsigned long long 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

double 型から unsigned long long 型への変換を行います。

小数部分は 0 方向へ丸めます。

以下の組み合わせの結果を返します。

	戻り値
非数，または±∞	0
-0x8000000000000000 より小さい	0
+0xFFFFFFFFFFFFFFFF より大きい	0
その他	変換後の整数値

___cvt.sd

float 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r6	変換前の値

[戻り値]

r7:r6	変換後の値

[詳細説明]

float 型から double 型への変換を行います。

以下の組み合わせの結果を返します。

	戻り値
非数	非数
±∞	±∞
その他	変換後の値

___cvt.ds

double 型から double 型への変換を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7:r6	変換前の値

[戻り値]

r6	変換後の値

[詳細説明]

double 型から double 型への変換を行います。

以下の組み合わせの結果を返します。

	戻り値
非数	非数
±∞	±∞
その他	変換後の値

___mul

int 型の乗算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	乗算の左項
r6	乗算の右項

[戻り値]

r6	乗算の結果

[詳細説明]

int 型の乗算を行います。

___mulu

unsigned int 型の乗算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	乗算の左項
r6	乗算の右項

[戻り値]

r6	乗算の結果

[詳細説明]

unsigned int 型の乗算を行います。

___div

int 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	除算の左項
r6	除算の右項

[戻り値]

r6	除算の結果

[詳細説明]

int 型の除算を行います。

ゼロ除算時は 0 が返ります。

___divu

unsigned int 型の除算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	除算の左項
r6	除算の右項

[戻り値]

r6	除算の結果

[詳細説明]

unsigned int 型の除算を行います。

ゼロ除算時は 0 が返ります。

___mod

int 型の剰余算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	剰余算の左項
r6	剰余算の右項

[戻り値]

r6	剰余算の結果

[詳細説明]

int 型の剰余算を行います。

ゼロ除算時は 0 が返ります。

___modu

unsigned int 型の剰余算を行います。

[所属]

ランタイム・ライブラリ

[引数]

r7	剰余算の左項
r6	剰余算の右項

[戻り値]

r6	剰余算の結果

[詳細説明]

unsigned int 型の剰余算を行います。

ゼロ除算時は 0 が返ります。

6. 4. 14 関数前後処理ランタイム関数

関数前後処理ランタイム関数は，関数のプロローグ／エピローグ処理で CX が自動的に呼び出すルーチンです。関数前後処理ランタイム関数は，“演算用ランタイム関数”と同様に，C ソースやアセンブラ・ソースで記述する関数ではありません。

V850Ex コアの場合，関数前後処理ランタイム関数の呼び出しに CALLT 命令を使用します。これらの関数を CALLT 命令によるテーブル呼び出しにすることにより，コード効率を上げます。

関数前後処理ランタイム関数呼び出しを有効にするためには，次のようになります。

- 最適化オプション“-Ospeed（実行速度優先最適化）”指定時以外のとき
- コンパイラ・オプション“-Xpro_epi_runtime=on”を指定したとき

表 6―31 関数前後処理ランタイム関数

関数／マクロ名	概要
___Epush250，___Epush251，___Epush252， ___Epush253，___Epush254，___Epush260， ___Epush261，___Epush262，___Epush263， ___Epush264，___Epush270，___Epush271， ___Epush272，___Epush273，___Epush274， ___Epush280，___Epush281，___Epush282， ___Epush283，___Epush284，___Epush290， ___Epush291，___Epush292，___Epush293， ___Epush294，___Epushlp0，___Epushlp1， ___Epushlp2，___Epushlp3，___Epushlp4	関数のプロローグ処理

関数／マクロ名	概要
___push2000, ___push2001, ___push2002, ___push2003, ___push2004, ___push2040, ___push2100, ___push2101, ___push2102, ___push2103, ___push2104, ___push2140, ___push2200, ___push2201, ___push2202, ___push2203, ___push2204, ___push2240, ___push2300, ___push2301, ___push2302, ___push2303, ___push2304, ___push2340, ___push2400, ___push2401, ___push2402, ___push2403, ___push2404, ___push2440, ___push2500, ___push2501, ___push2502, ___push2503, ___push2504, ___push2540, ___push2600, ___push2601, ___push2602, ___push2603, ___push2604, ___push2640, ___push2700, ___push2701, ___push2702, ___push2703, ___push2704, ___push2740, ___push2800, ___push2801, ___push2802, ___push2803, ___push2804, ___push2840, ___push2900, ___push2901, ___push2902, ___push2903, ___push2904, ___push2940, ___pushlp00, ___pushlp01, ___pushlp02, ___pushlp03, ___pushlp04, ___pushlp40	関数のプロローグ処理
___Epop250, ___Epop251, ___Epop252, ___Epop253, ___Epop254, ___Epop260, ___Epop261, ___Epop262, ___Epop263, ___Epop264, ___Epop270, ___Epop271, ___Epop272, ___Epop273, ___Epop274, ___Epop280, ___Epop281, ___Epop282, ___Epop283, ___Epop284, ___Epop290, ___Epop291, ___Epop292, ___Epop293, ___Epop294, ___Epoplp0, ___Epoplp1, ___Epoplp2, ___Epoplp3, ___Epoplp4	関数のエピローグ処理

関数／マクロ名	概要
___pop2000, ___pop2001, ___pop2002, ___pop2003, ___pop2004, ___pop2040, ___pop2100, ___pop2101, ___pop2102, ___pop2103, ___pop2104, ___pop2140, ___pop2200, ___pop2201, ___pop2202, ___pop2203, ___pop2204, ___pop2240, ___pop2300, ___pop2301, ___pop2302, ___pop2303, ___pop2304, ___pop2340, ___pop2400, ___pop2401, ___pop2402, ___pop2403, ___pop2404, ___pop2440, ___pop2500, ___pop2501, ___pop2502, ___pop2503, ___pop2504, ___pop2540, ___pop2600, ___pop2601, ___pop2602, ___pop2603, ___pop2604, ___pop2640, ___pop2700, ___pop2701, ___pop2702, ___pop2703, ___pop2704, ___pop2740, ___pop2800, ___pop2801, ___pop2802, ___pop2803, ___pop2804, ___pop2840, ___pop2900, ___pop2901, ___pop2902, ___pop2903, ___pop2904, ___pop2940, ___poplp00, ___poplp01, ___poplp02, ___poplp03, ___poplp04, ___poplp40	関数のエピローグ処理

6.5 ライブラリ消費スタック一覧

この節では，ライブラリに含まれている各種関数のスタック消費量について説明します。

6.5.1 標準ライブラリ

以下に，標準ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) 可変個引数関数

表 6―32 可変個引数関数

関数／マクロ名	スタック消費量
va_start	0
va_end	0
va_arg	0

(2) 文字列関数

表 6―33 文字列関数

関数／マクロ名	スタック消費量
memchr	0
memcmp	0
bcmp	0
memcpy	0
bcopy	0
memmove	0
memset	0

(3) メモリ管理関数

表 6―34 メモリ管理関数

関数／マクロ名	スタック消費量
index	0
strpbrk	0
rindex	0
strrchr	0
strchr	0

関数／マクロ名	スタック消費量
strstr	0
strspn	0
strcspn	0
strcmp	0
strncmp	0
strcpy	0
strncpy	0
strcat	0
strncat	0
strtok	0
strlen	0
strerror	0

(4) 文字変換関数

表 6―35 文字変換関数

関数／マクロ名	スタック消費量
toupper	0
_toupper	0
tolower	0
_tolower	0
toascii	0

(5) 文字分類関数

表 6―36 文字分類関数

関数／マクロ名	スタック消費量
isalnum	0
isalpha	0
isascii	0
isupper	0
islower	0
isdigit	0
isxdigit	0
iscntrl	0

関数／マクロ名	スタック消費量
ispunct	0
isspace	0
isprint	0
isgraph	0

（6）標準入出力関数

表 6―37 標準入出力関数

関数／マクロ名	スタック消費量
fread	40
getc	0
fgetc	0
fgets	0
fwrite	28
putc	0
fputc	0
fputs	0
getchar	0
gets	0
putchar	0
puts	0
sprintf	284
fprintf	276
vsprintf	268
printf	276
vfprintf	256
vprintf	264
sscanf	244
fscanf	236
scanf	236
ungetc	0
rewind	0
perror	288

(7) 標準ユーティリティ関数

表 6―38 標準ユーティリティ関数

関数／マクロ名	スタック消費量
abs	0
labs	0
llabs	0
bsearch	40
qsort	76
div	0
ldiv	0
lldiv	36
itoa	36
ltoa	36
ultoa	44
lltoa	88
ulltoa	76
ecvt	168
ecvtf	124
fcvt	168
fcvtf	124
gcvt	232
gcvtf	232
atoi	64
atol	64
atoll	72
strtol	72
strtoul	64
strtoll	72
strtoull	72
atoff	140
atof	140
strtodf	140
strtod	140
calloc	20
malloc	12
realloc	24
free	16

関数／マクロ名	スタック消費量
rand	0
srand	0

（8）非局所分岐関数

表 6―39 非局所分岐関数

関数／マクロ名	スタック消費量
longjmp	0
setjmp	0

6. 5. 2 数学ライブラリ

以下に，数学ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

（1）数学関数

表 6―40 数学関数

関数／マクロ名	スタック消費量
j0f	68
j1f	68
jnf	88
y0f	80
y1f	80
ynf	100
erff	64
erfcf	64
expf	40
exp	56
logf	44
log	68
log2f	44
log10f	44
log10	68
powf	52
pow	88
sqrtf	72

関数／マクロ名	スタック消費量
sqrt	52
cbrtf	44
cbrt	64
ceilf	0
ceil	0
fabsf	0
fabs	0
floorf	0
floor	4
fmodf	32
fmod	56
frexpf	32
frexp	44
ldexpf	32
ldexp	44
modff	0
modf	0
gammaf	52
hypotf	44
matherrf (matherr)	0
matherrd	0
cosf	40
cos	64
sinf	40
sin	68
tanf	52
tan	80
acosf	52
acos	68
asinf	52
asin	68
atanf	48
atan	68
atan2f	52
atan2	80
coshf	40
cosh	60

関数／マクロ名	スタック消費量
sinhf	40
sinh	60
tanhf	48
tanh	60
acoshf	44
asinhf	44
atanhf	44

6. 5. 3　初期化処理ライブラリ

以下に，初期化処理ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) 周辺装置の初期化関数

表 6―41　初期化処理ライブラリ

関数／マクロ名	スタック消費量
hdwinit	0

6. 5. 4　ROM 化用ライブラリ

以下に，ROM 化用ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) コピー関数

表 6―42　コピー関数

関数／マクロ名	スタック消費量
_rcopy	24
_rcopy1	24
_rcopy2	20
_rcopy4	20

6. 5. 5 マルチコア用ライブラリ

以下に，マルチコア用ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

（1）マルチコア用擬似 main 関数

表 6―43 マルチコア用擬似 main 関数

関数／マクロ名	スタック消費量
main_pe2	0
main_pe3	0
main_pe4	0
main_pe5	0
main_pe6	0
main_pe7	0
main_pe8	0
main_pe9	0
main_pe10	0
main_pe11	0
main_pe12	0
main_pe13	0
main_pe14	0
main_pe15	0
main_pe16	0
main_pe17	0
main_pe18	0
main_pe19	0
main_pe20	0
main_pe21	0
main_pe22	0
main_pe23	0
main_pe24	0
main_pe25	0
main_pe26	0
main_pe27	0
main_pe28	0
main_pe29	0
main_pe30	0
main_pe31	0

6. 5. 6 ランタイム・ライブラリ

以下に，ランタイム・ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) 演算用ランタイム関数

表 6―44 演算用ランタイム関数

関数／マクロ名	スタック消費量
___addf.s	84
___subf.s	84
___mulf.s	84
___divf.s	84
___cmpf.s	84
___fcmp.s	84
___negf.s	92
___notf.s	92
___addf.d	96
___subf.d	96
___mulf.d	96
___divf.d	140
___fcmp.d	72
___negf.d	108
___notf.d	84
___add.l	0
___sub.l	0
___mul.l	0
___div.l	32
___div.ul	16
___mod.l	40
___mod.ul	20
___shl.l	4
___shr.l	4
___sar.l	4
___inc.l	0
___dec.l	0
___not.l	0
___neg.l	0
___cmp.l	4
___cmp.ul	0

関数／マクロ名	スタック消費量
___bext.l	8
___bext.ul	8
___bins.l	12
___cvt.ws	16
___cvt.wd	8
___cvt.uws	16
___cvt.uwd	8
___cvt.ls	20
___cvt.ld	24
___cvt.uls	8
___cvt.uld	12
___trnc.sw	4
___trnc.dw	4
___trnc.suw	4
___trnc.duw	4
___trnc.sl	12
___trnc.dl	12
___trnc.sul	12
___trnc.dul	12
___cvt.sd	4
___cvt.ds	12
___mul	12
___mulu	12
___div	20
___divu	16
___mod	20
___modu	16

(2) 関数前後処理ランタイム関数

表 6―45 関数前後処理ランタイム関数

関数／マクロ名	スタック消費量
___Epush250	0
___Epush251	0
___Epush252	0
___Epush253	0

関数／マクロ名	スタック消費量
___Epush254	0
___Epush260	0
___Epush261	0
___Epush262	0
___Epush263	0
___Epush264	0
___Epush270	0
___Epush271	0
___Epush272	0
___Epush273	0
___Epush274	0
___Epush280	0
___Epush281	0
___Epush282	0
___Epush283	0
___Epush284	0
___Epush290	0
___Epush291	0
___Epush292	0
___Epush293	0
___Epush294	0
___Epushlp0	0
___Epushlp1	0
___Epushlp2	0
___Epushlp3	0
___Epushlp4	0
___Epop250	0
___Epop251	0
___Epop252	0
___Epop253	0
___Epop254	0
___Epop260	0
___Epop261	0
___Epop262	0
___Epop263	0
___Epop264	0
___Epop270	0

関数／マクロ名	スタック消費量
___Epop271	0
___Epop272	0
___Epop273	0
___Epop274	0
___Epop280	0
___Epop281	0
___Epop282	0
___Epop283	0
___Epop284	0
___Epop290	0
___Epop291	0
___Epop292	0
___Epop293	0
___Epop294	0
___Epoplp0	0
___Epoplp1	0
___Epoplp2	0
___Epoplp3	0
___Epoplp4	0
___push2000	0
___push2001	0
___push2002	0
___push2003	0
___push2004	0
___push2040	0
___push2100	0
___push2101	0
___push2102	0
___push2103	0
___push2104	0
___push2140	0
___push2200	0
___push2201	0
___push2202	0
___push2203	0
___push2204	0
___push2240	0

関数／マクロ名	スタック消費量
___push2300	0
___push2301	0
___push2302	0
___push2303	0
___push2304	0
___push2340	0
___push2400	0
___push2401	0
___push2402	0
___push2403	0
___push2404	0
___push2440	0
___push2500	0
___push2501	0
___push2502	0
___push2503	0
___push2504	0
___push2540	0
___push2600	0
___push2601	0
___push2602	0
___push2603	0
___push2604	0
___push2640	0
___push2700	0
___push2701	0
___push2702	0
___push2703	0
___push2704	0
___push2740	0
___push2800	0
___push2801	0
___push2802	0
___push2803	0
___push2804	0
___push2840	0
___push2900	0

関数／マクロ名	スタック消費量
___push2901	0
___push2902	0
___push2903	0
___push2904	0
___push2940	0
___pushlp00	0
___pushlp01	0
___pushlp02	0
___pushlp03	0
___pushlp04	0
___pushlp40	0
___pop2000	0
___pop2001	0
___pop2002	0
___pop2003	0
___pop2004	0
___pop2040	0
___pop2100	0
___pop2101	0
___pop2102	0
___pop2103	0
___pop2104	0
___pop2140	0
___pop2200	0
___pop2201	0
___pop2202	0
___pop2203	0
___pop2204	0
___pop2240	0
___pop2300	0
___pop2301	0
___pop2302	0
___pop2303	0
___pop2304	0
___pop2340	0
___pop2400	0
___pop2401	0

関数／マクロ名	スタック消費量
___pop2402	0
___pop2403	0
___pop2404	0
___pop2440	0
___pop2500	0
___pop2501	0
___pop2502	0
___pop2503	0
___pop2504	0
___pop2540	0
___pop2600	0
___pop2601	0
___pop2602	0
___pop2603	0
___pop2604	0
___pop2640	0
___pop2700	0
___pop2701	0
___pop2702	0
___pop2703	0
___pop2704	0
___pop2740	0
___pop2800	0
___pop2801	0
___pop2802	0
___pop2803	0
___pop2804	0
___pop2840	0
___pop2900	0
___pop2901	0
___pop2902	0
___pop2903	0
___pop2904	0
___pop2940	0
___poplp00	0
___poplp01	0
___poplp02	0

関数／マクロ名	スタック消費量
___poplp03	0
___poplp04	0
___poplp40	0

6. 5. 7 V850E2V3-FPU 使用ライブラリ

以下に，V850E2V3-FPU 使用ライブラリに含まれている各種関数のスタック消費量（単位：バイト）を示します。

(1) V850E2V3-FPU 使用関数

表 6―46 V850E2V3-FPU 使用関数

関数／マクロ名	スタック消費量
expf	212
exp	272
logf	140
log	204
log10f	120
log10	204
powf	180
pow	356
sqrtf	24
sqrt	36
ceilf	20
ceil	36
floorf	20
floor	36
fmodf	100
fmod	176
frexpf	56
frexp	84
ldexpf	136
ldexp	136
modff	36
modf	68
cosf	220
cos	352

関数／マクロ名	スタック消費量
sinf	220
sin	352
tanf	84
tan	176
acosf	80
acos	472
asinf	72
asin	384
atanf	108
atan	288
atan2f	160
atan2	360
coshf	268
cosh	352
sinhf	268
sinh	352
tanhf	272
tanh	380

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。

備考　マルチコア・プログラミングでは，マルチコア・プログラム用のスタートアップ・ルーチンが必要です。

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　スタートアップ・ルーチンのサンプル

格納場所	ファイル名	内容
Version Folder \ lib \ 850e	cstart.asm	V850Ex コア用スタートアップ・ルーチンのサンプル
	cstartN.asm	ROM 化をしない場合用 V850Ex コア用スタートアップ・ルーチンのサンプル
	cstartM.asm	マルチコア用スタートアップ・ルーチンのサンプル
	cstartMN.asm	ROM 化をしない場合用 マルチコア用スタートアップ・ルーチンのサンプル

スタートアップ・ルーチンを新たに作成する場合には，上記のサンプルをコピーして，プロジェクトに追加後，編集してください。

スタートアップ・ルーチンをプロジェクトに追加しなかった場合，CX はデフォルトのスタートアップ・ルーチン（オブジェクト）を自動的にリンクします。リンクされるファイルは，サンプルのスタートアップ・ルーチン“cstart.asm”，および“cstartN.asm”をコンパイル（アセンブル）したファイルです。

また，これらのオブジェクトは，アセンブラ・オプション“-Xcommon=v850e”を指定してアセンブルしており，V850 マイクロコントローラで共通に使用できるようにしたオブジェクトになっています。

7.3　スタートアップ・ルーチン

スタートアップ・ルーチンとは，V850 をリセットしたあと，main 関数を実行する前に，実行するルーチンを言います。基本的にはシステムをリセットしたあとの初期化を行います。具体的には，次のことを行います。

- リセットが入ったときの RESET ハンドラの設定
- スタートアップ・ルーチンのレジスタ・モード設定
- スタック領域の確保とスタック・ポインタの設定
- main 関数の引数領域の確保
- テキスト・ポインタ（tp）の設定
- グローバル・ポインタ（gp）の設定
- エレメント・ポインタ（ep）の設定
- main 関数実行前に行う必要のある周辺 I/O レジスタの初期化
- main 関数実行前に行う必要のあるユーザ・ターゲットの初期化
- sbss 領域のゼロクリア
- bss 領域のゼロクリア
- sebss 領域のゼロクリア
- tibss.byte 領域のゼロクリア
- tibss.word 領域のゼロクリア
- sibss 領域のゼロクリア
- 関数前後処理ランタイム関数用の CTBP 値の設定
- プログラマブル周辺 I/O レジスタ値の設定
- r6 と r7 を main 関数の引数に設定
- main 関数へ分岐する（リアルタイム OS を使用していない場合）
- リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）
- V850E2V3 マルチコア用スタートアップ・ルーチン

もちろん，システムによっては必要のない処理もありますので，それらに関しては省略できます。
また，これ以外にもユーザで行っておきたい処理があった場合は記述しておきます。
7.3.1 以降に記載している記述例は，様々なケースを想定して説明しています。
そのため，CubeSuite が提供しているスタートアップ・ルーチンの記述と異なる可能性があります。
なお，これらの処理は，基本的にアセンブラ命令で記述する必要があります。

7.3.1　リセットが入ったときの RESET ハンドラの設定

リセット（リセット割り込み）が入ったときの処理を記述します。V850 では，リセットが入ると 0x0 番地のハンドラ・アドレスに分岐します。そこで，0x0 番地にスタートアップ・ルーチンの先頭へ分岐する命令を配置します。リセット割り込みは C 言語上で #pragma interrupt 指定による記述ができないので，アセンブラ命令で記述します。記述は次のようになります。

```
RESET   .cseg   TEXT
        jr      __start
__start:
```

ハンドラ・アドレスへの配置には，.cseg 疑似命令を使用します。上記のように記述することによって RESET のハンドラ・アドレスに“jr __start”という命令が配置されます。

また，jr 命令で届かない場所，つまり，0x0 番地から± 2M バイト内に“__start”がなかった場合は，次のように jmp 命令を使用します。

```
RESET   .cseg   TEXT
    mov     #__start, lp
    jmp     [lp]
__start:
```

この場合，レジスタを 1 つ使用します。上記の例では lp（r31）レジスタを使用していますが，この時点で破壊してもよい汎用レジスタがあれば使用可能です。リセット時点では，関数からの戻りアドレスが入る lp（r31）レジスタを使用することはないので，lp（r31）レジスタの使用が安全です。

なお，これらの .cseg 疑似命令を記述は，特にスタートアップ・ルーチン内ではなくても問題ありません。

また，例ではスタートアップ・ルーチンのシンボルを“__start”としていますが，これも別の名前で問題ありません。

7.3.2　スタートアップ・ルーチンのレジスタ・モード設定

アセンブラ命令で記述するスタートアップ・ルーチンに，レジスタ・モードの設定する記述をします。

ただし，この設定をする必要があるのは，システム全体で 22 レジスタ・モード，26 レジスタ・モードを指定している場合です。32 レジスタ・モードを指定している場合は記述する必要がありません。

【22 レジスタ・モード時】

```
$ REG_MODE 22
```

【26 レジスタ・モード時】

```
$ REG_MODE 26
```

【共通レジスタ・モード時】

```
$ REG_MODE common
```

この設定をしていなかった場合，リンク時に次の警告メッセージが出力されます。

```
W0565308: input files have different register modes.
        use "-Xregmode_info" option for more information.
```

7.3.3　スタック領域の確保とスタック・ポインタの設定

システムで使用するスタック領域を確保し，その領域の末尾にスタック・ポインタ（SP=r3）を設定します。ただし，リアルタイム OS を使用している場合，ここで指定するスタックは，リアルタイム OS の初期化ルーチンに分岐するまでに使用するスタックとなります。

したがって，ほとんど使用しない，またはまったく使用しないことが多いので，ここで多く確保してしまうと RAM 領域が無駄になります。リアルタイム OS の初期化ルーチンに分岐するまでにスタックを使用しているかどうかを確認してください。特に割り込みには注意が必要ですが，スタートアップ・ルーチン内は割り込み禁止で実行することが通例です。

スタック領域の確保の方法は次のようになります。

```
STACKSIZE   .set    0x200
            .dseg   BSS
mov     #__stack + STACKSIZE, sp
```

上記は，確保するスタック・サイズは 0x200 バイトで，.bss 領域に確保する例です。スタックの内容は，初期値を持たないので“bss 属性の領域”に配置します。もちろん sbss 領域に配置することも可能ですが，sbss 領域は gp 相対 1 命令でアクセスできる領域のために，配置できるサイズに限界があります。他の変数等を配置した方がよいこともあるので，スタック・サイズが大きくなる場合は，bss 領域に配置することを推奨しています。

確保するスタック・サイズを変更するときは，.set 命令に書かれている数値を変更します。また，CX は，スタック・ポインタ（sp）相対でメモリを参照する場合，sp が 4 バイト境界に位置していることを前提としたコードを出力しています。そのため，スタック・ポインタは必ず“4 バイト境界”に配置するようにしてください。必要ならば疑似命令“.align 4”を使用し，.set 命令に指定する数値を 4 の倍数にしてください。

スタックはシステムの動作に大きく関わってきます。スタックが不足すると，確保した領域を越えて破壊するため，システムの暴走につながります。確保すべきスタック・サイズは，CX にパッケージされているスタック見積もりツールなどを用いて，関数で使用するスタック・サイズを見積もり，十分なサイズを確保してください。

7. 3. 4　main 関数の引数領域の確保

ANSI C 仕様では，main 関数の形式は，仮引数を持たない“int main (void) { ... }”として定義されるか，2 つの仮引数をもつ関数“int main (int argc, char *argv[ ]) { ... }”として定義されます。

ここで 2 つの仮引数を持つ関数の場合，argc は非負の値であり，仮引数の総計を示します。argv は引数文字列へのポインタの配列を示します。argv[argc] は NULL（空ポインタ）で，argc が 1 以上ならば argv[0] ～ argv[argc - 1] は文字列へのポインタになります。

この argc と argv の領域をスタートアップ・ルーチン内で確保します。確保の方法は次のとおりです。

```
        .dseg   DATA
        .align  4
__argc:
        .db4    0
__argv:
        .db4    #.L16
.L16:
        .db     0
        .db     0
        .db     0
        .db     0
```

この領域は初期値定義しておくため，“data 属性領域”に配置します。

“int main (void) { ... }”の形で main 関数を定義する場合は，上記の領域は不要です。

削除することにより，上記の分 RAM 領域を削減することができます。

なお，実際に main 関数の引数（r6 と r7）へ設定する処理は main 関数の直前で行います。r6 と r7 をスタートアップ・ルーチン内で使用しないのであれば，上記のプログラム直後に行っても問題ありません。設定する処理は「7. 3. 18　r6 と r7 を main 関数の引数に設定」を参照してください。

7. 3. 5　テキスト・ポインタ（tp）の設定

アプリケーションのテキスト領域であるプログラム・コードを参照する際に，配置される位置に依存しない参照（PIC：Position Independent Code）を実現するために用意されているポインタが“テキスト・ポインタ（tp）”です。たとえば，プログラム実行中に，コード内のある箇所を参照する必要がある場合，CX は tp 相対でアクセスするコードを出力します。

したがって，tp が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で tp を正しく設定する必要があります。

テキスト・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，テキスト・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__tp_TEXT@%TP_SYMBOL {TEXT};
```

このとき，テキスト・ポインタの値は“TEXT セグメント”の先頭になり，その値は“__tp_TEXT”に入ります。

スタートアップ・ルーチン内で tp をセットするには，次のように記述してください。

```
.extern __tp_TEXT, 4
mov     #__tp_TEXT, tp
```

7.3.6　グローバル・ポインタ（gp）の設定

アプリケーション内で定義した外部変数／データはメモリ上に配置されます。そのメモリに配置されている変数／データを参照する際，配置位置に依存することのない参照（PID：Position Independent Data）を実現するために用意されているポインタが“グローバル・ポインタ（gp）”です。gp 相対でアクセスするセクションが存在する場合，CX は gp 相対でアクセスするコードを出力します。

したがって，gp が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で gp を正しく設定する必要があります。

グローバル・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，グローバル・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__gp_DATA@%GP_SYMBOL {DATA};
```

また，gp シンボル値は，上記のように“DATA セグメントなどの「データ用セグメント」の先頭を gp シンボル値とする方法”のほかに，“テキスト・シンボルからのオフセットを gp シンボル値とする”方法もあります。

この方法の場合，gp シンボルを「tp の値と，tp からのオフセット値を加える」ことによって決定できます。つまり，配置に依存しないコードの生成が可能になります。たとえば，“プログラム・コード”と“そのコードが使用するデータ”を同時 RAM 領域にコピーしてから実行させたい場合，コードの先頭（コピー先の先頭アドレス）さえ分かれば，gp の値もすぐ導き出せるというメリットがあります。この場合のシンボル・ディレクティブ記述は次のようになります。

```
__tp_TEXT@%TP_SYMBOL {TEXT};
__gp_DATA@%GP_SYMBOL &__tp_TEXT {DATA};
```

グローバル・ポインタの値は，“__tp_TEXT に __gp_DATA の値を加えた値”となり，加える値（オフセット値）が“__gp_DATA”に入ります。したがって，スタートアップ・ルーチン内で gp をセットするには，次のように記述します。

```
.extern __tp_TEXT, 4
.extern __gp_DATA, 4
mov     #__tp_TEXT, tp
mov     #__gp_DATA, gp
add     tp, gp
```

これにより，正しいグローバル・ポインタの値が gp に設定されます。

7. 3. 7　エレメント・ポインタ（ep）の設定

アプリケーション内で定義した外部変数／データのうち，次に割り当てられているものは，エレメント・ポインタ（ep）からの相対でアクセスされます。

- sedata/sebss セクション
- sidata/sibss セクション
- tidata.byte/tibss.byte セクション
- tidata.word/tibss.word セクション

これらのセクションが存在する場合，CX は ep 相対でアクセスするコードを出力します。

したがって，ep が正しく設定されていることを前提としたコードを出力しているので，スタートアップ・ルーチン内で ep を正しく設定する必要があります。

エレメント・ポインタの値は，リンク時に決定され，リンク・ディレクティブ・ファイル内に書かれる“シンボル・ディレクティブ”に定義されたシンボルに入っています。たとえば，エレメント・ポインタのシンボル・ディレクティブが次のように記述されていたとします。

```
__ep_DATA@%EP_SYMBOL;
```

エレメント・ポインタの値は，デフォルトで“SIDATA セグメントの先頭”になり，その値は“__ep_DATA”に入ります。

したがって，スタートアップ・ルーチン内で ep をセットするには，次のように記述します。

```
.extern __ep_DATA, 4
mov     #__ep_DATA, ep
```

__ep_DATA の絶対アドレス参照を行い，その値を ep に設定します。

7. 3. 8　main 関数実行前に行う必要のある周辺 I/O レジスタの初期化

スタートアップ・ルーチン内で外部 RAM の初期化などを行う場合，まず周辺 I/O に対して外部メモリ設定等を行わないと，メモリ領域のアクセスができず，初期化ができません。その他，スタートアップ・ルーチンを実行するうえで，設定しなくてはならない周辺 I/O レジスタの初期化を行います。

なお，レジスタ設定は，アセンブラ命令でそのまま記述してもよいですし，いったんスタートアップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。C 言語で行うと，周辺 I/O への読み出しや代入を分かりやすく記述できます。たとえば，C 言語関数“void reset(void)”を作成して，スタートアップ・ルーチンから呼び出すときは，スタートアップ・ルーチン内に次の命令を記述します。

```
jarl    _reset, lp
```

アセンブラ命令の記述と C 言語記述の違いの例を示します。たとえば，P0（ポート 0）に“1”を代入する命令を，アセンブラ・ソース（r10 を使用）と C ソースで記述すると次のようになります。

【アセンブラ・ソース】

```
mov     1, r10
st.b    r10, P0
```

【C ソース】

```
#pragma ioreg
P0 = 1;
```

外部メモリ設定等は，デバイスによって異なりますので，使用する各デバイスのユーザーズ・マニュアルを参照してください。

また，クロック発生機能を使用し，V850 内蔵の各ユニットに供給される“内部システム・クロック”を発生させる必要がありますが，このとき，PLL（Phase locked loop）シンセサイザによって，クロックを逓倍して使用します。したがって，使用する周波数を正しく設定しなければ，想定している動作速度に誤差を生じます。

PLL のデフォルト値は，たいてい逓倍値が小さく，動作周波数が低くなっています。スタートアップ・ルーチンにおいても例外ではなく，「7. 3. 10　sbss 領域のゼロクリア」以降で説明する“メモリ領域のクリア”を，動作周波数が低いまま実行すると，実行完了までにたいへん時間がかかってしまいます。したがって，PLL の設定に関しては，スタートアップ・ルーチンの初期の方で行うことを推奨します。

他“システム・ウエイト・コントロール・レジスタ（VSWC）”やコマンド・レジスタ（PRCMD），必要であれば“ウォッチ・ドック・タイマ（WDT）”などの設定も必要になりますので，使用する各デバイスのユーザーズ・マニュアルを参考に正しく設定してください。

7. 3. 9　main 関数実行前に行う必要のあるユーザ・ターゲットの初期化

スタートアップ・ルーチン内でユーザ・ターゲットに初期化が必要なものがある場合は，その初期化処理を記述しておきます。

処理はアセンブラ・ソースで記述してもよいですし，いったんスタートアップ・ルーチンから C 言語関数へ分岐し，その C 言語関数内で行ってもよいです。

7. 3. 10　sbss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sbss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sbss 領域をゼロクリアすることを推奨します。

なお，sbss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sbss 領域のクリアを行うときは，CX で予約されているシンボル“__ssbss”と“__esbss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―2　sbss 領域のシンボル

シンボル名	意味
__ssbss	sbss 領域の先頭シンボル
__esbss	sbss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sbss 領域をゼロクリアするプログラムは次のとおりです。

```
    .extern __ssbss, 4
    .extern __esbss, 4
    mov     #__ssbss, r13
    mov     #__esbss, r12
    cmp     r12, r13
    jnl     .L11
.L12:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L12
.L11:
```

sbss 領域を 4 バイトずつゼロクリアしていきます。

7. 3. 11 bss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“bss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，bss 領域をゼロクリアすることを推奨します。

なお，bss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

bss 領域のクリアを行うときは，CX で予約されているシンボル“__sbss”と“__ebss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―3 bss 領域のシンボル

シンボル名	意味
__sbss	bss 領域の先頭シンボル
__ebss	bss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，bss 領域をゼロクリアするプログラムは次のとおりです（bss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __sbss, 4
    .extern __ebss, 4
    mov     #__sbss, r13
    mov     #__ebss, r12
    cmp     r12, r13
    jnl     .L14
.L15:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L15
.L14:
```

7. 3. 12 sebss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sebss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sebss 領域をゼロクリアすることを推奨します。

なお，sebss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sebss 領域のクリアを行うときは，CX で予約されているシンボル“__ssebss”と“__esebss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―4 sebss 領域のシンボル

シンボル名	意味
__ssebss	sebss 領域の先頭シンボル
__esebss	sebss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sebss 領域をゼロクリアするプログラムは次のとおりです（sebss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __ssebss, 4
    .extern __esebss, 4
    mov     #__ssebss, r13
    mov     #__esebss, r12
    cmp     r12, r13
    jnl     .L17
.L18:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L18
.L17:
```

7. 3. 13　tibss.byte 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“tibss.byte 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.byte 領域をゼロクリアすることを推奨します。

なお，tibss.byte セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.byte 領域のクリアを行うときは，CX で予約されているシンボル“__stibss.byte”と“__etibss.byte”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―5　tibss.byte 領域のシンボル

シンボル名	意味
__stibss.byte	tibss.byte 領域の先頭シンボル
__etibss.byte	tibss.byte 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.byte 領域をゼロクリアするプログラムは次のとおりです（tibss.byte 領域を 1 バイトずつゼロクリアしていきます）。

```
    .extern __stibss.byte, 4
    .extern __etibss.byte, 4
    mov     #__stibss.byte, r13
    mov     #__etibss.byte, r12
    cmp     r12, r13
    jnl     .L20
.L21:
    st.b    r0, [r13]
    add     1, r13
    cmp     r12, r13
    jl      .L21
.L20:
```

7. 3. 14　tibss.word 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“tibss.word 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，tibss.word 領域をゼロクリアすることを推奨します。

なお，tibss.word セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

tibss.word 領域のクリアを行うときは，CX で予約されているシンボル“__stibss.word”と“__etibss.word”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―6　tibss.word 領域のシンボル

シンボル名	意味
__stibss.word	tibss.word 領域の先頭シンボル
__etibss.word	tibss.word 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，tibss.word 領域をゼロクリアするプログラムは次のとおりです（tibss.word 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __stibss.word, 4
    .extern __etibss.word, 4
    mov     #__stibss.word, r13
    mov     #__etibss.word, r12
    cmp     r12, r13
    jnl     .L23
.L24:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L24
.L23:
```

7. 3. 15 sibss 領域のゼロクリア

初期値を持たない領域である“bss 属性”領域の 1 つである“sibss 領域”の初期化を行います。

V850 リセット後のメモリ内容は不定なので，sibss 領域をゼロクリアすることを推奨します。

なお，sibss セクションを作成していない場合や，ゼロクリアの必要がない場合は，この処理を行う必要はありません。

sibss 領域のクリアを行うときは，CX で予約されているシンボル“__ssibss”と“__esibss”を使用します。それぞれのシンボルの意味は次のとおりです。

表 7―7 sibss 領域のシンボル

シンボル名	意味
__ssibss	sibss 領域の先頭シンボル
__esibss	sibss 領域の最後尾シンボル

これらのシンボルの値（アドレス）は，リンク時に決定されます。このシンボルを使用して，sibss 領域をゼロクリアするプログラムは次のとおりです（sibss 領域を 4 バイトずつゼロクリアしていきます）。

```
    .extern __ssibss, 4
    .extern __esibss, 4
    mov     #__ssibss, r13
    mov     #__esibss, r12
    cmp     r12, r13
    jnl     .L26
.L25:
    st.w    r0, [r13]
    add     4, r13
    cmp     r12, r13
    jl      .L25
.L26:
```

7. 3. 16 関数前後処理ランタイム関数用の CTBP 値の設定

関数前後処理ランタイム関数を使用する場合にこの設定が必要になります。

関数前後処理ランタイム関数を呼び出すとき，CALLT 命令を使用するため，その CALLT 命令に必要な CTBP の値を，関数前後処理ランタイム関数の関数テーブルの先頭に設定しておく必要があります。

関数前後処理ランタイム関数を使用する設定になるのは，次の場合です。

- コンパイラ・オプション“-Xpro_epi_runtime=on”と“-Ospeed”を設定している

コンパイラの最適化オプション“-Ospeed”以外を指定していると，自動的に“-Xpro_epi_runtime=on”になります。

関数前後処理ランタイム関数の関数テーブルの先頭シンボルは，次のとおりです。

- ___PROLOG_TABLE

このシンボルを用いて，次のコードを記述します。

```
mov     #___PROLOG_TABLE, r12
ldsr    r12, 20
```

CTBP はシステム・レジスタ 20 番なので，ldsr 命令を使用して，値を設定します。

7. 3. 17　プログラマブル周辺 I/O レジスタ値の設定

プログラマブル周辺 I/O レジスタを搭載している V850 マイクロコントローラを使用していて，かつ，プログラマブル周辺 I/O レジスタを使用する場合に，BPC を設定する必要があります。

たとえば，V850E/IA1 の場合，次のようになっています。

図 7―1　BPC レジスタ

15															0
PA15	0	PA13	PA12	PA11	PA10	PA9	PA8	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

表 7―8　BPC レジスタ

ビット位置	ビット名	意味
15	PA15	プログラマブル周辺 I/O エリアの使用許可／不許可を設定 0： プログラマブル周辺 I/O 領域の使用を不許可 1： プログラマブル周辺 I/O 領域の使用を許可
13 ～ 0	PA13 ～ PA0	プログラマブル周辺 I/O エリアのアドレスを設定

BPC に値をセットするには，スタートアップ・ルーチン等で BPC レジスタに値を書き込む処理が必要となります。

V850E/IA1 の場合は，PA15 に 1 を立て，PA13 ～ PA0 にプログラマブル周辺 I/O エリアのアドレスを設定することになります。たとえば，プログラマブル周辺 I/O エリアのアドレスを 0x1234 としたい場合は，BPC レジスタへの設定は次のようになります。

```
mov     0x9234, r13
st.h    r13, BPC
```

PA15 に 1 を立てる必要があるため，0x1234 と 0x8000 の論理和（OR）を取った値を BPC に設定します。

プログラマブル周辺 I/O レジスタについての詳細は，各デバイスのユーザーズ・マニュアルを参照してください。

7. 3. 18　r6 と r7 を main 関数の引数に設定

main 関数を，2 つの仮引数をもつ関数“int main (int argc, char *argv[ ]) { /* ... */ }”というように，2 つの仮引数をもつ関数として定義した場合，main 関数へ分岐する前に，引数（r6 と r7）へ値を設定する処理が必要になります。領域の確保は「7. 3. 4　main 関数の引数領域の確保」を参照してください。

なお，リアルタイム OS を使用したアプリケーションでは，main 関数は作成しないため，この処理は必要ありません。

r6 と r7 へ値を設定する処理は次のようになります。

```
ld.w    $__argc, r6
movea   $__argv, gp, r7
```

main 関数の引数の領域を .data セクションに配置したので，gp 相対のアクセス・コードを記述します。

7. 3. 19　main 関数へ分岐する（リアルタイム OS を使用していない場合）

スタートアップ・ルーチンで行う必要のある処理がすべて終わったとき，main 関数への分岐命令を実行します。

ただし，リアルタイム OS を使用したアプリケーションの場合は，main 関数は作成しないため，この処理は必要ありません。代わりにリアルタイム OS の初期化ルーチンへ分岐する命令が必要になります。この詳細については「7. 3. 20　リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）」を参照してください。

main 関数への分岐には，次のコードを記述します。

```
jarl    _main, lp
```

また，main 関数の実行がすべて終わった後，この分岐命令の次の 4 バイトに戻ってくることになります。

戻ってこないことが分かっている場合は，次の命令も使用できます。

```
jr      _main
```

```
mov     #_main, lp
jmp     [lp]
```

jmp 命令を使用すると，32 ビット全空間をアクセスすることができます。

もし，“jarl _main, lp”を使用する場合は，main 関数実行後に戻ってくるので，戻ってきた後，デッドロックしないように，対策を施しておくと安全です。

7. 3. 20　リアルタイム OS の初期化ルーチンへ分岐する（リアルタイム OS を使用している場合）

リアルタイム OS を使用したアプリケーションで，スタートアップ・ルーチンで行う必要のある処理がすべて終わったとき，初期化ルーチンへ分岐します。リアルタイム OS を使用していないアプリケーションの場合，main 関数へ分岐することになりますので「7. 3. 19　main 関数へ分岐する（リアルタイム OS を使用していない場合）」を参照してください。

【RX850V4 の場合】

```
    .extern __kernel_sit
    .extern __kernel_start
    mov     #__kernel_sit, r6
    mov     #__kernel_start, r11
    jarl    __jump_kernel_start, lp
__boot_error:
    jbr     __boot_error
__jump_kernel_start:
    jmp     [r11]
```

詳細については，リアルタイム OS のユーザーズ・マニュアルを参照してください。

7. 3. 21　V850E2V3 マルチコア用スタートアップ・ルーチン

共通部領域の初期化処理は以下の手順で行ってください。

```
__start:
        ld.hu   PEID, r10
        add     -1, r10
        shl     2, r10

        mov32   0xFFFF6900, r11                 /* MEV0 レジスタのアドレス取得 */
        mov     1, r12

        caxi    [r11], r0, r12                  /* 共通部の初期化担当 PE を選ぶ */
        bnz     .Lsleep                         /* 担当外の PE は待機させる */

        jarl    _hdwinit, lp

        mov     -1, r7
        mov32   #__S_romp, r6                   /* 共通部の初期化 */
        jarl    __rcopy, lp

        mov32   #___PROLOG_TABLE, r12
        ldsr    r12, 20

        st.w    r0, MEV0                        /* 担当外を PE を復帰させる */
.Lwakeup:
        ld.w    #__table.__ssbss[r10], r6
        ld.w    #__table.__esbss[r10], r7       /* 各 PE 部の初期化 */
        jarl    __zeroclrw, lp

        mov32   #__exit, lp
        ld.w    #__table._main[r10], r10        /* 各 PE 部の main() へ分岐 */
        jmp     [r10]
__exit:
        br      __exit
.Lsleep:
        ld.w    MEV0, r12
        cmp     r0, r12                         /* MEV レジスタが 0 になるまで待機 */
        bz      .Lwakeup
        br      .Lsleep
```

7.4　コーディング例

スタートアップ・ルーチンの例を，次に示します。

表 7―9　スタートアップ・ルーチンの例

```
#-----------------------------------------------------------------------
# CX 予約シンボルの外部ラベル宣言 1（tp，gp，ep 用）
#-----------------------------------------------------------------------
        .extern __tp_TEXT, 4
        .extern __gp_DATA, 4
        .extern __ep_DATA, 4
#-----------------------------------------------------------------------
# CX 予約シンボルの外部ラベル宣言 2（bss 属性セクション初期化用）
# 使用していないセクションがある場合は削除。
# まだ使用するセクションが決まっていないような場合は，すべて書いておいてセクションの追加・削除によって
# スタート・アップ・ルーチンのアセンブル・エラーを出さないようにしておくとよい。
#-----------------------------------------------------------------------
        .extern __ssbss, 4
        .extern __esbss, 4
        .extern __sbss, 4
        .extern __ebss, 4
        .extern __ssebss, 4
        .extern __esebss, 4
        .extern __stibss.byte, 4
        .extern __etibss.byte, 4
        .extern __stibss.word, 4
        .extern __etibss.word, 4
        .extern __ssibss, 4
        .extern __esibss, 4
#-----------------------------------------------------------------------
# CX 予約シンボルの外部ラベル宣言
# V850Ex で関数前後処理ランタイム関数を使用するとき，
# 関数テーブルの先頭アドレスを外部ラベル宣言しておく。
#-----------------------------------------------------------------------
        .extern ___PROLOG_TABLE
#-----------------------------------------------------------------------
# main 関数の外部ラベル宣言
#-----------------------------------------------------------------------
        .extern _main
#-----------------------------------------------------------------------
# main 関数の引数の領域（void main(void) 型の場合は必要なし）
#-----------------------------------------------------------------------
        .dseg   DATA
        .align  4
__argc:
```

```
        .db4    0
__argv:
        .db4    #.L16
.L16:
        .db     0
        .db     0
        .db     0
        .db     0
#---------------------------------------------------------------------
# 以下はセクション生成のための“ダミーデータ”
# このダミーは，後で出てくる bss 属性のセクションをゼロクリアするためのもの。
#
# その先頭シンボルと最後尾シンボルは，リンク時に該当セクションにデータが存在したときに生成される。しかし，ま
# だ使用するセクションが決まっていないような場合は，リンク・ディレクティブ・ファイルを書き換えてセクションの
# 追加・削除するたびに，スタートアップ・ルーチンのアセンブル・エラーが出てしまう。これを避けるために，ダミ
# ー・データをセクションに配置することで，セクションの先頭シンボルと最後尾シンボルをとりあえず生成しておく
# bss セクションは，スタック生成コードでデータが割り当てられており，ここにダミーを作る必要がないため，記述し
# ていない。
#
# 使用するセクションが決まったときは，このダミー部分を削除し，また，ゼロクリア・ルーチンも必要なものだけ残し
# て削除すると，無駄がなくなり，コード効率が上がる。
#---------------------------------------------------------------------
.sbss   .dseg   sbss
        .ds(0)
.sebss  .dseg   sebss
        .ds(0)
.tibss.byte     .dseg   tibss.byte
        .ds(0)
.tibss.word     .dseg   tibss.word
        .ds(0)
.sibss  .dseg   sibss
        .ds(0)
#---------------------------------------------------------------------
# スタック確保
# bss 領域に 0x200 バイト確保
#---------------------------------------------------------------------
STACKSIZE       .set    0x200
        .dseg   BSS
#---------------------------------------------------------------------
# リセット・ハンドラ
# リセット・ハンドラに配置する命令を記述
#---------------------------------------------------------------------
RESET   .cseg   TEXT
        jr      __start
#---------------------------------------------------------------------
```

```
# スタートアップ・ルーチン本体
#-----------------------------------------------------------------------
        .cseg   text
        .align  4
        .public __start
        .public __exit
        .public __startend
__start:
#-----------------------------------------------------------------------
# __gp_DATA は tp からの相対値とすることをシンボル・ディレクティブで設定していることを想定。
# そのため gp は tp に __gp_DATA の値を加算する。
#-----------------------------------------------------------------------
        mov     #__tp_TEXT, tp
        mov     #__gp_DATA, gp
        add     tp, gp
        mov     #__stack + STACKSIZE, sp
        mov     #__ep_DATA, ep
#-----------------------------------------------------------------------
# sbss セクションのゼロクリア
# sbss セクションを使用していないときは，コード削減のため削除する。
#-----------------------------------------------------------------------
        mov     #__ssbss, r13
        mov     #__esbss, r12
        cmp     r12, r13
        jnl     .L11
.L12:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl      .L12
.L11:
#-----------------------------------------------------------------------
# bss セクションのゼロクリア
# bss セクションを使用していないときは，コード削減のため削除する。
#-----------------------------------------------------------------------
        mov     #__sbss, r13
        mov     #__ebss, r12
        cmp     r12, r13
        jnl     .L14
.L15:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl     .L15
.L14:
```

```
#------------------------------------------------------------------
# sebss セクションのゼロクリア
# sebss セクションを使用していないときは，コード削減のため削除する。
#------------------------------------------------------------------
        mov     #__ssebss, r13
        mov     #__esebss, r12
        cmp     r12, r13
        jnl     .L17
.L18:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl     .L18
.L17:
#------------------------------------------------------------------
# tibss.byte セクションのゼロクリア
# tibss.byte セクションを使用していないときは，コード削減のため削除する
。
#------------------------------------------------------------------
        mov     #__stibss.byte, r13
        mov     #__etibss.byte, r12
        cmp     r12, r13
        jnl     .L20
.L21:
        st.b    r0, [r13]
        add     1, r13
        cmp     r12, r13
        jl      .L21
.L20:
#------------------------------------------------------------------
# tibss.word セクションのゼロクリア
# tibss.word セクションを使用していないときは，コード削減のため削除する。
#------------------------------------------------------------------
        mov     #__stibss.word, r13
        mov     #__etibss.word, r12
        cmp     r12, r13
        jnl     .L23
.L24:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl     .L24
.L23:
#------------------------------------------------------------------
# sibss セクションのゼロクリア
# sibss セクションを使用していないときは，コード削減のため削除する。
```

```
#------------------------------------------------------------------
        mov     #__ssibss, r13
        mov     #__esibss, r12
        cmp     r12, r13
        jnl     .L26
.L25:
        st.w    r0, [r13]
        add     4, r13
        cmp     r12, r13
        jl     .L25
.L26:
#------------------------------------------------------------------
# 関数のプロローグ・エピローグ・ランタイム・ライブラリの設定
# ライブラリ関数テーブルの先頭アドレスを CTBP（システムレジスタ 20 番）にセット
# V850Ex 以外は，この記述は削除。
#------------------------------------------------------------------
        mov     #___PROLOG_TABLE, r12
        ldsr    r12, 20
#------------------------------------------------------------------
# プログラマブル周辺 I/O レジスタの設定
# プログラマブル周辺 I/O レジスタを持たない V850 の場合はこの記述は削除。
# 下記は BPC レジスタの値（設定アドレス）が 0x1234 の例
# 0x1234( アドレス ) と 0x8000( プログラマブル周辺 I/O 使用 ) の論理和を BPC に設定する。
#------------------------------------------------------------------
PIOADDR .set    0x12340000
USEBPC  .set    0x8000
        mov     USEBPC | (PIOADDR >> 14), r13
        st.w    r13, BPC
#------------------------------------------------------------------
# main 関数の引数を r6，r7 に設定
#------------------------------------------------------------------
        ld.w    $__argc, r6
        movea   $__argv, gp, r7
#------------------------------------------------------------------
# main 関数へ分岐
#------------------------------------------------------------------
        jarl    _main, lp
#------------------------------------------------------------------
# main 関数が戻ってきた後の処理
#------------------------------------------------------------------
__exit:
        br      __exit
__startend:
```

第 8 章 ROM 化

この章では，ROM 化の手順，操作方法などを説明します。

8.1 概　　要

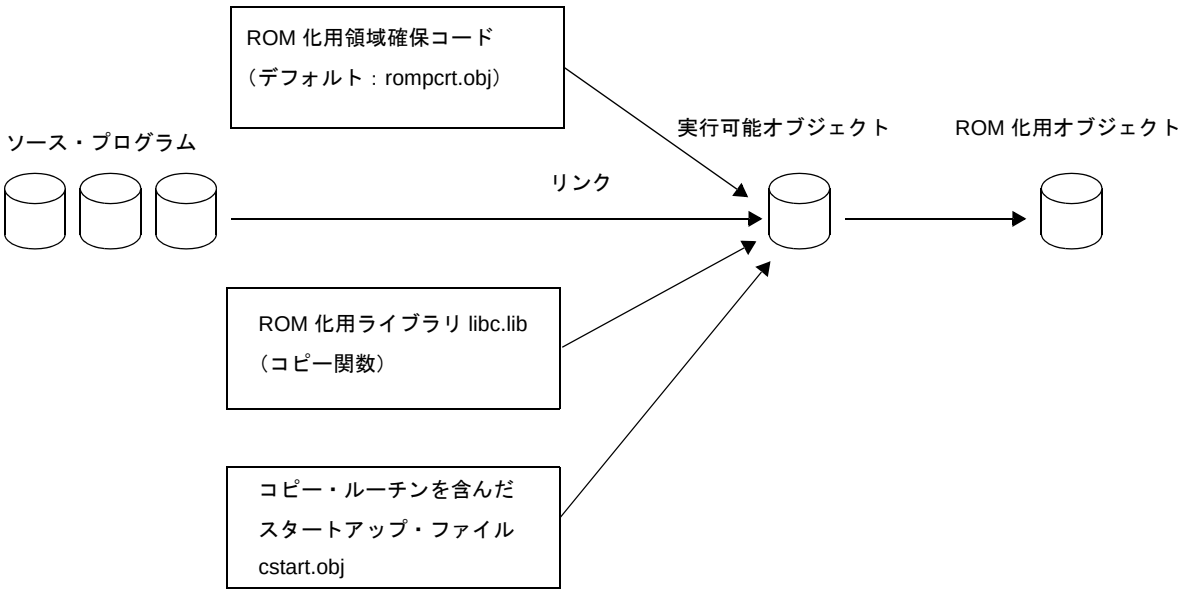
プログラム中で，グローバルに変数を宣言すると，初期値を持つ変数ならば data 属性のセクションへ，初期値を持たない変数ならば bss 属性のセクションというように，RAM 上のセクションに配置されます。特に初期値を持つ変数ならば，その初期値自体が RAM 上に配置されます。その他，アプリケーションの高速化のために，プログラム・コードを内蔵 RAM 領域へ配置する場合もあります。

組み込みシステムの場合，デバッグ時にインサーキット・エミュレータなどを使用する場合，実行可能なモジュールを配置イメージのままダウンロードして実行できます。しかし，実際にプログラムをターゲット・システムの ROM 領域に書き込んで実行する場合，data 属性のセクションにある初期値情報や，RAM 領域に配置するプログラム・コードを，実行前に RAM 上に展開されていなければなりません。つまり，RAM に展開するデータを ROM 上に持たせておき，それをアプリケーション実行前に ROM から RAM へコピーする作業が必要になります。

ROM 化は，data 属性セクションの変数の初期値情報や，RAM 上に配置するプログラムを，1 つのセクションにパッキングすることです。このセクションを ROM 上に配置し，CX で用意されているコピー関数を呼び出すことによって，初期値情報やプログラムを容易に RAM 上へ展開することができます。

ROM 化用オブジェクトを作成する流れの概要は，次図のようになります。

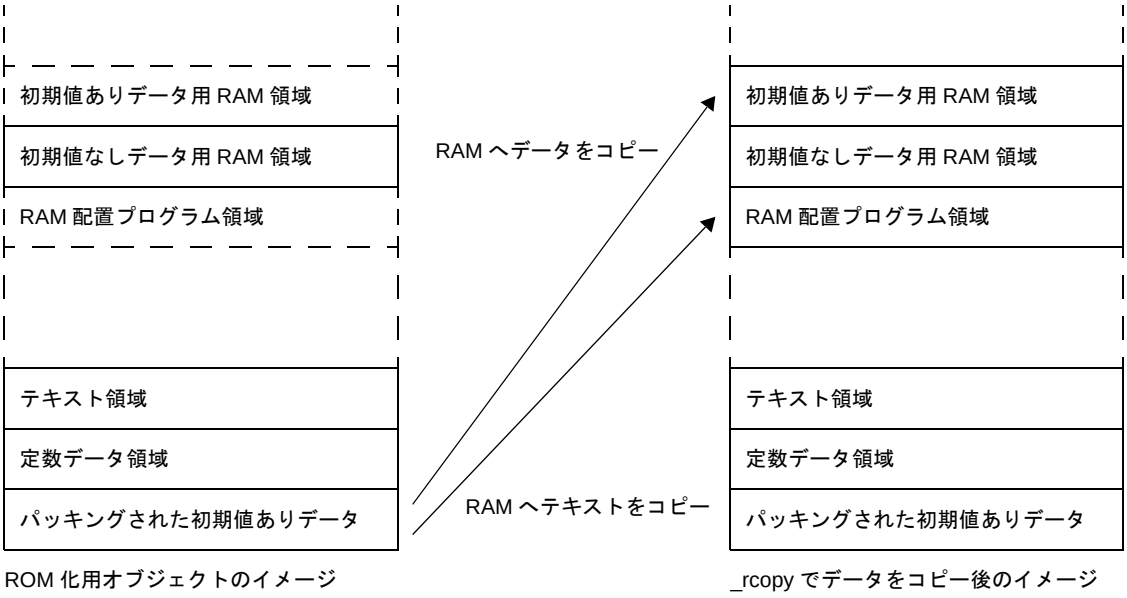
図 8―1　ROM 化用オブジェクトの作成



「表 8―1 コピー関数」のように ROM 化用オブジェクトを作成すると，コピー関数 _rcopy を実行することによって，RAM に配置するデータを，パッキングされた ROM からコピーします。

イメージは次のようになります。

図 8―2 コピー関数呼び出し前後のイメージ



ここで，ROM 化用オブジェクトに必要となるセクション名，およびそのセクションの先頭アドレス（ラベル名）は，デフォルトでは次のようになっています。

- パッキングしたセクション名→ rompsec セクション
- rompsec セクションの先頭アドレス（ラベル名）→ __S_romp

そして rompsec セクションから，RAM 領域へコピーする関数は次のとおりです。

- コピー関数→ _rcopy，_rcopy1，_rcopy2，_rcopy4

この関数は *Version Folder* \ lib850e にあるライブラリ“libc.lib”に格納されています。

__S_romp は *Version Folder* \ lib850e にある“rompcrt.obj”で定義されているラベルです（このソース・ファイルは rompcrt.asm）。rompcrt.obj をそのまま使用することにより，自動的に .text 属性の直後（4 バイトでアラインしたところ）に，rompsec セクションを作成します。そして __S_romp が rompsec セクションの先頭アドレスを指すラベルになります。

このように自動的に rompsec セクションを作成する方法のほかに，rompcrt.asm に相当するプログラムを独自に作成して配置することもできます。

また，リロケーション解決したオブジェクト・モジュール・ファイルにシンボル情報，デバッグ情報が含まれる場合，それらを削除することなく ROM 化用のオブジェクト・モジュール・ファイルを生成します。そのため，ROM 化後のオブジェクト・モジュール・ファイルでもデバッガによるソース・デバッグができます。

8.2 rompsec セクション

この節では，rompsec セクションについて説明します。

8.2.1 パッキングするセクションの種類

rompsec セクションとしてパッキングする対象となるものは，デフォルトでは「書き込み可能な属性を持つセクション[注]に割り当てられたデータ」です。その他，オプション（-Xrompsec_text オプション）を指定することによって「text 属性，const 属性を持つ任意のセクション」をパッキングすることも可能です。

注 書き込み可能ですが初期値を持たないことが明らかである bss 属性セクション，および sbss 属性セクションは，パッキングされません。

具体的には次に示すようになります。

- 予約セクション（.data，.sdata，.sedata，.sidata，.tidata，.tidata.byte，.tidata.word）
- アセンブラ・プログラムにおいてセクション定義疑似命令により sdata 属性，または data 属性を指定し，任意の名前で生成したセクション
- 内蔵命令 RAM に配置するセクション（V850E2 コアを持つデバイス指定時は対象となりません）

ただし，ユーザが指定する「text 属性，const 属性を持つ任意のセクション」をパッキングせず，さらに上記のセクションが実行可能モジュールに存在しない場合は，ROM 化オブジェクトを生成する必要はありません。

予約セクション（.data，.sdata，.sedata，.sidata，.tidata，.tidata.byte，.tidata.word）が存在するかしないかは，リンク・マップ・ファイルを参照してください。

ROM 化処理によって生成されたマップ・ファイルを参照することにより，.data セクションや .sdata セクション，内蔵 RAM に配置したセクション（割り込みハンドラのセクションも含む）などの代わりに，rompsec セクションが作成されていることを確認することができます。

なお，内蔵命令 RAM に配置されるセクション（割り込みハンドラのセクションも含む）をパッキングする際，各セクションの先頭アドレスは 4 バイト整列されている必要があります。

また，rompsec セクション内のオフセットも 4 バイト整列するため，パディング領域が発生し，rompsec セクションのサイズに加算します。

8.2.2 rompsec セクションのサイズ

rompsec セクションとして確保されるサイズについて説明します。

ROM 化用モジュールを作成するときは，rompsec セクションのサイズと，使用している CPU の内蔵 ROM 領域，ターゲット・システムの ROM 領域のアドレスやサイズに注意します。

rompsec セクションが，他のセクションとオーバラップしないようにリンク・ディレクティブ・ファイルを記述してください。

備考 リンク・ディレクティブ・ファイルの記述例については，「8.3 ROM 化用ロード・モジュールの作成」を参照してください。

次に rompsec セクションのサイズを求める計算式を示します。

rompsec セクションのサイズ（10 進数，単位：バイト）

＝ 8 ＋ 12 × ROM 化対象セクションの数＋ ROM 化対象セクションの合計サイズ＋パディング・サイズ[注]

注 ROM 化対象のセクションの整列条件により 1 セクションあたり 0 ～ 3 バイトになります。

たとえば，.data，.sdata セクションが存在し，それぞれのサイズが 1002 バイト，1000 バイトで，それぞれのセクションの整列条件が 4 バイトの場合，rompsec セクションのサイズは次のようになります。

8 ＋ 12 × 2 ＋ 1002 ＋ 1000 ＋ 2 ＝ 2036（バイト）

8. 2. 3 rompsec セクションとリンク・ディレクティブ

ROM 化時には，.text セクションの直後に rompsec セクションが追加されるよう，CX は ROM 化用領域確保コード・ファイル（rompcrt.obj）を最後にリンクします。

したがって，以下のようなリンク・ディレクティブによる rompsec セクションの配置は必要ありません。

ROM 化処理を考慮したリンク・ディレクティブを以下に示します。

```
SCONST : !LOAD ?R {                    # 内蔵 ROM に SCONST ／ CONST ／ TEXT を配置
        .sconst = $PROGBITS ?A .sconst;
};

CONST : !LOAD ?R {
        .const = $PROGBITS ?A .const;
};

TEXT : !LOAD ?RX {
        .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
        .text = $PROGBITS ?AX .text;
        rompsec = $PROGBITS ?AX rompsec # 内蔵 ROM の最後に rompsec を配置
};

DATA : !LOAD ?RW V0x100000 {           # 外部 RAM に DATA を配置
        .data = $PROGBITS ?AW;
        .sdata = $PROGBITS ?AWG;
        .sbss = $NOBIT ?AWG;
        .bss = $NOBIT ?AW;
};

SIDATA : !LOAD ?RW V0xFFE000 {         # 内蔵 RAM に SIDATA を配置
        .sidata = $PROGBITS ?AW .sidata;
        .sibss = $NOBITS ?AWG .sibss;
};

__tp_TEXT@%TP_SYMBOL;
```

```
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

rompsec セクションが内蔵 ROM 領域を越えた場合には，メッセージを出力して処理を中止します。

備考 Xromize_check_off=rom_less オプションを指定することで，内蔵 ROM 領域を無視することができます。また，-Xromize_check_off オプションを指定することで，メッセージを出力しつつ処理を続行することができます。

rompsec セクションを外部 ROM 領域の終端に配置する場合には，これらのチェックは行われません。メモリ・マップ・ファイルを参照して，ROM に収まっているかの判断を行ってください。

備考 メモリ・マップ・ファイルは，-Xmap オプションを指定することにより出力することができます。

なお，ROM の途中に rompsec セクションを配置する必要がある場合には，次のように rompsec セクションのサイズと配置アドレスから，rompsec セクションの配置される領域を認識した上で，rompsec セクション直後のセグメントに対して，適切なアドレス指定をしてください。

図 8―3 ROM 化処理を考慮したリンク・ディレクティブ（サイズ考慮）

```
# 内蔵 ROM に SCONST/CONST/TEXT を配置
SCONST:    !LOAD ?R {
    .sconst = $PROGBITS ?A .sconst;
};
# 内蔵 ROM の途中に .text を配置
#TEXT と CONST の間に rompsec
TEXT:      !LOAD ?RX {
    .pro_epi_runtime = $PROGBITS ?AX .pro_epi_runtime;
    .text = $PROGBITS ?AX .text;
    rompsec = $PROGBITS ?AX rompsec;
};
# 内蔵 ROM の最後に rompsec のサイズを考慮したアドレス指定を行って CONST を配置
CONST:     !LOAD ?R Vx3f800 {
    .const = $PROGBITS ?A .const;
};
# 外部 RAM に DATA を配置
DATA:      !LOAD ?RX V0x100000 {
    .data = $PROGBITS ?AW;
    .sdata = $PROGBITS ?AWG;
    .sbss = $NOBIT ?AWG;
    .bss = $NOBIT ?AW;
};
# 内蔵 RAM に SIDATA を配置
SIDATA:    !LOAD ?RX V0xFFE000 {
```

```
    .sidata = $PROGBITS ?AW .sidata;
    .sibss = $NOBIT ?AWG .sibss;
};
__tp_TEXT@%TP_SYMBOL;
__gp_DATA@%GP_SYMBOL &__tp_TEXT{DATA};
__ep_DATA@%EP_SYMBOL;
```

8. 3 ROM 化用ロード・モジュールの作成

この節では，ROM 化用ロード・モジュールの作成手順について説明します。

8. 3. 1 ROM 化用ロード・モジュールの作成手順（デフォルト）

デフォルトで用意されている ROM 化用領域確保コード・ファイル（rompcrt.obj）を使用した方法を以下に示します。

(1) コピー関数の呼び出し

スタートアップ・ルーチンに，必要な引数を持ったコピー関数 _rcopy( ) を起動するように記述をし，オブジェクト・モジュール・ファイルを作成します。

CX では，標準のスタートアップ・ルーチンにこの記述がされているので，デフォルトで ROM 化を行う場合には，特に意識する必要はありません。

また，-Xno_romize オプションで ROM 化を抑止する場合についても，ドライバで入力するスタートアップ・ファイルの切り替えを行うため，特に意識する必要はありません。

ただし，_rcopy の代わりに _rcopy2/_rcopy4 を使用したい場合や，コピーするセクションを指定したい場合は，スタートアップ・ルーチンを書き換え，cstart.asm から cstart.obj を作成する必要があります。

図 8―4 コピー関数 _rcopy の使用例（cstrat.asm）

```
.extern _hdwinit
.extern __S_romp
.extern __rcopy
     :
jarl    _hdwinit, lp
    :
mov32   #__S_romp, r6
mov     -1, r7
jarl    __rcopy, lp
```

備考 1. コピー関数の詳細については，「8. 4 コピー関数」を参照してください。

2. スタートアップ・ルーチンを書き換える場合は，-Xno_startup オプション，または -Xstartup オプションを指定して，使用するスタートアップ・ルーチンを切り替えてください。

(2) rompsec セクションの配置指定

ROM 化時には，.text セクションの直後に rompsec セクションが追加されるよう，cx は ROM 化用領域確保コード・ファイル（rompcrt.obj）を最後にリンクします。

したがって，リンク・ディレクティブによる rompsec セクションの配置は必要ありません。

備考 詳細については，「8. 2. 3 rompsec セクションとリンク・ディレクティブ」を参照してください。

(3) rompsec セクション用の領域確保

rompsec セクション用の領域を確保し，その先頭アドレスを示すオブジェクト・モジュール・ファイルを作成します。

標準の ROM 化用領域確保コード・ファイルが，デフォルトでリンクされるため，特に意識する必要はありません。

__S_romp というラベルが，オブジェクト・モジュール・ファイル内の text セクションの終端を越える最初の（4 バイトの整列条件で整列された）アドレスを絶対アドレスとして指すようなコード生成します。

(4) リンク処理

リンク処理は，以下の順序でファイルを指定するようにドライバが制御します。

- cstart.obj（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）
- ユーザ指定のオブジェクト・モジュール・ファイル
- libc.lib（hdwinit 関数や _rcopy 関数を含む標準ライブラリ）
- rompcrt.obj（ROM 化用領域確保コード・ファイル）

rompcrt.obj を最後にリンク[注]することにより，.text セクションの直後に rompsec セクション用の領域が確保されることになります。

注 -Xrescan オプションを指定した場合，rompcrt.obj の後にライブラリ・ファイルがリンクされ，ROM 化処理中にエラーとなることがあります。その場合には，rompsec セクションの領域を明示的に確保するようにしてください。

詳細については，「8. 2. 3 rompsec セクションとリンク・ディレクティブ」を参照してください。

(5) ROM 化処理

ROM 化処理にて，初期値を持ち RAM に置かれる属性である data 属性，または sdata 属性を持つセクション，および，内蔵命令 RAM に配置されるセクション（割り込みハンドラのセクション等，リンク・ディレクティブで内蔵命令 RAM に配置指定したすべてのセクション）の代わりに，rompsec セクションを持つオブジェクト・モジュール・ファイルが生成されます。

(6) ヘキサ処理

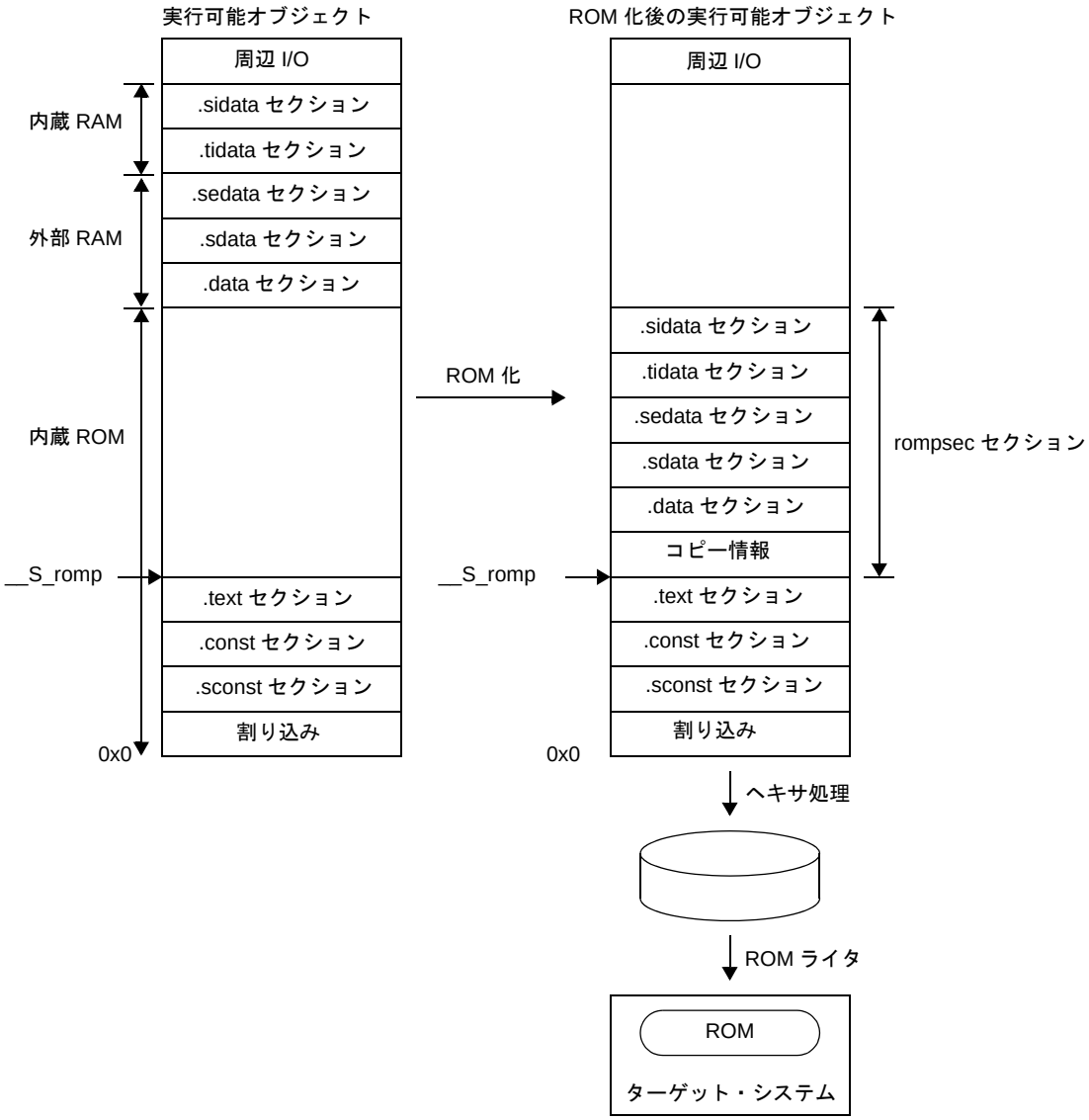
ヘキサ処理にて，ヘキサ・データが作成されます。

ヘキサ処理は，ドライバにより自動的に呼び出されます。-Xhex オプションの指定により，出力ファイル名の指定を行うことができます。

（7）ROM へのダウンロード

作成したヘキサ・データをターゲット・システムの ROM にロードします。

図 8―5 ROM 化のイメージ（デフォルト）



8. 3. 2 ROM 化用ロード・モジュールの作成手順（カスタマイズ）

ROM 化用領域確保コード・ファイルを独自に作成し，.rompack セクションの先頭アドレスや配置場所を自分で決定する方法を示します。

（1）ROM 化用領域確保コード・ファイルの記述

標準の ROM 化用領域確保コード・ファイル rompcrt.asm に相当するコードを記述します。

ここでは，ROM 化用領域確保コード・ファイルのソース・ファイルを“rompack.asm”，ROM 化用領域の先頭を指すシンボル名を“__rompack”とします。

また，このシンボルの存在するセクションを“.rompack セクション”とします。

この場合，rompack.asm は，以下のようなコードになります。

例 rompack.asm

```
.rompack        .cseg   text
        .align  4
        .public __rompack, 4
__rompack:
```

rompack.asm を記述したのち，アセンブルして，ROM 化用領域確保コード・ファイルのオブジェクト・ファイル rompack.obj を生成します。

（2）コピー関数の呼び出し

スタートアップ・ルーチン内でコピー関数 _rcopy を呼び出します。

例 コピー関数 _rcopy の呼び出し

```
.extern _hdwinit
.extern __rompack
.extern __rcopy
        :
jarl    _hdwinit, lp
        :
mov32   #__rompack, r6
mov     -1, r7
jarl    __rcopy, lp
```

備考 1. コピー関数の詳細については，「8. 4 コピー関数」を参照してください。

2. 標準以外のスタートアップ・ルーチンを使用する場合は，-Xno_startup オプション，または -Xstartup オプションを指定して，使用するスタートアップ・ルーチンを切り替えてください。

(3) .rompack セクションの配置指定

リンク・ディレクティブにおいて，作成した .rompack セクションを定義します。

これと同時にアドレスを指定すると，.rompack セクションの配置場所を任意に決定することもできます。

.rompack セクションを含むセグメントを ROMPACK とし，このセグメントを 0x3000 番地に配置する場合，リンク・ディレクティブは以下のようになります。

```
TEXT:    !LOAD ?RX V0x1000 {
        .text = $PROGBITS ?AX .text;
};

ROMPACK: !LOAD ?RX V0x3000 {
        .rompack = $PROGBITS ?AX .rompack;
};
                :
```

このとき，ROMPACK セグメントの配置アドレスが前後のセグメントと重ならないように，.rompack セクションのサイズを「8. 2. 2 rompsec セクションのサイズ」に従って見積もり，リンク・ディレクティブ・ファイルに反映します。

(4) ROM 化用領域確保コード・ファイルの指定

-Xrompcrt オプションで，ROM 化用領域確保コード・ファイル rompack.obj を指定します。

(5) rompack セクションの先頭ラベルを指定

-Xrompsec_start オプションのパラメータとして，“__rompack”を指定します。

これにより，ラベル __rompack が .rompack セクションと同じアドレスを指すコードを生成します。

(6) リンク処理

リンク処理は，以下の順序でファイルを指定するようにドライバが制御します。

- cstart.obj（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）
- ユーザ指定のオブジェクト・モジュール・ファイル
- libc.lib（hdwinit 関数や _rcopy 関数を含む標準ライブラリ）
- rompack.obj（ROM 化用領域確保コード・ファイル）

rompack.obj を最後にリンク[注]することにより，.text セクションの直後に .rompack セクション用の領域が確保されることになります。

注 -Xrescan オプションを指定した場合，rompack.obj の後にライブラリ・ファイルがリンクされ，ROM 化処理中にエラーとなることがあります。その場合には，.rompack セクションの領域を明示的に確保するようにしてください。
詳細については，「8. 2. 3 rompsec セクションとリンク・ディレクティブ」を参照してください。

（7）ROM 化処理

ROM 化処理にて，初期値を持ち RAM に置かれる属性である data 属性，または sdata 属性を持つセクション，および，内蔵命令 RAM に配置されるセクション（割り込みハンドラのセクション等，リンク・ディレクティブで内蔵命令 RAM に配置指定したすべてのセクション）の代わりに，.rompack セクションを持つオブジェクト・モジュール・ファイルが生成されます。

（8）ヘキサ処理

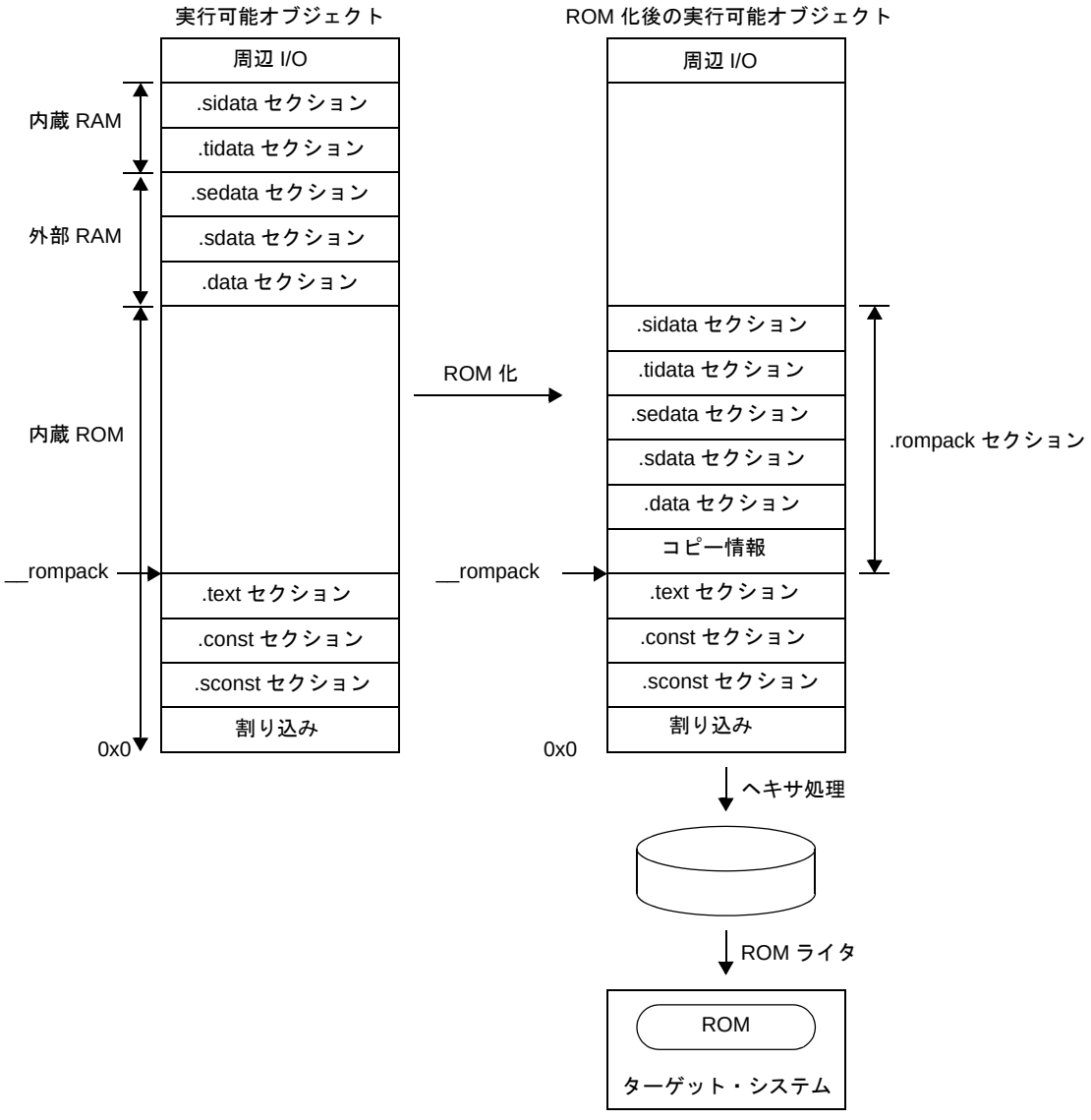
ヘキサ処理にて，ヘキサ・データが作成されます。

ヘキサ処理は，ドライバにより自動的に呼び出されます。-Xhex オプションの指定により，出力ファイル名の指定を行うことができます。

（9）ROM へのダウンロード

作成したヘキサ・データをターゲット・システムの ROM にロードします。

図 8―6 ROM 化のイメージ（カスタマイズ）



8.4 コピー関数

ここでは，ROM 化するプログラムに必要なコピー関数について説明します。

表 8―1 コピー関数

関数名	機能
_rcopy	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy1 と同じ）
_rcopy1	パッキング・データを 1 バイトずつ RAM へコピーする（_rcopy と同じ）
_rcopy2	パッキング・データを 2 バイトずつ RAM へコピーする
_rcopy4	パッキング・データを 4 バイトずつ RAM へコピーする

転送先の RAM の仕様に応じて，1 バイト転送，2 バイト転送，4 バイト転送を使い分けてください。

_rcopy

初期値データ /RAM テキスト注のコピー（1 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ

[指定形式]

int _rcopy(const unsigned long * *label*, long *number*);

extern const unsigned long _S_romp 注;

注 _S_romp は，パッキング・データの先頭アドレスです。

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 1 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。

[注意事項]

- 本関数は ROM 化処理で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。
- *label* の示すアドレス以降の 4 バイトの領域に，ROM 化処理で生成されたオブジェクトであることを示すマジック・ナンバが存在しない場合，コピーを行いません。
- *number* に指定するセクション番号は，正の整数です。
  セクション名とセクション番号の関連については，メモリ・マップ・ファイルを参照してください。
- *number* にセクション番号，または -1 以外の指定を行った場合，コピーを行いません。

- 複数の RAM が存在し，複数のコピー・ルーチンを使い分ける場合に，*number* に -1 の指定を行なうと，すべてのセクションを各々のコピー・ルーチンで複数回転送することになり，セクションの整列等の問題により，正常にコピーされません。このような場合，*number* には -1 の指定をせず，セクション番号を指定してください。
- *number* に -1 を指定した場合，セクション番号順にコピーを行います。上記の各問題によりコピーが行なわれないセクションが発生した場合，戻り値として -1 を返却し，問題となったセクションよりも後ろのセクションは，コピーを行いません。
- 本関数は _rcopy1 と同じ機能です。

［使用例］

```
main(){
        _rcopy( &_S_romp, -1 );
}
```

_rcopy1

初期値データ /RAM テキスト注のコピー（1 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ

[指定形式]

int _rcopy1(const unsigned long * *label*, long *number*);

extern const unsigned long _S_romp 注;

注 _S_romp は，パッキング・データの先頭アドレスです。

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 1 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。

[注意事項]

- 本関数は ROM 化処理で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。
- *label* の示すアドレス以降の 4 バイトの領域に，ROM 化処理で生成されたオブジェクトであることを示すマジック・ナンバが存在しない場合，コピーを行いません。
- *number* に指定するセクション番号は，正の整数です。
  セクション名とセクション番号の関連については，メモリ・マップ・ファイルを参照してください。
- *number* にセクション番号，または -1 以外の指定を行った場合，コピーを行いません。

- 複数の RAM が存在し，複数のコピー・ルーチンを使い分ける場合に，*number* に -1 の指定を行なうと，すべてのセクションを各々のコピー・ルーチンで複数回転送することになり，セクションの整列等の問題により，正常にコピーされません。このような場合，*number* には -1 の指定をせず，セクション番号を指定してください。
- *number* に -1 を指定した場合，セクション番号順にコピーを行います。上記の各問題によりコピーが行なわれないセクションが発生した場合，戻り値として -1 を返却し，問題となったセクションよりも後ろのセクションは，コピーを行いません。
- 本関数は _rcopy と同じ機能です。

_rcopy2

初期値データ /RAM テキスト注のコピー（2 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ

[指定形式]

int _rcopy2(const unsigned long * *label*, long *number*);

extern const unsigned long _S_romp 注;

注 _S_romp は，パッキング・データの先頭アドレスです。

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 2 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。

[注意事項]

- コピー元の先頭アドレス（rompsec セクション内のオフセット）と，コピー先の先頭アドレスが 2 バイト整列されていない場合には，コピーを行いません。
- コピーするセクションのサイズが 2 の倍数でない場合，最後の端数バイトに加え，セクション終端直後のパディング領域もコピーを行います。後ろに続くセクションを 2 バイト整列とするか，アドレス昇順にコピーを行なうことで，後ろに続くセクションを上書きしないようにしてください。
- 本関数は ROM 化処理で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。

- *label* の示すアドレス以降の 4 バイトの領域に，ROM 化処理で生成されたオブジェクトであることを示すマジック・ナンバが存在しない場合，コピーを行いません。
- *number* に指定するセクション番号は，正の整数です。
  セクション名とセクション番号の関連については，メモリ・マップ・ファイルを参照してください。
- *number* にセクション番号，または -1 以外の指定を行った場合，コピーを行いません。
- 複数の RAM が存在し，複数のコピー・ルーチンを使い分ける場合に，*number* に -1 の指定を行なうと，すべてのセクションを各々のコピー・ルーチンで複数回転送することになり，セクションの整列等の問題により，正常にコピーされません。このような場合，*number* には -1 の指定をせず，セクション番号を指定してください。
- *number* に -1 を指定した場合，セクション番号順にコピーを行います。上記の各問題によりコピーが行なわれないセクションが発生した場合，戻り値として -1 を返却し，問題となったセクションよりも後ろのセクションは，コピーを行いません。

_rcopy4

初期値データ /RAM テキスト注のコピー（4 バイト）を行います。

注 RAM に配置する初期値ありデータ・セクション，および内蔵 RAM 用テキスト・セクションです。

[所属]

ROM 化用ライブラリ

[指定形式]

int _rcopy4(const unsigned long * *label*, long *number*);

extern const unsigned long _S_romp 注;

注 _S_romp は，パッキング・データの先頭アドレスです。

[戻り値]

0	正常終了（正しくコピーされた場合）
-1	異常終了（正しくコピーされなかった場合）

[詳細説明]

label の示すアドレス以降に存在する rompsec セクション内の情報を元に，コピーしたいセクション番号 *number* の初期値データ，または RAM に配置するテキストを，RAM 領域に 4 バイトずつコピーします。*number* に -1 を指定した場合，rompsec セクション内のすべてのセクションをコピーします。セクション番号 *number* は，1 から始まる正数です。

デフォルトは，セクションが入力ファイル中に出現した順番に割り当てられます。

[注意事項]

- コピー元の先頭アドレス（rompsec セクション内のオフセット）と，コピー先の先頭アドレスが 4 バイト整列されていない場合には，コピーを行いません。
- コピーするセクションのサイズが 4 の倍数でない場合，最後の端数バイトに加え，セクション終端直後のパディング領域もコピーを行います。後ろに続くセクションを 4 バイト整列とするか，アドレス昇順にコピーを行なうことで，後ろに続くセクションを上書きしないようにしてください。
- 本関数は ROM 化処理で生成された情報にしたがってコピーを行います。本関数実行時にコピー先のアドレスにオフセットを加えるような処理はできません。
- コピーを行うとオーバライトが生じる場合，コピーを行いません。
- 本関数の第一引数 *label* には，絶対値を持つグローバルなラベル，または絶対アドレスを指定してください。これら以外のものを指定した場合，その結果は保証されません。

- *label* の示すアドレス以降の 4 バイトの領域に，ROM 化処理で生成されたオブジェクトであることを示すマジック・ナンバが存在しない場合，コピーを行いません。
- *number* に指定するセクション番号は，正の整数です。
  セクション名とセクション番号の関連については，メモリ・マップ・ファイルを参照してください。
- *number* にセクション番号，または -1 以外の指定を行った場合，コピーを行いません。
- 複数の RAM が存在し，複数のコピー・ルーチンを使い分ける場合に，*number* に -1 の指定を行なうと，すべてのセクションを各々のコピー・ルーチンで複数回転送することになり，セクションの整列等の問題により，正常にコピーされません。このような場合，*number* には -1 の指定をせず，セクション番号を指定してください。
- *number* に -1 を指定した場合，セクション番号順にコピーを行います。上記の各問題によりコピーが行なわれないセクションが発生した場合，戻り値として -1 を返却し，問題となったセクションよりも後ろのセクションは，コピーを行いません。

第 9 章　コンパイラとアセンブラの相互参照

この章では，CX 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。

9.1　引数，自動変数のアクセス方法

(1) アセンブラ関数への引数

CX は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。アセンブラ関数内で引数値を使用するときは，それぞれに格納された値を参照してください。

ただし，構造体を返すアセンブラ関数の場合，3 ワード分の引数を“引数用レジスタ（r7 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。そして，r6 レジスタには戻り値を格納するアドレスを格納しますので，引数格納場所に注意が必要です。

なお，引数値は C 言語関数において，引数に指定された値そのもので，この値をアセンブラ関数内で変更しても，C 言語関数の動作に影響を及ぼすことはありません。

(2) C 言語関数への引数

CX は 4 ワード分の引数を“引数用レジスタ（r6 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。4 ワード分を越えた引数は，SP の指すアドレスから，上位方向に向かって格納してください。

ただし，構造体を返す C 言語関数の場合，3 ワード分の引数を“引数用レジスタ（r7 ～ r9）”に格納し，それを越えた分の引数は，呼び出し側のスタック・フレームに格納します。そして，r6 レジスタには戻り値を格納するアドレスを格納します。

9.2　戻り値の格納方法

(1) アセンブラ関数からの戻り値

CX は「関数の戻り値は“レジスタ r10”に格納される」ことを想定したコードを生成します。アセンブラ関数からの戻り値は r10 に格納するようにしてください。

なお，構造体を返す関数の場合は“呼び出し側の関数のスタック・フレーム内”に戻り値である構造体が格納されます。

(2) C 言語関数からの戻り値

CX は「関数の戻り値は“レジスタ r10”に格納される」ことを想定したコードを生成します。C 言語関数からの戻り値を使用する場合は，r10 レジスタを参照してください。

なお，構造体を返す関数の場合は，呼び出し側の戻り値用の領域に値が格納され，その領域のアドレスを引数として渡す形のコードを出力します。呼び出し側であらかじめ戻り値用の領域を確保しておく必要があります。

9.3　C 言語からアセンブリ言語ルーチンの呼び出し

C 言語関数からアセンブラ関数を呼び出すときの注意点について説明します。

(1) 識別子について

CX では C ソース内で外部名，たとえば，関数や外部変数が記述された場合，それらの名前をアセンブラへ出力すると，先頭に“_（アンダースコア）”を付けた名前になります。

表 9―1　識別子について

C	アセンブラ
func1 ( )	_func1

アセンブラ命令で関数や外部変数を定義するときは，識別子の先頭に“_”をつけ，C 言語関数から参照するときは“_”を取った形で行ってください。

(2) スタック・フレームに関して

CX は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，C ソースからアセンブラ関数へ分岐後は，アセンブラ関数内では，SP の指すアドレスよりも下位のアドレス領域は自由に使用することができます。逆に上位のアドレス領域の内容を変更した場合，C 言語関数で使用していた領域を破壊することにつながり，以降の動作を保証できませんので注意が必要です。上記を回避するためには，アセンブラ関数の先頭で SP を変更してからスタックを使用してください。

ただし，その際は呼び出しの前後で SP の値が保持されるようにしてください。

また，アセンブラ関数内でレジスタ変数用レジスタを使用する場合は，アセンブラ関数の呼び出し前後でレジスタ値が保持されるようにしてください（使用前にレジスタ変数用レジスタの値を退避し，使用後は復帰してください）。

レジスタ変数用レジスタは，レジスタ・モードにより異なります。

表 9―2　レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

(3) C 言語関数への戻り先アドレス

CX は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。アセンブラ関数へ分岐するとき，lp に関数の戻り先アドレスが格納されているので，C 言語関数へ戻るときは“jmp [lp]”を実行してください。

9.4　アセンブリ言語から C 言語ルーチンの呼び出し

アセンブラ関数から C 言語関数を呼び出すときの注意点について説明します。

(1) スタック・フレームについて

CX は「スタック・ポインタ（SP）が，常にスタック・フレームの最下位アドレスを指している」ことを想定したコードを出力します。そのため，アセンブラ関数から C 言語関数へ分岐する前に，スタック領域中の未使用領域の上位アドレスを指すように SP を設定してください。これは下位アドレスの方向にスタック・フレームが取られるためです。

(2) 作業用レジスタ

CX は C 言語関数呼び出しの前後において，レジスタ変数用レジスタの値は保持しますが，作業用レジスタの値は保持しません。そのため，保持しなくてはならない値を作業用レジスタに割り当てたままにしないでください。

レジスタ変数用レジスタ，作業用レジスタは，レジスタ・モードにより異なります。

表 9―3　レジスタ変数用レジスタ

レジスタ・モード	レジスタ変数用レジスタ
22 レジスタ・モード	r25，r26，r27，r28，r29
26 レジスタ・モード	r23，r24，r25，r26，r27，r28，r29
32 レジスタ・モード	r20，r21，r22，r23，r24，r25，r26，r27，r28，r29

表 9―4　作業用レジスタ

レジスタ・モード	作業用レジスタ
22 レジスタ・モード	r10，r11，r12，r13，r14
26 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16
32 レジスタ・モード	r10，r11，r12，r13，r14，r15，r16，r17，r18，r19

(3) アセンブラ関数への戻り先アドレス

CX は「関数の戻り先アドレスは“リンク・ポインタ lp（r31）”に格納される」ことを想定したコードを生成します。C 言語関数へ分岐するとき，lp に関数の戻り先アドレスを格納する必要あります。

一般的には jarl 命令によって，C 言語関数へ分岐します。

9.5　他言語で定義された変数の参照

アセンブリ言語で定義した変数を C 言語側で参照する方法を以下に示します。

【C 言語のプログラム例】

```
extern  char    c;
extern  int     i;

void subf() {
    c = 'A';
    i = 4;
}
```

CX アセンブラでは，次のように行います。

```
    .public  _i
    .public  _c
    .dseg   SDATA
_i:
    .db4    0x0
_c:
    .db     0x0
```

第 10 章　注意事項

この章では，CX を用いる際に注意すべき点について説明します。

10. 1　フォルダ／パスの区切り

“\”と“/”の両方が区切りとみなされます。

10. 2　関数宣言／定義における K&R 形式との混在

関数の宣言と定義において，K&R 形式と ANSI 規格形式が混在している場合，K&R 形式における引数拡張処理の結果，CX によるコンパイル時にエラーとなる場合があります。

たとえば，次の例では，関数宣言を ANSI 規格で行っていますが，関数定義は K&R 形式で行っているため，引数の型に不整合が生じ，CX は“関数の再宣言”エラーを出力します。

【エラーとなる例】

```
void    func(int a, int b, float c);
/*ANSI 規格形式で宣言 */
/* 第 3 引数は float 型として宣言 */
      :
void func(a, b, c)
int     a, b;
float   c;
{
    /*K&R 形式で定義 */
    /* 第 3 引数は，K&R のデフォルトの拡張のため，double 型 */
      :
}
```

この例の場合，関数宣言で“void func ( );”として K&R 形式に統一する，または関数定義で“void func (int a, int b, float c);”として ANSI 規格形式に統一することにより，正常にコンパイルができます。

ただし，CX では，ANSI 規格の形式で統一することを推奨しています。

10. 3　ポジション・インディペンデントではないコード出力

CX は，基本的に，位置に依存しない（ポジション・インディペンデント）コードを出力します。ただし，「自動変数以外のポインタ型の変数に対する，数値以外の初期値による初期化指示」に対しては，次の例に示すようなコードを出力します。

例

```
【C 言語の記述】
char    *ptr = "test\n";
```

```
【出力コード】
LL20    .ds     (6)
LL20:
    .db     "test\n\0"
    .align  4
    .public _ptr, 4
_ptr:
    .db4    #LL20   -- ラベルの絶対アドレス参照
```

10.4 オプション指定によるライブラリ・ファイル検索

CX では，オプション（-L，-l）によるライブラリ・ファイルの検索[注]の結果，指定されたライブラリ・ファイルが存在しなくてもメッセージを表示しません。ただし，ライブラリ・ファイル名をコマンド・ライン，およびコマンド・ファイルで直接指定した場合はフォルダ，メッセージを表示します

注 -L オプションを指定しない場合，標準のライブラリ・フォルダ（*Version Folder* \ lib850e）で検索します。

例

```
> cx -Cf3507 a.c usr.lib
```

```
F0560001: 入力ファイル“usr.lib”をオープンできません。
```

10.5 volatile 修飾子

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外され，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。また，volatile 指定された変数のアクセス幅も変更されません。

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，冗長な命令と解釈されて最適化により命令が削除されることもあります。特に周辺 I/O レジスタへアクセスする変数や，割り込み処理で値が変更される変数，また，外部から値が変更される変数に対しては，volatile 指定する必要があります。ただし，CX では，#pragma ioreg 指令を使って周辺 I/O レジスタにアクセスする場合，内部的に volatile 指定されているコードが出力されるので，改めて volatile 修飾子をつけて宣言する必要ありません。

volatile 指定すべきところで指定されていなかった場合，次の現象が起こることがあります。

- 正しい計算結果が得られない
- ループ内で変数を使っていた場合，ループから抜け出せない

ただし，volatile 指定した変数を使用する際，ある区間でその変数の値が外部から変更されないことが自明な場合，volatile 指定されていない変数に，その値を代入してその変数を参照することにより，その変数が最適化され，実行速度が向上する可能性があります。

【volatile 指定しなかった場合のソースと出力コードの例】

“変数 a”，“変数 b”，および“変数 c”を volatile 指定しなかった場合，これらの変数がレジスタに割り付けられ，最適化されます。たとえば，この間に割り込みが入り，割り込み内で変数値を変更しても，値が反映されないことになります。

```
int a;
int b;
int c;

void func(void) {
    if(a <= 0) {
        b++;
    } else {
        c++;
    }
    b++;
    c++;
}
```

```
_func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    ld.w    $_a, r12
    cmp     r0, r12
    jgt     .L2
    ld.w    $_b, r11
    ld.w    $_c, r10
    add     1, r11
    jbr     .L3
.L2:
    ld.w    $_c, r10
    ld.w    $_b, r11
    add     1, r10
.L3:
    addi    1, r11, r13
    st.w    r13, $_b
    addi    1, r10, r14
    st.w    r14, $_c
    #@B_EPILOGUE
    jmp     [lp]
    #@E_EPILOGUE
```

【volatile 指定した場合のソースと出力コードの例】

“変数 a”，“変数 b”，および“変数 c”を volatile 指定した場合，これらの変数値を必ずメモリから読み込み，操作後にメモリへ書き込むコードが出力されます。たとえば，この間に割り込みが入り，割り込み内で変数値が変更されても，その変更が反映された結果を取得することができます（このような例の場合，割り込みのタイミングによっては，変数の操作区間内を割り込み禁止にするなどの処置が必要となります）。

volatile 指定をすると，メモリの読み込み／書き込み処理が入るため，volatile 指定しなかった場合よりもコード・サイズは大きくなります。

```
volatile int    a;
volatile int    b;
volatile int    c;

void func(void) {
    if(a <= 0) {
        b++;
    } else {
        c++;
    }
    b++;
    c++;
}
```

```
func:
    #@B_PROLOGUE
    #@E_PROLOGUE
    ld.w    $_a, r10
    cmp     r0, r10
    jgt     .L2
    ld.w    $_b, r11
    add     1, r11
    st.w    r11, $_b
    jbr     .L3
.L2:
    ld.w    $_c, r12
    add     1, r12
    st.w    r12, $_c
.L3:
    ld.w    $_b, r13
    add     1, r13
    st.w    r13, $_b
    ld.w    $_c, r14
    add     1, r14
    st.w    r14, $_c
    #@B_EPILOGUE
    jmp     [lp]
    #@E_EPILOGUE
```

10. 6　関数宣言での余計なかっこ

関数宣言で，かっこ“( )”が余計に記述されている場合，ANSI-C では次のように規定されていますが，CX では，エラーとなります。

例

```
typedef int Int;

void    f1((Int));
```

【ANSI-C での規定】

仮引数宣言内では，かっこで囲まれた型定義名は，単一の仮引数をもつ関数を指定する抽象宣言子と解釈する。宣言子の識別子を囲む冗長なかっことは解釈しない。

したがって，上記の例では，ANSI-C では次のように解釈されます。

```
void    f(int(*)(int));
```

かっこを余計に記述してしまった場合には，次のように不必要なかっこを削除してください。

例

```
typedef int Int;


void    f1(Int);
```

付録 A 索 引

【D】

【E】

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.10.01	―	初版発行

CubeSuite Ver.1.40 ユーザーズマニュアル
コーディング編（CX コンパイラ）

発行年月日　2010 年 10 月 1 日　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/inquiry

CubeSuite Ver.1.40

ルネサス エレクトロニクス株式会社

R20UT0259JJ0100