User's Manual

CC-RH

コンパイラ

ユーザーズマニュアル

対象リビジョン
V1.00.00 - V2.07.00

対象デバイス
RH850 ファミリ

対象CPUコア
G3M, G3K, G3MH, G3KH, G4MH

ルネサス エレクトロニクス
www.renesas.com

Rev.1.12　2024.12

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。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、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

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

   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合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いますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。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. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

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直接的、間接的に支配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。

(Rev.5.0-1 2020.10)

本社所在地

〒 135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。

このマニュアルの使い方

このマニュアルは，RH850 ファミリ用アプリケーション・システムを開発する際のコンパイラ（CC-RH）について説明します。

対象者

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

構成

このマニュアルは，大きく分けて次の内容で構成しています。

読み方

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

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

目次

1. 概　　説

本書は「RH850 ファミリ用 C コンパイラ」CC-RH V1.00 ～ V2.07 向けのユーザーズマニュアルです。
この章では，CC-RH の全体概要について説明します。

1.1 概　　要

CC-RH 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

CC-RH は，次の特長を備えています。

(1) 規格に準拠した言語仕様
C 言語仕様は，C90 および C99 規格に準拠しています。

(2) 高度な最適化
高度な最適化を適用し，従来の最適化に加えて大域最適化を実現します。
これにより，コード・サイズ，および実行速度に優れたコードを生成し，かつコンパイル時間も短縮されます。

(3) 高い移植性
既存コンパイラ（SuperH RISC engine C/C++ コンパイラ）向けプログラムからの移行をサポートします。
また，デバッグ情報には業界標準フォーマットである DWARF2/3 を採用しています。

(4) 多機能性
CS+ との連携による静的解析機能などを提供します。

1.3 著作権について

本ソフトウェアは LLVM 及び Protocol Buffers を利用しています。
・LLVM は University of Illinois at Urbana-Champaign が著作権を有します。
・Protocol Buffers は Google Inc. が著作権を有します。
その他のソフトウェア構成物はルネサスエレクトロニクス株式会社が著作権を有します。

1.4 ライセンスについて

コンパイラのライセンスは，ライセンス・マネージャにより管理します。
ご使用のライセンスに応じて，standard 版，または professional 版として動作します。
standard 版と professional 版については「1.5 standard 版と professional 版について」を参照してください。
ライセンスが確認できない場合には，無償評価版として動作します。
無償評価版については「1.6 無償評価版について」を参照してください。
ライセンスおよびライセンス・マネージャの詳細は，ライセンス・マネージャのユーザーズマニュアルを参照してください。
CC-RH V1.05 以降では，ライセンス・マネージャは V2.00 以降のバージョンをご使用ください。
RH850 G4 コア向けの開発を行う場合は，CC-RH V2 以降のライセンスをご使用ください。

1.5 standard 版と professional 版について

コンパイラのエディションとして，standard 版と professional 版の 2 種類があります。
standard 版では，C90 および C99 規格に準拠した C 言語仕様をサポートし，組み込みプログラム記述に必要な基本機能を使用することができます。
professional 版では，standard 版に加えて，プログラムの品質向上と開発期間の短縮に貢献する付加機能を使用することができます。
professional 版の付加機能は，オプションまたは #pragma 指令により有効になります。
professional 版のみで使用可能なオプションは「表 2.2 コンパイル・オプション」，または各オプションの説明の項を参照してください。
professional 版のみでサポートしている #pragma 指令は「表 4.10 サポートしている #pragma 指令」を参照してください。
professional 版のみでサポートしているライブラリは「7.1 提供ライブラリ」を参照してください。

1.6 無償評価版について

無償評価版には試用期間（コンパイラの初回起動から 60 日）があり，試用期間内は，professional 版と同等の機能を使用することができます。
試用期間後は，professional 版の付加機能を使用できないほか，リンクサイズに制限があります。

- リンクサイズの制限は，ROM 領域に配置されるセクション・サイズの合計が 256K バイトまでになります。
  256K バイトを超えた場合はリンカエラーになります。

無償評価版として動作している場合は最適化リンカのバージョン表記が W，製品版の場合はバージョン表記が V となります。
以下に出力例を示します。

- 無償評価版のバージョン表記例
  Renesas Optimizing Linker W1.01.01 [25 Apr 2014]
- 製品版のバージョン表記例
  Renesas Optimizing Linker V1.01.01 [25 Apr 2014]

無償評価版では，以下サービス提供の対象外となります。以下のサービスが必要な場合には，製品版の購入をご検討ください。

- 技術的なお問い合わせに対するサポート
- リビジョンアップ情報などの案内メール送信

2. コマンド・リファレンス

ここでは，ビルド・ツール（CC-RH）に含まれる各コマンドの仕様についての詳細を説明します。

2.1 概　　要

CC-RH は，C 言語やアセンブリ言語で記述したソース・プログラムから，ターゲット・システムで実行可能なファイルを生成します。

以下のコマンドで構成されており，1 つのコンパイル・ドライバ（ccrh）がコンパイルからリンクまでの全フェーズを制御します。

ccrh　：コンパイル・ドライバの起動コマンド
asrh　：アセンブラの起動コマンド
rlink　：最適化リンカの起動コマンド

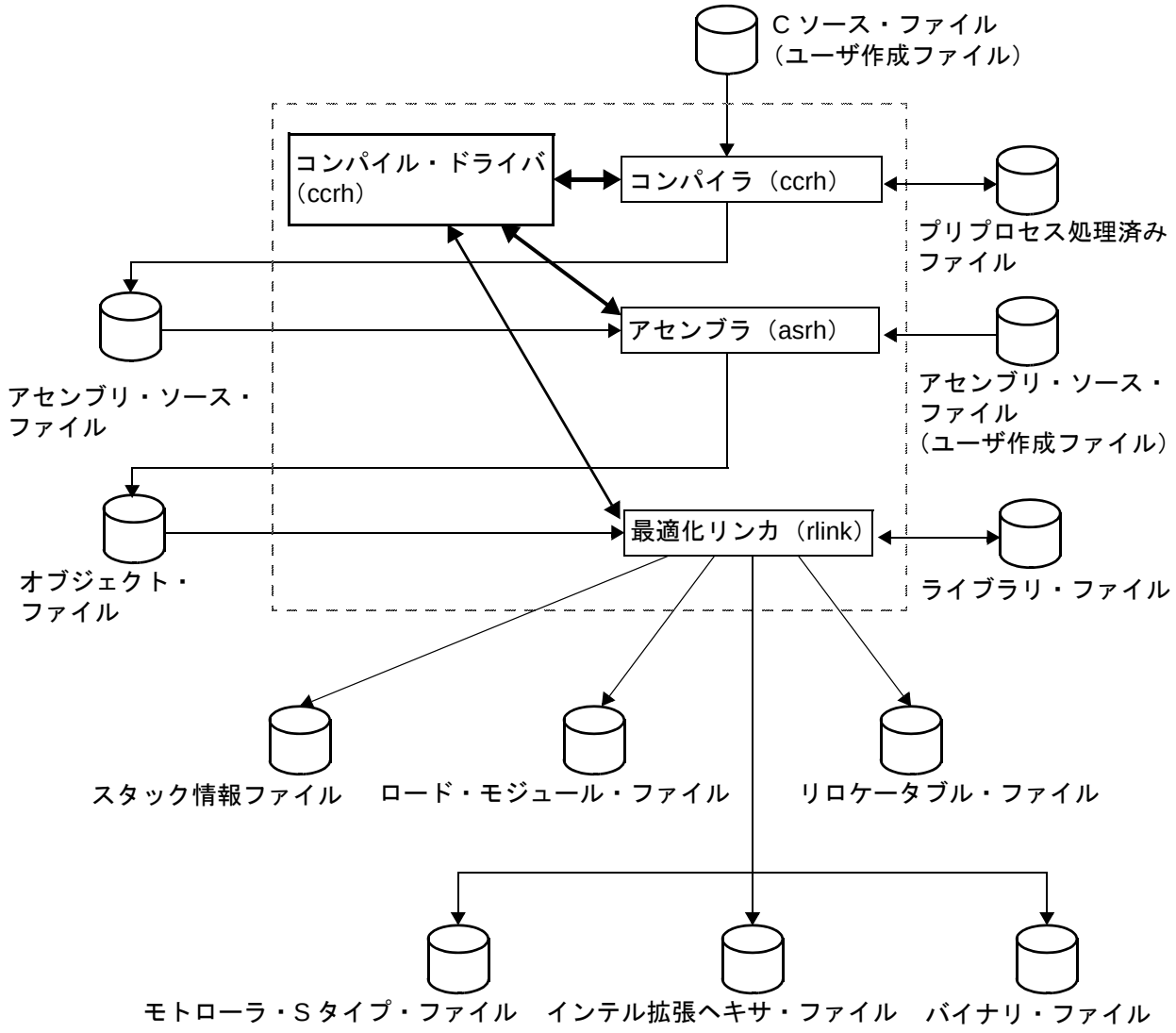
各コマンドの処理について説明します。

(1) コンパイラ（ccrh）
C ソース・プログラムに対して，プリプロセス指令の処理，コメント処理，最適化を行い，アセンブリ・ソース・プログラムを生成します。

(2) アセンブラ（asrh）
アセンブリ・ソース・プログラムを機械語命令に変換して，再配置可能なオブジェクト・ファイルを生成します。

(3) 最適化リンカ（rlink）
オブジェクト・ファイル，ライブラリ・ファイルをリンクし，ターゲット・システムで実行可能なオブジェクト・ファイル（ロード・モジュール・ファイル）を生成します。
また，組み込みアプリケーション向けの ROM イメージ作成支援，リロケータブル・ファイル結合時の最適化，ライブラリ・ファイルの作成や編集，インテル拡張ヘキサ・ファイルやモトローラ・S タイプ・ファイルへの変換を行います。

図 2.1 ccrh における処理の流れ



2.2 入出力ファイル

コンパイル・ドライバである ccrh コマンドの入出力ファイルを以下に示します。

表 2.1 ccrh コマンドの入出力ファイル

ファイル種別	拡張子	入出力	説明
C ソース・ファイル	.c	入力	C 言語で記述したソース・ファイル ユーザ作成ファイルです。
プリプロセス処理済みファイル	.i 注 1	出力	入力ファイルに対してプリプロセス処理を実行した結果を出力したファイル ASCII イメージ・ファイルです。 -P オプション指定時に出力します。
アセンブリ・ソース・ファイル	.asm 注 1	出力	コンパイルにより C ソースから生成したアセンブリ言語ファイル -S オプション指定時に出力します。
	.asm .s	入力	アセンブリ言語で記述したソース・ファイル ユーザ作成ファイルです。

ファイル種別	拡張子	入出力	説明
ヘッダ・ファイル	任意	入力	ソース・ファイルで参照するファイル C 言語，またはアセンブリ言語で記述したファイルです。 ユーザ作成ファイルです。 拡張子は任意ですが，以下を推奨します。 - #include 指令：.h - $include 制御命令：.inc
オブジェクト・ファイル	.obj 注 1	入出力	機械語情報と機械語の配置アドレスに関する再配置情報，およびシンボル情報を含んだ ELF 形式ファイル
アセンブル・リスト・ファイル注 2	.prn 注 1	出力	アセンブル結果の情報を持つリスト・ファイル -Xasm_option=-Xprn_path オプション指定時に出力します。
ライブラリ・ファイル	.lib 注 1	入出力	複数のオブジェクト・ファイルが登録された ELF 形式ファイル -Xlk_option=-form=library オプション指定時に出力します。
ロード・モジュール・ファイル	.abs 注 1	入出力	リンク結果のオブジェクト・コードの ELF 形式ファイル ヘキサ・ファイルを出力する際の入力ファイルとなります。 -Xlk_option=-form=absolute オプション指定時に出力します。 ただし，-Xlk_option オプションで -form オプションが指定されていない場合は，上記オプションが指定されたものと解釈して処理します。
リロケータブル・ファイル	.rel 注 1	出力	リロケータブルなオブジェクト・ファイル -Xlk_option=-form=relocate オプション指定時に出力します。
インテル拡張ヘキサ・ファイル注 2	.hex 注 1	入出力	ロード・モジュール・ファイルをインテル拡張ヘキサ・フォーマットに変換したファイル -Xlk_option=-form=hexadecimal オプション指定時に出力します。
モトローラ・S タイプ・ファイル注 2	.mot 注 1	入出力	ロード・モジュール・ファイルをモトローラ・S タイプに変換したファイル -Xlk_option=-form=stype オプション指定時に出力します。
バイナリ・ファイル	.bin 注 1	出力	ロード・モジュール・ファイルをバイナリ・フォーマットに変換したファイル -Xlk_option=-form=binary オプション指定時に出力します。
シンボル・アドレス・ファイル	.fsy	入出力	外部定義シンボルをアセンブラ制御命令で記述したアセンブリ・ソース・ファイル -Xlk_option=-fsymbol オプション指定時に出力します。
リンク・マップ・ファイル注 2	.map 注 1	出力	リンク結果の情報を持つリスト・ファイル -Xlk_option=-list オプション指定時に出力します。
ライブラリ・リスト・ファイル注 2	.lbp 注 1	出力	ライブラリ作成結果の情報を持つリスト・ファイル -Xlk_option=-list オプション指定時に出力します。
スタック情報ファイル	.sni	出力	スタック使用量の情報を持つファイル -Xlk_option=-stack オプション指定時に出力します。

ファイル種別	拡張子	入出力	説明
外部シンボル割り付け情報ファイル	.bls 注 1	出力	コンパイラが外部変数アクセス最適化で使用する外部変数割り付け情報ファイル -Omap オプション指定時に出力します。
静的解析情報ファイル	任意	入出力	本製品が利用する情報を持つファイル 拡張子は任意ですが，“.cref” を推奨します。 -Xcref オプション指定時に出力します。
エラー・メッセージ・ファイル	任意	出力	エラー・メッセージを内容とするファイル 拡張子は任意ですが，“.err” を推奨します。 -Xerror_file オプション指定時に出力します。
サブコマンド・ファイル	任意	入力	実行プログラムのパラメータを内容とするファイル ユーザ作成ファイルです。
排他制御チェック設定ファイル	任意	入力	CS+ から入力するファイルです。
ツール使用情報ファイル	.ud .udm	出力	ツールの使用情報を収集するために出力するファイルです。

注 1. オプション指定により拡張子の変更が可能です。

注 2. 各ファイルについての詳細は，「3. 出力ファイル」を参照してください。

2.3 環境変数

ここでは，環境変数について説明します。
最適化リンカの環境変数とコマンド・ラインでの指定例を以下に示します。

- HLNK_LIBRARY1，HLNK_LIBRARY2，HLNK_LIBRARY3
  最適化リンカが使用するデフォルト・ライブラリ名を指定します。
  ライブラリは，-library オプションで指定したものを優先してリンクします。
  その後，未解決のシンボルがある場合，HLNK_LIBRARY1，HLNK_LIBRARY2，HLNK_LIBRARY3 の順に，デフォルト・ライブラリを検索します。

  例

```
>set HLNK_LIBRARY1=usr1.lib
>set HLNK_LIBRARY2=usr2.lib
>set HLNK_LIBRARY3=usr3.lib
```

- HLNK_TMP
  最適化リンカがテンポラリ・ファイルを作成するフォルダを指定します。
  この環境変数の指定がない場合は，カレント・フォルダに作成します。

  例

```
>set HLNK_TMP=D:¥workspace¥tmp
```

- HLNK_DIR
  最適化リンカの入力ファイル格納フォルダを指定します。
  -input オプション，-library オプションで指定したファイルの検索順序は，カレント・フォルダ，HLNK_DIR 指定フォルダとなります。
  ただし，ワイルドカードで指定したファイルは，カレント・フォルダ内だけを検索します。

  例

```
>set HLNK_DIR=D:¥workspace¥obj1;D:¥workspace¥obj2
```

2.4 操作方法

ここでは，各コマンドの操作方法について説明します。

- コマンド・ラインでの操作方法
- サブコマンド・ファイルの使用方法

2.4.1 コマンド・ラインでの操作方法

コンパイル・ドライバである ccrh コマンドの起動により，コンパイル，アセンブル，リンクなどを行うことができます。
また，アセンブラ（asrh），最適化リンカ（rlink）は単体起動することも可能です。

(1) 指定形式
各コマンドは，コマンド・ラインで以下のように入力します。

```
>ccrh[ △ option]... △ file[ △ file| △ option]...
```

```
>asrh[ △ option]... △ file[ △ file| △ option]...
```

```
>rlink[{ △ file| △ option}...]
```

option	:	オプション名
file	:	ファイル名
[ ]	:	[ ] 内は省略可能
...	:	直前の [ ] 内のパターンの繰り返しが可能
{ }	:	| で区切られた項目を選択
△	:	1 個以上の空白
[, ...]	:	カンマで区切って，直前のパターンの繰り返すことが可能
[: ...]	:	コロンで区切って，直前のパターンの繰り返すことが可能
string := *A*	:	*string* を，*A* で置換する
string := *A* | *B* | *C*	:	*string* を *A*，*B*，*C* のいずれか 1 つで置換する

コマンドを入力する際の注意事項を以下に示します。

- オプションの指定形式は，使用するコマンドによって異なります。
  各コマンドのオプションに対する注意事項については，「2.5.1 コンパイル・オプション」，「2.5.2 アセンブル・オプション」，「2.5.3 リンク・オプション」を参照してください。
- ファイル名は，OS で認められるものであれば指定可能です。
  相対パス，もしくはドライブ名からはじまる絶対パスで指定します。
  ただし，“@” はサブコマンド・ファイル指定と判断するため，“@” をファイル名の先頭文字として使用することはできません。
  また，“-” はオプション指定と判断するため，“-” もファイル名の先頭文字として使用することはできません。
  さらに，“(”，“)” はリンク・オプションの一部と判断するため，“(”，“)” をファイル名に使用することはできません。
  そして以下の文字も使用できません。

OS	使用できない文字
Windows	"^|;*?<>¥/:
Linux	"$&'()^|[];<>`¥/

  この他にも，内部処理でサブコマンド・ファイルを使用するため，ファイル名やパス名の使用に注意が必要な文字があります。
  「2.4.2 サブコマンド・ファイルの使用方法」も合わせてご確認ください。
- 指定可能なファイル名の長さは，OS に依存します（Windows では，259 文字まで）。
- Windows では，ファイル名のアルファベットは，大文字／小文字を区別しません。

- 入力として複数のファイルが指定可能です。
  異なる種類のファイル（C ソース・ファイルとアセンブリ・ソース・ファイル，またはオブジェクト・ファイルなど）を混在して指定することも可能です。
  ただし，拡張子を除いたソース・ファイル名が同じファイルを複数指定することはできません（異なるフォルダにある場合も含みます）。
  複数のファイルを指定した場合は，1 つのファイルでエラーとなっても，残りのファイルの処理が継続可能であれば，処理を継続します。
  なお，生成したオブジェクト・ファイルはリンク後も削除しません。

(2) 操作例
コマンド・ラインでの操作例を以下に示します。

備考 各オプションについての詳細は，「2.5 オプション」を参照してください。

(a) コンパイル，アセンブル，リンクを 1 コマンドで行う場合
C ソース・ファイル file1.c を ccrh でコンパイルして，アセンブリ・ソース・ファイル file1.asm を生成します。
次に，アセンブリ・ソース・ファイル file1.asm と file2.asm を asrh でアセンブルして，オブジェクト・ファイル file1.obj，file2.obj を生成します。
また，アセンブル・リスト・ファイルをカレント・フォルダに出力します。
最後に，オブジェクト・ファイル file1.obj，file2.obj，file3.obj を rlink でリンクして，リンク・マップ・ファイル sample.map，およびロード・モジュール・ファイル sample.abs を生成します。

```
>ccrh file1.c file2.asm file3.obj -Xasm_option=-Xprn_path -Xlk_option=-list -
osample.abs -Xcommon=rh850
```

備考 ccrh に対して，asrh のみに指定可能なオプションを指定する場合は -Xasm_option オプション，rlink のみに指定可能なオプションを指定する場合は -Xlk_option オプションを使用します。

(b) コンパイル，アセンブルを 1 コマンドで行い，リンクは単独で行う場合
C ソース・ファイル file1.c を ccrh でコンパイルして，アセンブリ・ソース・ファイル file1.asm を生成します。
次に，アセンブリ・ソース・ファイル file1.asm と file2.asm を asrh でアセンブルして，オブジェクト・ファイル file1.obj，file2.obj を生成します。
また，アセンブル・リスト・ファイルをカレント・フォルダに出力します。

```
>ccrh -c file1.c file2.asm -Xasm_option=-Xprn_path -Xcommon=rh850
```

備考 ccrh に対して，asrh のみに指定可能なオプションを指定する場合は -Xasm_option オプションを使用します。

オブジェクト・ファイル file1.obj，file2.obj，file3.obj を rlink でリンクして，リンク・マップ・ファイル sample.map，およびロード・モジュール・ファイル sample.abs を生成します。

```
>rlink file1.obj file2.obj file3.obj -output=sample.abs -list
```

(c) コンパイル，アセンブル，リンクともに単独で行う場合
C ソース・ファイル file1.c を ccrh でコンパイルして，アセンブリ・ソース・ファイル file1.asm を生成します。

```
>ccrh -S file1.c -Xcommon=rh850
```

アセンブリ・ソース・ファイル file1.asm と file2.asm を asrh でアセンブルして，オブジェクト・ファイル file1.obj，file2.obj を生成します。
また，アセンブル・リスト・ファイルをカレント・フォルダに出力します。

```
>asrh file1.asm -Xprn_path -Xcommon=rh850
>asrh file2.asm -Xprn_path -Xcommon=rh850
```

オブジェクト・ファイル file1.obj，file2.obj，file3.obj を rlink でリンクして，リンク・マップ・ファイル sample.map，およびロード・モジュール・ファイル sample.abs を生成します。

```
>rlink file1.obj file2.obj file3.obj -output=sample.abs -list
```

2.4.2 サブコマンド・ファイルの使用方法

サブコマンド・ファイルとは，コマンドに対して指定するオプションやファイル名を記述したファイルです。
コマンドは，サブコマンド・ファイルの内容をコマンド・ラインの引数のように扱います。
サブコマンド・ファイルは，コマンド・ラインで引数を指定しきれない場合や，コマンドを実行するたびに同じオプションを繰り返し指定するような場合に使用します。

(1) コンパイラ，アセンブラでサブコマンド・ファイルを使用する場合

(a) サブコマンド・ファイルの記述に関する注意事項

- 指定する引数は，複数行に分けて記述することができます。
  ただし，オプションやファイル名の途中で改行することはできません。
- サブコマンド・ファイル内でサブコマンド・オプションを指定する場合，同名のサブコマンド・ファイルを指定することはできません。
- サブコマンド・ファイルの文字コードは，-Xcharacter_set オプションで指定することはできません。
  サブコマンド・ファイル中に ASCII 文字以外を使用する場合は，BOM 付き UTF-8，または SJIS のファイルにしてください。
- 以下の文字は，特殊文字として扱います。
  これらの特殊文字自体は，ccrh コマンドへのコマンド・ラインの中に含まずに削除します。

"（ダブルクォーテーション）	次のダブルクォーテーションまでの文字列を連続した文字列として扱います。
#（シャープ）	行頭に指定した場合は，行末までをコメントとして扱います。
^（ハット）	直後の文字を特殊文字として扱いません。

(b) サブコマンド・ファイルの指定例
サブコマンド・ファイル sub.txt をエディタで作成します。

```
-Xcommon=rh850
-c
-Dtest
-Idir
-Osize
```

コマンド・ラインにおいて，サブコマンド・ファイル指定オプション @ により sub.txt を指定します。

```
>ccrh @sub.txt -ofile.obj file.c
```

コマンド・ラインは以下のように展開されます。

```
>ccrh -Xcommon=rh850 -c -Dtest -Idir -Osize -ofile.obj file.c
```

(2) 最適化リンカでサブコマンド・ファイルを使用する場合

(a) サブコマンド・ファイルの記述に関する注意事項

- オプションは，先頭のハイフン（-）を省略して指定することができます。
- オプションとパラメータの区切りは，“=” の代わりに空白を使用することも可能です。
- 1 行につき 1 つのオプションを指定します。
  1 行に記述できない場合は，“&” を使用して複数行に記述することができます。
- サブコマンド・ファイル内にサブコマンド・オプションを指定することはできません。【V1.04.00 以前】
- サブコマンド・ファイル内でサブコマンド・オプションを指定する場合，同名のサブコマンド・ファイルを指定することはできません。【V1.05.00 以降】
- 以下の文字は，特殊文字として扱います。
  これらの特殊文字自体は，rlink コマンドへのコマンド・ラインの中に含まずに削除します。

&（アンド）	次の行を継続行として扱います。

;（セミコロン）	行末までをコメントとして扱います。

(b) サブコマンド・ファイルの指定例
サブコマンド・ファイル sub.txt をエディタで作成します。

```
input file2.obj file3.obj       ; ここはコメントです。
library lib1.lib, &             ;“&” は継続行を示します。
lib2.lib
```

コマンド・ラインにおいて，サブコマンド・ファイル指定オプション -subcommand により sub.txt を指定します。

```
>rlink file1.obj -subcommand=sub.txt file4.obj
```

コマンド・ラインは以下のように展開されます。

```
>rlink file1.obj file2.obj file3.obj -library=lib1.lib,lib2.lib file4.obj
```

2.5 オプション

ここでは，ccrh のオプションについて，各フェーズごとに説明します。
コンパイル・フェーズ→「2.5.1 コンパイル・オプション」参照
アセンブル・フェーズ→「2.5.2 アセンブル・オプション」参照
リンク・フェーズ→「2.5.3 リンク・オプション」参照

2.5.1 コンパイル・オプション

ここでは，コンパイル・フェーズのオプションについて説明します。

オプションに関する注意事項を以下に示します。

- オプションの大文字／小文字は区別します。
- パラメータとして数値を指定する場合は，10 進数，または “0x”（“0X”）で始まる 16 進数での指定が可能です。16 進数のアルファベットは，大文字／小文字を区別しません。
- パラメータとしてファイル名を指定する場合は，パス付き（絶対パス，または相対パス）での指定が可能です。パスなし，および相対パスで指定する場合は，カレント・フォルダを基準とします。
- パラメータ中に空白を含める場合（パス名など）は，そのパラメータ全体をダブルクォーテーション（"）で囲んでください。

オプションの分類と説明を以下に示します。

表 2.2 コンパイル・オプション

分類	オプション	説明
バージョン／ヘルプ表示指定	-V	ccrh のバージョン情報を表示します。
	-h	ccrh のオプションの説明を表示します。
出力ファイル指定	-o	出力ファイル名を指定します。
	-Xobj_path	コンパイル途中に生成されるオブジェクト・ファイルを保存するフォルダを指定します。
	-Xasm_path	コンパイル途中に生成されるアセンブリ・ソース・ファイルを保存するフォルダを指定します。
	-Xprep_path	プリプロセス処理済みファイルを保存するフォルダを指定します。
ソース・デバッグ制御	-g	ソース・デバッグ用の情報を出力します。
	-g_line 【V1.05.00 以降】	最適化時にソース・デバッグ用の情報を強化します。
デバイス指定	-Xcommon	デバイス共通のオブジェクト・ファイルを生成することを指定します。
	-Xcpu	指定したコア向けのオブジェクトを生成することを指定します。
処理中断指定	-P	入力した C ソース・ファイルに対してプリプロセス処理のみ実行します。
	-S	アセンブル以降の処理を実行しません。
	-c	リンク以降の処理を実行しません。
プリプロセッサ制御	-D	プリプロセッサ・マクロ，およびアセンブラ・シンボルを定義します。
	-U	-D オプションによるプリプロセッサ・マクロ，およびアセンブラ・シンボルの定義を解除します。
	-I	インクルード・ファイルを検索するフォルダを指定します。
	-Xpreinclude	コンパイル単位の先頭にインクルードするファイルを指定します。
	-Xpreprocess	プリプロセス結果の出力を制御します。

分類	オプション	説明
C 言語制御	-lang 【V1.07.00 以降】	言語規格を指定します。
	-strict_std 【V1.07.00 以降】	C ソース・プログラムを言語規格に厳密にあわせて処理します。
	-Xenum_type	列挙型に対して，どの整数型として扱うかを指定します。
	-Xvolatile	外部変数の volatile 化を指定します。
	-Xcheck	C ソース・ファイルの互換性をチェックします。
	-Xmisra2004 【Professional 版のみ】	MISRA-C:2004 ルールによるソース・チェックを行います。
	-Xmisra2012 【Professional 版のみ】	MISRA-C:2012 ルールによるソース・チェックを行います。
	-Xignore_files_misra 【Professional 版のみ】	MISRA-C:2004 または MISRA-C:2012 ルールによるソース・チェックの対象外のファイルを指定します。
	-Xcheck_language_extension 【Professional 版のみ】	言語拡張により部分抑止される MISRA-C:2004 または MISRA-C:2012 ルールのソース・チェックを有効にします。
	-misra_intermodule 【Professional 版のみ】【V2.01.00 以降】	複数ファイルにまたがる MISRA-C:2012 ルールによるソース・チェックを行います。
	-Xuse_fp16 【Professional 版のみ】【V1.05.00 以降】	半精度浮動小数点型の使用を指定します。
日本語／中国語文字列制御	-Xcharacter_set	日本語／中国語の文字コードを指定します。
最適化指定	-O	最適化のレベル，または各最適化項目の詳細を指定します。
	-Xintermodule	大域最適化を行います。
	-Xinline_strcpy	標準ライブラリ関数 strcpy，strcmp，memcpy，memset の呼び出しをインライン展開します。
	-Xmerge_string	文字列定数をマージします。
	-Xalias	ポインタ指示先の型を考慮した最適化の指定を行います。
	-Xmerge_files	複数の C ソース・ファイルをマージしてコンパイルを行います。
	-Xwhole_program	コンパイル対象ファイルがプログラム全体であることを仮定して最適化を行います。
	-library 【V2.00.00 以降】	標準ライブラリ関数の呼び出しをインライン展開します。
	-goptimize 【V2.01.00 以降】	リンク時最適化用の情報を生成します。

分類	オプション	説明
生成コード制御	-Xpack	構造体パッキングを行います。
	-misalign 【V2.04.00 以降】	ミスアライン・メモリ・アクセスを行う命令列を生成します。
	-Xbit_order	ビット・フィールドのメンバの並び順を指定します。
	-Xpass_source	出力するアセンブリ・ソース・ファイル中に C ソース・プログラムをコメントとして出力します。
	-Xswitch	switch 文のコード出力方式を指定します。
	-Xreg_mode	レジスタ・モードを指定します。
	-Xreserve_r2	r2 レジスタを予約します。
	-r4 【V1.07.00 以降】	r4 レジスタの扱い方を指定します。
	-Xep	ep レジスタの扱い方を指定します。
	-Xfloat	浮動小数点演算命令の生成を制御します。
	-Xfxu 【V2.00.00 以降】	FXU 命令の使用を制御します。
	-Xcall_jump	関数呼び出しの分岐命令の生成を制御します。
	-Xfar_jump	far jump の出力を制御します。
	-Xdiv	除算に対して，div，および divu 命令を生成します。
	-Xcheck_div_ov	除算時に OV フラグのチェックを行います。
	-relaxed_math 【V2.00.00 以降】	厳密さより効率を重視した浮動小数点演算コードを生成します。
	-Xuse_fmaf	積和演算命令を生成します。
	-use_recipf 【V2.00.00 以降】	recipf 命令を生成します。
	-approximate 【V2.02.00 以降】	浮動小数点演算を，同等の近似計算で置き換えます。
	-Xunordered_cmpf	浮動小数点比較において，無効演算例外の検出を行います。
	-Xmulti_level	マルチコア用プログラムの生成を指定します。
	-Xpatch	パッチを適用します。
	-Xdbl_size	double 型および long double 型の精度を指定します。
	-Xround	浮動小数点定数の丸めモードを指定します。
	-Xalign4	分岐先アドレスのアライメントを指定します。
	-Xstack_protector/ -Xstack_protector_all 【Professional 版のみ】	スタック破壊検出コード生成を指定します。
	-Xsection	データのデフォルト・セクションを指定します。
	-stuff 【V2.03.00 以降】	変数をアライメント数に応じたセクションに分けて配置します。
	-Xcheck_exclusion_control 【V1.04.00 以降】	排他制御チェックを有効にします。
	-Xresbank_mode 【V2.00.00 以降】	resbank 命令の動作モードを指定します。

分類	オプション	説明
生成コード制御	-insert_dbtag_with_label【V1.06.00 以降】	dbtag 命令の挿入を制御します。
	-store_reg 【Professional 版のみ】【V1.06.00 以降】	制御レジスタへの書き込み処理の検出や，レジスタ間の同期化処理の挿入を制御します。
	-control_flow_integrity【Professional 版のみ】【V1.07.00 以降】	不正な間接関数呼び出しを検出するコードを生成します。
	-pic 【V1.07.00 以降】	PIC 機能を有効にします。
	-pirod 【V1.07.00 以降】	PIROD 機能を有効にします。
	-pid 【V1.07.00 以降】	PID 機能を有効にします。
情報ファイル出力制御	-Xcref	静的解析情報ファイルを出力します。
エラー出力制御	-Xerror_file	エラー・メッセージをファイルに出力します。
警告メッセージ出力制御	-Xno_warning	指定した警告メッセージの出力を抑止します。
	-change_message【V1.07.00 以降】	指定した警告メッセージをエラーメッセージに変更します。
フェーズ個別オプション指定	-Xasm_option	アセンブル・オプションを指定します。
	-Xlk_option	リンク・オプションを指定します。
サブコマンド・ファイル指定	@	サブコマンド・ファイルを指定します。

バージョン／ヘルプ表示指定

バージョン／ヘルプ表示指定オプションには，次のものがあります。

- -V
- -h

-V

ccrh のバージョン情報を表示します。

[指定形式]

```
-V
```

- 省略時解釈
  ccrh のバージョン情報を表示せずに，コンパイルを行います。

[詳細説明]

- ccrh のバージョン情報を標準エラー出力に出力します。
  コンパイルは行いません。

[使用例]

- ccrh のバージョン情報を標準エラー出力に出力します。

```
>ccrh -V -Xcommon=rh850
```

-h

ccrh のオプションの説明を表示します。

[指定形式]

```
-h
```

- 省略時解釈
  ccrh のオプションの説明を表示しません。

[詳細説明]

- ccrh のオプションの説明を標準エラー出力に出力します。
  コンパイルは行いません。

[使用例]

- ccrh のオプションの説明を標準エラー出力に出力します。

```
>ccrh -h -Xcommon=rh850
```

出力ファイル指定

出力ファイル指定オプションには，次のものがあります。

- -o
- -Xobj_path
- -Xasm_path
- -Xprep_path

-o

出力ファイル名を指定します。

[指定形式]

```
-ofile
```

- 省略時解釈
  出力ファイル名は，指定オプションにより異なります。
  なお，ファイルの出力先はカレント・フォルダとなります。
  - -P オプションを指定している場合
    出力ファイル名は，入力ファイル名の拡張子を “.i” に置き換えたものとなります。
  - -S オプションを指定している場合
    出力アセンブリ・ソース・ファイル名は，ソース・ファイル名の拡張子を “.asm” に置き換えたものとなります。
  - -c オプションを指定している場合
    出力オブジェクト・ファイル名は，ソース・ファイル名の拡張子を “.obj” に置き換えたものとなります。
  - 上記以外の場合
    出力ロード・モジュール・ファイル名は，最初に入力したファイル名の拡張子を “.abs” に置き換えたものとなります。

[詳細説明]

- 出力ファイル名を *file* に指定します。
- *file* がすでに存在する場合は，そのファイルを上書きします。
- -P/-S/-c オプションと同時に指定することにより処理を中断した場合にも，本オプションは有効となります。
  - -P オプションと同時に指定した場合
    file には，入力ファイルに対してプリプロセス処理を行った結果のファイル名を指定したものとみなします。
  - -S オプションと同時に指定した場合
    file には，アセンブリ・ソース・ファイル名を指定したものとみなします。
  - -c オプションと同時に指定した場合
    file には，オブジェクト・ファイル名を指定したものとみなします。
  - 上記以外の場合
    file には，ロード・モジュール・ファイル名を指定したものとみなします。
- 出力ファイルが複数の場合は，エラーとなります。
- *file* を省略した場合は，エラーとなります。

[使用例]

- ロード・モジュール・ファイルをファイル名 sample.abs で出力します。

```
>ccrh -osample.abs -Xcommon=rh850 main.c
```

-Xobj_path

コンパイル途中に生成されるオブジェクト・ファイルを保存するフォルダを指定します。

[指定形式]

```
-Xobj_path[=path]
```

- 省略時解釈
  カレント・フォルダに，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。

[詳細説明]

- コンパイル途中に生成されるオブジェクト・ファイルを保存するフォルダを *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。
  存在しないフォルダ名を指定した場合は，エラーとなります。
- *path* には存在するファイル名を指定することも可能です。
  出力するオブジェクト・ファイルが 1 つの場合は，*path* というファイル名で保存します。
  出力するオブジェクト・ファイルが複数の場合は，エラーとなります。
  存在しないファイル名を指定した場合は，エラーとなります。
- =*path* を省略した場合は，カレント・フォルダに，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。
- ソース・ファイルとして同じ名前のファイル（異なるフォルダにある場合を含む）を複数指定した場合は，警告を出力して，最後に指定したソース・ファイルに対するオブジェクト・ファイルのみを保存します。

[使用例]

- コンパイル途中に生成されるオブジェクト・ファイルをフォルダ D:¥sample に保存します。

```
>ccrh -Xobj_path=D:¥sample -Xcommon=rh850 main.c
```

-Xasm_path

コンパイル途中に生成されるアセンブリ・ソース・ファイルを保存するフォルダを指定します。

[指定形式]

```
-Xasm_path[=path]
```

- 省略時解釈
  アセンブリ・ソース・ファイルを出力しません（-S オプション指定時を除く）。

[詳細説明]

- コンパイル途中に生成されるアセンブリ・ソース・ファイルを保存するフォルダを *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，C ソース・ファイル名の拡張子を “.asm” で置き換えたファイル名でアセンブリ・ソース・ファイルを保存します。
  存在しないフォルダ名を指定した場合は，エラーとなります。
- *path* には存在するファイル名を指定することも可能です。
  出力するアセンブリ・ソース・ファイルが 1 つの場合は，*path* というファイル名で保存します。
  出力するアセンブリ・ソース・ファイルが複数の場合は，エラーとなります。
  存在しないファイル名を指定した場合は，エラーとなります。
- =*path* を省略した場合は，カレント・フォルダに，C ソース・ファイル名の拡張子を “.asm” で置き換えたファイル名でアセンブリ・ソース・ファイルを保存します。
- ソース・ファイルとして同じ名前のファイル（異なるフォルダにある場合を含む）を複数指定した場合は，警告を出力して，最後に指定したソース・ファイルに対するアセンブリ・ソース・ファイルのみを保存します。

[使用例]

- コンパイル途中に生成されるアセンブリ・ソース・ファイルをフォルダ D:¥sample に保存します。

```
>ccrh -Xasm_path=D:¥sample -Xcommon=rh850 main.c
```

-Xprep_path

プリプロセス処理済みファイルを保存するフォルダを指定します。

[指定形式]

```
-Xprep_path[=path]
```

- 省略時解釈
  プリプロセス処理済みファイルを出力しません（-P オプション指定時を除く）。

[詳細説明]

- -P オプションを指定したとき，プリプロセス処理済みファイルを保存するフォルダを *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，C ソース・ファイル名の拡張子を “.i” で置き換えたファイル名でプリプロセス処理済みファイルを保存します。
  存在しないフォルダ名を指定した場合は，エラーとなります。
- *path* には存在するファイル名を指定することも可能です。
  出力するプリプロセス処理済みファイルが 1 つの場合は，*path* というファイル名で保存します。
  出力するプリプロセス処理済みファイルが複数の場合は，エラーとなります。
  存在しないファイル名を指定した場合は，エラーとなります。
- =*path* を省略した場合は，カレント・フォルダに，C ソース・ファイル名の拡張子を “.i” で置き換えたファイル名でプリプロセス処理済みファイルを保存します。
- ソース・ファイルとして同じ名前のファイル（異なるフォルダにある場合を含む）を複数指定した場合は，警告を出力して，最後に指定したソース・ファイルに対するプリプロセス処理済みファイルのみを保存します。

[使用例]

- プリプロセス処理済みファイルをフォルダ D:¥sample に保存します。

```
>ccrh -Xprep_path=D:¥sample -Xcommon=rh850 main.c
```

ソース・デバッグ制御

ソース・デバッグ制御オプションには，次のものがあります。

- -g
- -g_line 【V1.05.00 以降】

-g

ソース・デバッグ用の情報を出力します。

[指定形式]

```
-g
```

- 省略時解釈
  ソース・デバッグ用の情報を出力しません。

[詳細説明]

- ソース・デバッグ用の情報を出力ファイル中に出力します。
- 本オプションを指定することにより，ソース・デバッグが可能となります。
- 最適化オプションと同時に指定した場合は，デバッグのしやすさに影響があります。

[使用例]

- ソース・デバッグ用の情報を出力ファイル中に出力します。

```
>ccrh -g -Xcommon=rh850 main.c
```

-g_line 【V1.05.00 以降】

最適化時にソース・デバッグ用の情報を強化します。

[指定形式]

```
-g_line
```

- 省略時解釈
  最適化時にソース・デバッグ用の情報を強化しません。

[詳細説明]

- -g オプションと同時に指定した場合にのみ有効となります。
- 最適化を行った場合に，デバッグ時により正確にソースレベルのステップ実行を行うことができるように，デバッグ情報を強化します。
- デバッグ情報量が増加し，ステップ実行が遅くなる可能性があります。

[使用例]

- 出力ファイル中の，ソース・デバッグ用の情報を強化して出力します。

```
>ccrh -g -g_line main.c
```

デバイス指定

デバイス指定オプションには，次のものがあります。

- -Xcommon
- -Xcpu

-Xcommon

デバイス共通のオブジェクト・ファイルを生成することを指定します。

[指定形式]

```
-Xcommon=series
```

- 省略時解釈
  なし

[詳細説明]

- デバイス共通のオブジェクト・ファイルを生成することを指定します。
- V2.00.00 以降では，本オプションは無効です。指定した場合，無視されますが，従来バージョンとの互換性のため，エラーにはなりません。このとき，警告を出力しません。
- *series* には v850e3v5 または rh850 を指定可能です。
- 次の場合はエラーとなります。
  - *series* を省略した場合
  - *series* に指定可能なパラメータ以外を指定した場合
  - 本オプション指定を省略した場合 【V1.01.00 以前】

[備考]

本オプションは出力コードに影響しません。
使用する命令セットを選択する場合は，-Xcpu オプションを指定してください。

-Xcpu

指定したコア向けのオブジェクトを生成することを指定します。

[指定形式]

```
-Xcpu=core
```

- 省略時解釈
  G3M 向けオブジェクトを生成します。

[詳細説明]

- コア *core* 向けのオブジェクトを生成することを指定します。
- *core* に指定可能なものを以下に示します。

g3m	G3M 向けオブジェクトを生成します。
g3k	G3K 向けオブジェクトを生成します。
g3mh	G3MH 向けオブジェクトを生成します。【V1.02.00 以降】
g3kh	G3KH 向けオブジェクトを生成します。【V1.03.00 以降】
g4mh	G4MH 向けオブジェクトを生成します。【V2.00.00 以降】

- 本オプションを複数回指定した場合，最後の指定が有効になります。
- 次の場合はエラーとなります。
  - パラメータを省略した場合
  - 指定可能なパラメータ以外を指定した場合

処理中断指定

処理中断指定オプションには，次のものがあります。

- -P
- -S
- -c

-P

入力した C ソース・ファイルに対してプリプロセス処理のみ実行します。

[指定形式]

```
-P
```

- 省略時解釈
  プリプロセス処理以降も処理を継続します。
  ファイルは出力しません。

[詳細説明]

- 入力した C ソース・ファイルに対してプリプロセス処理のみ実行して，結果をファイルに出力します。
- 出力ファイル名は，入力ファイル名の拡張子を “.i” に置き換えたものになります。
- -o オプションと同時に指定することにより，出力ファイル名を指定することができます。
- -Xpreprocess オプションを指定することにより，出力ファイルの内容を制御することができます。

[使用例]

- 入力した C ソース・ファイルに対してプリプロセス処理のみ実行して，結果をファイル main.i に出力します。

```
>ccrh -P -Xcommon=rh850 main.c
```

-S

アセンブル以降の処理を実行しません。

[指定形式]

```
-S
```

- 省略時解釈
  アセンブル以降も処理を継続します。

[詳細説明]

- アセンブル以降の処理を実行しません。
- ソース・ファイル名の拡張子を “.asm” で置き換えた名前でアセンブリ・ソース・ファイルを出力します。
- -o オプションと同時に指定することにより，出力ファイル名を指定することができます。

[使用例]

- アセンブル以降の処理を実行せず，アセンブリ・ソース・ファイル main.asm を出力します。

```
>ccrh -S -Xcommon=rh850 main.c
```

-c

リンク以降の処理を実行しません。

[指定形式]

```
-c
```

- 省略時解釈
  リンク以降も処理を継続します。

[詳細説明]

- リンク以降の処理を実行しません。
- ソース・ファイル名の拡張子を “.obj” で置き換えた名前でオブジェクト・ファイルを出力します。
- -o オプションと同時に指定することにより，出力ファイル名を指定することができます。

[使用例]

- リンク以降の処理を実行せず，オブジェクト・ファイル main.obj を出力します。

```
>ccrh -c -Xcommon=rh850 main.c
```

プリプロセッサ制御

プリプロセッサ制御オプションには，次のものがあります。

- -D
- -U
- -I
- -Xpreinclude
- -Xpreprocess

-D

プリプロセッサ・マクロ，およびアセンブラ・シンボルを定義します。

[指定形式]

```
-Dname[=def][,name[=def]]...
```

- 省略時解釈
  なし

[詳細説明]

- プリプロセッサ・マクロ，およびアセンブラのユーザ定義シンボルとして *name* を定義します。
- ソース・プログラムの前に，#define *name def*，および .SET *name def*（アセンブリ・ソース・プログラムの場合のみ）を記述するのと同等です。
- *name* が，アセンブラ・シンボルには使用可能であるがプリプロセッサ・マクロには使用できない文字（@，.，~）を含む場合，警告を出力してアセンブラ・シンボルとしてのみ定義します。
- 本オプションで C 言語の既定義マクロ __LINE__，__FILE__，__DATE__，__TIME__，__CCRH__ を再定義することはできません（-D__CCRH__[=1]，および -D__CCRH[=1] を除く）。
  入力ファイルが C ソース・ファイルの場合にこれらを再定義すると，再定義は無視され警告となります。
- *name* を省略した場合は，エラーとなります。
- *=def* を省略した場合，*def* は 1 とみなします。
- 本オプションは，複数指定が可能です。
- 同じプリプロセッサ・マクロ，およびアセンブラ・シンボルに対して，本オプションと -U オプションを同時に指定した場合は，あとから指定したものが有効となります。

[使用例]

- プリプロセッサ・マクロとして sample=256 を定義します。

```
>ccrh -Dsample=256 -Xcommon=rh850 main.c
```

-U

-D オプションによるプリプロセッサ・マクロ，およびアセンブラ・シンボルの定義を解除します。

[指定形式]

```
-Uname[,name]...
```

- 省略時解釈
  なし

[詳細説明]

- -D オプションによるプリプロセッサ・マクロ，およびアセンブラのユーザ定義シンボル *name* の定義を解除します。
- ソース・プログラムの前に，#undef *name* を記述するのと同等です。
- *name* を省略した場合は，エラーとなります。
- 本オプションでは，#define *name def*，および .SET *name def*（アセンブリ・ソース・プログラムの場合のみ）の記述による定義は解除できません。
- 本オプションにより，C 言語の既定義マクロを解除することもできますが，__LINE__，__FILE__，__DATE__，__TIME__，__CCRH__，__CCRH を解除することはできません。
  入力ファイルが C ソース・ファイルの場合に *name* にこれらを指定すると，エラーとなります。
- 本オプションは，複数指定が可能です。
- 同じプリプロセッサ・マクロ，およびアセンブラ・シンボルに対して，本オプションと -D オプションを同時に指定した場合は，あとから指定したものが有効となります。

[使用例]

- -D オプションによるプリプロセッサ・マクロ test の定義を解除します。

```
>ccrh -Utest -Xcommon=rh850 main.c
```

-I

インクルード・ファイルを検索するフォルダを指定します。

[指定形式]

```
-Ipath[,path]...
```

- 省略時解釈
  インクルード・ファイルを ソース・ファイルのあるフォルダと標準インクルード・ファイル・フォルダから検索します。

[詳細説明]

- プリプロセッサ指令 #include，およびアセンブラ制御命令 $INCLUDE/$BINCLUDE で読み込むインクルード・ファイルを検索するフォルダを *path* に指定します。
  インクルード・ファイルの検索は，以下の順番で行います。

(1) #include の場合

(a) ソース・ファイルのあるフォルダ（ファイルを ” ” で指定した場合）

(b) -I オプションで指定したフォルダ（指定が複数ある場合は，コマンド・ラインで指定した順（左から右の順））

(c) 標準インクルード・ファイル・フォルダ

(2) $INCLUDE/$BINCLUDE の場合

(a) -I オプションで指定したフォルダ（指定が複数ある場合は，コマンド・ラインで指定した順（左から右の順））

(b) ソース・ファイルのあるフォルダ

(c) カレント・フォルダ

- *path* が存在しない場合は，警告を出力します。
- *path* を省略した場合は，エラーとなります。

[使用例]

- インクルード・ファイルをカレント・フォルダ，フォルダ D:¥include，標準フォルダの順で検索します。

```
>ccrh -ID:¥include -Xcommon=rh850 main.c
```

-Xpreinclude

コンパイル単位の先頭にインクルードするファイルを指定します。

[指定形式]

```
-Xpreinclude=file[,file]...
```

- 省略時解釈
  コンパイル単位の先頭にインクルードするファイルはないものとみなします。

[詳細説明]

- コンパイル単位の先頭にインクルードするファイルを *file* に指定します。
- *file* が相対パス指定の場合はコンパイラを起動したフォルダから検索します。

[使用例]

- コンパイル単位の先頭にファイル sample.h をインクルードします。

```
>ccrh main.c -Xpreinclude=sample.h -Xcommon=rh850
```

-Xpreprocess

プリプロセス結果の出力を制御します。

[指定形式]

```
-Xpreprocess=string[,string]
```

- 省略時解釈
  プリプロセス結果のファイルに，C ソースのコメント，および行番号情報を出力しません。

[詳細説明]

- プリプロセス結果のファイルに，C ソースのコメント，および行番号情報を出力します。
- 本オプションは，-P オプション指定時のみ有効です。
  -P オプションを指定しない場合は，本オプションを無視します。
- *string* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

comment	C ソースのコメントを出力します。
line	行番号情報を出力します。

< 行番号情報のフォーマット >

```
#line 行番号 " ファイル名 "
```

- *行番号*は 10 進数で，最大値は unsigned int の最大数となります。
- *ファイル名*はフルパスで，パス中の ¥ は ¥¥ に，" は ¥" に変換します。
  印字可能文字（スペースを含む）でない場合は，*¥3 桁の 8 進数* ("¥¥%03o") で出力します。
  改行文字は ¥¥n に変換します。
- 入力ソース・ファイル中に前処理指令 # *数値* " *文字列* "，または #line *数値* " *文字列* " を記述している場合は，*数値*を*行番号*，*文字列*を*ファイル名*として適用します。

- *string* を省略した場合は，エラーとなります。
- OS 標準の文字コードで出力します。

[使用例]

- プリプロセス結果のファイルに，C ソースのコメント，および行番号情報を出力します。

```
>ccrh -Xpreprocess=comment,line -P -Xcommon=rh850 main.c
```

以下は上記の例と同等です。

```
>ccrh -Xpreprocess=comment -Xpreprocess=line -P -Xcommon=rh850 main.c
```

C 言語制御

C 言語制御オプションには，次のものがあります。

- -lang 【V1.07.00 以降】
- -strict_std 【V1.07.00 以降】
- -Xenum_type
- -Xvolatile
- -Xcheck
- -Xmisra2004 【Professional 版のみ】
- -Xmisra2012 【Professional 版のみ】
- -Xignore_files_misra 【Professional 版のみ】
- -Xcheck_language_extension 【Professional 版のみ】
- -misra_intermodule 【Professional 版のみ】【V2.01.00 以降】
- -Xuse_fp16 【Professional 版のみ】【V1.05.00 以降】

-lang 【V1.07.00 以降】

言語規格を指定します。

[指定形式]

```
-lang={c|c99}
```

- 省略時解釈
  C90 規格に沿ってコンパイルします。

[詳細説明]

- C ソース・ファイルの言語規格を指定します。
- -lang=c オプション指定時またはオプション省略時は，C90 規格に沿ってコンパイルします。
- -lang=c99 オプション指定時は，C99 規格に沿ってコンパイルします。
- c，c99 以外を指定した場合は，エラーとします。

[備考]

- 本コンパイラは，言語規格の一部をサポートしません。
  - C90/C99 言語規格の一部の標準ライブラリ関数
  - C99 言語規格の複素数型
  - C99 言語規格の可変長配列

-strict_std 【V1.07.00 以降】

C ソース・プログラムを言語規格に厳密にあわせて処理します。

[指定形式]

```
-strict_std  【V1.07.00 以降】
-Xansi       【V1.06.00 以前との互換用途】
```

- 省略時解釈
  従来の C 言語の仕様との両立性を持たせ，警告を出力して処理を続行します。-lang=c99 オプションを指定していない場合でも，C99 で追加された仕様の一部も受容します。

[詳細説明]

- C ソース・プログラムを -lang オプションで指定した言語規格に厳密にあわせて処理し，規格に反する記述に対してエラーや警告を出力します。
- 本オプション指定時，または未指定時に有効になる既定義マクロは，「4.2.2 マクロ」を参照してください。
- 言語規格に厳密なコンパイル時の処理は，以下のようになります。
  - C90 準拠時
    - ビット・フィールド
      ビット・フィールドに int，signed int，unsigned int 型以外の型を指定した場合は，エラーとなります。
      本オプションを指定しない場合は，int 型以外の型の指定を許可します（警告は出力しません）。
    - # 行番号
      エラーとなります。
      本オプションを指定しない場合は，“#line 行番号 ” と同様に扱います。
    - #pragma inline 指定された関数の引数
      指定した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が型変換が可能である場合は，エラーとなります。
      本オプションを指定しない場合は，戻り値の型は呼び出し側の型に，引数は関数定義での型に変換して，インライン展開を行います。
    - 基本型
      _Bool，long long，unsigned long long，__fp16 型をエラーとします。
    - 構造体指定子，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エラーメッセージを出力します。
  - C99 準拠時
    - # 行番号
      エラーとなります。
      本オプションを指定しない場合は，“#line 行番号 ” と同様に扱います。
    - #pragma inline 指定された関数の引数
      指定した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が型変換が可能である場合は，エラーとなります。
      本オプションを指定しない場合は，戻り値の型は呼び出し側の型に，引数は関数定義での型に変換して，インライン展開を行います。
    - 基本型
      __fp16 型をエラーとします。
    - 構造体指定子，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，エラーとします。

-Xenum_type

列挙型に対して，どの整数型として扱うかを指定します。

[指定形式]

```
-Xenum_type=string
```

- 省略時解釈
  列挙型に対して，signed int として扱います。

[詳細説明]

- 列挙型に対して，どの整数型として扱うかを指定します。
- *string* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

auto	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。

- *string* を省略した場合は，エラーとなります。

[使用例]

- 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。

```
>ccrh -Xenum_type=auto -Xcommon=rh850 main.c
```

-Xvolatile

外部変数の volatile 化を指定します。

[指定形式]

```
-Xvolatile
```

- 省略時解釈
  volatile 修飾のある変数のみを volatile 宣言したものとして扱います。

[詳細説明]

- すべての外部変数を volatile 宣言したものとして扱います。
  外部変数のアクセス回数，アクセス順序は C ソース・ファイルで記述したとおりになります。

[使用例]

- すべての外部変数を volatile 宣言したものとして扱います。

```
>ccrh -Xvolatile -Xcommon=rh850 main.c
```

-Xcheck

C ソース・ファイルの互換性をチェックします。

[指定形式]

```
-Xcheck=comp
```

- 省略時解釈
  C ソース・ファイルの互換性のチェックを行いません。

[詳細説明]

- *comp* に指定したコンパイラ用にコーディングした C ソース・ファイルを本コンパイラでコンパイルする際，互換性に影響するオプション指定，およびソース記述のチェックを行い，影響のある部分について警告，またはエラーを出力します。
- *comp* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

shc	SHC 用にコーディングした C ソース・ファイルをチェックします。

- *comp* を省略した場合は，エラーとなります。
- 主なチェック項目を以下に示します。
  - オプション：-Xbit_order=*pos*
    言語仕様で規定されていない実装依存の内容がコンパイラ間で異なっています。
    メッセージに出力されたオプションの選択を確認してください。
  - 拡張機能：#pragma section，#pragma entry，#pragma stacksize，#pragma address，#pragma global_register
    プログラムの動作に影響を及ぼす可能性がある拡張仕様です。
    メッセージに出力された拡張仕様の記述を確認してください。
  - volatile 修飾した変数
    読み出しや書き込みのサイズがコンパイラ間で異なる場合があります。
    volatile 修飾したビット・フィールドは，本コンパイラでは宣言型より小さいサイズでアクセスすることがありますが，SH コンパイラでは宣言型のサイズどおりにアクセスします。
  - 2 項演算の整数拡張
    unsigned int 型と long 型をオペランドとする 2 項演算（加減乗除や比較など）の結果が SH コンパイラと異なる場合があります。
    SH コンパイラでは -strict_ansi オプションを指定しない場合 signed long 型で演算します。
    本コンパイラではオペランドを unsigned int 型に変換してから演算します。
  - signed long 型を超える整数定数の型
    SH コンパイラでは unsigned long 型で表現可能な範囲の値を signed long long 型とします。
    本コンパイラでは unsigned long 型で表現可能な範囲の値は unsigned long 型とします。
  - ビット・フィールドの割り付け
    SH コンパイラではビット・フィールドの型が直前のビット・フィールドの型と異なる場合はビットを連続して配置しません。
    本コンパイラでは -Xpack オプションの指定に応じて連続して配置する場合があります。
- 構造体，およびビット・フィールド・メンバの割り付けについては，メッセージを出力しません。
  割り付けを意識した宣言を行っている場合は，「4.1.3 データの内部表現と領域」を参照してください。

［使用例］

- SH コンパイラ用にコーディングした C ソース・ファイルの互換性をチェックします。

```
>ccrh -Xcheck=shc -Xcommon=rh850 main.c
```

-Xmisra2004 【Professional 版のみ】

MISRA-C:2004 ルールによるソース・チェックを行います。

[指定形式]

```
-Xmisra2004=item[=value]
```

- 省略時解釈
  MISRA-C:2004 ルールによるソース・チェックを行いません。

[詳細説明]

- MISRA-C:2004 ルールによるソース・チェックを行います。
  指定したチェック項目 *item* に該当した場合，メッセージを出力します。
- 本オプションは -Xmisra2012 オプションと同時に指定することはできません。
- -lang=c99 オプションを同時に指定している場合，本オプションを無視します。このとき警告を出力します。
- *item* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

チェック項目（*item*）	パラメータ（*value*）	説明
all	なし	サポートしているすべてのルールをチェック対象とします。
apply	*num*[,*num*]...	サポートしているルールのうち，*num* で指定した番号のルールをチェック対象とします。
ignore	*num*[,*num*]...	サポートしているルールのうち，*num* で指定した番号以外のルールをチェック対象とします。
required	なし	サポートしているルールのうち，ルールの分類が “required” になっているルールをチェック対象とします。
required_add	*num*[,*num*]...	サポートしているルールのうち，ルールの分類が “required” になっているルールと *num* で指定した番号のルールをチェック対象とします。
required_remove	*num*[,*num*]...	サポートしているルールのうち，ルールの分類が “required” になっているルールから *num* で指定した番号を除いたルールをチェック対象とします。
file		サポートしているルールのうち，指定したファイル *file* に記載した番号のルールをチェック対象とします。 ファイル内では，1 行につき 1 ルール番号を指定します。

- *num* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

  2.2 2.3
  4.1 4.2
  5.2 5.3 5.4 5.5 5.6
  6.1 6.2 6.3 6.4 6.5
  7.1
  8.1 8.2 8.3 8.5 8.6 8.7 8.11 8.12
  9.1 9.2 9.3
  10.1 10.2 10.3 10.4 10.5 10.6
  11.1 11.2 11.3 11.4 11.5
  12.1 12.3 12.4 12.5 12.6 12.7 12.8 12.9 12.10 12.11 12.12 12.13
  13.1 13.2 13.3 13.4

14.2 14.3 14.4 14.5 14.6 14.7 14.8 14.9 14.10
15.1 15.3 15.4 15.5
16.1 16.3 16.5 16.6 16.9
17.5
18.1 18.4
19.3 19.6 19.7 19.8 19.11 19.13 19.14 19.15
20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12

- *item* を省略した場合は，エラーとなります。
- __fp16 型は，float 型とみなしてチェックします。影響については -Xcheck_language_extension 【Professional 版のみ】オプションの説明を参照してください。

［使用例］

- MISRA-C:2004 ルール番号 5.2，5.3，5.4 のルールをチェック対象としたソース・チェックを行います。

```
>ccrh -Xmisra2004=apply=5.2,5.3,5.4 -Xcommon=rh850 main.c
```

-Xmisra2012 【Professional 版のみ】

MISRA-C:2012 ルールによるソース・チェックを行います。

[指定形式]

```
-Xmisra2012=item[=value]
```

- 省略時解釈
  MISRA-C:2012 ルールによるソース・チェックを行いません。

[詳細説明]

- MISRA-C:2012 ルールによるソース・チェックを行います。
  指定したチェック項目 *item* に該当した場合，メッセージを出力します。
- 本オプションは -Xmisra2004 オプションと同時に指定することはできません。
- *item* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。
  なお，ルールの分類が “mandatory” になっているルールは以下の指定に関わらず必ずチェック対象になります。

チェック項目（*item*）	パラメータ（*value*）	説明
all	なし	サポートしているすべてのルールをチェック対象とします。
apply	*num*[,*num*]...	サポートしているルールのうち，*num* で指定した番号のルールをチェック対象とします。
ignore	*num*[,*num*]...	サポートしているルールのうち，*num* で指定した番号以外のルールをチェック対象とします。
required	なし	サポートしているルールのうち，ルールの分類が “mandatory” および “required” になっているルールをチェック対象とします。
required_add	*num*[,*num*]...	サポートしているルールのうち，ルールの分類が “mandatory” および “required” になっているルールと *num* で指定した番号のルールをチェック対象とします。
required_remove	*num*[,*num*]...	サポートしているルールのうち，ルールの分類が “required” になっているルールから *num* で指定した番号を除いたルールをチェック対象とします。
file		サポートしているルールのうち，指定したファイル *file* に記載した番号のルールをチェック対象とします。 ファイル内では，1 行につき 1 ルール番号を指定します。

- V2.02.00 以降では，MISRA-C:2012 Amendment 1 をもとに，次のルール番号をサポートしています。
  これ以外のものを指定した場合は，エラーとなります。
  2.2 2.6 2.7
  3.1 3.2
  4.1 4.2
  5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9
  6.1 6.2
  7.1 7.2 7.3 7.4
  8.1 8.2 8.3 8.4 8.5 8.6 8.8 8.9 8.11 8.12 8.13 8.14
  9.1 9.2 9.3 9.4 9.5
  10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8
  11.1 11.2 11.3 11.4 11.5 11.6 11.7 11.8 11.9
  12.1 12.2 12.3 12.4 12.5

13.1 13.2 13.3 13.4 13.5 13.6
14.2 14.3 14.4
15.1 15.2 15.3 15.4 15.5 15.6 15.7
16.1 16.2 16.3 16.4 16.5 16.6 16.7
17.1 17.3 17.4 17.5 17.6 17.7 17.8
18.4 18.5 18.7
19.2
20.1 20.2 20.3 20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12 20.13 20.14
21.1 21.2 21.3 21.4 21.5 21.6 21.7 21.8 21.9 21.10 21.11 21.12 21.13 21.15 21.16

- *item* を省略した場合は，エラーとなります。
- __fp16 型は，float 型とみなしてチェックします。影響については -Xcheck_language_extension 【Professional 版のみ】オプションの説明を参照してください。

［使用例］

- MISRA-C:2012 ルール番号 5.2，5.3，5.4 のルールをチェック対象としたソース・チェックを行います。

```
>ccrh -Xmisra2012=apply=5.2,5.3,5.4 -Xcommon=rh850 main.c
```

-Xignore_files_misra 【Professional 版のみ】

MISRA-C:2004 または MISRA-C:2012 ルールによるソース・チェックの対象外のファイルを指定します。

[指定形式]

```
-Xignore_files_misra=file[,file]...
```

- 省略時解釈
  すべての C ソース・ファイルをチェック対象とします。

[詳細説明]

- *file* で指定したファイルを MISRA-C:2004 または MISRA-C:2012 ルールによるソース・チェックの対象外とします。
- 本オプションは，-Xmisra2004 または -Xmisra2012 オプション指定時のみ有効です。
  -Xmisra2004 または -Xmisra2012 オプションを指定しない場合は，本オプションを無視します（警告は出力しません）。

[使用例]

- sample.c を MISRA-C:2004 ルールによるソース・チェックの対象外とします。

```
>ccrh -Xmisra2004=all -Xignore_files_misra=sample.c -Xcommon=rh850 main.c
```

-Xcheck_language_extension 【Professional 版のみ】

言語拡張により部分抑止される MISRA-C:2004 または MISRA-C:2012 ルールのソース・チェックを有効にします。

[指定形式]

```
-Xcheck_language_extension
```

- 省略時解釈
  言語拡張により部分抑止される MISRA-C:2004 または MISRA-C:2012 ルールのソース・チェックを無効にします。

[詳細説明]

- C 言語規格から独自に拡張した言語仕様のために MISRA-C:2004 または MISRA-C:2012 ルールのソース・チェックが抑止される以下の場合について，ソース・チェックを有効にします。
  - プロトタイプ宣言がなく（MISRA-C:2004 ルール 8.1，MISRA-C:2012 ルール 8.4），該当関数に #pragma interrupt 指定がある場合
- 本オプションは，-Xmisra2004 または -Xmisra2012 オプション指定時のみ有効です。
  -Xmisra2004 または -Xmisra2012 オプションを指定しない場合は，本オプションを無視します（警告は出力しません）。
  __fp16 型は，float 型とみなしてチェックします。次のルールが影響します。
  - MISRA-C:2004 ルール 10.2
  - MISRA-C:2012 ルール 10.3
  - MISRA-C:2012 ルール 10.4

[使用例]

- 言語拡張により部分抑止される MISRA-C:2004 ルールのソース・チェックを有効にします。

```
>ccrh -Xmisra2004=all -Xcheck_language_extension -Xcommon=rh850 main.c
```

-misra_intermodule 【Professional 版のみ】 【V2.01.00 以降】

複数ファイルにまたがる MISRA-C:2012 ルールによるソース・チェックを行います。

[指定形式]

```
-misra_intermodule=file
```

- 省略時解釈
  なし

[詳細説明]

- 複数ファイルのシンボル情報を *file* に収集して，複数ファイルにまたがる MISRA-C:2012 ルールによるソース・チェックを行います。
  file が存在しない場合は新規作成し，*file* が存在する場合は追記します。
- MISRA-C:2012 において解析範囲が「システム」であるルールに対して本オプションを 適用します。
  本オプションでチェック可能な MISRA-C:2012 ルールは以下の通りです。
  5.1 5.6 5.7 5.8 5.9
  8.3 8.5 8.6
- *file* の拡張子に .{c|a|f} は指定できません。指定した場合はエラーとなります。
- *file* が他の入出力ファイルと重複する場合は，動作を保証しません。
- 本オプションを複数回指定した場合，最後に指定したものが有効となります。その際，警告を出力します。
- -Xmisra2012 オプションを同時に指定しない場合は，本オプションを無視します。その際，警告を出力します。
- 次の場合は，エラーになります。
  - パラメータを省略した場合

[備考]

- *file* 作成後にソース・ファイルを修正した場合は，再コンパイルすることで *file* の情報が更新されます。
  プロジェクトからソース・ファイルを削除するか，ソース・ファイル名を変更した場合は，*file* を削除して MISRA-C:2012 のチェックをやり直してください。
- チェック対象のファイル数が多く，*file* に入るシンボル情報が膨大になると，コンパイル速度が遅くなります。
- 本オプションは，パラレル・ビルドなど並列でコンパイルする場合には，正しいチェックができません。並列コンパイルはせずにご指定してください。

[使用例]

- a1.c，a2.c，a3.c にまたがるチェックを行います。

```
>ccrh -Xmisra2012=all -misra_intermodule=info.mi a1.c a2.c a3.c
```

-Xuse_fp16 【Professional 版のみ】 【V1.05.00 以降】

半精度浮動小数点型を有効にします。

[指定形式]

```
-Xuse_fp16[=value]
```

- 省略時解釈
  半精度浮動小数点型を無効にします。

[詳細説明]

- C 言語規格から独自に拡張した半精度浮動小数点型を有効にします。
- *value* には on または off を指定します。*value* を省略した場合は on を指定したとみなします。
- 次の場合，警告を出力して本オプションを無視します。
  - -strict_std オプションと同時指定する。
  - -Xcpu=g3k オプションと同時指定する。
  - -Xfloat=soft オプションと同時指定する。
  - -Xround=zero オプションと同時指定する。

[使用例]

- __fp16 型を有効にします。

```
>ccrh -Xuse_fp16 main.c
```

日本語／中国語文字列制御

日本語／中国語文字列制御オプションには，次のものがあります。

- -Xcharacter_set

-Xcharacter_set

日本語／中国語の文字コードを指定します。

[指定形式]

```
-Xcharacter_set=code
```

- 省略時解釈
  日本語の文字コードを SJIS として扱います。

[詳細説明]

- ソース・ファイル中の日本語／中国語のコメント，文字列に対して，使用する文字コードを指定します。
- *code* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。
  なお，ソース・ファイル中で使用している文字コードと異なるものを指定した場合，動作は保証されません。

none	日本語／中国語の文字コードを解釈しません
euc_jp	EUC（日本語）
sjis	SJIS
utf8	UTF-8
big5	繁体字中国語
gb2312	簡体字中国語

- *code* を省略した場合は，エラーとなります。

[使用例]

- ソース・ファイル中の日本語のコメント，文字列に対して，使用する文字コードに EUC を指定します。

```
>ccrh -Xcharacter_set=euc_jp -Xcommon=rh850 main.c
```

最適化指定

最適化指定オプションには，次のものがあります。

- -O
- -Xintermodule
- -Xinline_strcpy
- -Xmerge_string
- -Xalias
- -Xmerge_files
- -Xwhole_program
- -library 【V2.00.00 以降】
- -goptimize 【V2.01.00 以降】

-O

最適化のレベル，または各最適化項目の詳細を指定します。

[指定形式]

```
-O[level]
-O[item[=value][,item[=value]]...]
```

- 省略時解釈
  デバッグに影響しない最適化を行います（-Odefault オプションの指定と同じです）。

[詳細説明]

- 最適化のレベル，または各最適化項目の詳細を指定します。
- *level* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

nothing	デバッグ優先の最適化 デバッグのしやすさを重視し，冗長なコードの削除など最小限の最適化に抑えます。
default	デフォルト デバッグに影響しない範囲の最適化（式の最適化，およびレジスタ割り付けなど）を行います。
size	オブジェクト・サイズ優先の最適化 ROM/RAM 容量の削減を重視して，一般的なプログラムに対して有効な最大限の最適化を行います。
speed	実行速度優先の最適化 実行速度の短縮を重視して，一般的なプログラムに対して有効な最大限の最適化を行います。

- *level*，および *item* を省略した場合は，size を指定したものとみなします。
- *item*，および *value* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

最適化項目 （*item*）	パラメータ （*value*）	説明
unroll	0 ～ 4294967295 （整数値）	ループ展開 ループ文（for，while，do-while）を展開します。 *value* で，最大で何倍の展開を行うかを指定します。 *value* の値 0 は値 1 と同じ意味となります。 *value* を省略した場合は，4 を指定したものとみなします。 本項目は，-Ospeed オプションを指定した場合は，-Ounroll=4 オプションを指定したものとみなします。

最適化項目 （*item*）	パラメータ （*value*）	説明
inline	0 ～ 3 （整数値）	関数のインライン展開 *value* は，展開のレベルを表します。 0：#pragma inline 指定した関数を含めて，すべてのインライン展開を抑止します。 1：#pragma inline 指定した関数のみ展開します。 2：自動的に展開対象の関数を判別して展開します。 3：コード・サイズがなるべく増加しない範囲で，自動的に展開対象の関数を判別して展開します。 ただし，1 ～ 3 を指定した場合でも，関数の内容やコンパイル状況により，#pragma inline 指定した関数が展開されない場合があります。 *value* を省略した場合は，2 を指定したものとみなします。 本項目は，-Osize，または -Ospeed オプションを指定した場合に有効です（-Osize オプションを指定した場合は -Oinline=3，-Ospeed オプションを指定した場合は -Oinline=2 オプションを指定したものとみなします）。 -Osize，-Ospeed，-Oinline オプションのいずれも指定していない場合は，-Oinline=1 オプション指定したものとみなします。 -Onothing オプションを指定した場合は，-Oinline=0 オプションを指定したものとみなします。
inline_size	0 ～ 65535 （整数値）	インライン展開サイズ コード・サイズが何%増加するまでインライン展開を行うかを指定します。 *value* を省略した場合は，100 を指定したものとみなします。 本項目は，-Oinline=2 オプションを指定した場合に有効です（-Ospeed オプションによる指定も含みます）。
inline_init 【V1.07.00 以降】	on，または off	自動変数の初期化子の即値化 on を指定した場合，自動変数の初期化を常に即値の代入で行います。 off を指定した場合，初期化をメモリ間コピーで行うか，即値の代入で行うかを，コンパイラが自動的に選択します。 *value* を省略した場合は，on を指定したものとみなします。
delete_static_func	on，または off	未使用 static 関数の削除 *value* を省略した場合は，on を指定したものとみなします。 本項目は，-Onothing オプション指定時以外に有効です。
pipeline	on，または off	パイプライン最適化 *value* を省略した場合は，on を指定したものとみなします。
tail_call	on，または off	末尾呼び出し最適化 on を指定した場合，関数の末尾が関数呼び出しであり，かつ一定の条件を満たす場合に，その呼び出しに対して関数呼び出しを無条件分岐に変換して lp の退避／復帰コードを削除し，コード・サイズを削減します。 ただし，一部のデバッグ機能を使用することができなくなります。 *value* を省略した場合は，on を指定したものとみなします。 本項目は，-Ospeed，または -Osize オプションを指定した場合に有効です。
map	*ファイル名*	外部変数アクセス最適化 最適化リンカが生成する外部シンボル割り付け情報を元にベースアドレスを設定し，外部変数，または静的変数のアクセスをベース・アドレス相対で行うコードを生成します。 また，スタートアップ・ルーチンでシンボル “__gp_data” を定義して gp に値を設定するコードを記述した場合，外部シンボル割り付け情報を元に，可能であれば変数のアクセスを gp 相対で行うコードを生成します。 *ファイル名*には，最適化リンカが生成した外部シンボル割り付け情報ファイルを指定します。 *ファイル名*を省略した場合は，一度リンク処理まで実行し，外部シンボル割り付け情報ファイルを作成したのち，再度コンパイルからリンクまでの処理を行います。

最適化項目（*item*）	パラメータ（*value*）	説明
smap	なし	コンパイル単位内で定義された外部変数に対する外部変数アクセス最適化 コンパイル対象ファイル内で定義された外部変数，または静的変数についてベース・アドレスを設定し，アクセスをベース・アドレス相対で行うコードを生成します。
align 【V2.03.00 以降】	on, または off	整列条件の変更による最適化の実施 たとえば構造体型の変数の中にある，連続した領域にアクセスする際に，変数の整列条件を変更した上で，複数のアクセスを１回に統合することで生成する命令数を少なくし，コードサイズ削減，実行速度の向上を図ります。 整列条件を変更する結果として，パディング・データの埋め込みが発生し，データを記憶する領域の使用量が増加する場合があります。 value を省略した場合は，on を指定したものとみなします。 本項目は，-Osize または -Ospeed オプションを指定した場合に有効です。 -stuff オプションを同時に指定した場合，本項目は無効になり，off を指定した場合と同じ動作になります。

- 同じ *item* について，本オプションを複数指定した場合は，あとから指定したものが有効となります。
- -O*item* のあとに -O*level* を指定した場合，さきに指定した -O*item* は無効になります。ただし，-Omap，-Osmap は，-O*level* の影響をうけません。
- -O*item* を指定しなかった場合の最適化項目は，-O*level* の指定によって以下のように解釈します。

最適化項目（*item*）	最適化レベル（*level*）			
	nothing	default	size	speed
unroll	1	1	1	4
inline	0	1	3	2
inline_size	—	—	—	100
inline_init	off	off	off	on
delete_static_func	off	on	on	on
pipeline	off	off	off	on
tail_call	off	off	on	on
map	—	—	—	—
smap	—	—	—	—
align	off	off	on 注	on

注 -misalign オプションを同時に指定した場合は，off と解釈します。

この表は，ある level を指定すると同時に，item の指定内容を他の level の item の設定に合わせたとしても，他の level と同等の最適化を実施することを示すものではありません。例えば，“-Ospeed” という指定と，“-Osize -Ounroll=4 -Oinline=2 -Oinline_size=100 -Opipeline=on” という指定では，同じ出力コードにならない場合があります。

［使用例］

- オブジェクト・サイズ優先の最適化を行います。

```
>ccrh -Osize -Xcommon=rh850 main.c
```

-Xintermodule

大域最適化を行います。

[指定形式]

```
-Xintermodule
```

- 省略時解釈
  大域最適化を行いません。

[詳細説明]

- 大域最適化を行います。
- 主な最適化内容を以下に示します。
  - 手続き間別名解析を利用した最適化
    出力コード例を以下に示します。

```
[C ソース ]
extern int x[2];
static int func1(int *a, int *b) {
    *a=0;
    *b=1;
    return *a;
}
int func2() {
    return func1(&x[0], &x[1]);
}

[ 出力アセンブラ・ソース ]
_func1.1:
    .stack  _func1.1 = 0
    mov     #_x, r2
    st.w    r0, 0x00000000[r2]
    mov     0x00000001, r5
    st.w    r5, 0x00000004[r2]
    mov     0x00000000, r10     ; a と b の指すアドレスが違うため 0 を直接代入
    jmp     [r31]
_func2:
    .stack  _func2 = 0
    mov     #_x, r6
    addi    0x00000004, r6, r7
    br9     _func1.1
```

- パラメータ，戻り値の定数伝播
  出力コード例を以下に示します。

```
[C ソース ]
static int func(int x, int y, int z) {
    return x-y+z;
}
int func2() {
    return func(3,4,5);
}

[ 出力アセンブラ・ソース ]
_func.1:
    .stack  _func.1 = 0
    mov     0x00000004, r10     ; 4(=3-4+5) を直接代入
    jmp     [r31]
_func2:
    .stack  _func2 = 0
    mov     0x00000005, r8
    mov     0x00000004, r7
    mov     0x00000003, r6
    br9     _func.1
```

［使用例］

- ソース・ファイル main.c，sub.c の大域最適化を行います。

```
>ccrh -Xintermodule -Osize -Xcommon=rh850 main.c sub.c
```

-Xinline_strcpy

標準ライブラリ関数 strcpy，strcmp，memcpy，memset の呼び出しをインライン展開します。

［指定形式］

```
-Xinline_strcpy
```

- 省略時解釈
  標準ライブラリ関数 strcpy，strcmp，memcpy，memset のインライン展開を行いません。

［詳細説明］

- 標準ライブラリ関数 strcpy，strcmp，memcpy，memset の呼び出しをインライン展開します。
- 本オプションは，-Xpack オプションと同時に指定することはできません。
- strcpy については，第二引数が文字列の場合にのみ，インライン展開を行います。
- 本オプションを指定した場合，配列，および文字列は自動的に 4 バイト境界に配置されます。
- 生成されるプログラムの実行速度は高速になりますが，コード・サイズは増大します。

［使用例］

- 標準ライブラリ関数 strcpy，strcmp，memcpy，memset の呼び出しをインライン展開します。

```
>ccrh -Xinline_strcpy -Xcommon=rh850 main.c
```

-Xmerge_string

文字列定数をマージします。

[指定形式]

```
-Xmerge_string
```

- 省略時解釈
  ソース・ファイル内で同じ文字列定数が複数存在する場合，それぞれを別々の領域に割り付けます。

[詳細説明]

- ソース・ファイル内で同じ文字列定数が複数存在する場合，これらをまとめて 1 つの領域に割り付けます。
- #pragma section の指定に依らず，同じ文字列定数を同一領域に割り付けます。
  ただし，異なるセクションを指定した場合，文字列定数の割り付け先のセクションはソース中での出現順序に依存します。

[使用例]

- ソース・ファイル内で同じ文字列定数が複数存在する場合，これらをまとめて 1 つの領域に割り付けます。

```
>ccrh -Xmerge_string -Xcommon=rh850 main.c
```

-Xalias

ポインタ指示先の型を考慮した最適化の指定を行います。

[指定形式]

```
-Xalias=value
```

- 省略時解釈
  ANSI 規格に基づくポインタ指示先の型を考慮した最適化を行いません。

[詳細説明]

- ANSI 規格に基づくポインタ指示先の型を考慮した最適化を行うかどうかを指定します。
- *value* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

ansi	ANSI 規格に基づき，ポインタ指示先の型を考慮した最適化を行います。
noansi	ANSI 規格に基づくポインタ指示先の型を考慮した最適化を行いません。

- *value* を省略した場合は，エラーとなります。

[使用例]

- ANSI 規格に基づき，ポインタ指示先の型を考慮した最適化を行います。

```
>ccrh -Xalias=ansi -Xcommon=rh850 -Osize main.c
```

-Xmerge_files

複数の C ソース・ファイルをマージしてコンパイルを行います。

[指定形式]

```
-Xmerge_files
```

- 省略時解釈
  マージせずに入力ファイル単位でコンパイルを行います。

[詳細説明]

- 複数の C ソース・ファイルをマージしてコンパイルを行い，1 ファイルで出力します。
- 出力ファイル名は，-o オプションを指定した場合は指定したファイル名となり，-o オプションを指定しなかった場合は最初に指定した C ソース・ファイル名に対して -o オプションの省略時解釈に従ったファイル名となります。
- 本オプションは，入力 C ソース・ファイルが 1 ファイルの場合，および -P オプションと同時に指定した場合は無効となります。
- 本オプションは，-S，または -c オプションと同時に指定した場合は，2 番目以降に指定した C ソース・ファイル名について -o オプションの省略時解釈に従ったファイル名の空ファイルを出力します。
- -Oinline オプションと同時に指定した場合は，ファイル間インライン展開を行います。
- 本オプションを指定して生成したオブジェクト・ファイルをリンクする際にリンク・オプション -delete，-rename，-replace のいずれかを同時に指定した場合，動作は保証されません。

[使用例]

- main.c，sub.c をマージしてコンパイルを行い，1 ファイルで出力します。

```
>ccrh -Xmerge_files -Xwhole_program -Xcommon=rh850 main.c sub.c
```

-Xwhole_program

コンパイル対象ファイルがプログラム全体であることを仮定して最適化を行います。

[指定形式]

```
-Xwhole_program
```

- 省略時解釈
  コンパイル対象ファイルがプログラム全体であることを仮定しません。

[詳細説明]

- コンパイル対象ファイルがプログラム全体であることを仮定して最適化を行います。
- 以下の条件を満たすことを前提にコンパイルを行い，条件を満たさなかった場合の動作は保証されません。
  - コンパイル対象ファイル内で定義した extern 変数の値，およびアドレスが，コンパイル対象ファイル以外で変更，および参照されない。
  - コンパイル対象ファイル内からコンパイル対象ファイル以外で定義した関数を呼び出したとしても，呼び出された関数からコンパイル対象ファイル内の関数が呼ばれることがない。
- 本オプションを指定した場合，-Xintermodule オプションを指定したものとみなしてコンパイルを行います。
  また，入力 C ソース・ファイルが複数の場合，-Xmerge_files オプションを指定したものとみなしてコンパイルを行います。

[使用例]

- コンパイル対象ファイルがプログラム全体であることを仮定して最適化を行います。

```
>ccrh -Xwhole_program -Xcommon=rh850 -Osize main.c
```

-library 【V2.00.00 以降】

標準ライブラリ関数の呼び出しをインライン展開します。

[指定形式]

```
-library={function|intrinsic}
```

- 省略時解釈
  function を指定した場合と同じ意味になります。

[詳細説明]

- 次の標準ライブラリ関数を関数呼び出しにするか，インライン展開するかを制御します。
  - abs()，labs()，llabs()
  - fabs()，fabsf()
  - sqrt()，sqrtf()
  - fmax()，fmaxf()
  - fmin()，fminf()
  - copysign()，copysignf()
- function を指定した場合，常に対象関数を呼び出すコードを生成します。
- intrinsic を指定した場合，可能であれば対象関数の呼び出しをインライン展開します。
- パラメータは小文字で指定してください。
- 本オプションを複数回指定した場合，最後の指定が有効になります。
- 次の場合はエラーとなります。
  - パラメータを省略した場合
  - パラメータに function，intrinsic 以外を指定した場合
- 本オプション指定により，対象のライブラリ関数呼び出しがインライン展開された場合，展開後のコードは変数 errno を更新しません。また，次の入力に対する動作が関数呼び出し時と異なります。
  - sqrt，sqrtf：-0.0，負数，非数
  - fmax，fmaxf，fmin，fminf：+0.0 と -0.0，非数

-goptimize 【V2.01.00 以降】

リンク時最適化用の情報を生成します。

[指定形式]

```
-goptimize
```

- 省略時解釈
  なし

[詳細説明]

- リンク時最適化時に使用する付加情報を出力ファイル内部に生成します。
- 本オプションを指定したファイルは，リンク時にリンク時最適化の対象になります。
  リンク時最適化の詳細については，リンク・オプション -OPtimize を参照してください。

生成コード制御

生成コード制御オプションには，次のものがあります。

- -Xpack
- -misalign 【V2.04.00 以降】
- -Xbit_order
- -Xpass_source
- -Xswitch
- -Xreg_mode
- -Xreserve_r2
- -r4 【V1.07.00 以降】
- -Xep
- -Xfloat
- -Xfxu 【V2.00.00 以降】
- -Xcall_jump
- -Xfar_jump
- -Xdiv
- -Xcheck_div_ov
- -relaxed_math 【V2.00.00 以降】
- -Xuse_fmaf
- -use_recipf 【V2.00.00 以降】
- -approximate 【V2.02.00 以降】
- -Xunordered_cmpf
- -Xmulti_level
- -Xpatch
- -Xdbl_size
- -Xround
- -Xalign4
- -Xstack_protector/-Xstack_protector_all 【Professional 版のみ】
- -Xsection
- -stuff 【V2.03.00 以降】
- -Xcheck_exclusion_control 【V1.04.00 以降】
- -Xresbank_mode 【V2.00.00 以降】
- -insert_dbtag_with_label 【V1.06.00 以降】
- -store_reg 【Professional 版のみ】【V1.06.00 以降】
- -control_flow_integrity 【Professional 版のみ】【V1.07.00 以降】
- -pic 【V1.07.00 以降】
- -pirod 【V1.07.00 以降】
- -pid 【V1.07.00 以降】

-Xpack

構造体パッキングを行います。

[指定形式]

```
-Xpack=num
```

- 省略時解釈
  構造体パッキングを行いません。

[詳細説明]

- 構造体パッキングを行います。
- 本オプションを指定した場合，構造体のメンバをその型でアライメントせず，指定した *num* バイトのアライメントに詰めてコードを生成します。
- *num* には，1，2，4 のいずれかを指定することができます。
  これ以外のものを指定した場合は，エラーとなります。
- *num* を省略した場合は，エラーとなります。
- 本オプションは，-Xinline_strcpy オプションと同時に指定することはできません。
- C ソース中に #pragma 指令で構造体パッキングを指定している場合に本オプションを指定すると，最初の #pragma 指令が出現するまでは，オプションの指定値をすべての構造体に適用します。
  それ以降は，#pragma 指令の値を適用します。
  ただし，#pragma 指令の出現後でも，指定がデフォルトになった部分（#pragma 指令でパッキング値を指定しない場合）は，オプションの指定値を適用します。

[使用例]

- 構造体のメンバを 1 バイトのアライメントに詰めてコードを生成します。

```
>ccrh -Xpack=1 -Xcommon=rh850 main.c
```

-misalign 【V2.04.00 以降】

ミスアライン・メモリ・アクセスを行う命令列を生成します。

[指定形式]

```
-misalign
```

[詳細説明]

- メモリ・アクセスに対して，整列されていないアドレスへのアクセスをデバイスがサポートしていると想定した，より効率的な命令列を生成します。
- 本オプションを複数回指定した場合，１回指定した場合と同じ意味になります。このとき，警告を出力しません。

[備考]

- 本オプションを指定する場合は，デバイスのミスアライン・メモリ・アクセス機能を有効にしてください。詳細はデバイスのユーザーズマニュアルを参照してください。

-Xbit_order

ビット・フィールドのメンバの並び順を指定します。

[指定形式]

```
-Xbit_order=pos
```

- 省略時解釈
  ビット・フィールドのメンバを下位ビットから割り付けます。

[詳細説明]

- ビット・フィールドのメンバの並び順を指定します。
- *pos* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

left	上位ビットからメンバを割り付けます。
right	下位ビットからメンバを割り付けます。

- *pos* を省略した場合は，エラーとなります。
- C ソース中に #pragma 指令でビット・フィールドのメンバの並び順を指定している場合，最初の #pragma bit_order 指令が出現するまでは，オプション指定値がすべてのビット・フィールドのメンバに適用されます。
  それ以降は，#pragma 指令の値が適用されます。

[使用例]

- ビット・フィールドのメンバを上位ビットから割り付けます。

```
>ccrh -Xbit_order=left -Xcommon=rh850 main.c
```

-Xpass_source

出力するアセンブリ・ソース・ファイル中にC ソース・プログラムをコメントとして出力します。

[指定形式]

```
-Xpass_source
```

- 省略時解釈
  出力するアセンブリ・ソース・ファイル中にC ソース・プログラムをコメントとして出力しません。

[詳細説明]

- 出力するアセンブリ・ソース・ファイル中にC ソース・プログラムをコメントとして出力します。
- 出力するコメントは，あくまで参考であり，厳密にはコードと対応していない場合もあります。
  また，実行文以外の行にはコメントとして出力されないものもあります（型宣言やラベルなど）。
  たとえば，グローバル変数とローカル変数，関数宣言などのコメントの出力位置がずれることがあります。
  また，最適化オプションを指定した場合などにより，コードが削除され，コメントのみが残ることもあります。

[使用例]

- 出力するアセンブリ・ソース・ファイル中にC ソース・プログラムをコメントとして出力します。

```
>ccrh -Xpass_source -S -Xcommon=rh850 main.c
```

-Xswitch

switch 文のコード出力形式を指定します。

[指定形式]

```
-Xswitch=type
```

- 省略時解釈
  コンパイラが switch 文ごとに最適な出力形式を 自動的に選択します。

[詳細説明]

- switch 文のコード出力形式を指定します。
- *type* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

ifelse	case ラベルを 1 つずつ比較する形式で出力します。case 文の数が少ないときに，本項目を指定します。
binary	バイナリ・サーチ形式で出力します。 バイナリ・サーチ・アルゴリズムに用いて合致する case 文を探します。 ラベル数が多いときに本項目を選択すると，どの case 文も同じくらいの速さで見つけることができます。
table	テーブル・ジャンプ形式で出力します。 case 文の値を基にインデックス化したテーブルを参照し，switch 文の値により case ラベルを選択して処理を行います。 どの case 文にも同じくらい速く分岐します。 ただし，case 値が連続していないときは無駄な領域ができます。 また，case ラベルの最大値と最小値の差が 8192 を超える場合は，本オプションを無視して，switch 文ごとに最適な出力形式を 自動的に選択します。

- *type* を省略した場合は，エラーとなります。

[使用例]

- switch 文のコードに対して，バイナリ・サーチ形式で出力します。

```
>ccrh -Xswitch=binary -Xcommon=rh850 main.c
```

-Xreg_mode

レジスタ・モードを指定します。

[指定形式]

```
-Xreg_mode=mode
```

- 省略時解釈
  32 レジスタ・モードのオブジェクト・ファイルを生成します。

[詳細説明]

- 指定したレジスタ・モードのオブジェクト・ファイルを生成します。
- ccrh が使用するレジスタを 32 本（32 レジスタ・モード），22 本（22 レジスタ・モード，または common レジスタ・モード）のいずれかに制限し，オブジェクト・ファイル内にレジスタ・モードを示すマジック・ナンバを埋め込みます。
- common レジスタ・モードは，レジスタ・モードに依存しないオブジェクト・ファイルを生成するために使用します。
- *mode* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

レジスタ・モード（*mode*）	作業用レジスタ	レジスタ変数用レジスタ
common	r10 ～ r14	r25 ～ r29
22	r10 ～ r14	r25 ～ r29
32	r10 ～ r19	r20 ～ r29

- *mode* を省略した場合は，エラーとなります。
- C ソース・ファイルに対して，使用可能なレジスタのみを用いたコードを生成します。
- 32 レジスタ・モードのオブジェクトファイルと，22 レジスタ・モードのオブジェクト・ファイルが混在している場合は，リンク時にエラーとなります。

[使用例]

- 22 レジスタ・モードのオブジェクト・ファイルを生成します。

```
>ccrh -Xreg_mode=22 -Xcommon=rh850 main.c
```

-Xreserve_r2

r2 レジスタを予約します。

[指定形式]

```
-Xreserve_r2
```

- 省略時解釈
  r2 レジスタの予約を行わずに，コンパイラで使用します。

[詳細説明]

- r2 レジスタを予約し，コンパイラでは r2 レジスタを使用しないコードを生成します。

[使用例]

- r2 レジスタを予約し，コンパイラでは r2 レジスタを使用しないコードを生成します。

```
>ccrh -Xreserve_r2 -Xcommon=rh850 main.c
```

-r4 【V1.07.00 以降】

r4 レジスタの扱い方を指定します。

[指定形式]

```
-r4=mode
```

- 省略時解釈
  r4 レジスタの値をプロジェクト全体で固定します。

[詳細説明]

- r4 レジスタの扱い方を指定します。
- *mode* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

fix	r4 レジスタの値をプロジェクト全体で固定します。GP 相対セクション[注]を使用する場合は，本パラメータを指定してください。
none	コンパイラは r4 レジスタを使用しません。

注 GP 相対セクションについては「4.2.6.1 関数とデータのセクション割り当て」を参照してください。

-Xep

ep レジスタの扱い方を指定します。

[指定形式]

```
-Xep=mode
```

- 省略時解釈
  ep レジスタを関数呼び出し前後で値を保証するレジスタとして扱います。

[詳細説明]

- ep レジスタの扱い方を指定します。
- *mode* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

fix	ep レジスタの値をプロジェクト全体で固定します。 プロジェクト内で EP 相対セクション[注]を使用する場合は，本パラメータを指定してください。
callee	ep レジスタを関数呼び出し前後で値を保証するレジスタとして扱います。 -Omap，または -Osmap オプションを指定した場合は，本パラメータを指定してください。

注 EP 相対セクションについては「4.2.6.1 関数とデータのセクション割り当て」を参照してください。

- *mode* を省略した場合は，エラーとなります。
- 本オプションは，すべてのソース・ファイルに対して同じ指定をする必要があります。
  ソース・ファイルごとに指定を変えることはできません。
  指定の異なるオブジェクト・ファイルが混在している場合は，リンク時にエラーとなります。

[使用例]

- ep レジスタの値をプロジェクト全体で固定します。

```
>ccrh -Xep=fix -Xcommon=rh850 main.c
```

-Xfloat

浮動小数点演算命令の生成を制御します。

[指定形式]

```
-Xfloat=type
```

- 省略時解釈
  -Xcpu=g3k オプションを指定している場合は，-Xfloat=soft とみなします。
  それ以外の場合は，-Xfloat=fpu とみなします。

[詳細説明]

- 浮動小数点演算命令の生成を制御します。
- *type* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

soft	浮動小数点演算に対して，ランタイム関数の呼び出し命令を生成します。
fpu	浮動小数点演算に対して，FPU（浮動小数点ユニット）の浮動小数点演算命令を生成します。 ただし -Xcpu=g3kh を指定した場合，倍精度演算に対してはランタイム関数の呼出し命令を生成します。 -Xcpu=g3k オプションと同時に指定した場合は無効となり，-Xfloat=soft とみなします。

- *type* を省略した場合は，エラーとなります。
- *type* に soft を指定した場合，-Xround=zero オプションは無効となり，常に -Xround=nearest が有効となります。
- 本オプションの指定をソース・ファイルごとに変更した場合，例外ハンドラ内のレジスタ管理が正しく行われない場合があります。

[使用例]

- 浮動小数点演算に対して，ランタイム関数の呼び出し命令を生成します。

```
>ccrh -Xfloat=soft -Xcommon=rh850 main.c
```

-Xfxu 【V2.00.00 以降】

FXU 命令の使用を制御します。

[指定形式]

```
-Xfxu[={on|off}]
```

- 省略時解釈
  -Xcpu=g4mh 指定時は，-Xfxu=on を指定したとみなします。
  -Xcpu=g4mh 以外を指定した場合は，-Xfxu=off を指定したとみなします。

[詳細説明]

- 例外ハンドラ内での，FXU 用システムレジスタの扱いを制御します。
- on を指定した場合，プログラム中で FXU 命令を使用するとみなします。
  off を指定した場合，プログラム中で FXU 命令を使用しないとみなします。
- パラメータを省略すると，on を指定した場合と同じ意味になります。
- 本オプションを複数回指定した場合，最後の指定が有効になります。
- 本オプションの指定をソース・ファイルごとに変更した場合，例外ハンドラ内のレジスタ管理が正しく行われない場合があります。
- -Xcpu=g4mh 以外を指定している場合，本オプションの指定を無視します。この時警告を出力します。
- 次の場合はエラーとなります。
  - on，off 以外を指定した場合

[備考]

V2.00.00 では，本オプションを指定しても FXU 命令を生成しません。例外ハンドラの生成コードにのみ影響します。

-Xcall_jump

関数呼び出しの分岐命令の生成を制御します。

[指定形式]

```
-Xcall_jump=num
```

- 省略時解釈
  関数呼び出しの分岐に対して，jarl，および jr 命令を生成します。

[詳細説明]

- 関数呼び出しの分岐命令の生成を制御します。
- *num* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

22	関数呼び出しの分岐に対して，jarl，および jr 命令を生成します。
32	関数呼び出しの分岐に対して，jarl32，および jr32 命令を生成します。

- *num* を省略した場合は，エラーとなります。

[使用例]

- 関数呼び出しの分岐に対して，jarl32，および jr32 命令を生成します。

```
>ccrh -Xcall_jump=32 -Xcommon=rh850 main.c
```

-Xfar_jump

far jump の出力を制御します。

[指定形式]

```
-Xfar_jump=file
```

- 省略時解釈
  -Xcall_jump オプションに従った命令を生成します。

[詳細説明]

- C ソース・ファイルについて，far jump 呼び出し関数一覧ファイル *file* 中で指定した関数への分岐に対して，32 ビット長の分岐距離を持つ命令を生成します。
- *file* の推奨拡張子は，".fjp" です。
- *file* が存在しない場合は，エラーとなります。
- *file* を省略した場合は，エラーとなります。
- -Xcall_jump=22 オプションを指定した際，分岐命令と分岐先関数との距離が 22 ビット長（±2M バイト）を超えていて，リンク時にエラーとなる場合，本オプションを使用してコンパイルし直します。
- 本オプションを複数指定した場合は，最後に指定したものが有効となります。
- 出力コード例を以下に示します。
  - C ソース

```
far_func();     /* デフォルトでは jarl _far_func, lp を出力する */
```

  - 出力アセンブリ・ソース

```
jarl32  _far_func, lp
```

備考 far jump 呼び出し関数一覧ファイルの書式に関する注意事項を以下に示します。

- 1 行に 1 個の関数名を記述します。
  複数の関数名を記述した場合は，最初の名前のみを有効とします。
- 記述する関数名は，C 言語の関数名の先頭に "_" を付けた名前（アセンブリ・ソースでのラベル名）とします。
  ただし，関数名の代わりに，以下の形式で指定することもできます。

形式	意味
{all_function}	すべての関数呼び出しを対象とします。

- C ソース・ファイル上から呼び出す関数だけでなく，演算用ランタイム関数も指定可能です。
  演算用ランタイム関数を指定する場合は，関数名の先頭に "_" を追加せず，「7.4.13 演算用ランタイム関数」の表 7.16 に記載されている関数名をそのまま指定してください。
- 関数名の前後に，空白やタブを挿入することが可能です。
- 使用可能な文字は，ASCII 文字のみです。
  なお，行頭の空白を除いたあと，次の空白，または行末までの非空白文字列を関数名とし，空白から行末までは無視します。

- コメントを挿入することはできません。
- 1 行に記述可能な文字数は，1023 文字までです（空白やタブも含みます）。

関数の指定例を以下に示します。

```
_func_led
_func_beep
_func_motor
    :
_func_switch
_COM_div64
```

［使用例］

- func.fjp 中で指定した関数への分岐に対して，32 ビット長の分岐距離を持つ命令を使用したコードを生成します。

```
>ccrh -Xfar_jump=func.fjp -Xcommon=rh850 main.c
```

-Xdiv

除算に対して，div，および divu 命令を生成します。

[指定形式]

```
-Xdiv
```

- 省略時解釈
  除算に対して，divq，および divqu 命令を生成します。

[詳細説明]

- 除算に対して，divq，および divqu 命令を生成する代わりに，div，および divu 命令を生成します。
- divq，および divqu 命令は高速ですが，実行サイクル数がオペランドの値により変わります。
  そのため，リアルタイム性の保証などのために，常に実行サイクル数が一定であることが必要な場合に，本オプションを指定します。

[使用例]

- 除算に対して，div，および divu 命令を生成します。

```
>ccrh -Xdiv -Xcommon=rh850 main.c
```

-Xcheck_div_ov

除算時に OV フラグのチェックを行います。

[指定形式]

```
-Xcheck_div_ov=num
```

- 省略時解釈
  除算時に OV フラグのチェックを行わないコードを生成します。

[詳細説明]

- 除算命令の後に OV フラグのチェックを行い，OV フラグが 1 のときに FE レベル・ソフトウエア例外を発生させるコード（fetrap 命令）を生成します。
- *num* に指定可能な値は，1 ～ 15（fetrap 命令のオペランドで指定可能な値）です。
  これ以外のものを指定した場合は，エラーとなります。
- *num* を省略した場合は，エラーとなります。

[使用例]

- 除算時に OV フラグのチェックを行います。

```
>ccrh -Xcheck_div_ov=1 -Xcommon=rh850 main.c
```

-relaxed_math 【V2.00.00 以降】

厳密さより効率を重視した浮動小数点演算コードを生成します。

[指定形式]

```
-relaxed_math
```

- 省略時解釈
  なし

[詳細説明]

- 浮動小数点演算に対して，C 言語規格や IEEE754 に厳密ではないが，コードサイズや実行速度について効率のよい演算コードの生成を行います。
- また，次のオプションを同時に指定したとみなします。
  - -Xuse_fmaf
  - -use_recipf
  - -approximate 【V2.02.00 以降】
- 本オプションを複数回指定した場合，1 回指定した場合と同じ意味になります。このとき，警告を出力しません。

[備考]

本オプションを指定した場合，次のように浮動小数点演算を扱い，演算結果が C 言語規格や IEEE754 の厳密な規定と異なる場合があります。

- 0.0 の符号の意味を無視します。
- 演算により例外や精度誤差が発生しないと仮定して，代数的な性質を利用した数式の変形を行います。
- 比較演算，その他の演算に非数，無限大の入出力が無いと仮定します。
  これらの値を扱うプログラムである場合，意図しない実行結果になる可能性があるため，本オプションを使用する場合は注意が必要です。

  例）
  このプログラムは，通常は x または y が非数である場合，関数 func2 を呼び出しません。
  しかし，本オプション指定時は，非数の入力を想定しないかわりに効率のよい (*) コードを生成するため，関数 func2 を呼び出す場合があります。

  (*) -Xfloat=soft 指定時に顕著に変化します。

```
void func1(double x, double y) {
  if (x < y) {
    func2();
  }
}
```

-Xuse_fmaf

積和演算命令を生成します。

[指定形式]

```
-Xuse_fmaf
```

- 省略時解釈
  積和演算命令を生成しません。

[詳細説明]

- 単精度浮動小数点積和演算に対して，積和演算命令（fmaf.s，fmsf.s，fnmaf.s，fnmsf.s）を生成します。
- 本オプションを指定すると，実行速度は上がりますが，演算精度が変わります。

[使用例]

- 単精度浮動小数点積和演算に対して，積和演算命令を生成します。

```
>ccrh -Xuse_fmaf -Xcommon=rh850 main.c
```

-use_recipf 【V2.00.00 以降】

recipf 命令を生成します。

[指定形式]

```
-use_recipf
```

- 省略時解釈
  なし

[詳細説明]

- recipf.d，recipf.s 命令を生成します。
- 本オプションを複数回指定した場合，1 回指定した場合と同じ意味になります。このとき，警告を出力しません。
- -Xcpu オプション，または -Xfloat オプションで FPU を使用しない設定にした場合，本オプションを無視します。このとき，警告を出力しません。

[備考]

本オプション指定により recipf 命令を生成した場合，演算結果が本オプションを指定しない場合と異なる可能性があります。
recipf 命令は，常に FPU の不正確演算例外を発生させるため，適切に FPU 例外処理を設定してください。

-approximate 【V2.02.00 以降】

浮動小数点演算を，同等の近似計算で置き換えます。

[指定形式]

```
-approximate
```

- 省略時解釈
  なし

[詳細説明]

- 浮動小数点演算を，同等の近似計算で置き換えます。
  この置き換えにより，コードサイズや実行速度について効率のよい演算コードを生成します。

[備考]

本オプションを指定した場合，次のように浮動小数点演算を扱い，演算結果が C 言語規格や IEEE754 の厳密な規定と異なる場合があります。

- 0.0 の符号の意味を無視します。演算により例外や精度誤差が発生しないと仮定して，代数的な性質を利用した数式の変形を行います。

-Xunordered_cmpf

浮動小数点比較において，無効演算例外の検出を行います。

[指定形式]

```
-Xunordered_cmpf
```

- 省略時解釈
  浮動小数点比較において，qNaN を含む場合は無効演算例外の検出を行いません。

[詳細説明]

- 浮動小数点比較において，比較する値に qNaN が含まれている場合に無効演算例外を発生する比較条件を使用するコードを生成します。
- 本オプションは FPU（浮動小数点ユニット）の命令を使用した浮動小数点比較にのみ有効です。

[使用例]

- 浮動小数点比較において，無効演算例外の検出を行います。

```
>ccrh -Xunordered_cmpf -Xcommon=rh850 main.c
```

-Xmulti_level

マルチコア用プログラムの生成を指定します。

[指定形式]

```
-Xmulti_level=level
```

- 省略時解釈
  シングルコア用プログラムを生成します。

[詳細説明]

- 指定したコア用のプログラムを生成します。
- *level* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

0	シングルコア用プログラムを生成します。 プログラム中の #pragma pmodule 指定は無視します。
1	マルチコア用プログラムを生成します。 プログラム中の #pragma pmodule 指定は有効となります。

[使用例]

- マルチコア用プログラムを生成します。

```
>ccrh -Xmulti_level=1 -Xcommon=rh850 file1.c file2.c
```

-Xpatch

パッチを適用します。

[指定形式]

```
-Xpatch=string[,string]...
```

- 省略時解釈
  既定のパッチを適用します。

[詳細説明]

string には以下のいずれかを指定します。これ以外のものを指定した場合はエラーとなります。

- dw_access
  ld.dw/st.dw 命令の生成を抑止したコードを生成します。
- switch 【V1.03.00 以降】
  -Xcpu=g3m を指定した場合は，本オプションを指定しない状態では switch 命令の生成を抑止します。本オプションを指定することで，抑止を解除して switch 命令を生成するようになります。
- syncp 【V1.03.00 以降】
  -Xcpu=g3m を同時に指定している場合は，#pragma interrupt で priority=SYSERR/FPI/FENMI/FEINT/EIINT_PRIORITYX(X：0 ～ 15) を指定，もしくは priority も channel も指定していない割り込み関数の関数先頭に syncp 命令を挿入します。
  -Xcpu=g3m 以外を指定している場合は，本オプション指定を無視します。
- br【V2.04.00 以降】
  -Xcpu=g3m を指定した場合は，本オプションを指定しない状態では，特定の条件を満たす br disp9 命令の生成を抑止します。本オプションを指定することで，抑止を解除して br disp9 命令を生成するようになります。
  -Xcpu=g3m 以外を指定した場合は，本オプション指定によらず，br disp9 命令の生成を抑止しません。
- br_jr【V2.04.01 以降】
  -Xcpu=g3kh を同時に指定している場合は，br disp9 命令，jr disp22 命令，および jr disp32 命令の生成を抑止します。
  -Xcpu=g3kh 以外を指定している場合は，本オプション指定を無視します。

本オプションを指定しない場合に適用する既定のパッチは次の通りです。

- -Xcpu=g3m 指定時に，次のパッチを適用します。
  - switch 命令の生成の抑止
  - 特定の条件を満たす br disp9 命令の生成の抑止

[使用例]

- ld.dw/st.dw 命令の生成を抑止したコードを生成します。

```
>ccrh -Xpatch=dw_access -Xcommon=rh850 main.c
```

-Xdbl_size

double 型および long double 型の精度を指定します。

[指定形式]

```
-Xdbl_size=num
```

- 省略時解釈
  double 型および long double 型を倍精度浮動小数点型 (8 バイト) として扱う (-Xdbl_size=8 と同じ)。

[詳細説明]

- double 型および long double 型の精度を指定します。
- num には以下のいずれかを指定します。これ以外のものを指定した場合はエラーとなります。
  - 4
    double 型および long double 型を単精度浮動小数点型 (4 バイト) として扱う。
  - 8
    double 型および long double 型を倍精度浮動小数点型 (8 バイト) として扱う。

-Xround

浮動小数点定数の丸めモードを指定します。

[指定形式]

```
-Xround=mode
```

- 省略時解釈
  浮動小数点定数を表現可能な最も近い値に丸めます（-Xround=nearest と同じ）。

[詳細説明]

- 浮動小数点定数の丸めモードを指定します。
- mode には以下のいずれかを指定します。これ以外のものを指定した場合はエラーとなります。
  - nearest
    浮動小数点定数を表現可能な最も近い値に丸めます。
  - zero
    浮動小数点定数を 0 の方に丸めます。
- -Xfloat=soft と同時に指定した場合，本オプションは無効になり，常に浮動小数点定数を表現可能な最も近い値に丸めます。

-Xalign4

分岐先アドレスのアライメントを指定します。

[指定形式]

```
-Xalign4[=mode]
```

- 省略時解釈
  分岐先アドレスのアライメントを 2 にします。

[詳細説明]

- *mode* で指定した分岐先アドレスのアライメントを 4 にします。
- *mode* には以下のいずれかを指定します。これ以外のものを指定した場合はエラーとなります。
  - function
    関数の先頭アドレスのアライメントを 4 にします。
  - loop
    関数の先頭とすべてのループの先頭アドレスのアライメントを 4 にします。
  - innermostloop
    関数の先頭と最内側ループの先頭アドレスのアライメントを 4 にします。
  - all
    関数の先頭とすべての分岐先アドレスのアライメントを 4 にします。
  - =*mode* を省略した場合
    関数の先頭アドレスのアライメントを 4 にします (function と同じ)。
- 本オプションを使用しないでコンパイルして生成したオブジェクト・モジュール・ファイルや標準ライブラリとリンクした場合，リンク時に W0561322 の警告を出力しますが，動作は問題ありません。

-Xstack_protector/-Xstack_protector_all 【Professional 版のみ】

スタック破壊検出コード生成を指定します。

[指定形式]

```
-Xstack_protector[=num]
-Xstack_protector_all[=num]
```

- 省略時解釈
  スタック破壊検出コードを生成しません。

[詳細説明]

- 関数の入口・出口にスタック破壊検出コードを生成します。スタック破壊検出コードとは次に示す 3 つの処理を実行するための命令を指します。
  (1) 関数の入口で，当該関数スタック・フレームのローカル変数領域の直前（0xFFFFFFFF 番地に向かう方向）に 4 バイトの領域を確保し，*num* で指定した値を確保した領域に格納します。
  (2) 関数の出口で，*num* を格納した 4 バイトの領域が書き換わっていないことをチェックします。
  (3) (2) で書き換わっている場合には，スタックが破壊されたとして __stack_chk_fail 関数を呼び出します。
- *num* には 0 から 4294967295 までの整数値を指定します。*num* の指定を省略した場合には，コンパイラが自動的に数値を指定します。
- __stack_chk_fail 関数はユーザが定義する必要があり，スタックの破壊検出時に実行する処理を記述します。
  __stack_chk_fail 関数を定義する際には，次の項目に注意してください。
  - 返却値の型は void 型のみであり，仮引数を持たない関数です。
  - static 指定をしないでください。
  - 通常の関数のように呼び出すことは禁止します。
  - __stack_chk_fail 関数は，オプション -Xstack_protector，-Xstack_protector_all と #pragma stack_protector によるスタック破壊検出コードの生成の対象にはなりません。
  - 関数内では abort() を呼び出してプログラムを終了させるなど，呼び出し元であるスタックの破壊を検出した関数にリターンしないようにしてください。
  - __stack_chk_fail 関数内で関数を呼び出す場合は，呼び出した先の関数内で再帰的にスタックの破壊を検出しないように注意してください。
  - PIC（「8.6 PIC/PID 機能」を参照してください）である関数に対して本機能を使用する場合，__stack_chk_fail 関数も PIC 対象としてください。
- -Xstack_protector を指定すると，関数が持つ構造体，共用体または配列のローカル変数に対して，コンパイラが 8byte を超えるオブジェクトとしてスタックに領域を確保する場合に，スタック破壊検出コードを生成する対象となります。-Xstack_protector_all を指定した場合には全ての関数に対してスタック破壊検出コードを生成します。
- 本オプションと #pragma stack_protector とを同時に使用した場合は，#pragma stack_protector の指定が有効になります。
- 以下の #pragma が指定された関数は，本オプションを指定した場合でもスタック破壊検出コードを生成しません。
  #pragma inline，inline キーワード，#pragma inline_asm，#pragma no_stack_protector

［例］

- ＜C ソース＞

```
#include <stdio.h>
#include <stdlib.h>

void f1()   // スタックが破壊されるプログラムの例
{
    volatile char str[10];
    int i;
    for (i = 0; i <= 10; i++){
      str[i] = i; // i=10 の場合にスタックが破壊される
    }
}

void __stack_chk_fail(void)
{
    printf("stack is broken!");
    abort();
}
```

- ＜出力コード＞

-Xstack_protector=1234 を指定してコンパイルした場合

```
_f1:
        .stack _f1 = 16
        add 0xFFFFFFF0, r3
        movea 0x000004D2, r0, r1  ; 指定した＜数値＞ 1234 をスタックの領域へ格納する
        st.w r1, 0x0000000C[r3]
        mov 0x00000000, r2
        br9 .BB.LABEL.1_2
.BB.LABEL.1_1:  ; bb
        movea 0x00000002, r3, r5
        add r2, r5
        st.b r2, 0x00000000[r5]
        add 0x00000001, r2
.BB.LABEL.1_2:  ; bb7
        cmp 0x0000000B, r2
        blt9 .BB.LABEL.1_1
.BB.LABEL.1_3:  ; return
        ld.w 0x0000000C[r3], r1   ; 関数の入口で＜数値＞を格納した位置からロードし，
        movea 0x000004D2, r0, r12 ; 指定した＜数値＞1234 と，
        cmp r12, r1               ; 比較する
        bnz9 .BB.LABEL.1_5        ; 異なっている場合には，分岐する
.BB.LABEL.1_4:  ; return
        dispose 0x00000010, 0x00000000, [r31]
.BB.LABEL.1_5:  ; return
        br9 ___stack_chk_fail     ; __stack_chk_fail を呼び出す

___stack_chk_fail:
        .stack ___stack_chk_fail = 4
        prepare 0x00000001, 0x00000000
        mov #.STR.1, r6
        jarl _printf, r31
        jarl _abort, r31
        dispose 0x00000000, 0x00000001, [r31]
```

-Xsection

データのデフォルト・セクションを指定します。

[指定形式]

```
-Xsection=string=value[,string=value]
```

- 省略時解釈
  初期値なしデータは .bss，初期値ありデータは .data，定数データは .const をデフォルトとします。

[詳細説明]

- データのデフォルト・セクション属性を指定します。
- *string* と *value* に指定可能な文字列と，各々の場合のデフォルト・セクションは以下の通りです。*string* と *value* にこれ以外のものを指定した場合はエラーとなります。

string	value	デフォルト・セクション		
		初期値なしデータ	初期値ありデータ	定数データ
data	r0_disp16	.zbss	.zdata	-
	r0_disp23	.zbss23	.zdata23	-
	ep_disp16	.ebss	.edata	-
	ep_disp23	.ebss23	.edata23	-
	gp_disp16	.sbss	.sdata	-
	gp_disp23	.sbss23	.sdata23	-
const	zconst	-	-	.zconst
	zconst23	-	-	.zconst23
	pcconst16 【V1.07.00 以降】	-	-	.pcconst16
	pcconst23 【V1.07.00 以降】	-	-	.pconst23

- #pragma section で属性を変更した場合は，#pragma section で指定した属性が有効になります。

本オプションと他のオプションの，エラーになる組み合わせを次に示します。

-Xsection=data=ep_disp16 -Xsection=data=ep_disp23	-Omap を同時に指定する -Osmap を同時に指定する
-Xsection=data=gp_disp16 -Xsection=data=gp_disp23	-r4=none を同時に指定する
-Xsection=data=r0_disp16 -Xsection=data=r0_disp23	-pid を同時に指定する
-Xsection=const=zconst -Xsection=const=zconst23	-pirod を同時に指定する
-Xsection=const=pcconst16 -Xsection=const=pcconst23	-pirod を同時に指定しない

-stuff 【V2.03.00 以降】

変数をアライメント数に応じたセクションに分けて配置します。

[指定形式]

```
-stuff[=< 変数種別 >[,...]]
< 変数種別 >:{bss|data|const}
```

- 省略時解釈
  セクションを分けずに変数を配置します。

[詳細説明]

- stuff オプションを指定した場合，指定した < *変数種別* > に属する変数をアライメント数に応じたセクションに分けて配置します。
- bss 指定は初期値なし変数を，data 指定は初期値あり変数を，const 指定は const 変数を対象とします。
- < *変数種別* > を省略した場合は，全ての種別の変数が対象となります。
- 本オプションを複数回指定した場合，指定した全ての変数種別の変数が対象となります。
- 同じ変数種別を複数回指定した場合，１回指定した場合と同じ意味になります。このとき，警告を出力しません。
- < *変数種別* > に bss, data, const 以外を指定した場合はエラーとします。
- 変数の出力先はセクション名に "_< *アライメント数* >" を付加したセクションとなります。
  ただし，アライメント数が 4 の場合は，"_4" は付加されません。
  例）
  変数のアライメント数が 4：.bss
  変数のアライメント数が 2：.bss_2
  変数のアライメント数が 1：.bss_1

[使用例]

```
const char c=1;
const short s=2;
const long l=3;
```

デフォルト	-stuff 指定

`.section .const, const`
`_c:`
`.db 0x01`
`.align 2`
`_s:`
`.dhw 0x0002`
`.align 4`
`_l:`
`.dw 0x00000003`
	`.section .const_1, const, align=1`
	`_c:`
	`.db 0x01`
	
	`.section .const_2, const, align=2`
	`.align 2`
	`_s:`
	`.dhw 0x0002`
	
	`.section .const, const`
	`.align 4`
	`_l:`
	`.dw 0x00000003`

[備考]

- 各セクション名は，次のオプションまたは #pragma section での指定を反映します。
  -Xsection, -Xmulti_level

-Xcheck_exclusion_control 【V1.04.00 以降】

排他制御チェックを有効にします。

[指定形式]

```
-Xcheck_exclusion_control=< ファイル名 >
```

- 省略時解釈
  排他制御チェックを無効にします。

[詳細説明]

- 設定ファイルを読み込み，指定した位置に dbtag 命令を挿入します。
- 本機能は CS+ 経由での使用が前提であり，ユーザは直接使用しません。

-Xresbank_mode 【V2.00.00 以降】

resbank 命令の動作モードを指定します。

[指定形式]

```
-Xresbank_mode=num
```

- 省略時解釈
  -Xresbank_mode=0 を指定した場合と同じ意味になります。

[詳細説明]

- RBCR0.MD（レジスタ･バンク機能の退避モード指定レジスタ）に *num* で指定された値が設定されている状態で resbank 命令が動作するとみなして，コードを生成します。
- *num* には 0 または 1 を指定できます。RBCR0.MD への設定値と同じ値を指定してください。
- 本オプションを複数回指定した場合，最後の指定が有効になります。
- 本オプションの指定をソース・ファイルごとに変更した場合，例外ハンドラ内のレジスタ管理が正しく行われない場合があります。全てのソース・ファイルで同じ指定にしてください。
- 本オプションは，#pragma interrupt 指令で resbank を指定した例外ハンドラにのみ有効です。
- 次の場合はエラーとなります。
  - *num* を省略した場合
  - 0，1 以外を指定した場合
  - -Xcpu オプションで g4mh 以外を指定した場合

[備考]

本オプションを指定しても，RBCR0.MD へ値を設定するコードは生成されません。ユーザ・プログラムで直接設定してください。

-insert_dbtag_with_label 【V1.06.00 以降】

dbtag 命令の挿入を制御します。

[指定形式]

```
-insert_dbtag_with_label=file, line, label, tagid
```

- 省略時解釈
  dbtag 命令の挿入を行いません。

[詳細説明]

- ソース・デバッグ情報の出力を元にして，指定した位置にローカル・ラベルと dbtag 命令を挿入します。
- 本オプション指定時は，-g オプションも同時に有効になります。
- 本機能は CS+ 経由での使用が前提であり，ユーザは直接使用しません。

-store_reg 【Professional 版のみ】【V1.06.00 以降】

制御レジスタへの書き込み処理の検出や，レジスタ間の同期化処理の挿入を制御します。

[指定形式]

```
-store_reg[=mode]
```

- 省略時解釈
  #pragma register_group を記述している場合に，-store_reg=list 指定時と同じ動作をします。

[詳細説明]

- #pragma register_group を有効な #pragma 指令として認識し，*mode* で指定した動作を行います。
  #pragma register_group については，「4.2.6.14 制御レジスタへの書き込みの検出，同期化処理挿入 【Professional 版のみ】【V1.06.00 以降】」を参照してください。
- *mode* には次のいずれかを指定します。これ以外のものを指定した場合はエラーとなります。
  - list
    #pragma register_group で指定した制御レジスタへの書き込みを検出して，ソース・ファイル上の記述位置を標準エラー出力に表示します。このとき，同一グループへの書き込みが後に続くと分かる場合は表示しません。
  - list_all
    #pragma register_group で指定した制御レジスタへの書き込みを検出して，ソース・ファイル上の記述位置を標準エラー出力に表示します。同一グループへの書き込みが後に続くかどうかにかかわらず表示します。
  - sync
    #pragma register_group で指定した制御レジスタへの書き込みを検出して，書き込みの後に同期化処理を挿入します。このとき，同一グループへの書き込みが後に続くと分かる場合は挿入しません。
  - ignore
    #pragma register_group を，警告を出力せずに無視します。
  - *=mode* を省略した場合
    =list 指定と同じ動作をします。

-control_flow_integrity 【Professional 版のみ】 【V1.07.00 以降】

不正な間接関数呼び出しを検出するコードを生成します。

[指定形式]

```
-control_flow_integrity
```

- 省略時解釈
  不正な間接関数呼び出しを検出するコードを生成しません。

[詳細説明]

- 不正な間接関数呼び出しを検出するコードを生成します。
  本オプションを指定すると，C ソース・プログラム内に次の処理を行うコードを生成します。

(1) 関数の間接呼び出しが行われる直前に，間接呼び出し先のアドレスを引数に持つチェック関数 __control_flow_integrity を呼び出します。

(2) チェック関数内で，引数のアドレスが，間接呼び出しされる可能性のある関数アドレスのリスト（以降，関数リストと呼びます）の中に含まれるかをチェックし，含まれていない場合は，不正な間接呼び出しとみなして __control_flow_chk_fail 関数を呼び出します。

このように，間接関数呼び出しなどプログラムの流れを変える処理について，その正しさを検証することを，Control Flow Integrity（CFI）と呼びます。

- チェック関数は下記のように定義され，ライブラリ関数として提供しています。
  void __control_flow_integrity(void *addr);
  チェック関数を通常の関数のように呼び出すことは禁止します。
- コンパイラは，間接呼び出しされる可能性のある関数の情報を C ソース・プログラムから自動で抽出します。リンカがその情報を統合して関数リストを作成します。リンカで関数リストを作成するにはリンク・オプション -CFI の指定が必要です。
  詳細は「2.5.3 リンク・オプション」を参照してください。
- __control_flow_chk_fail 関数には不正な間接関数呼び出しの検出時に実行する処理を記述します。
  この関数はユーザが定義する必要があります。
  __control_flow_chk_fail 関数を定義する際には，次の項目に注意してください。
  - 戻り値および引数の型を void 型としてください。
  - static 関数にしないでください。
  - 通常の関数のように呼び出すことは禁止します。
  - __control_flow_chk_fail 関数は，不正な間接関数呼び出しを検出するコードの生成の対象になりません。
  - __control_flow_chk_fail 関数内では abort() を呼び出してプログラムを終了するなど，チェック関数に戻らないように注意してください。
  - -pic オプションを同時に指定した場合，エラーとなります。

［例］

- ＜C ソース＞

```
#include <stdlib.h>

int glb;

void __control_flow_chk_fail(void)
{
    abort();
}

void func1(void) // 関数リストに追加される
{
    ++glb;
}

void func2(void) // 関数リストに追加されない
{
    --glb;
}

void (*pf)(void) = func1;

void main(void)
{
    pf(); // func1 関数の間接呼び出し
    func2();
}
```

- ＜出力コード＞
  -S -control_flow_integrity を指定してコンパイルした場合

```
___control_flow_chk_fail:
    .stack ___control_flow_chk_fail = 4
    prepare 0x00000001, 0x00000000
    jarl _abort, r31
    dispose 0x00000000, 0x00000001, [r31]
_func1:
    .stack _func1 = 0
    movhi HIGHW1(#_glb), r0, r2
    ld.w LOWW(#_glb)[r2], r5
    add 0x00000001, r5
    st.w r5, LOWW(#_glb)[r2]
    jmp [r31]
_func2:
    .stack _func2 = 0
    movhi HIGHW1(#_glb), r0, r2
    ld.w LOWW(#_glb)[r2], r5
    add 0xFFFFFFFF, r5
    st.w r5, LOWW(#_glb)[r2]
    jmp [r31]
_main:
    .stack _main = 8
    prepare 0x00000041, 0x00000000
    movhi HIGHW1(#_pf), r0, r20
    ld.w LOWW(#_pf)[r20], r20
    mov r20, r6
    jarl ___control_flow_integrity, r31 ; チェック関数の呼び出し
    jarl [r20], r31 ; func1 関数の間接呼び出し
    jarl _func2, r31 ; func2 関数の直接呼出し
    dispose 0x00000000, 0x00000041, [r31]
    .section .bss, bss
    .align 4
_glb:
    .ds (4)
    .section .data, data
    .align 4
_pf:
    .dw #_func1
    .section .const, const
```

-pic 【V1.07.00 以降】

PIC 機能を有効にします。

[指定形式]

```
-pic
```

- 省略時解釈
  PIC 機能を無効にします。

[詳細説明]

- PIC 機能とは，関数の配置先セクションを位置独立にする機能です。
  PIC 機能の詳細については，「4.2.6.1 関数とデータのセクション割り当て」「8.6 PIC/PID 機能」を参照してください。
- 本オプションを指定すると，関数コードの出力先セクションを，text 属性セクションから pctext 属性セクションに変更します。
- pctext 属性セクションに配置した関数に対する参照は，すべて PC 相対で行います。これによって，pctext 属性セクションは，リンク後に任意のアドレスに配置することができます。
- 本オプション指定時は，既定義マクロ __PIC が有効になります。
- -pirod オプションと同時に指定しない場合，エラーとなります。

-pirod 【V1.07.00 以降】

PIROD 機能を有効にします。

[指定形式]

```
-pirod
```

- 省略時解釈
  PIROD 機能を無効にします。

[詳細説明]

- PIROD 機能とは，const 変数や文字列リテラルなどの，定数データの配置先セクションを位置独立にする機能です。
  PIROD 機能の詳細については，「4.2.6.1 関数とデータのセクション割り当て」「8.6 PIC/PID 機能」を参照してください。
- 本オプションを指定すると，定数データの出力先セクションを，const 属性セクションから pcconst32 属性セクションに変更します。
- pcconst32 属性セクションに配置した定数データに対する参照は，すべて PC 相対で行います。これによって，pcconst32 属性セクションは，リンク後に任意のアドレスに配置することができます。
- 本オプション指定時は，既定義マクロ __PIROD が有効になります。
- -pic オプションと同時に指定しない場合，エラーとなります。
- -Omap，-Osmap オプションと同時に指定した場合，エラーとなります。

-pid 【V1.07.00 以降】

PID 機能を有効にします。

［指定形式］

```
-pid
```

- 省略時解釈
  PID 機能を無効にします。

［詳細説明］

- PID 機能とは，変数データの配置先セクションを位置独立にする機能です。
  PID 機能の詳細については，「4.2.6.1 関数とデータのセクション割り当て」「8.6 PIC/PID 機能」を参照してください。
- 本オプションを指定すると，変数データの出力先セクションを，data，bss 属性セクションから sdata32，sbss32 属性セクションに変更します。
- sdata32，sbss32 属性セクションに配置した変数データに対する参照は，すべて GP 相対で行います。これによって，sdata32，sbss32 属性セクションは，リンク後に任意のアドレスに配置することができます。
- 本オプション指定時は，既定義マクロ __PID が有効になります。
- -r4=none オプションと同時に指定した場合，エラーとなります。
- -Omap，-Osmap オプションと同時に指定した場合，エラーとなります。

情報ファイル出力制御

情報ファイル出力制御オプションには，次のものがあります。

- -Xcref

-Xcref

静的解析情報ファイルを出力します。

[指定形式]

```
-Xcref=path
```

- 省略時解釈
  静的解析情報ファイルを出力しません。

[詳細説明]

- コンパイル途中に生成される静的解析情報ファイルの保存先を *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，C ソース・ファイル名の拡張子を “.cref” で置き換えたファイル名で静的解析情報ファイルを保存します。
- *path* に存在するファイル名を指定した場合，および存在しないフォルダ名，またはファイル名を指定した場合は，出力する静的解析情報ファイルが 1 つの場合は，*path* というファイル名で保存します。
  出力する静的解析情報ファイルが複数の場合は，エラーとなります。
- *=path* を省略した場合は，エラーとなります。
- ソース・ファイルとして同じ名前のファイル（異なるフォルダにある場合を含む）を複数指定した場合は，警告を出力して，最後に指定したソース・ファイルに対する静的解析情報ファイルのみを保存します。

[使用例]

- 静的解析情報ファイルをファイル名 info.cref で出力します。

```
>ccrh -Xcref=info.cref -Xcommon=rh850 main.c
```

エラー出力制御

エラー出力制御オプションには，次のものがあります。

- -Xerror_file

-Xerror_file

エラー・メッセージをファイルに出力します。

[指定形式]

```
-Xerror_file=file
```

- 省略時解釈
  エラー・メッセージを標準エラー出力のみに出力します。

[詳細説明]

- エラー・メッセージを標準エラー出力，およびファイル *file* に出力します。
- *file* がすでに存在する場合は，そのファイルを上書きします。
- *file* を省略した場合は，エラーとなります。

[使用例]

- エラー・メッセージを標準エラー出力，およびファイル err に出力します。

```
>ccrh -Xerror_file=err -Xcommon=rh850 main.c
```

警告メッセージ出力制御

警告メッセージ出力制御オプションには，次のものがあります。

- -Xno_warning
- -change_message 【V1.07.00 以降】

-Xno_warning

指定した警告メッセージの出力を抑止します。

[指定形式]

```
-Xno_warning={num|num1-num2}[, ...]
```

- 省略時解釈
  すべての警告メッセージを出力します。

[詳細説明]

- 指定した警告メッセージの出力を抑止します。
- *num*，*num1*，*num2* には，メッセージ番号を指定します。
  存在しないメッセージ番号を指定した場合は，無視します。
- *num*，または *num1*，および *num2* を省略した場合は，エラーとなります。
- *num1-num2* の形式で指定すると，その範囲に含まれるメッセージ番号を指定したものとみなします。
- 本オプションを複数指定した場合は，すべての指定が有効になります。
- 本オプションで指定するメッセージ番号は，W に続く 7 桁の数字のうち，下位 5 桁です。
  メッセージ番号については，「10. メッセージ」を参照してください。
- -change_message オプションでエラーに変更したメッセージは，本オプションの制御の対象外になります。
- 本オプションが対象とするメッセージ番号は次の通りです。
  - W0520000 ～ W0529999，W0550000 ～ W0559999 【V1.06.00 以前】
  - W0510000 ～ W0559999 【V1.07.00 以降】

[使用例]

- 警告メッセージ W0520111 の出力を抑止します。

```
>ccrh -Xno_warning=20111 -Xcommon=rh850 main.c
```

-change_message 【V1.07.00 以降】

指定した警告メッセージをエラーメッセージに変更します。

[指定形式]

```
-change_message=error={num|num1-num2}[, ...]
```

[詳細説明]

- 指定した警告メッセージをエラーメッセージに変更します。
  本オプションが対象とするメッセージ番号は，W0510000 ～ W0549999 です。
- *num*，*num1*，*num2* には，メッセージ番号の下位 5 桁を指定します。
  num1-num2 の形式で指定すると，その範囲に含まれるメッセージ番号を指定したものとみなします。
- *num*，または *num1-num2* を省略した場合は，対象の全ての警告メッセージをエラーメッセージに変更します。
- 本オプションを複数指定した場合は，すべての指定が有効になります。
- 存在しないメッセージ番号を指定した場合は，無視します。
  メッセージ番号については，「10. メッセージ」を参照してください。

[使用例]

- W0520000 ～ W0549999 をエラーに変更します。

```
>ccrh -change_message=error=20000-49999 a.c
```

フェーズ個別オプション指定

フェーズ個別オプション指定オプションには，次のものがあります。

- -Xasm_option
- -Xlk_option

-Xasm_option

アセンブル・オプションを指定します。

[指定形式]

```
-Xasm_option=arg
```

- 省略時解釈
  指定したオプションはすべて ccrh のドライバが解釈します。

[詳細説明]

- *arg* をアセンブル・オプションとして，アセンブラに渡します。
- *arg* が存在しないアセンブル・オプションである場合は，エラーとなります。
- *arg* を省略した場合は，エラーとなります。

[使用例]

- -Xprn_path オプションをアセンブラに渡します。

```
>ccrh -Xasm_option=-Xprn_path -Xcommon=rh850 main.c
```

上記の例は，以下と同じ意味となります。

```
>ccrh -S -Xcommon=rh850 main.c
>asrh -Xprn_path -Xcommon=rh850 main.asm
```

-Xlk_option

リンク・オプションを指定します。

[指定形式]

```
-Xlk_option=arg
```

- 省略時解釈
  指定したオプションはすべて ccrh のドライバが解釈します。

[詳細説明]

- *arg* をリンク・オプションとして，最適化リンカに渡します。
- ccrh のドライバが最適化リンカへの入力として認識しない識別子を持つファイルを最適化リンカに渡す際には，本オプションを使用します。
- *arg* が存在しないリンク・オプションである場合は，エラーとなります。
- *arg* を省略した場合は，エラーとなります。

[使用例]

- -form=relocate オプションを最適化リンカに渡します。

```
>ccrh -Xlk_option=-form=relocate -Xcommon=rh850 main.c
```

上記の例は，以下と同じ意味となります。

```
>ccrh -c -Xcommon=rh850 main.c
>rlink -form=relocate main.obj
```

サブコマンド・ファイル指定

サブコマンド・ファイル指定オプションには，次のものがあります。

- @

@

サブコマンド・ファイルを指定します。

[指定形式]

```
@file
```

- 省略時解釈
  コマンド・ラインで指定したオプション，およびファイル名のみを認識します。

[詳細説明]

- *file* をサブコマンド・ファイルとして扱います。
- *file* が存在しない場合は，エラーとなります。
- *file* を省略した場合は，エラーとなります。
- サブコマンド・ファイルについての詳細は，「2.4.2 サブコマンド・ファイルの使用方法」を参照してください。

[使用例]

- command.txt をサブコマンド・ファイルとして扱います。

```
>ccrh @command.txt -Xcommon=rh850
```

2.5.2 アセンブル・オプション

ここでは，アセンブル・フェーズのオプションについて説明します。

オプションに関する注意事項を以下に示します。

- オプションの大文字／小文字は区別します。
- パラメータとして数値を指定する場合は，10 進数，または “0x”（“0X”）で始まる 16 進数での指定が可能です。16 進数のアルファベットは，大文字／小文字を区別しません。
- パラメータとしてファイル名を指定する場合は，パス付き（絶対パス，または相対パス）での指定が可能です。パスなし，および相対パスで指定する場合は，カレント・フォルダを基準とします。
- パラメータ中に空白を含める場合（パス名など）は，そのパラメータ全体をダブルクォーテーション（"）で囲んでください。
- ccrh コマンドに対して，-Xprn_path，または -Xasm_far_jump オプションを指定する場合は，-Xasm_option オプションを使用する必要があります。

オプションの分類と説明を以下に示します。

表 2.3 アセンブル・オプション

分類	オプション	説明
バージョン／ヘルプ表示指定	-V	asrh のバージョン情報を表示します。
	-h	asrh のオプションの説明を表示します。
出力ファイル指定	-o	出力ファイル名を指定します。
	-Xobj_path	アセンブル途中に生成されるオブジェクト・ファイルの保存先を指定します。
	-Xprn_path	アセンブル・リスト・ファイルの保存先を指定します。
ソース・デバッグ制御	-g	ソース・デバッグ用の情報を出力します。
デバイス指定	-Xcommon	デバイス共通のオブジェクト・ファイルを生成することを指定します。
	-Xcpu	指定したコア向けのオブジェクトを生成することを指定します。
最適化	-goptimize 【V2.01.00 以降】	リンク時最適化用の情報を生成します。
シンボル定義指定	-D	アセンブラ・シンボルを定義します。
	-U	-D オプションによるアセンブラ・シンボルの定義を解除します。
インクルード・ファイル読み込みパス指定	-I	インクルード・ファイルを検索するフォルダを指定します。
日本語／中国語文字列制御	-Xcharacter_set	日本語／中国語の文字コードを指定します。

分類	オプション	説明
生成コード制御	-Xreg_mode	レジスタ・モードを指定します。
	-Xreserve_r2	r2 レジスタを予約します。
	-Xep	ep レジスタの扱い方を指定します。
	-pic 【V1.07.00 以降】	PIC 機能を有効にします。
	-pirod 【V1.07.00 以降】	PIROD 機能を有効にします。
	-pid 【V1.07.00 以降】	PID 機能を有効にします。
アセンブラ制御指定	-Xasm_far_jump	アセンブリ・ソース・ファイルに対して，far jump の出力を制御します。
エラー出力制御	-Xerror_file	エラー・メッセージをファイルに出力します。
警告メッセージ出力制御	-Xno_warning	指定した警告メッセージの出力を抑止します。
サブコマンド・ファイル指定	@	サブコマンド・ファイルを指定します。

バージョン／ヘルプ表示指定

バージョン／ヘルプ表示指定オプションには，次のものがあります。

- -V
- -h

-V

asrh のバージョン情報を表示します。

[指定形式]

```
-V
```

- 省略時解釈
  asrh のバージョン情報を表示せずに，アセンブルを行います。

[詳細説明]

- asrh のバージョン情報を標準エラー出力に出力します。
  アセンブルは行いません。

[使用例]

- asrh のバージョン情報を標準エラー出力に出力します。

```
>asrh -V -Xcommon=rh850
```

-h

asrh のオプションの説明を表示します。

[指定形式]

```
-h
```

- 省略時解釈
  asrh のオプションの説明を表示しません。

[詳細説明]

- asrh のオプションの説明を標準エラー出力に出力します。
  アセンブルは行いません。

[使用例]

- asrh のオプションの説明を標準エラー出力に出力します。

```
>asrh -h -Xcommon=rh850
```

出力ファイル指定

出力ファイル指定オプションには，次のものがあります。

- -o
- -Xobj_path
- -Xprn_path

-o

出力ファイル名を指定します。

[指定形式]

```
-ofile
```

- 省略時解釈
  カレント・フォルダにファイルを出力します。
  出力オブジェクト・ファイル名は，ソース・ファイル名の拡張子を “.obj” に置き換えたものとなります。

[詳細説明]

- 出力オブジェクト・ファイル名を *file* に指定します。
- *file* がすでに存在する場合は，そのファイルを上書きします。
- 本 オプションを指定しても，エラーがある場合には，オブジェクト・ファイルの出力は行いません。
- 出力ファイルが複数の場合は，エラーとなります。
- *file* を省略した場合は，エラーとなります。

[使用例]

- オブジェクト・ファイルをファイル名 sample.obj で出力します。

```
>asrh -osample.obj -Xcommon=rh850 main.asm
```

-Xobj_path

アセンブル途中に生成されるオブジェクト・ファイルを保存するフォルダを指定します。

[指定形式]

```
-Xobj_path[=path]
```

- 省略時解釈
  カレント・フォルダに，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。

[詳細説明]

- アセンブル途中に生成されるオブジェクト・ファイルを保存するフォルダを *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。
  存在しないフォルダ名を指定した場合は，エラーとなります。
- *path* には存在するファイル名を指定することも可能です。
  出力するオブジェクト・ファイルが 1 つの場合は，*path* というファイル名で保存します。
  出力するオブジェクト・ファイルが複数の場合は，エラーとなります。
  存在しないファイル名を指定した場合は，エラーとなります。
- =*path* を省略した場合は，カレント・フォルダに，ソース・ファイル名の拡張子を “.obj” で置き換えたファイル名でオブジェクト・ファイルを保存します。
- ソース・ファイルとして同じ名前のファイル（異なるフォルダにある場合を含む）を複数指定した場合は，警告を出力して，最後に指定したソース・ファイルに対するオブジェクト・ファイルのみを保存します。

[使用例]

- アセンブル途中に生成されるオブジェクト・ファイルをフォルダ D:¥sample に保存します。

```
>asrh -Xobj_path=D:¥sample -Xcommon=rh850 main.asm
```

-Xprn_path

アセンブル・リスト・ファイルを保存するフォルダを指定します。

[指定形式]

```
-Xprn_path[=path]
```

- 省略時解釈
  アセンブル・リスト・ファイルを出力しません。

[詳細説明]

- アセンブル時に出力するアセンブル・リスト・ファイルを保存するフォルダを *path* に指定します。
- *path* に存在するフォルダ名を指定した場合は，フォルダ *path* に，ソース・ファイル名の拡張子を “.prn” で置き換えたファイル名でアセンブル・リスト・ファイルを保存します。
  存在しないフォルダ名を指定した場合は，エラーとなります。
- *path* には存在するファイル名を指定することも可能です。
- *path* というファイル名でアセンブル・リスト・ファイルを保存します。
  存在しないファイル名を指定した場合は，エラーとなります。
- =*path* を省略した場合は，カレント・フォルダに，ソース・ファイル名の拡張子を “.prn” で置き換えたファイル名でアセンブル・リスト・ファイルを保存します。

[使用例]

- アセンブル時に出力するアセンブル・リスト・ファイルをフォルダ D:¥sample に保存します。

```
>asrh -Xprn_path=D:¥sample -Xcommon=rh850 main.asm
```

ソース・デバッグ制御

ソース・デバッグ制御オプションには，次のものがあります。

- -g

-g

ソース・デバッグ用の情報を出力します。

[指定形式]

```
-g
```

- 省略時解釈
  ソース・デバッグ用の情報を出力しません。

[詳細説明]

- ソース・デバッグ用の情報を出力ファイル中に出力します。
- 本オプションを指定することにより，ソース・デバッグが可能となります。

[使用例]

- ソース・デバッグ用の情報を出力ファイル中に出力します。

```
>asrh -g -Xcommon=rh850 main.asm
```

デバイス指定

デバイス指定オプションには，次のものがあります。

- -Xcommon
- -Xcpu

-Xcommon

デバイス共通のオブジェクト・ファイルを生成することを指定します。

[指定形式]

```
-Xcommon=series
```

- 省略時解釈
  なし

[詳細説明]

- デバイス共通のオブジェクト・ファイルを生成することを指定します。
- V2.00.00 以降では，本オプションは無効です。指定した場合，無視されますが，従来バージョンとの互換性のため，エラーにはなりません。このとき，警告を出力しません。
- *series* には v850e3v5 または rh850 を指定可能です。
- 次の場合はエラーとなります。
  - *series* を省略した場合
  - *series* に指定可能なパラメータ以外を指定した場合
  - 本オプション指定を省略した場合 【V1.01.00 以前】

[備考]

本オプションは出力コードに影響しません。
使用する命令セットを選択する場合は，-Xcpu オプションを指定してください。

-Xcpu

指定したコア向けのオブジェクトを生成することを指定します。

[指定形式]

```
-Xcpu=core
```

- 省略時解釈
  G3M 向けオブジェクトを生成します。

[詳細説明]

- コア *core* 向けのオブジェクトを生成することを指定します。
- *core* に指定可能なものを以下に示します。

g3m	G3M 向けオブジェクトを生成します。
g3k	G3K 向けオブジェクトを生成します。
g3mh	G3MH 向けオブジェクトを生成します。【V1.02.00 以降】
g3kh	G3KH 向けオブジェクトを生成します。【V1.03.00 以降】
g4mh	G4MH 向けオブジェクトを生成します。【V2.00.00 以降】

- 本オプションを複数回指定した場合，最後の指定が有効になります。
- 次の場合はエラーとなります。
  - パラメータを省略した場合
  - 指定可能なパラメータ以外を指定した場合

最適化

最適化オプションには，次のものがあります。

- -goptimize 【V2.01.00 以降】

-goptimize 【V2.01.00 以降】

リンク時最適化用の情報を生成します。

[指定形式]

```
-goptimize
```

- 省略時解釈
  なし

[詳細説明]

- リンク時最適化時に使用する付加情報を出力ファイル内部に生成します。
- 本オプションを指定したファイルは，リンク時にリンク時最適化の対象になります。
  リンク時最適化の詳細については，リンク・オプション -OPtimize を参照してください。

シンボル定義指定

シンボル定義指定オプションには，次のものがあります。

- -D
- -U

-D

アセンブラ・シンボルを定義します。

[指定形式]

```
-Dname[=def][name[=def]]...
```

- 省略時解釈
  なし

[詳細説明]

- アセンブラ・シンボルとして *name* を定義します。
- *def* の指定方式は次の通りです。
  - 整数値のみ指定可能
  - 整数値以外を指定した場合は 0 とみなす
  - 整数値は 10 進，prefix 方式の 8 進 (0...)，16 進記法 (0x...) が可能
  - 先頭の - 符号は指定可能だが，+ 符号は不可
  - 負数は 2 の補数に変換される
- アセンブリ・ソース・プログラムの前に，*name* .SET *def* を記述するのと同等です。
- *name* を省略した場合は，エラーとなります。
- =*def* を省略した場合，*def* は 1 とみなします。
- 本オプションは，複数指定が可能です。
- 同じアセンブラ・シンボルに対して，本オプションと -U オプションを同時に指定した場合は，あとから指定したものが有効となります。

[使用例]

- アセンブラ・シンボルとして sample=256 を定義します。

```
>asrh -Dsample=256 -Xcommon=rh850 main.asm
```

-U

-D オプションによるアセンブラ・シンボルの定義を解除します。

[指定形式]

```
-Uname[,name]...
```

- 省略時解釈
  なし

[詳細説明]

- -D オプションによるアセンブラ・シンボル *name* の定義を解除します。
- *name* を省略した場合は，エラーとなります。
- 本オプションでは，*name* .SET *def* の記述による定義は解除できません。
- 本オプションは，複数指定が可能です。
- 同じアセンブラ・シンボルに対して，本オプションと -D オプションを同時に指定した場合は，あとから指定したものが有効となります。

[使用例]

- -D オプションによるアセンブラ・シンボル test の定義を解除します。

```
>asrh -Utest -Xcommon=rh850 main.asm
```

インクルード・ファイル読み込みパス指定

インクルード・ファイル読み込みパス指定オプションには，次のものがあります。

- -I

-I

インクルード・ファイルを検索するフォルダを指定します。

[指定形式]

```
-Ipath[,path]...
```

- 省略時解釈
  インクルード・ファイルを 標準インクルード・ファイル・フォルダからのみ検索します。

[詳細説明]

- アセンブラ制御命令 $INCLUDE/$BINCLUDE で読み込むインクルード・ファイルを検索するフォルダを *path* に指定します。
  インクルード・ファイルの検索は，以下の順番で行います。
  (1) -I オプションで指定したフォルダ（指定が複数ある場合は，コマンド・ラインで指定した順（左から右の順））
  (2) ソース・ファイルのあるフォルダ
  (3) カレント・フォルダ

- *path* が存在しない場合は，警告を出力します。
- *path* を省略した場合は，エラーとなります。

[使用例]

- インクルード・ファイルをフォルダ D:¥include，D:¥src，カレント・フォルダの順で検索します。

```
>asrh -ID:¥include -Xcommon=rh850 D:¥src¥main.asm
```

日本語／中国語文字列制御

日本語／中国語文字列制御オプションには，次のものがあります。

- -Xcharacter_set

-Xcharacter_set

日本語／中国語の文字コードを指定します。

[指定形式]

```
-Xcharacter_set=code
```

- 省略時解釈
  日本語の文字コードを SJIS として扱います。

[詳細説明]

- ソース・ファイル中の日本語／中国語のコメント，文字列に対して，使用する文字コードを指定します。
- *code* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。
  なお，ソース・ファイル中で使用している文字コードと異なるものを指定した場合，動作は保証されません。

none	日本語／中国語の文字コードを解釈しません
euc_jp	EUC（日本語）
sjis	SJIS
utf8	UTF-8
big5	繁体字中国語
gb2312	簡体字中国語

- *code* を省略した場合は，エラーとなります。

[使用例]

- ソース・ファイル中の日本語のコメント，文字列に対して，使用する文字コードに EUC を指定します。

```
>asrh -Xcharacter_set=euc_jp -Xcommon=rh850 main.asm
```

生成コード制御

生成コード制御オプションには，次のものがあります。

- -Xreg_mode
- -Xreserve_r2
- -Xep
- -pic 【V1.07.00 以降】
- -pirod 【V1.07.00 以降】
- -pid 【V1.07.00 以降】

-Xreg_mode

レジスタ・モードを指定します。

[指定形式]

```
-Xreg_mode=mode
```

- 省略時解釈
  32 レジスタ・モードのオブジェクト・ファイルを生成します。

[詳細説明]

- 指定したレジスタ・モードのオブジェクト・ファイルを生成します。
- ccrh が使用するレジスタを 32 本（32 レジスタ・モード），22 本（22 レジスタ・モード，または common レジスタ・モード）のいずれかに制限し，オブジェクト・ファイル内にレジスタ・モードを示すマジック・ナンバを埋め込みます。
- common レジスタ・モードは，レジスタ・モードに依存しないオブジェクト・ファイルを生成するために使用します。
- *mode* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

レジスタ・モード（*mode*）	作業用レジスタ	レジスタ変数用レジスタ
common	r10 ～ r14	r25 ～ r29
22	r10 ～ r14	r25 ～ r29
32	r10 ～ r19	r20 ～ r29

- *mode* を省略した場合は，エラーとなります。
- 32 レジスタ・モードのオブジェクト・ファイルと，22 レジスタ・モードのオブジェクト・ファイルが混在している場合は，リンク時にエラーとなります。

[使用例]

- 22 レジスタ・モードのオブジェクト・ファイルを生成します。

```
>asrh -Xreg_mode=22 -Xcommon=rh850 main.asm
```

-Xreserve_r2

r2 レジスタを予約します。

[指定形式]

```
-Xreserve_r2
```

- 省略時解釈
  r2 レジスタの予約を行わずに，コンパイラで使用します。

[詳細説明]

- r2 レジスタを予約し，コンパイラでは r2 レジスタを使用しないコードを生成します。

[使用例]

- r2 レジスタを予約し，コンパイラでは r2 レジスタを使用しないコードを生成します。

```
>asrh -Xreserve_r2 -Xcommon=rh850 main.asm
```

-Xep

ep レジスタの扱い方を指定します。

[指定形式]

```
-Xep=mode
```

- 省略時解釈
  ep レジスタを関数呼び出し前後で値を保証するレジスタとして扱います。

[詳細説明]

- ep レジスタの扱い方を指定します。
- *mode* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

fix	ep レジスタの値をプロジェクト全体で固定します。 プロジェクト内で EP 相対セクションを使用する場合は，本パラメータを指定してください。
callee	ep レジスタを関数呼び出し前後で値を保証するレジスタとして扱います。 -Omap，または -Osmap オプションを指定した場合は，本パラメータを指定してください。

- *mode* を省略した場合は，エラーとなります。
- 本オプションは，すべてのソース・ファイルに対して同じ指定をする必要があります。ソース・ファイルごとに指定を変えることはできません。指定の異なるオブジェクト・ファイルが混在している場合は，リンク時にエラーとなります。

[使用例]

- ep レジスタの値をプロジェクト全体で固定します。

```
>asrh -Xep=fix main.asm
```

-pic 【V1.07.00 以降】

PIC 機能を有効にします。

［指定形式］

```
-pic
```

- 省略時解釈
  PIC 機能を無効にします。

［詳細説明］

- 本オプションを指定すると，.cseg，.section 疑似命令に指定可能な再配置属性が変更されます。
  次の再配置属性を指定するとエラーになります。
  　オプション指定時：TEXT
  　オプション未指定時：PCTEXT
- 本オプション指定時は，既定義マクロ __PIC が有効になります。
- -pirod オプションと同時に指定しない場合，エラーとなります。
- 本オプションは，指定可能な再配置属性を制御するのみです。関数への参照コードに対するエラー判定処理は行いません。

-pirod 【V1.07.00 以降】

PIROD 機能を有効にします。

[指定形式]

```
-pirod
```

- 省略時解釈
  PIROD 機能を無効にします。

[詳細説明]

- 本オプションを指定すると，.cseg，.section 疑似命令に指定可能な再配置属性が変更されます。
  次の再配置属性を指定するとエラーになります。
  オプション指定時：CONST，ZCONST，ZCONST23
  オプション未指定時：PCCONST16，PCCONST23，PCCONST32
- 本オプション指定時は，既定義マクロ __PIROD が有効になります。
- -pic オプションと同時に指定しない場合，エラーとなります。
- 本オプションは，指定可能な再配置属性を制御するのみです。定数データへの参照コードに対するエラー判定処理は行いません。

-pid 【V1.07.00 以降】

PID 機能を有効にします。

[指定形式]

```
-pid
```

- 省略時解釈
  PID 機能を無効にします。

[詳細説明]

- 本オプションを指定すると，.dseg，.section 疑似命令に指定可能な再配置属性が変更されます。
  次の再配置属性を指定するとエラーになります。
  オプション指定時：DATA，ZDATA，ZDATA23，BSS，ZBSS，ZBSS23
  オプション未指定時：SDATA32，SBSS32，EDATA32，EBSS32
- 本オプション指定時は，既定義マクロ __PID が有効になります。
- 本オプションは，指定可能な再配置属性を制御するのみです。データへの参照コードに対するエラー判定処理は行いません。

アセンブラ制御指定

アセンブラ制御指定オプションには，次のものがあります。

- -Xasm_far_jump

-Xasm_far_jump

アセンブリ・ソース・ファイルに対して，far jump の出力を制御します。

[指定形式]

```
-Xasm_far_jump
```

- 省略時解釈
  jarl，または jr 命令としてアセンブルを行います。

[詳細説明]

- アセンブリ・ソース・ファイルに対して，ソース中に記述されたすべての jarl，および jr 命令を jarl32，および jr32 命令とみなしてアセンブルを行います。
- 個別の命令ごとに制御したい場合は，ソース中で jarl22/jarl32，jr22/jr32 のように明記します。
- 本オプションは，jump 命令には影響しません。
- 本オプションは，C ソース・ファイルに対しては無効です。

[使用例]

- アセンブリ・ソース中に記述されたすべての jarl，および jr 命令を jarl32，および jr32 命令とみなしてアセンブルを行います。

```
>asrh -Xasm_far_jump -Xcommon=rh850 main.asm
```

エラー出力制御

エラー出力制御オプションには，次のものがあります。

- -Xerror_file

-Xerror_file

エラー・メッセージをファイルに出力します。

[指定形式]

```
-Xerror_file=file
```

- 省略時解釈
  エラー・メッセージを標準エラー出力のみに出力します。

[詳細説明]

- エラー・メッセージを標準エラー出力，およびファイル *file* に出力します。
- *file* がすでに存在する場合は，そのファイルを上書きします。
- *file* を省略した場合は，エラーとなります。

[使用例]

- エラー・メッセージを標準エラー出力，およびファイル err に出力します。

```
>asrh -Xerror_file=err -Xcommon=rh850 main.asm
```

警告メッセージ出力制御

警告メッセージ出力制御オプションには，次のものがあります。

- -Xno_warning

-Xno_warning

指定した警告メッセージの出力を抑止します。

[指定形式]

```
-Xno_warning={num|num1-num2}[, ...]
```

- 省略時解釈
  すべての警告メッセージを出力します。

[詳細説明]

- 指定した警告メッセージの出力を抑止します。
- *num*，*num1*，*num2* には，メッセージ番号を指定します。
  存在しないメッセージ番号を指定した場合は，無視します。
- *num*，または *num1*，および *num2* を省略した場合は，エラーとなります。
- *num1-num2* の形式で指定すると，その範囲に含まれるメッセージ番号を指定したものとみなします。
- 本オプションで指定するメッセージ番号は，W に続く 7 桁の数字のうち，下位 5 桁です。
  メッセージ番号については，「10. メッセージ」を参照してください。
- 本オプションで制御することができるのは，メッセージの番号（コンポーネント番号を含む）が 0550000 ～ 0559999 であるもののみです。

[使用例]

- 警告メッセージ W0550002，W0550003 の出力を抑止します。

```
>asrh -Xno_warning=50002,50003 -Xcommon=rh850 main.asm
```

サブコマンド・ファイル指定

サブコマンド・ファイル指定オプションには，次のものがあります。

- @

@

サブコマンド・ファイルを指定します。

[指定形式]

```
@file
```

- 省略時解釈
  コマンド・ラインで指定したオプション，およびファイル名のみを認識します。

[詳細説明]

- *file* をサブコマンド・ファイルとして扱います。
- *file* が存在しない場合は，エラーとなります。
- *file* を省略した場合は，エラーとなります。
- サブコマンド・ファイルについての詳細は，「2.4.2 サブコマンド・ファイルの使用方法」を参照してください。

[使用例]

- command.txt をサブコマンド・ファイルとして扱います。

```
>asrh @command.txt -Xcommon=rh850
```

2.5.3 リンク・オプション

ここでは，リンク・フェーズのオプションについて説明します。

オプションに関する注意事項を以下に示します。

- オプションの大文字／小文字は区別しません。
- オプション，およびパラメータの大文字は，短縮形の指定が可能であることを表しています。
  大文字以降の文字の指定は任意です。

  例 -FOrm=Absolute の場合，例えば以下のように指定することも可能です。
  -fo=a
  -fo=abs
  -for=absolu

- パラメータとしてファイル名を指定する場合，“(”，および “)” を使用することはできません。
- ccrh コマンドに対して，リンク・オプションを指定する場合は，-Xlk_option オプションを使用する必要があります。

オプションの分類と説明を以下に示します。

表 2.4 リンク・オプション

分類	オプション	説明
入力制御	-Input	入力ファイルを指定します。
	-LIBrary	入力ライブラリ・ファイルを指定します。
	-Binary	入力バイナリ・ファイルを指定します。
	-DEFine	未定義シンボルを強制定義します。
	-ENTry	実行開始アドレスを指定します。
	-ALLOW_DUPLICATE_MODULE_NAME 【V2.02.00 以降】	複数の同じモジュール名の指定を許可します。

分類	オプション	説明
出力制御	-FOrm	出力形式を指定します。
	-DEBug	出力ファイル中にデバッグ情報を出力します。
	-NODEBug	デバッグ情報を出力しません。
	-RECord	出力するデータ・レコードのサイズを指定します。
	-END_RECORD 【V1.06.00 以降】	エンドレコードを指定します。
	-ROm	ROM から RAM へマップするセクションを指定します。
	-OUtput	出力ファイルを指定します。
	-MAp	外部シンボル割り付け情報ファイルを出力します。
	-SPace	出力範囲のメモリの空き領域を充てんします。
	-Message	インフォメーション・メッセージを出力します。
	-NOMessage	インフォメーション・メッセージの出力を抑止します。
	-MSg_unused	参照されない外部定義シンボルをユーザに通知します。
	-BYte_count	データ・レコードのバイト数の最大値を指定します。
	-FIX_RECORD_LENGTH_AND_ALIGN 【V1.07.00 以降】	データ・レコードの出力フォーマットを固定します。
	-PADDING	セクションの終端にデータを埋め込みます。
	-OVERRUN_FETCH	オーバーラン・フェッチに伴う未初期化領域の読み出しを回避します。
	-RESERVE_PREFETCH_AREA【V2.04.01 以降】	プリフェッチの可能性がある領域にセクションを生成して確保します。
	-CRc	CRC コードを出力します。
	-CFI 【Professional 版のみ】【V1.07.00 以降】	不正な間接関数呼び出し検出で用いる関数リストを生成します。
	-CFI_ADD_Func 【Professional 版のみ】【V1.07.00 以降】	不正な間接関数呼び出し検出で用いる関数リストに追加する関数シンボルまたはアドレスを指定します。
	-CFI_IGNORE_Module【Professional 版のみ】【V1.07.00 以降】	不正な間接関数呼び出し検出で用いる関数リストから除外するモジュールを指定します。
リスト出力	-LISt	リスト・ファイルを出力します。
	-SHow	リスト・ファイルへの出力情報を指定します。
最適化	-OPtimize / -NOOPtimize【V2.01.00 以降】	リンク時最適化の実行有無を指定します。
	-SEction_forbid 【V2.01.00 以降】	特定セクションの，リンク時最適化を抑止します。
	-Absolute_forbid 【V2.01.00 以降】	特定アドレス範囲内の，リンク時最適化を抑止します。
	-SYmbol_forbid 【V2.01.00 以降】	特定シンボルの，リンク時最適化を抑止します。
	-ALLOW_OPTIMIZE_ENTRY_BLOCK【V2.06.00 以降】	実行開始シンボルより前に配置されている領域を最適化の対象にします。

分類	オプション	説明
セクション指定	-STARt	セクションの開始アドレスを指定します。
	-FSymbol	外部定義シンボルをシンボル・アドレス・ファイルに出力します。
	-ALIGNED_SECTION	セクションのアライメント数を 16 バイトに変更します。
ベリファイ指定	-CPu	セクションの割り付けアドレスの整合性をチェックします。
サブコマンド・ファイル指定	-SUbcommand	オプションをサブコマンド・ファイルで指定します。
その他	-S9	S9 レコードを終端に出力します。
	-STACk	スタック情報ファイルを出力します。
	-COmpress	デバッグ情報を圧縮します。
	-NOCOmpress	デバッグ情報を圧縮しません。
	-MEMory	リンク時に使用するメモリ量を指定します。
	-REName	外部シンボル名，セクション名を変更します。
	-LIB_REName 【V2.01.00 以降】	ライブラリから入力されたシンボル名，セクション名を変更します。
	-DELete	外部シンボル名，またはライブラリ・モジュールを削除します。
	-REPlace	ライブラリ・モジュールを置換します。
	-EXTract	ライブラリ・モジュールを抽出します。
	-STRip	ロード・モジュール・ファイル，ライブラリ・ファイルのデバッグ情報を削除します。
	-CHange_message	インフォメーション，ワーニング，エラーのメッセージ種別を変更します。
	-Hide	出力ファイル内のローカル・シンボル名情報を消去します。
	-Total_size	リンク後の合計セクション・サイズを標準エラー出力に表示します。
	-VERBOSE 【V2.03.00 以降】	詳細情報を標準エラー出力に表示します。
	-LOgo	コピーライトを出力します。
	-NOLOgo	コピーライトの出力を抑止します。
	-END	本オプションより前に指定したオプション列を実行します。
	-EXIt	オプション指定の終了を指定します。

入力制御

入力制御オプションには，次のものがあります。

- -Input
- -LIBrary
- -Binary
- -DEFine
- -ENTry
- -ALLOW_DUPLICATE_MODULE_NAME 【V2.02.00 以降】

-Input

入力ファイルを指定します。

[指定形式]

```
-Input=suboption [{,|△} ...]
  suboption := file
             |  file ( module [, ...] )
```

- 省略時解釈
  なし

[詳細説明]

- 入力ファイル *file* を指定します。
  複数指定する場合は，カンマ（,），または空白文字で区切ります。
- ワイルドカード（*，?）も使用可能です。
  ワイルドカードで指定した文字列は，アルファベット順に展開します。
  数字と英文字では数字を先に，英大文字と英小文字では英大文字を先に展開します。
- 入力ファイルとして指定できるのは，コンパイラ，またはアセンブラが出力したオブジェクト・ファイル，最適化リンカが出力したリロケータブル・ファイル，ロード・モジュール・ファイル，インテル拡張ヘキサ・ファイル，およびモトローラ・S タイプ・ファイルです。
  また，“*library*(*module*)” の形式で，ライブラリ内モジュールを指定することもできます。
  モジュール名は拡張子なしで指定します。
- 入力ファイル名に拡張子の指定がないとき，モジュール名の指定がない場合は “.obj”，モジュール名の指定がある場合は “.lib” を指定したものとみなします。

[注意]

- 本オプションは，サブコマンド・ファイル内のみで使用することができます。
  本オプションをコマンド・ライン上で指定した場合は，エラーとなります。
  コマンド・ライン上で入力ファイルを指定する場合は，-input オプションなしで指定してください。

[使用例]

- a.obj と lib1.lib 内のモジュール e を入力します。
  ＜コマンド・ライン＞

```
>rlink -subcommand=sub.txt
```

＜サブコマンド・ファイル sub.txt ＞

```
-input=a.obj lib1(e)
```

- “c” で始まり，拡張子が “.obj” であるファイルをすべて入力します。
  ＜コマンド・ライン＞

```
>rlink -subcommand=sub.txt
```

＜サブコマンド・ファイル sub.txt ＞

```
-input=c*.obj
```

［備考］

- 本オプションは，-form=object，および -extract オプションを指定した場合は無効となります。
- 入力ファイルにインテル拡張ヘキサ・ファイルを指定した場合は -form=hexadecimal オプション，モトローラ・S タイプ・ファイルを指定した場合は -form=stype オプションのみを指定することができます。
  出力ファイル名を指定していない場合は，“ *先頭入力ファイル名* _combine. *拡張子* ” となります（入力ファイルが a.mot の場合，出力ファイルは a_combine.mot となります）。

-LIBrary

入力ライブラリ・ファイルを指定します。

[指定形式]

```
-LIBrary=file[,file]...
```

- 省略時解釈
  なし

[詳細説明]

- 入力ライブラリ・ファイル *file* を指定します。
  複数指定する場合は，カンマ（,）で区切ります。
- ワイルドカード（*，?）も使用可能です。
  ワイルドカードで指定した文字列は，アルファベット順に展開します。
  数字と英文字では数字を先に，英大文字と英小文字では英大文字を先に展開します。
- 入力ファイル名に拡張子の指定がない場合は “.lib” を指定したものとみなします。
- -form=library，または -extract オプションと同時に指定した場合は，指定したライブラリ・ファイルを編集対象ライブラリとして入力します。
  それ以外の場合は，入力ファイルとして指定されたファイル間でのリンク処理後に，未定義シンボルをライブラリ・ファイルから検索します。
- ライブラリ・ファイル内シンボルの検索は，以下の順序で行います。
  - 本オプションで指定したユーザ・ライブラリ・ファイル（指定順）
  - 本オプションで指定したシステム・ライブラリ・ファイル（指定順）
  - デフォルト・ライブラリ（環境変数 HLNK_LIBRARY1，HLNK_LIBRARY2，HLNK_LIBRARY3 [注]の順）

  注 環境変数についての詳細は，「2.3 環境変数」を参照してください。

[使用例]

- a.lib と b.lib を入力します。

```
rlink main.obj -library=a.lib,b
```

- “c” で始まり，拡張子が “.lib” であるファイルをすべて入力します。

```
rlink main.obj -library=c*.lib
```

-Binary

入力バイナリ・ファイルを指定します。

[指定形式]

```
-Binary=suboption[, ...]
  suboption := file( section[:alignment][/attribute][,symbol] )
```

- 省略時解釈
  なし

[詳細説明]

- 入力バイナリ・ファイル *file* を指定します。
  複数指定する場合は，カンマ（,）で区切ります。
- 入力ファイル名に拡張子の指定がない場合は “.bin” を指定したものとみなします。
- 入力したバイナリ・データは，指定したセクション *section* のデータとして配置します。
  セクションのアドレスは -start オプションで指定します。
  section を省略した場合は，エラーとなります。
- シンボル *symbol* を指定すると，定義シンボルとしてリンクすることもできます。
  C プログラムで参照している変数名の場合，プログラム中での参照名の先頭に “_” を付加します。
- 本オプションで指定したセクションには，セクション属性，アライメント数の指定が可能です。
- セクション属性 *attribute* に指定可能なものは，CODE，または DATA です。
  attribute を省略した場合は，デフォルトとして，書き込み，読み取り，実行すべての属性が有効になります。
- アライメント数 *alignment* に指定可能な値は 2 の累乗（1，2，4，8，16，32）です。
  それ以外の値を指定することはできません。
  alignment を省略した場合は，デフォルトとして，1 が有効となります。

[使用例]

- b.bin を D1bin セクションとして，0x200 番地から配置します。
  c.bin を D2bin セクション（アライメント数 4）として，D1bin の後に配置します。
  c.bin データを定義シンボル _datab としてリンクします。

```
>rlink a.obj -start=D*/200 -binary=b.bin(D1bin),c.bin(D2bin:4,_datab)
```

[備考]

- 本オプションは，-form={object|library} オプション，または -strip オプションを指定した場合は無効となります。
- 入力オブジェクト・ファイルを指定していない場合，本オプションは指定することができません。

-DEFine

未定義シンボルを強制定義します。

[指定形式]

```
-DEFine=suboption[, ...]
  suboption := symbol1=symbol2
             |  symbol1=value
```

- 省略時解釈
  なし

[詳細説明]

- 未定義シンボル *symbol1* を外部定義シンボル *symbol2*，または数値 *value* で強制定義します。
- *value* は 16 進数で指定します。
  先頭が A ～ F の場合は，先にシンボルを検索し，該当するシンボルがなければ数値と解釈します。
  先頭が 0 の場合は，常に数値と解釈します。
- シンボル名が C 変数名の場合は，プログラム中での定義名の先頭に “_” を付加します。

[使用例]

- _sym1 を外部定義シンボル data と同値として定義します。

```
>rlink -define=_sym1=data a.obj b.obj
```

- _sym2 を 0x4000 として定義します。

```
>rlink -define=_sym2=4000 a.obj b.obj
```

[備考]

- 本オプションは，-form={object|relocate|library} オプションを指定した場合は無効となります。

-ENTry

実行開始アドレスを指定します。

[指定形式]

```
-ENTry={symbol|address}
```

- 省略時解釈
  なし

[詳細説明]

- 実行開始アドレスを外部定義シンボル *symbol*，またはアドレス *address* で指定します。
- *address* は 16 進数で指定します。
  先頭が A ～ F の場合は，先に定義シンボルを検索し，該当するシンボルがなければアドレスと解釈します。
  先頭が 0 の場合は，常にアドレスと解釈します。
- シンボル名が C 変数名の場合は，プログラム中での定義名の先頭に “_” を付加します。

[使用例]

- C の main 関数を実行開始アドレスとして指定します。

```
>rlink -entry=_main a.obj b.obj
```

- 0x100 を実行開始アドレスとして指定します。

```
>rlink -entry=100 a.obj b.obj
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，または -strip オプションを指定した場合は無効となります。
- リンク時最適化（optimize[= symbol_delete]）指定時には，本オプションで *symbol* を指定する必要があります。指定がない場合は，リンク時最適化指定は無効となります 。また，本オプションで *address* を指定している場合は，リンク時最適化を無効にします。
- start オプションで配置アドレスを指定したセクションのリスト中に，entry で指定したアドレスが所属するとき，その start オプションで指定した配置アドレスから，entry で指定したアドレスまでの領域を対象にした最適化は抑止します。

-ALLOW_DUPLICATE_MODULE_NAME 【V2.02.00 以降】

複数の同じモジュール名からの，ライブラリ生成を許可します。

[指定形式]

```
-allow_duplicate_module_name
```

- 省略時解釈
  なし

[詳細説明]

- ライブラリ生成時に，複数の同じモジュール名の入力ファイル指定を許容します。
- ライブラリ内に既に名前が重複するモジュールがあれば，モジュール名の末尾に ".<N>" を加えてライブラリに登録します。
- <N> にはライブラリ中で重複しないモジュール名になるよう番号を設定します。重複しない番号を見つけられない場合はエラーを出力して終了します。

[使用例]

- ライブラリ（a.lib）を同じモジュール名（mod）を持つ複数の入力ファイルから生成します。

```
> rlink -allow_duplicate_module_name -form=lib -output=a.lib b¥mod.obj c¥mod.obj
d¥mod.obj
```

生成したライブラリ（a.lib）は次のように構成されます。
- mod（b¥mod.obj から）
- mod.1（c¥mod.obj から）
- mod.2（d¥mod.obj から）

[備考]

- 本オプションは，-form={object|absolute|relocate|hexadecimal|stype|binary} オプション，-strip オプション，または -extract オプションを指定した場合は無効となります。

出力制御

出力制御オプションには，次のものがあります。

- -FOrm
- -DEBug
- -NODEBug
- -RECord
- -END_RECORD 【V1.06.00 以降】
- -ROm
- -OUtput
- -MAp
- -SPace
- -Message
- -NOMessage
- -MSg_unused
- -BYte_count
- -FIX_RECORD_LENGTH_AND_ALIGN 【V1.07.00 以降】
- -PADDING
- -OVERRUN_FETCH
- -RESERVE_PREFETCH_AREA【V2.04.01 以降】
- -CRc
- -CFI 【Professional 版のみ】【V1.07.00 以降】
- -CFI_ADD_Func 【Professional 版のみ】【V1.07.00 以降】
- -CFI_IGNORE_Module 【Professional 版のみ】【V1.07.00 以降】

-FOrm

出力形式を指定します。

[指定形式]

```
-FOrm=format
```

- 省略時解釈
  ロード・モジュール・ファイルを出力します（-form=absolute オプションの指定と同じです）。

[詳細説明]

- 出力形式 *format* を指定します。
- *format* に指定可能なものを以下に示します。

Absolute	ロード・モジュール・ファイルを出力します。
Relocate	リロケータブル・ファイルを出力します。
Object	オブジェクト・ファイルを出力します。 -extract オプションでライブラリから 1 個のモジュールをオブジェクト・ファイルとして取り出すときに使用します。
Library[={S|U}]	ライブラリ・ファイルを出力します。 library=s を指定した場合は，出力ファイルをシステム・ライブラリ・ファイルとします。 library=u を指定した場合は，出力ファイルをユーザ・ライブラリ・ファイルとします。 library のみを指定した場合は，library=u を指定したものとみなします。
Hexadecimal	インテル拡張ヘキサ・ファイルを出力します。 詳細については，「3.5 インテル拡張ヘキサ・ファイル」を参照してください。
Stype	モトローラ・S タイプ・ファイルを出力します。 詳細については，「3.6 モトローラ・S タイプ・ファイル」を参照してください。
Binary	バイナリ・ファイルを出力します。

[備考]

- 出力形式と入力ファイル，他のオプションとの関係を以下に示します。

表 2.5 出力形式と入力ファイル，他のオプションとの関係

出力形式	指定オプション	入力可能なファイル形式	指定可能なオプション[注 1]
Absolute	-strip あり	ロード・モジュール・ファイル	-input，-output
	上記以外	オブジェクト・ファイル リロケータブル・ファイル バイナリ・ファイル ライブラリ・ファイル	-input，-library，-binary，-debug，-nodebug，-cpu，-start，-rom，-entry，-output，-map，-padding，-hide，-optimize/-nooptimize，-absolute_forbid，-symbol_forbid，-section_forbid，-compress，-nocompress，-rename，-lib_rename，-delete，-define，-fsymbol，-stack，-memory，-msg_unused，-show={all|symbol|reference|xreference|total_size|struct|relocation_attribute|cfi}，-aligned_section，-overrun_fetch，-cfi，-cfi_add_func，-cfi_ignore_module

出力形式	指定オプション	入力可能なファイル形式	指定可能なオプション注 1
Relocate	-extract あり	ライブラリ・ファイル	-library, -output
	上記以外	オブジェクト・ファイル リロケータブル・ファイル バイナリ・ファイル ライブラリ・ファイル	-input, -library, -binary, -debug, -nodebug, -output, -hide, -rename, -lib_rename, -delete, -show={all|symbol|xreference|total_size}
Object	-extract あり	ライブラリ・ファイル	-library, -output
Hexadecimal Stype Binary		オブジェクト・ファイル リロケータブル・ファイル バイナリ・ファイル ライブラリ・ファイル	-input, -library, -binary, -cpu, -start, -rom, -entry, -output, -map, -space, -optimize/-nooptimize, -absolute_forbid, -symbol_forbid, -section_forbid, -rename, -lib_rename, -delete, -define, -fsymbol, -stack, -record 注 2, -end_record 注 2, -s9 注 2, -byte_count 注 3, -fix_record_length_and_align 注 7, -padding, -memory, -msg_unused, -show={all|symbol|reference|xreference|total_size|struct|relocation_attribute|cfi}, -aligned_section, -overrun_fetch 注 4, -crc, -cfi, -cfi_add_func, -cfi_ignore_module
		ロード・モジュール・ファイル	-input, -output, -record 注 2, -end_record 注 2, -s9 注 2, -byte_count 注 3, -fix_record_length_and_align 注 7, -show={all|symbol|reference|xreference}, -crc
		インテル拡張ヘキサ・ファイル注 5	-input, -output
		モトローラ・S タイプ・ファイル注 5	-input, -output, -s9 注 2
Library	-strip あり	ライブラリ・ファイル	-library, -output, -memory 注 6, -show
	-extract あり	ライブラリ・ファイル	-library, -output
	上記以外	オブジェクト・ファイル リロケータブル・ファイル	-input, -library, -output, -hide, -rename, -delete, -replace, -memory 注 6, -show={all|symbol|section}, -allow_duplicate_module_name

注 1. 以下のオプションは，常に指定可能です。
-message，-nomessage，-change_message，-logo，-nologo，-form，-list，-subcommand

注 2. -form=stype オプションを指定した場合のみ指定可能です。

注 3. -byte_count オプションは，-form= hexadecimal，または -form=stype オプションを指定した場合のみ指定可能です。

注 4. -overrun_fetch オプションは，-form=hexadecimal，または -form=stype オプションを指定した場合のみ指定可能です。

注 5. 入力ファイルにインテル拡張ヘキサ・ファイルを指定した場合は -form=hexadecimal オプション，モトローラ・S タイプ・ファイルを指定した場合は -form=stype オプションのみを指定することができます。

注 6. -hide オプションを指定した場合は指定することはできません。

注 7. -fix_record_length_and_align オプションは，-form=hexadecimal，または -form=stype オプションを指定した場合のみ指定可能です。

［使用例］

- a.obj，b.obj からリロケータブル・ファイル c.rel を出力します。

```
>rlink a.obj b.obj -form=relocate -output=c.rel
```

- lib.lib からモジュール a を取り出し，オブジェクト・ファイルとして出力します。

```
>rlink -library=lib.lib -extract=a -form=object
```

- lib.lib からモジュール a を取り出し，ライブラリ・ファイル exta.lib を出力します。

```
>rlink -library=lib.lib -extract=a -form=library -output=exta
```

- lib.lib からモジュール a を取り出し，リロケータブル・ファイル a.rel を出力します。

```
>rlink -library=lib.lib -extract=a -form=relocate
```

-DEBug

出力ファイル中にデバッグ情報を出力します。

[指定形式]

```
-DEBug
```

- 省略時解釈
  出力ファイル中にデバッグ情報を出力します（-debug オプションの指定と同じです）。

[詳細説明]

- 出力ファイル中にデバッグ情報を出力します。

[使用例]

- 出力ファイル中にデバッグ情報を出力します。

```
>rlink a.obj b.obj -debug -output=c.abs
```

[備考]

- 本オプションは，-form={object|library|hexadecimal|stype|binary}，-strip オプション，または -extract オプションを指定した場合は無効となります。
- -form=absolute オプション，および -output オプションで出力ファイル名を複数した場合は，デバッグ情報を出力しません。

-NODEBug

デバッグ情報を出力しません。

[指定形式]

```
-NODEBug
```

- 省略時解釈
  出力ファイル中にデバッグ情報を出力します（-debug オプションの指定と同じです）。

[詳細説明]

- デバッグ情報を出力しません。

[使用例]

- デバッグ情報を出力しません。

```
>rlink a.obj b.obj -nodebug -output=c.abs
```

[備考]

- 本オプションは，-form={object|library|hexadecimal|stype|binary}，-strip オプション，または -extract オプションを指定した場合は無効となります。

-RECord

出力するデータ・レコードのサイズを指定します。

[指定形式]

```
-RECord=record
```

- 省略時解釈
  それぞれのアドレスにあわせて，混在したデータ・レコードを出力します。

[詳細説明]

- アドレス範囲に関係なく，一定のデータ・レコード *record* で出力します。
- *record* に指定可能なものを以下に示します。

H16	ヘキサ・レコード
H20	拡張ヘキサ・レコード
H32	32 ビット・ヘキサ・レコード
S1	S1 レコード
S2	S2 レコード
S3	S3 レコード

- 指定したデータ・レコードより大きいアドレスが存在した場合に，アドレスに合わせてデータ・レコードを選択します。

[使用例]

- アドレス範囲に関係なく，32 ビット・ヘキサ・レコードで出力します。

```
>rlink a.obj b.obj -record=H32 -form=hexadecimal -output=c.hex
```

[備考]

- 本オプションは，-form={hexadecimal|stype} オプションを指定していない場合は無効となります。
- -form=hexadecimal オプション指定時に -record={S1|S2|S3} オプションを指定した場合，および，-form=stype オプション指定時に -record={H16|H20|H32} オプションを指定した場合は，エラーとなります。

-END_RECORD 【V1.06.00 以降】

エンドレコードを指定します。

[指定形式]

```
-END_RECORD=record
```

- 省略時解釈
  エントリ・ポイント・アドレスに合わせてエンドレコードを出力します。

[詳細説明]

- モトローラ・S タイプ・ファイルのエンドレコードを指定します。
- *record* に指定可能なものを以下に示します。

S7	S7 レコード
S8	S8 レコード
S9	S9 レコード

- 指定したアドレスフィールドより大きいエントリ・ポイント・アドレスの場合，エントリ・ポイント・アドレスに合わせてエンドレコードを選択します。

[使用例]

- アドレス範囲に関係なく，32 ビット・S タイプ・エンド・レコードで出力します。

```
> rlink a.obj b.obj -end_record=S7 -form=stype -output=c.mot
```

[備考]

- -form={stype} 指定がない時，本オプションはエラーを出力して終了します。

-ROm

ROM から RAM へマップするセクションを指定します。

[指定形式]

```
-ROm=ROMsection=RAMsection[,ROMsection=RAMsection]...
```

- 省略時解釈
  なし

[詳細説明]

- 初期値ありデータ領域の ROM 用，RAM 用領域を確保し，ROM セクション内定義シンボルを RAM セクション内アドレスでリロケーションします。
- ROM セクション *ROMsection* には，初期値のあるリロケータブル・セクションを指定します。
- RAM セクション *RAMsection* には，存在しないセクション，またはサイズ 0 のリロケータブル・セクションを指定します。
- *ROMsection*，*RAMsection* には，ワイルドカード記号 '*' を使用することができます。【V2.06.00 以降】
  - 初期値のあるリロケータブル・セクション（ROM 側セクション）の名前が，*ROMsection* のワイルドカード表現と一致する場合，*RAMsection* 内のワイルドカード記号 '*' の部分を，ROM 側セクション名の中のワイルドカード記号 '*' に一致する部分と置き換えて，RAM 側セクションの名前として処理します。

    例） ROM 側セクションが .data，.data_1，.sdata，.sdata_1 の 4 つあるとき，-rom=*data*=*data*_R と指定すると，.data_R，.data_1_R，.sdata_R，.sdata_1_R の 4 つの RAM 側セクションが生成されます。

    注 置換後の RAM 側セクション名は，-start オプションなどで適切に対応する必要があります。
  - ワイルドカード記号 '*' は複数個指定できますが，*ROMsection*，*RAMsection* で個数が一致している必要があります。

    例）

```
-rom=.data*=.data*_R      # 問題なし
-rom=.data*=.data*_R_*    # RAMsection 側のワイルドカード記号が多すぎるのでエラー
```

  - 置換によってできたセクション名と同名のセクションがすでにあった場合は，エラーになります。

[使用例]

- .data セクションと同サイズの .data.R セクションを確保し，.data セクション内定義シンボルを .data.R セクション上のアドレスでリロケーションします。

```
>rlink a.obj b.obj -rom=.data=.data.R -start=.data/100,.data.R/8000
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，または -strip オプションを指定した場合は無効となります。

-OUtput

出力ファイルを指定します。

[指定形式]

```
-OUtput=suboption[, ...]
  suboption := file
             |  file=range
             |  file=/load-address
             |  file=range/load-address
  range := address1-address2
         |  section[: ...]
```

- 省略時解釈
  出力ファイル名を「*先頭入力ファイル名．デフォルト拡張子*」とします。
  デフォルト拡張子を以下に示します。
  -form=absolute オプション指定時 : abs
  -form=relocate オプション指定時 : rel
  -form=object オプション指定時 : obj
  -form=library オプション指定時 : lib
  -form=hexadecimal オプション指定時 : hex
  -form=stype オプション指定時 : mot
  -form=binary オプション指定時 : bin

[詳細説明]

- 出力ファイル *file* を指定します。
- *address1*，*address2* には，出力範囲の先頭アドレス，終了アドレスを 16 進数で指定します。
  出力範囲に "-" を指定した場合は常にアドレスと解釈します。
- *section* には，出力するセクションを指定します。
  複数指定する場合は，コロン（:）で区切ります。
- *load-address* を指定すると，インテル拡張ヘキサ・ファイルまたはモトローラ・S タイプ・ファイルを出力する際，出力ファイル上の先頭ロード・アドレスを *load-address* で指定した値に変更します。【V2.00.00 以降】
- -form={absolute|hexadecimal|stype|binary} オプションと同時に指定した場合は，複数のファイルを指定することができます。

[使用例]

- 0 ～ 0xffff 間を file1.abs に，0x10000 ～ 0x1ffff 間を file2.abs に出力します。

```
>rlink a.obj b.obj -output=file1.abs=0-ffff,file2.abs=10000-1ffff
```

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

```
>rlink a.obj b.obj -output=file1.abs=sec1:sec2,file2.abs=sec3
```

[備考]

- *load-address* は，form={ hexadecimal | stype } 指定時に指定できます。
- 入力ファイルがインテル拡張・ヘキサ・ファイル，またはモトローラ・S タイプ・ファイルの場合，本オプションで複数の出力ファイルを指定することはできません。
  本オプションを省略した場合，出力ファイル名は "*先頭入力ファイル名* _combine. *拡張子*" となります（入力ファイルが a.mot の場合，出力ファイルは a_combine.mot となります）。

-MAp

外部変数割り付け情報ファイルを出力します。

[指定形式]

```
-MAp[=file]
```

- 省略時解釈
  なし

[詳細説明]

- コンパイラが外部変数アクセス最適化で使用する外部変数割り付け情報ファイル *file* を出力します。
- ファイル名の指定を省略した場合は，-output オプションで指定したファイル名，または “ *先頭入力ファイル名* .bls” というファイル名となります。
- 外部変数割り付け情報ファイル作成時の変数宣言順と，再コンパイル後のオブジェクトを読み込んだ時の変数宣言順が異なる場合は，エラーとなります。
- 以下に該当する場合，リンカは外部変数割り付け情報ファイルと，-LIst オプション指定がある場合はリスト・ファイルを出力して正常終了します。このときロード・モジュール・ファイルは出力しません。【V1.05.00 以降】
  - セクションの配置アドレスが，使用可能なアドレス範囲を超える場合
    外部変数割り付け情報ファイルには，配置可能な領域内に配置できたシンボル情報とセクション情報のみ出力します。

[使用例]

- 外部変数割り付け情報ファイル file.bls を出力します。

```
>rlink a.obj b.obj -output=c.abs -map=file.bls
```

[備考]

- 本オプションは，-form={absolute|hexadecimal|stype|binary} オプションを指定した場合のみ有効となります。

-SPace

出力範囲のメモリの空き領域を充てんします。

[指定形式]

```
-SPace[=data]
```

- 省略時解釈
  なし

[詳細説明]

- 出力範囲のメモリの空き領域を，ユーザが指定するデータ *data* で充てんします。
- *data* に指定可能なものを以下に示します。

数値	16 進数の数値
Random	乱数

- 空き領域の充てん方法は，指定する出力範囲指定方法によって，以下のように異なります。
  - -Output オプションの出力範囲がセクション指定の場合
    指定したセクション間に空き領域が存在した場合に，指定データを出力します。
  - -Output オプションの出力範囲がアドレス範囲指定の場合
    指定した範囲内に空き領域が存在した場合に，指定データを出力します。
  - -FIX_RECORD_LENGTH_AND_ALIGN オプションを指定した場合
    - セクション先頭アドレスをアライメント数で割り切れるアドレスに整合したとき，セクション先頭に空き領域が存在する場合に指定データを出力します。
    - オプションで指定したデータ・レコード長にセクション終端が満たない場合に指定データを出力します。
- 出力データ・サイズは，1，2，4 バイト単位で有効となり，本オプションで指定する 16 進数の数値で決まります。
  3 バイト・データを指定した場合は，上位桁を 0 拡張し，4 バイトのデータとして扱います。
  奇数桁データを指定した場合は，上位桁に 0 拡張して，偶数桁入力として扱います。
- 空き領域のサイズが出力データ・サイズの倍数でない場合は，出力できるだけ出力し，警告を出力します。

[使用例]

- 100H 番地から 2FFH 番地の範囲内で，メモリの空き領域に対して ffH で充てんします。

```
>rlink a.obj b.obj -form=hexadecimal -output=file1=100-2ff -start=P1/100,P2/200
-space=ff
```

[備考]

- 本オプションにてデータの指定を省略した場合は，空き領域への充てんを行いません。
- 本オプションは，-form={binary|stype|hexadecimal} オプションを指定した場合のみ有効となります。
- 本オプションは，-output オプションにて出力範囲を指定しなかった場合，または -fix_record_length_and_align オプションを指定しなかった場合は無効となります。

-Message

インフォメーション・メッセージを出力します。

[指定形式]

```
-Message
```

- 省略時解釈
  インフォメーション・メッセージの出力を抑止します（-nomessage オプションの指定と同じです）。

[詳細説明]

- インフォメーション・メッセージを出力します。

[使用例]

- インフォメーション・メッセージを出力します。

```
>rlink a.obj b.obj -message
```

-NOMessage

インフォメーション・メッセージの出力を抑止します。

[指定形式]

```
-NOMessage[={num|num-num}[, ...]]
```

- 省略時解釈
  インフォメーション・メッセージの出力を抑止します。

[詳細説明]

- インフォメーション・メッセージの出力を抑止します。
- メッセージ番号 *num* を指定すると，指定した番号のメッセージの出力を抑止します。
  また，ハイフン（-）を使用して，メッセージ番号の範囲を指定することもできます。
- *num* には，コンポーネント番号（05），発生フェーズ（6）に続けて出力される 4 桁の数値（M0560004 の場合は 0004）を指定します。
  4 桁の数値の先頭が 0 で始まる場合は，0 を省略することができます（M0560004 の場合は 4）。
- ワーニング，またはエラー種別のメッセージ番号を指定した場合は，-change_message オプションでインフォメーション種別に変更したと仮定し，メッセージの出力を抑止します。

[使用例]

- M0560004，M0560100 ～ M0560103，および M0560500 のメッセージの出力を抑止します。

```
>rlink a.obj b.obj -nomessage=4,100-103,500
```

-MSg_unused

参照されない外部定義シンボルをユーザに通知します。

[指定形式]

```
-MSg_unused
```

- 省略時解釈
  なし

[詳細説明]

- リンク処理の中で一度も参照されることのなかった外部定義シンボルを，メッセージ出力によってユーザに通知します。

[使用例]

- 参照されない外部定義シンボルをユーザに通知します。

```
>rlink a.obj b.obj -message -msg_unused
```

[備考]

- 本オプションは，入力ファイルがロード・モジュール・ファイルの場合は無効となります。
- 本オプションは，-message オプションと同時に指定する必要があります。
- コンパイル時にインライン展開された関数に対して，メッセージ出力する場合があります。
  その場合は，関数定義に static 宣言を追加することにより，メッセージ出力を抑止することができます。
- 同一ファイル内の定数シンボルへの参照がある場合は，参照関係の解析が正しく行うことができず，メッセージ出力により通知される情報が不正確となります。

-BYte_count

データ・レコードのバイト数の最大値を指定します。

[指定形式]

```
-BYte_count=num
```

- 省略時解釈
  -form=hexadecimal 指定時は，バイト数の最大値を 0xFF として，インテル拡張ヘキサ・ファイルを生成します。
  -form=stype 指定時は，バイト数の最大値を 0x10 として，モトローラ・S タイプ・ファイルを生成します。

[詳細説明]

- インテル拡張ヘキサ・ファイルまたはモトローラ・S タイプ・ファイルを生成する際に，データ・レコード長を指定するためのオプションです。
- インテル拡張ヘキサ・ファイルを生成する場合，数値には，01 ～ FF（16 進数）を指定することができます。
- モトローラ・S タイプ・ファイルを生成する場合，数値には，次のいずれかの値を指定することができます。
  - S1 形式 : 01 ～ FC（16 進数）
  - S2 形式 : 01 ～ FB（16 進数）
  - S3 形式 : 01 ～ FA（16 進数）

[使用例]

- データ・レコードのバイト数の最大値として 0x10 を指定します。

```
>rlink a.obj b.obj -form=hexadecimal -byte_count=10
```

[備考]

- 本オプションは，-form=hexadecimal オプションを指定していない場合は無効となります。【V1.06.00 以前】
- 本オプションは，-form={hexadecimal|stype} オプションを指定していない場合は無効となります。【V1.07.00 以降】

-FIX_RECORD_LENGTH_AND_ALIGN 【V1.07.00 以降】

出力範囲の開始アドレスを整列します。

[指定形式]

```
-FIX_RECORD_LENGTH_AND_ALIGN=align
```

- 省略時解釈
  なし

[詳細説明]

- インテル拡張ヘキサ・ファイル，またはモトローラ・S タイプ・ファイルを生成する際に，指定したアライメント数で整合したアドレスから常に一定数のレコード長で出力します。
- 出力開始アドレスは，セクション先頭アドレス以前かつ指定したアライメント数で割り切れる最も大きなアドレスとなります。
- レコード長は，-BYte_count オプションで指定した個数または規定値を常に出力します。
- レコード長を一定にすることにより，一つのレコードに複数のセクションが出力される場合があります。
- 空き領域には，-SPace オプションが指定された場合はその値，-SPace オプションを指定せず，-CRC オプションを指定している場合は 0xFF，-SPace オプションも -CRC オプションも指定していない場合は 0 を出力します。

[使用例]

- レコードの出力を 8 で割り切れるアドレスから開始し，レコード長を 16 バイト（16 進数で 10）に固定します。

```
>rlink a.obj b.obj -form=hexadecimal -byte_count=10 -fix_record_length_and_align=8
```

[備考]

- 本オプションは，-form={hexadecimal|stype} オプションを指定していない場合は無効となります。

-PADDING

セクションの終端にデータを埋め込みます。

[指定形式]

```
-PADDING
```

- 省略時解釈
  なし

[詳細説明]

- セクション・サイズが，セクションのアライメントの倍数となるように，セクションの終端にデータを埋め込みます。

[使用例]

- 以下の場合，.const セクションに 2 バイトのパディング・データを埋め込み，サイズを 0x08 バイトにしてリンク処理を行います。
  .const セクションのアライメント:4 バイト
  .const セクションのサイズ : 0x06 バイト
  .text セクションのアライメント: 2 バイト
  .text セクションのサイズ : 0x02 バイト

```
>rlink a.obj b.obj -start=.const,.text/0 -padding
```

- 以下の場合，.const セクションに 2 バイトのパディング・データを埋め込み，サイズを 0x08 バイトにしてリンク処理を行うと，.text セクションと重複してしまうため，エラーを出力します。
  .const セクションのアライメント:4 バイト
  .const セクションのサイズ : 0x06 バイト
  .text セクションのアライメント: 2 バイト
  .text セクションのサイズ : 0x02 バイト

```
>rlink a.obj b.obj -start=.const/0,.text/6 -padding
```

[備考]

- 生成するパディング・データの値は 0x00 です。
- 絶対アドレス・セクションにはパディングを行わないため，絶対アドレス・セクションのサイズはユーザにて調整してください。
- V1.00.01 ではテキストデータ，const 変数，および初期値がある変数のセクションのみを対象としてパディング・データを埋め込んでいましたが，V1.01.00 以降では初期値がない変数のセクションも対象とします。

-OVERRUN_FETCH

オーバーラン・フェッチに伴う未初期化領域の読み出しを回避します。

[指定形式]

```
-OVERRUN_FETCH
```

- 省略時解釈
  なし

[詳細説明]

- 各 ROM セクションの直後に 128 バイト以上の空き領域が存在する場合，その ROM セクションを読み出す際に，未初期化のコード・フラッシュ領域をプリフェッチする可能性があるため，空き領域に対して 128 バイト分の NOP 命令を挿入します。

  注　128 バイトは，マイコンごとのプリフェッチのサイズに依存しない目安の値です。コード・フラッシュ領域の書き込み単位が 256 バイトであることから，後続するセクションとの間が 128 バイト未満である場合は未初期化領域が発生しないため，NOP 命令の挿入を行いません。

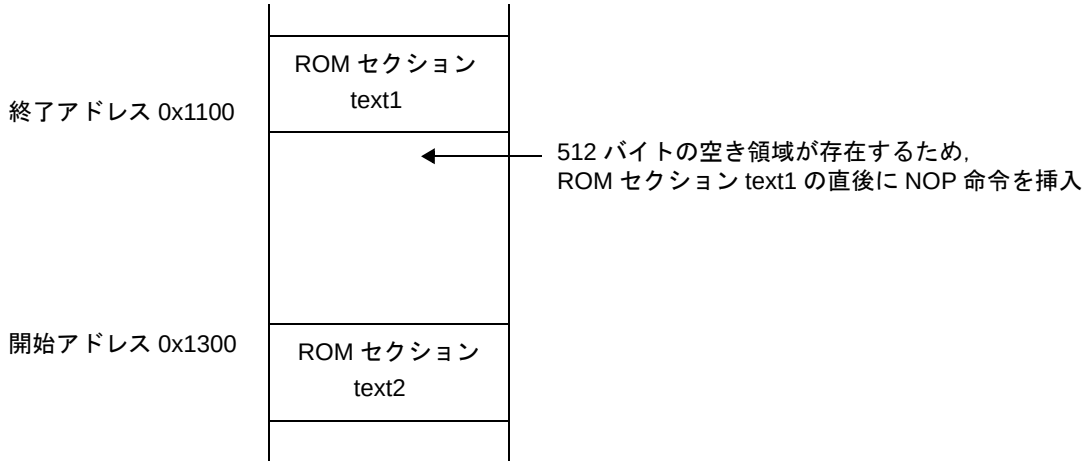
NOP 命令の挿入は -start オプションの配置に従って挿入し，挿入する直前のセクション名に対して以下のセクション名で出力します。

```
$sss_fetch??
```

sss ： 挿入する直前のセクション名
?? ： 01 ～ 99

(1) セクション間への挿入

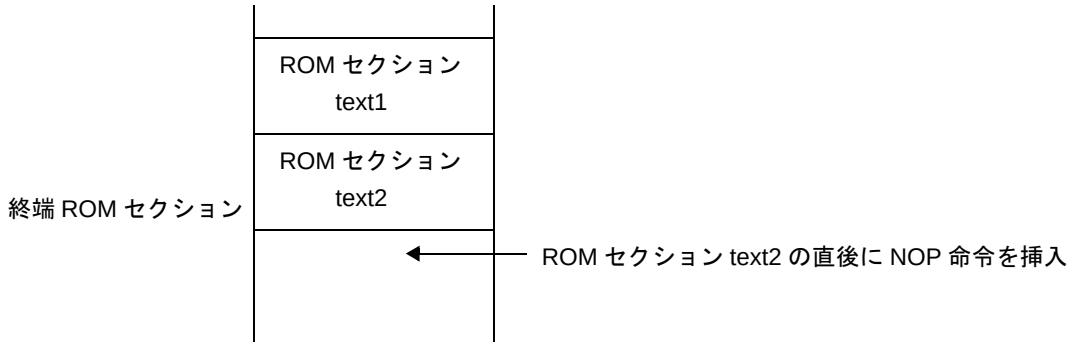
- ROM セクション（実行可能なプログラム・セクション，またはプログラム・セクション以外の ROM 領域配置セクション）と ROM セクション，および ROM セクションと RAM セクション（RAM 領域配置セクション）の間に 128 バイト以上の空き領域が存在する場合に NOP 命令を挿入します。



- ROM セクションと RAM セクションは，ROM セクションより大きいアドレスに RAM セクションが配置された場合に限り，NOP 命令を挿入します（ROM セクションより小さいアドレスに RAM セクションが配置された場合は NOP 命令を挿入しません）。

(2) 終端セクション直後への挿入

- 終端 ROM セクションの直後に NOP 命令を挿入します。



- -cpu オプションでアドレス範囲を指定する際，終端セクションの終端アドレスとして指定した終了アドレスの間に 128 バイトの空き領域がない場合，NOP 命令は挿入されません。
- 終端セクションの終端アドレスと 8M バイト境界のアドレスとの間に 128 バイトの空き領域がない場合，NOP 命令は挿入されません。

［使用例］

- オーバーラン・フェッチに伴う空き領域の読み出しを回避します。

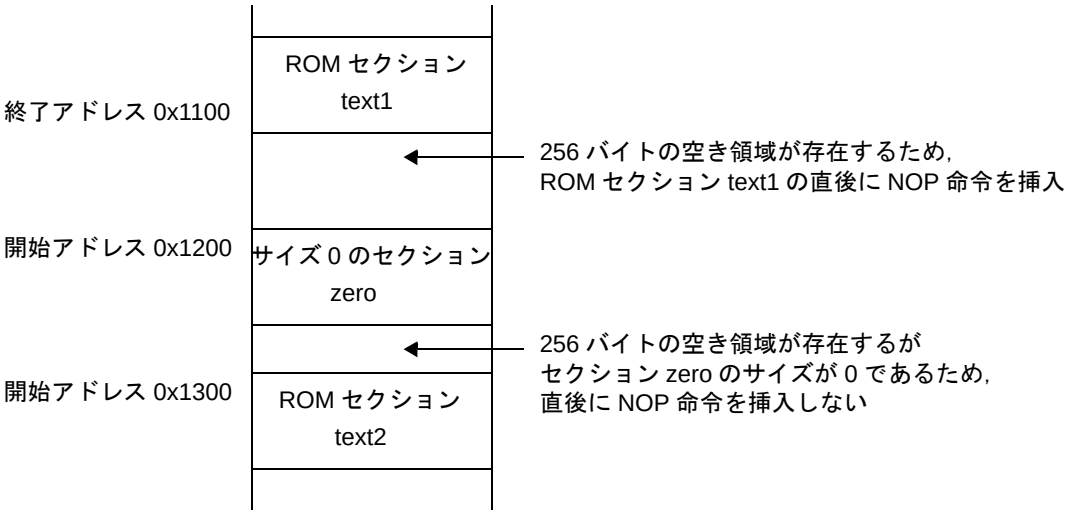
```
>rlink a.obj b.obj -overrun_fetch
```

［備考］

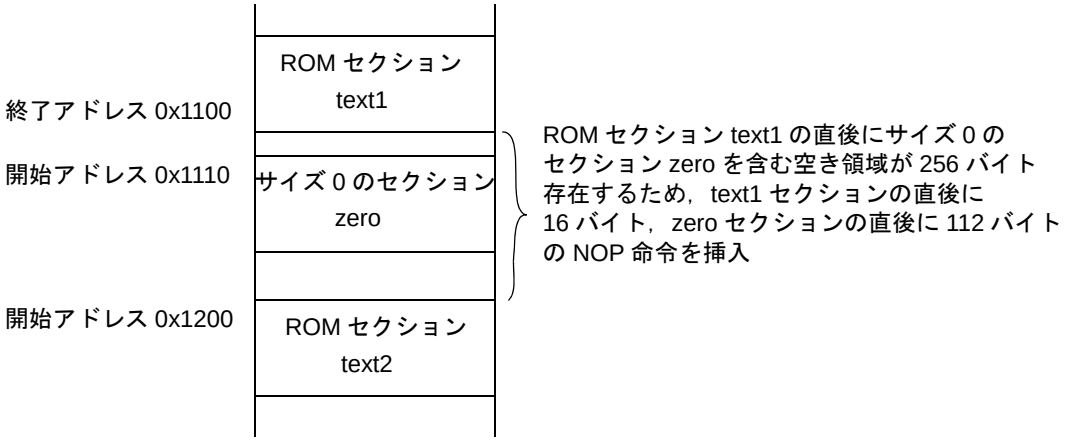
- -cpu オプションでアドレス範囲を指定していない場合，かつ終端セクションが ROM セクションの場合，終端 ROM セクションの直後に常に NOP 命令を挿入します。
- 本オプションは，入力ファイルがロード・モジュール・ファイルの場合は無効となります。
- 本オプションは，-form={object|relocate|library} オプションを指定した場合は無効となります。
- -start オプションでオーバーレイ配置指定を行った場合，オーバーレイ配置指定を含む start 列には NOP 命令の挿入を行いません。

例 1. -start=A,B,E/400,C,D:F:G/8000 オプションを指定した場合，オーバーレイ配置指定を含むセクション C，D，F，および G に対するセクション間，および終端コード・セクションに対して，NOP 命令の挿入を行いません。

例 2. サイズ 0 のセクションの直後に 128 バイト以上の空き領域が存在する場合，NOP 命令の挿入を行いません。



例 3. ROM セクションの直後に配置されたサイズ 0 のセクションを含む 128 バイト以上の空き領域が存在する場合，NOP 命令を挿入します。



-RESERVE_PREFETCH_AREA【V2.04.01 以降】

プリフェッチの可能性がある領域にセクションを生成して確保します。

[指定形式]

```
-RESERVE_PREFETCH_AREA[= セクション ]
```

- 省略時解釈
  セクションを生成しません。

[詳細説明]

- プリフェッチの可能性がある領域にセクションを生成して確保します。
- プロセッサのプリフェッチ動作によりアクセスされる可能性があるプログラム・セクション終端以後の 16byte を NOP 命令のセクションとして確保します。
- 挿入する直前のセクション名に対して下記セクション名で出力します。

  出力セクション名

  ```
  $sss_fetch??
  ```

  sss : 挿入する直前のセクション名
  ?? : 01 ～ 99
- セクションの領域が確保できない場合はエラーになります。

[備考]

- 入力ファイルが absolute 形式の場合，本オプション指定はエラー出力して終了します。
- form={object | relocate | library | binary} 指定時，本オプションはエラー出力して終了します。
- start オプションでオーバーレイ配置指定を行った場合，オーバーレイ配置指定を含む start 列には NOP 命令を挿入しません。

例 1. -start=A,B,E/400,C,D:F:G/8000
オーバーレイ配置指定を含むセクション C，D，F および G に対するセクション間，および終端コード・セクションに対して NOP 命令を挿入しません。

- プログラム・セクションの直後に配置されたサイズ 0（ゼロ）のセクションの場合は，その直後に NOP 命令を挿入します。

-CRc

CRC コードを出力します。

[指定形式]

```
-CRc = <出力位置>=<計算範囲>[/<演算方法>][(<初期値>)][:<エンディアン>]
  <計算範囲> := { <先頭アドレス>-<終了アドレス> | <セクション> }[,...]
  <エンディアン> := { BIG | LITTLE }[-<サイズ>-<オフセット>]
```

[詳細説明]

- 指定した範囲のセクションのデータを，下位アドレスから上位アドレスの順で CRC（Cyclic Redundancy Check）演算を行い，演算結果を出力アドレスへエンディアンの指定方法で出力します。
- 演算方法には以下のいずれかを指定することができます。演算方法の指定を省略した場合は，32-ETHERNET を指定したものとして演算を行います。

演算方法	内容
CCITT	CRC-16-CCITT で MSB First，初期値 0xFFFF，XOR 反転による演算結果を得ることができます。 生成多項式は $x^{16}+x^{12}+x^5+1$ です。
16-CCITT-MSB	CRC-16-CCITT で MSB First による演算結果を得ることができます。 生成多項式は $x^{16}+x^{12}+x^5+1$ です。
16-CCITT-MSB-LITTLE-4	入力を LITTLE エンディアン 4 バイト単位とし CRC-16-CCITT で MSB First による演算結果を得ることができます。 生成多項式は $x^{16}+x^{12}+x^5+1$ です。
16-CCITT-MSB-LITTLE-2	入力を LITTLE エンディアン 2 バイト単位とし CRC-16-CCITT で MSB First による演算結果を得ることができます。 生成多項式は $x^{16}+x^{12}+x^5+1$ です。
16-CCITT-LSB	CRC-16-CCITT で LSB First による演算結果を得ることができます。 生成多項式は $x^{16}+x^{12}+x^5+1$ です。
16	CRC-16 で LSB First による演算結果を得ることができます。 生成多項式は $x^{16}+x^{15}+x^2+1$ です。
SENT-MSB	入力を LITTLE エンディアン 1 バイト中下位 4bit 単位とし SENT 準拠で初期値 0x5，MSB First による演算結果を得ることができます。 生成多項式は $x^4+x^3+x^2+1$ です。
32-ETHERNET	CRC-32-ETHERNET による演算結果を得ることができます。演算結果は初期値 0xFFFFFFFF，XOR 反転，ビットリバースされています。 生成多項式は $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ です。

- < 初期値 > として指定可能な値の範囲は，演算方法が 32-ETHERNET では 0x0 ～ 0xFFFFFFFF，SENT-MSB では 0x0 ～ 0xF，それ以外は 0x0 ～ 0xFFFF です。
- < 初期値 > を省略した場合は，演算方法が SENT-MSB では 0x5，CCITT では 0xFFFF，32-ETHERNET では 0xFFFFFFFF，それ以外は 0x0 を指定したものとして演算を行います。
- 演算結果の出力アドレスへの出力は，サイズで確保した領域の先頭からオフセットの位置に，BIG か LITTLE で指定したバイトオーダーで書き込みます。確保した領域の先頭からオフセットの位置直前までは 0 を出力します。
- サイズとオフセットが省略された場合，サイズは 2 バイトで，オフセットは 0 とします。
- エンディアンが省略された場合は，リトルエンディアンで書き込みます。

- 計算範囲にある空き領域は space オプションが指定されていない場合は，space=FF が指定されていると仮定して，CRC 演算を行います。ただし，CRC 演算は，空き領域では 0xFF で計算を行いますが，0xFF を埋めることはありません。
- 演算範囲として指定した下位アドレスから上位アドレスの順に演算を行います。
- 本オプションを複数回指定した場合，すべての指定が有効です。【V2.05.00 以降】

［使用例］

- 0x1000 ～ 0x2FFD の領域に対して CRC 演算を行い，その結果を 0x2FFE 番地に出力します。

```
>rlink *.obj -form=stype -start=.SEC1,.SEC2/1000,.SEC3/2000
 -crc=2FFE=1000-2FFD -output=out.mot=1000-2FFF
```

- 0x1000 ～ 0x1FFF の領域に対する CRC 演算結果を 0x2FFC 番地に，また，0x2000 ～ 0x2FFB の領域に対する CRC 演算結果を 0x2FFE 番地に出力します。

```
>rlink *.obj -form=stype -start=.SEC1,.SEC2/1000,.SEC3/2000 -output=out.mot=1000-2FFF
-crc=2FFC=1000-1FFF -crc=2FFE=2000-2FFB
```

［備考］

- 複数のロード・モジュール・ファイル入力時は，本オプションはワーニングを出力して無視します。
- 本オプションは，次の場合に有効です。これ以外の場合はエラーを出力して終了します。
  - -form={hexadecimal | stype } 指定時 【V1.07.00 以前】
  - -form={hexadecimal | stype | bin } 指定時 【V2.00.00 以降】
- space オプションが指定されていない場合で，計算範囲に出力されない空き領域があるとき，空き領域には 0xFF が設定されているものとして CRC の計算が行われます。
- CRC 演算の計算範囲にオーバーレイ指定されている領域が含まれる場合はエラーを出力して終了します。
- エンディアンの指定で，サイズとオフセットには，以下が指定できます。これ以外の場合はエラーを出力して終了します。
  - LITTLE
  - LITTLE-2-0
  - LITTLE-4-0
  - BIG
  - BIG-2-0
  - BIG-4-0

-CFI 【Professional 版のみ】 【V1.07.00 以降】

不正な間接関数呼び出し検出で用いる関数リストを生成します。

[指定形式]

```
-CFI
```

- 省略時解釈
  不正な間接関数呼び出し検出で用いる関数リストを生成しません。

[詳細説明]

- 不正な間接関数呼び出し検出で用いる関数リストを生成します。
  不正な間接関数呼び出し検出の詳細については，コンパイル・オプション「-control_flow_integrity 【Professional 版のみ】【V1.07.00 以降】」を参照してください。
  リンカは，関数リストを .const セクションに生成します。そのため，リンク時に -start オプションで .const セクションを指定する必要があります。
- コンパイル時に -control_flow_integrity を指定して作られたオブジェクト・ファイルの場合，リンカは，コンパイラが自動抽出した情報を元に関数リストを生成します。
- コンパイル時に -control_flow_integrity を指定せずに作られたオブジェクト・ファイルの場合，リンカはオブジェクト・ファイル内のリロケーション解決したシンボル全ての関数リストを生成します。
- 特定の関数を関数リストに追加する場合は，リンク・オプション -CFI_ADD_Func を指定してください。
  特定のオブジェクト・ファイル内の関数を関数リストから除外する場合は，リンク・オプション -CFI_IGNORE_Module を指定してください。

-CFI_ADD_Func 【Professional 版のみ】 【V1.07.00 以降】

不正な間接関数呼び出し検出で用いる関数リストに追加する関数シンボルまたはアドレスを指定します。

[指定形式]

```
-CFI_ADD_Func={symbol|address}[, ...]
```

- 省略時解釈
  なし

[詳細説明]

- 不正な間接関数呼び出し検出で用いる関数リストに，関数のシンボルまたはアドレスを登録します。
  不正な間接関数呼び出し検出の詳細については，コンパイル・オプション「-control_flow_integrity 【Professional 版のみ】【V1.07.00 以降】」を参照してください。
- アドレスは 16 進数で指定します。
- 指定した関数シンボルがロード・モジュール内に存在しない場合，エラーとなります。
- 本オプションを複数回指定した場合は，指定した全ての関数シンボルまたはアドレスを関数リストに登録します。
- 本オプションを使用する場合，-CFI オプションの指定が必要です。-CFI オプションの指定が無い場合はエラーとなります。

[使用例]

- C ソースの main 関数，関数アドレス 0x100，C ソースの sub 関数を関数リストに登録します。

```
>rlink -cfi -cfi_add_func=_main,100 -cfi_add_func=_sub a.obj b.obj
```

-CFI_IGNORE_Module 【Professional 版のみ】 【V1.07.00 以降】

不正な間接関数呼び出し検出で用いる関数リストから除外するオブジェクト・ファイルを指定します。

[指定形式]

```
-CFI_IGNORE_Module=suboption [, ...]
  suboption := file
             |  file ( module [, ... ] )
```

- 省略時解釈
  なし

[詳細説明]

- 不正な間接関数呼び出し検出で用いる関数リストから除外するオブジェクト・ファイルや，ライブラリ・ファイル（【V2.00.00 以降】）を指定します。
  不正な間接関数呼び出し検出の詳細については，コンパイル・オプション「-control_flow_integrity 【Professional 版のみ】【V1.07.00 以降】」を参照してください。
- ライブラリ・ファイルを指定した場合，ライブラリ内モジュール名を指定できます。
- 指定したファイルが存在しない場合，エラーとなります。
- 本オプションを複数回指定した場合は，指定した全てのファイル内の関数を関数リストに登録しません。
- 本オプションを使用する場合，-CFI オプションの指定が必要です。-CFI オプションの指定が無い場合はエラーとなります。

[使用例]

- a.obj，b.obj，c.obj 内の関数を関数リストから除外します。

```
>rlink -cfi -cfi_ignore_module=a.obj,b.obj -cfi_ignore_module=c.obj
```

- b.lib ライブラリ内の c モジュール内の関数を関数リストから除外します。【V2.00.00 以降】

```
>rlink -cfi -cfi_ignore_module=b.lib(c) -lib=b.lib a.obj
```

リスト出力

リスト出力オプションには，次のものがあります。

- -LISt
- -SHow

-LISt

リスト・ファイルを出力します。

[指定形式]

```
-LISt[=file]
```

- 省略時解釈
  なし

[詳細説明]

- リスト・ファイル *file* を出力します。
- ファイル名の指定を省略した場合は，以下のリスト・ファイルを出力します。

指定オプション	ファイルの種類	ファイル名
-form=library オプション，または -extract オプション	ライブラリ・リスト・ファイル	*出力ファイル名*[注] .lbp
上記以外	リンク・マップ・ファイル	*出力ファイル名*[注] .map

注 出力ファイルが複数存在する場合は，先頭出力ファイル名となります。

- -MAp オプションと同時指定した場合，セクションの配置アドレスが，使用可能なアドレス範囲を超える場合でも，リンク・マップ情報とシンボル情報を出力します。ただし，使用可能なアドレス範囲を超える場合は “**OVER**” と表示します。【V1.05.00 以降】

[使用例]

- リンク・マップ・ファイルを file.map に出力します。

```
>rlink a.obj b.obj -list=file.map
```

-SHow

リスト・ファイルへの出力情報を指定します。

[指定形式]

```
-SHow[=info[,info]...]
```

- 省略時解釈
  なし

[詳細説明]

- リスト・ファイルへの出力情報 *info* を指定します。
- *info* に指定可能なものを以下に示します。

出力情報（*info*）	-form=library オプション指定時	-form=library オプション以外指定時
SYmbol	モジュール内シンボル名を出力	シンボル・アドレス，サイズ，種別，最適化内容を出力
Reference	指定不可	シンボル・アドレス，サイズ，種別，最適化内容，シンボルの参照回数を出力
SEction	モジュール内セクション名を出力	指定不可
Xreference	指定不可	クロス・リファレンス情報を出力
Total_size	指定不可	ROM 配置対象，RAM 配置対象ごとに，セクションの合計サイズを出力
STRUCT	指定不可	シンボル情報内にコンパイル時に -g を指定したファイル内で定義した構造体／共用体メンバ情報を出力 （-form=rel もしくは obj を指定した場合は無効）
RELOCATION_ATTRIBUTE 【V1.06.00 以降】	指定不可	再配置属性を出力
CFI 【V1.07.00 以降】	指定不可	-form=absolute オプション指定時 　不正な間接関数呼び出し検出で用いる関数リストを出力 -form=hex/bin/stype オプション指定時かつ入力ファイルが absolute/hex/stype 以外 　不正な間接関数呼び出し検出で用いる関数リストを出力
ALL	モジュール内シンボル名，セクション名を出力	-form=relocate オプション指定時 　-show=symbol,xreference,total_size オプション指定時と同じ情報を出力 -form=absolute オプション指定時 　-show=symbol,reference,xreference,total_size,struct 指定時と同じ情報を出力 -form=hexadecimal/stype/binary オプション指定時 　-show=symbol,reference,xreference,total_size,struct 指定時と同じ情報を出力 -form=object オプション指定時 　指定不可

備考 出力情報の詳細については，「3.2 リンク・マップ・ファイル」，「3.4 ライブラリ・リスト・ファイル」を参照してください。

- 出力情報の指定を省略した場合については，［備考］を参照してください。
- -extract オプションを指定した場合，出力情報（*info*）を指定することはできません。

［使用例］

- リンク・マップ・ファイルにシンボル・アドレス，サイズ，種別，最適化内容，シンボルの参照回数を出力します。

```
>rlink a.obj b.obj -list -show=symbol,reference
```

［備考］

- -form オプションと -show，または -show=all オプションの指定により，出力情報 *info* が有効／無効になる組み合わせを以下に示します。

		Symbol	Reference	Section	Xreference	Total_size	STRUCT	RELOCATION_ATTRIBUTE	CFI
-form=absolute	-show のみ	有効	有効	無効	無効	無効	無効	無効	無効
	-show=all	有効	有効	無効	有効	有効	有効注 4	無効	無効
-form=library	-show のみ	有効	無効	有効	無効	無効	無効	無効	無効
	-show=all	有効	無効	有効	無効	無効	無効	無効	無効
-form=relocate	-show のみ	有効	無効	無効	無効	無効	無効	無効	無効
	-show=all	有効	無効	無効	有効注 1	有効	無効	無効	無効
-form=object	-show のみ	有効	有効	無効	無効	無効	無効	無効	無効
	-show=all	無効	無効	無効	無効	無効	無効	無効	無効
-form=hexadecimal注 2/stype注 3/binary	-show のみ	有効	有効	無効	無効	無効	無効	無効	無効
	-show=all	有効	有効	無効	有効	有効注 1	有効注 4	無効	無効

注 1. 入力ファイルがロード・モジュール・ファイルの場合は無効となります。

注 2. 入力ファイルがインテル拡張ヘキサ・ファイルの場合は，-show オプションを指定することはできません。

注 3. 入力ファイルがモトローラ・S タイプ・ファイルの場合は，-show オプションを指定することはできません。

注 4. -rename，-lib_rename，-hide，-compress オプション，リンク時最適化（-optimize=symbol_delete）指定時には，無効です。

なお，クロス・リファレンス情報の出力については以下の制限があります。

- 入力ファイルがロード・モジュール・ファイルの場合，参照側アドレスの情報は出力されません。
- 同一ファイル内の定数シンボルへの参照についての情報は出力されません。
- コンパイル時に最適化が有効で，直下の関数を呼び出す場合についての情報は出力されません。
- -show=total_size オプションで表示する情報は，-total_size オプションにより表示する情報と同じです。

最適化

最適化オプションには，次のものがあります。

- -OPtimize / -NOOPtimize 【V2.01.00 以降】
- -SEction_forbid 【V2.01.00 以降】
- -Absolute_forbid 【V2.01.00 以降】
- -SYmbol_forbid 【V2.01.00 以降】
- -ALLOW_OPTIMIZE_ENTRY_BLOCK【V2.06.00 以降】

-OPtimize / -NOOPtimize 【V2.01.00 以降】

リンク時最適化の実行有無を指定します。

[指定形式]

```
-OPtimize[=SYmbol_delete]
-NOOPtimize
```

- 省略時解釈
  リンク時最適化を実行します。-optimize オプション指定と同じです。

[詳細説明]

- コンパイル，アセンブル時に -goptimize オプションを指定したファイルに対して，リンク時最適化（モジュール間最適化）を実行します。
- -optimize オプションは，最適化を実行します。
  -nooptimize オプションは，最適化を抑止します。
- CC-RH のリンク時最適化では，プログラム内で 1 度も参照のない変数／関数を削除します。
- 参照の無い変数／関数を探索するために，実行開始アドレスを指定する -entry オプションが必要です。
- パラメータに symbol_delete を指定できます。ただし，その意味はパラメータ指定が無い場合と同じです。
- -optimize および -nooptimize オプションを複数回指定した場合は，最後の指定が有効になります。
- 次の場合は，警告を出力して，リンク時最適化を実施しません。
  - -entry オプションを同時に指定していない場合
- 次の場合は，エラーになります。
  - -optimize オプションのパラメータに無効な文字列を指定した場合

-SEction_forbid 【V2.01.00 以降】

特定セクションの，リンク時最適化を抑止します。

［指定形式］

```
-SEction_forbid=sub [, ...]
    sub : = ( section-name[, ...] )
       |   file ( section-name[, ...] )
       |   module ( section-name[, ...] )
```

- 省略時解釈
  なし

［詳細説明］

- 特定セクションの，リンク時最適化を抑止します。
- リンク時最適化を抑止したいセクションを *section-name* で指定します。
- *section-name* の前に *file* または *module* を指定することにより，特定の入力ファイル，または特定のライブラリ・モジュールに含まれるセクションのみを，リンク時最適化の抑止対象にできます。
  - 入力ファイル名を指定する場合は，リンク・マップ・ファイルに表示されたとおりの入力ファイル名を，大文字／小文字を区別して指定してください。
    ［リンク・マップ・ファイルの出力例］

    ```
    *** Options ***
    -Input=DefaultBuild¥main.obj
    ```
- 本オプションを複数回指定した場合，すべての指定が有効です。
- 次の場合は，エラーになります。
  - 指定した *section-name*，*file*，*module* がみつからない場合
  - *section-name* を指定していない場合
- 次の場合は，警告を出力して，本オプション指定を無視します。
  - -nooptimize オプションを同時に指定している場合

［使用例］

- .SEC1 セクションへのリンク時最適化を抑止します。

  ```
  >rlink a.obj b.obj -optimize -section_forbid=(.SEC1)
  ```
- a.obj 内の .SEC1，.SEC2 セクションへのリンク時最適化を抑止します。

  ```
  >rlink a.obj b.obj -optimize -section_forbid=a.obj(.SEC1,.SEC2)
  ```

-Absolute_forbid 【V2.01.00 以降】

特定アドレス範囲内の，リンク時最適化を抑止します。

[指定形式]

```
-Absolute_forbid=range [, ...]
    range := address[+size]
```

- 省略時解釈
  なし

[詳細説明]

- 特定アドレス範囲内の，リンク時最適化を抑止します。
- リンク時最適化を抑止したい範囲を，*address*，*size* で指定します。
  address+size の範囲に含まれるセクションが，最適化抑止の対象になります。
- *address*，*size* は，0 ～ ffffffff までの 16 進数で指定します。
- +*size* を省略した場合は，+0 を指定したとみなします。
- 本オプションを複数回指定した場合，すべての指定が有効です。
- 次の場合は，警告を出力して，本オプション指定を無視します。
  - -nooptimize オプションを同時に指定している場合

[備考]

- -start オプションによるオーバーレイ配置と，本オプションの指定範囲が重なる場合，重なる範囲に含まれる全てのオーバーレイ・セクションが最適化抑止の対象になります。
  特定のセクションのみ最適化を抑止したい場合は，-section_forbid オプションを使用してください。

[使用例]

- アドレス 0x1000 ～ 0x11ff へのリンク時最適化を抑止します。

```
>rlink a.obj b.obj -optimize -absolute_forbid=1000+200
```

-SYmbol_forbid 【V2.01.00 以降】

特定シンボルの，リンク時最適化を抑止します。

［指定形式］

```
-SYmbol_forbid=symbol[, ...]
```

- 省略時解釈
  なし

［詳細説明］

- 特定シンボルの，リンク時最適化を抑止します。
- リンク時最適化で削除したくない変数／関数名を *symbol* で指定します。C 言語で定義した変数名，関数名は，プログラム中での定義名先頭に _ を付加します。
- *symbol* で指定した変数／関数から参照している変数／関数も，削除しません。
- 本オプションを複数回指定した場合，すべての指定が有効です。
- 次の場合は，エラーになります。
  - 指定した *symbol* がみつからない場合
  - *symbol* を指定していない場合
- 次の場合は，警告を出力して，本オプション指定を無視します。
  - -nooptimize オプションを同時に指定している場合

［使用例］

- C 言語上の関数 “sub()” の削除を抑止します。

```
>rlink a.obj b.obj -optimize -symbol_forbid=_sub
```

-ALLOW_OPTIMIZE_ENTRY_BLOCK【V2.06.00 以降】

実行開始シンボルより前に配置されている領域を最適化の対象にします。

[指定形式]

```
-ALLOW_OPTIMIZE_ENTRY_BLOCK
```

- 省略時解釈
  実行開始シンボルより前に配置されている領域を最適化の対象外にします。

[詳細説明]

- 実行開始シンボルより前に配置されている領域を最適化の対象にします。
- 本オプションを複数回指定した場合，１回指定した場合と同じ意味になります。このとき，警告を出力します。

[備考]

- 本オプションは，最適化を使用しないリンク処理では無効です。
- 本オプションは，-entry オプションでアドレスを指定した場合は警告を出力して無視します。
- 本オプションは，-entry オプションを指定していない場合は無効です。

[使用例]

- 実行開始シンボルより前に配置されている領域を含めて最適化を実施します。

```
>rlink a.obj b.obj -optimize -entry=_main -allow_optimize_entry_block
```

セクション指定

セクション指定オプションには，次のものがあります。

- -STARt
- -FSymbol
- -ALIGNED_SECTION

-STARt

セクションの開始アドレスを指定します。

[指定形式]

```
-STARt=suboption [, ...]
  suboption := placement-unit[, ...][/address]
  placement-unit := overlay-sections
                  |  section
  overlay-sections := ( section-list : section-list [: ...] )
  section-list := section [, ...]
```

- 省略時解釈
  絶対アドレス・セクションをアドレスの小さい順に配置し，絶対アドレス・セクションの末尾から指定された入力ファイル順に出現する相対アドレス・セクションを配置します。

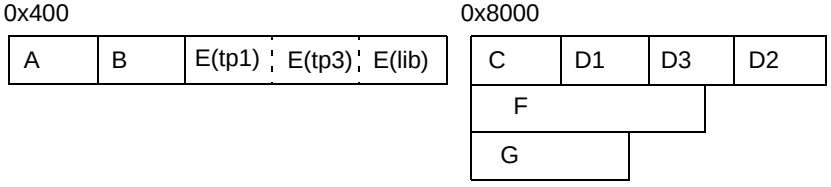
[詳細説明]

- セクション *section* の開始アドレス *address* を指定します。
  address は 16 進数で指定します。
- *section* にはワイルドカード（*）も使用可能です。
  ワイルドカードで指定したセクションは，入力順に展開します。
- セクションをコロン（:）で区切ることにより，複数のセクション（カンマ（,）で区切って指定）を同一アドレスに割り付ける（セクション・オーバーレイ配置）ことが可能です。
  同一アドレスに割り付け指定したセクション間は，指定順に割り付けます。
  また，丸かっこ “()” で囲むことにより，オーバーレイ配置する対象セクションを変更することができます。
- 同一セクション内のオブジェクトは，入力ファイルの指定順，入力ライブラリの指定順に割り付けます。
- アドレスの指定を省略した場合は，0 番地から割り付けます。
- -start オプションで指定していないセクションは，最終割り付けアドレスに続いて割り付けます。

[使用例]

- 以下の順番でオブジェクトを入力する場合のセクション配置を示します（かっこ内は各オブジェクトが持つセクションです）。
  tp1.obj(A,D1,E)
  tp2.obj(B,D3,F)
  tp3.obj(C,D2,E,G)
  lib.lib(E)

  - -start=A,B,E/400,C,D*:F:G/8000 オプションを指定した場合

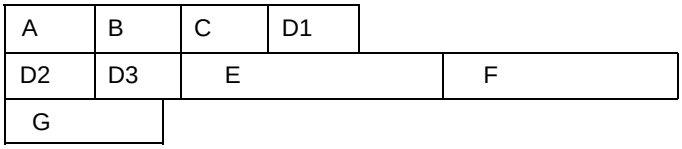


  - “:” で区切った C，F，G セクションは，同一アドレスに割り付きます。
  - ワイルドカードで記述したセクション（ここでは D で始まる名前のセクション）は，入力した順番で割り付きます。
  - 同名セクション内（ここでは E セクション）は，入力したオブジェクトから順番に割り付きます。

- ライブラリ入力による同名セクション（ここではEセクション）は，入力オブジェクトの次に割り付きます。

- -start=A,B,C,D1:D2,D3,E,F:G/400 オプションを指定した場合

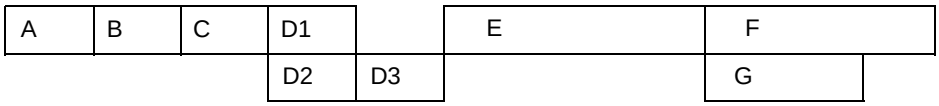
0x400



- ":" で区切った直後のセクション（この例の場合はA，D2，G）を先頭として，それぞれの先頭が同一アドレスに割り付きます。

- -start=A,B,C,(D1:D2,D3),E,(F:G)/400 オプションを指定した場合

0x400



- "()" で同一アドレス配置を囲んだ場合，"()" の直前のセクション（この例の場合はC，E）の直後を先頭として，"()" 内の同一アドレス配置が行われます。
- "()" の直後のセクション（この例の場合はE）は，"()" 内の最後尾のセクションの直後に続けて配置されます。

［備考］

- 本オプションは，-form={object|relocate|library} オプション，または -strip オプションを指定した場合は無効となります。
- "()" は，ネストして記述することはできません。
- "()" 内では，1つ以上のコロン ":" の記述が必要です。
  ":" を記述しない場合，"()" を記述することはできません。
- "()" を記述した場合，"()" の外にコロン ":" を記述することはできません。
- "()" を使用して本オプションを記述した場合，リンカの最適化機能は無効になります。

-FSymbol

外部定義シンボルをシンボル・アドレス・ファイルに出力します。

[指定形式]

```
-FSymbol=section[,section]...
```

- 省略時解釈
  なし

[詳細説明]

- セクション *section* 内の外部定義シンボルをアセンブラ制御命令形式でファイル（シンボル・アドレス・ファイル）に出力します。
  ファイル名は “ *出力ファイル名* .fsy” となります。

[使用例]

- セクション sct2，sct3 内の外部定義シンボルをシンボル・アドレス・ファイル test.fsy に出力します。

```
>rlink a.obj b.obj -fsymbol=sct2,stc3 -output=test.abs
```

シンボル・アドレス・ファイル test.fsy の出力例を以下に示します。

```
;RENESAS OPTIMIZING LINKER GENERATED FILE xxxx.xx.xx
;fsymbol = sct2,sct3

;SECTION NAME = sct2
    .public _f
_f .equ 0x0
    .public _g
_g .equ 0x16
;SECTION NAME = sct3
    .public _main
_main .equ 0x20
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，または -strip オプションを 指定した場合は無効となります。

-ALIGNED_SECTION

セクションのアライメント数を 16 バイト に変更します。

[指定形式]

```
-ALIGNED_SECTION=section[,section]...
```

- 省略時解釈
  なし

[詳細説明]

- セクション *section* のアライメント数を 16 バイト に変更します。

[使用例]

- A セクションのアライメント数を 16 バイト に変更します。

```
>rlink a.obj b.obj -aligned_section=A
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，および -extract，-strip オプションを 指定した場合は無効となります。

ベリファイ指定

ベリファイ指定オプションには，次のものがあります。

- -CPu

-CPu

セクションの割り付けアドレスの整合性をチェックします。

[指定形式]

```
-CPU=suboption[, ...]
  suboption := type=address1-address2
```

- 省略時解釈
  なし

[詳細説明]

- セクションの割り付けアドレスの整合性をチェックします。
  メモリ種別 *type* のセクションの割り付けアドレスに対して，指定したアドレス範囲に収まらない場合は，エラーを出力します。
- *type* に指定可能なものを以下に示します。
  これ以外のものを指定した場合は，エラーとなります。

ROm	セクションの割り付け領域を ROM とします。
RAm	セクションの割り付け領域を RAM とします。
FIX	セクションの割り付け領域をアドレス固定の領域（I/O エリアなど）とします。 アドレス範囲が ROM，および RAM と重複した場合は，“FIX” を有効とします。

- *address1*，*address2* には，整合性をチェックするアドレス範囲の先頭アドレス，終了アドレスを 16 進数で指定します。

[使用例]

- .text セクションが 0x100 ～ 0x1FF，.bss セクションが 0x200 ～ 0x2FF の範囲に収まる場合は正常終了します。
  収まらない場合はエラーを出力します。

```
>rlink a.obj b.obj -start=.text/100,.bss/200 -cpu=ROM=100-1FF,RAM=200-2FF
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，または -strip オプションを指定した場合は無効となります。

サブコマンド・ファイル指定

サブコマンド・ファイル指定オプションには，次のものがあります。

- -SUbcommand

-SUbcommand

オプションをサブコマンド・ファイルで指定します。

[指定形式]

```
-SUbcommand=file
```

- 省略時解釈
  なし

[詳細説明]

- オプションをサブコマンド・ファイル *file* で指定します。
- サブコマンド・ファイルで指定したオプション内容を，コマンド・ライン上の本オプション指定位置に展開し，実行します。
- サブコマンド・ファイルについての詳細は，「2.4.2 サブコマンド・ファイルの使用方法」を参照してください。

[使用例]

- サブコマンド・ファイル test.sub を以下の内容で作成します。

```
input file2.obj file3.obj        ; ここはコメントです。
library lib1.lib, &              ;“&” は継続行を示します。
lib2.lib
```

コマンド・ライン上でサブコマンド・ファイル test.sub を指定します。

```
>rlink file1.obj -subcommand=test.sub file4.obj
```

コマンド・ラインは以下のように展開され，ファイルの入力順序は，file1.obj，file2.obj，file3.obj，file4.obj となります。

```
>rlink file1.obj file2.obj file3.obj -library=lib1.lib,lib2.lib file4.obj
```

その他

その他のオプションには，次のものがあります。

- -S9
- -STACk
- -COmpress
- -NOCOmpress
- -MEMory
- -REName
- -LIB_REName 【V2.01.00 以降】
- -DELete
- -REPlace
- -EXTract
- -STRip
- -CHange_message
- -Hide
- -Total_size
- -VERBOSE 【V2.03.00 以降】
- -LOgo
- -NOLOgo
- -END
- -EXIt

-S9

S9 レコードを終端に出力します。

[指定形式]

```
-S9
```

- 省略時解釈
  なし

[詳細説明]

- エントリ・ポイント・アドレスが 0x10000 を超える場合でも，S9 レコードを終端に出力します。

[使用例]

- エントリ・ポイント・アドレスが 0x10000 を超える場合でも，S9 レコードを終端に出力します。

```
>rlink a.obj b.obj -form=stype -s9
```

[備考]

- 本オプションは，-form=stype オプションを指定していない場合は無効となります。

-STACk

スタック情報ファイルを出力します。

[指定形式]

```
-STACk
```

- 省略時解釈
  なし

[詳細説明]

- スタック情報ファイルを出力します。
- ファイル名は，“ *出力ファイル名* .sni” となります。

[使用例]

- スタック情報ファイル c.sni を出力します。

```
>rlink a.obj b.obj -stack
```

[備考]

- 本オプションは，-form={object|relocate|library} オプション，および -strip オプションを指定した場合は無効となります。

-COmpress

デバッグ情報を圧縮します。

[指定形式]

```
-COmpress
```

- 省略時解釈
  デバッグ情報を圧縮しません（-nocompress オプションの指定と同じです）。

[詳細説明]

- デバッグ情報を圧縮します。
- デバッグ情報を圧縮すると，デバッガのロード速度が速くなります。

[使用例]

- デバッグ情報を圧縮します。

```
>rlink a.obj b.obj -compress
```

[備考]

- 本オプションは，-form={object|relocate|library|hexadecimal|stype|binary} オプション，または -strip オプションを指定した場合は無効となります。

-NOCOmpress

デバッグ情報を圧縮しません。

[指定形式]

```
-NOCOmpress
```

- 省略時解釈
  デバッグ情報を圧縮しません。

[詳細説明]

- デバッグ情報を圧縮しません。
- 本オプションを指定すると，-compress オプションを指定した場合に比べてリンク時間が短くなります。

[使用例]

- デバッグ情報を圧縮しません。

```
>rlink a.obj b.obj -nocompress
```

-MEMory

リンク時に使用するメモリ量を指定します。

[指定形式]

```
-MEMory=[occupancy]
```

- 省略時解釈
  従来通りの処理を行います（-memory=high オプションの指定と同じです）。

[詳細説明]

- リンク時に使用するメモリ量 *occupancy* を指定します。
- *occupancy* に指定可能なものを以下に示します。

High	従来通りの処理を行います。
Low	リンク時に必要な情報のロードを細かく行うことにより，使用するメモリ量の削減を行います。 ファイル・アクセスの頻度が増えるため，メモリ使用量が実装メモリを超えない状況では High を指定した場合よりも処理が遅くなります。

- *occupancy* を省略した場合は，High を指定したものとみなします。
- 大規模なプロジェクトをリンクした際，最適化リンカのメモリ使用量が稼動マシンの実装メモリ量を超えてしまい，動作が遅くなっているような場合は，*occupancy* に Low を指定してください。

[使用例]

- 使用するメモリ量の削減を行います。

```
>rlink a.obj b.obj -memory=low
```

[備考]

- 以下の場合，-memory=low オプションの指定は無効となります。
  - -form={absolute|hexadecimal|stype|binary} オプションと以下のオプションを同時に指定した場合
    -optimize，-compress，-delete，-rename，-lib_rename，-map，-stack オプションのいずれか
    -list と，-show={reference|xreference|struct} のいずれか
  - -form=library オプションと以下のオプションを同時に指定した場合
    -delete，-rename，-extract，-hide，-replace，-allow_duplicate_module_name オプションのいずれか
  - -form={object|relocate} オプションと以下のオプションを同時に指定した場合
    -extract オプション

  また，入力ファイルや出力ファイルの形式によっても無効となる組み合わせがあります。
  詳細については，「表 2.5 出力形式と入力ファイル，他のオプションとの関係」を参照してください。

-REName

外部シンボル名，セクション名を変更します。

[指定形式]

```
-REName=suboption[, ...]
  suboption := ( names )
             |  file ( names )
             |  module ( names )
  names := name1=name2[, ...]
```

- 省略時解釈
  なし

[詳細説明]

- 外部シンボル名，セクション名を変更します。
- *name1* には変更対象のシンボル名，またはセクション名，*name2* には変更後のシンボル名，またはセクション名を指定します。
- *file* を指定することで，*file* に含まれるセクションだけ名前を変更することができます。
- ライブラリ出力時（-form=library 指定時）には，*module* を指定することで，入力ライブラリ内の *module* に含まれるセクションだけ名前を変更することができます。
  それ以外の場合で入力ライブラリ内のセクション名を変更する場合は，-lib_rename オプションを使用してください。
- *file*，*module* を指定して，それらに含まれる外部シンボルだけ名前を変更することもできます。
- C 変数名を指定する場合は，プログラム中での定義名の先頭に “_” を付加します。
- 指定した名前がセクション，シンボルの両方に存在した場合は，シンボル名を優先します。
- 同一のファイル名，モジュール名が複数存在する場合は，先に入力した方を優先します。
- 本オプションを複数回指定した場合，すべての指定が有効になります。
- 次の場合はエラーとなります。
  - 指定した *name*，*file*，*module* がみつからない場合
  - -extract オプション，または -strip オプションと同時に指定した場合

[使用例]

- シンボル名 _sym1 を data に変更します。

```
>rlink a.obj b.obj -rename=(_sym1=data)
```

- ライブラリ・モジュール lib1 内の .text セクションを P セクションに変更します。

```
>rlink -library=lib.lib -rename=lib1(.text=P) -form=library
```

[備考]

- -form={absolute|hexadecimal|stype|binary} オプションを指定した場合は，入力されたライブラリのセクション名を変更することはできません。
- コンパイル・オプション -Xmerge_files と本オプションを組み合わせて使用した場合，動作は保証されません。

-LIB_RENname 【V2.01.00 以降】

ライブラリから入力されたシンボル名，セクション名を変更します。

[指定形式]

```
-lib_rename = name1=name2[, ...]
-lib_rename = file(name1=name2[, ...])
-lib_rename = "file|module[|module ...](name1=name2[, ...])"
```

- 省略時解釈
  なし

[詳細説明]

- -library オプションで指定したライブラリ内モジュールに含まれる外部シンボル名，セクション名を変更します。
- *name1* には変更対象のシンボル名，またはセクション名，*name2* には変更後のシンボル名，またはセクション名を指定します。
- C 変数名を指定する場合は，プログラム中での定義名の先頭に “_” を付加します。
- 指定した名前がセクション，シンボルの両方に存在した場合は，シンボル名を優先します。
- 同一のファイル名，モジュール名が複数存在する場合は，先に入力した方を優先します。
- 本オプションを複数回指定した場合，すべての指定が有効になります。
- 次の場合はエラーとなります。
  - 指定した *name*，*file*，*module* がみつからない場合
  - パラメータを省略した場合
  - -form={object,library} オプション，-extract オプション，または -strip オプションと同時に指定した場合

[備考]

- -form={absolute|hexadecimal|stype|binary} オプションを指定した場合は，-show=struct オプションを同時に指定できません。
- 入力されたライブラリのセクション名を変更することはできません。
- コンパイル・オプション -Xmerge_files と本オプションを組み合わせて使用した場合，動作は保証されません。

[使用例]

- b.lib 及び c.lib にある _sym1 を _data に変更します。

```
>rlink a.obj -lib=b.lib,c.lib  -lib_rename=(_sym1=_data)
```

- b.lib にある全てのモジュール内の _sym1 を _data に変更します。

```
>rlink a.obj -lib=b.lib,c.lib  -lib_rename=b.lib(_sym1=_data)
```

- b.lib にあるモジュール m1 及び m2 内の _sym1 を _data に変更します。

```
>rlink a.obj -lib=b.lib,c.lib  -lib_rename="b.lib|m1|m2(_sym1=_data)"
```

-DELete

外部シンボル名，またはライブラリ・モジュールを削除します。

[指定形式]

```
-DELete=suboption[, ...]
  suboption := (symbol[, ...])
             |  file ( symbol[, ...] )
             |  module
```

- 省略時解釈
  なし

[詳細説明]

- 外部シンボル名 *symbol*，またはライブラリ・モジュール *module* を削除します。
- 特定のファイル *file* に含まれるシンボル名，モジュールを削除することもできます。
- C 変数名，C 関数名を指定する場合は，プログラム中での定義名の先頭に “_” を付加します。
- 同一ファイル名が複数存在する場合は，先に入力した方を優先します。
- 本オプションでシンボル名の削除を指定した場合，オブジェクトは削除されず，属性が内部シンボルに変更されます。

[使用例]

- 全ファイル中のシンボル名 _sym1 を削除します。

```
>rlink a.obj -delete=(_sym1)
```

- b.obj 内のシンボル名 _sym2 を削除します。

```
>rlink a.obj b.obj -delete=b.obj(_sym2)
```

[備考]

- 本オプションは，-extract オプション，または -strip オプションと同時に指定した場合は無効となります。
- -form=library オプションを指定した場合は，ライブラリ・モジュールを削除することができます。
- -form={absolute|relocate|hexadecimal|stype|binary} オプションを指定した場合は，外部シンボルを削除することができます。
- コンパイル・オプション -Xmerge_files と本オプションを組み合わせて使用した場合，動作は保証されません。

-REPlace

ライブラリ・モジュールを置換します。

[指定形式]

```
-REPlace=suboption[, ...]
  suboption := file
             |  file ( module [, ...] )
```

- 省略時解釈
  なし

[詳細説明]

- 指定したファイル *file*，またはライブラリ・モジュール *module* と -library オプションで指定したライブラリ・ファイル内の同名モジュールを置換します。

[使用例]

- file1.obj とライブラリ・ファイル lib1.lib 内のモジュール file1 を置換します。

```
>rlink -library=lib1.lib -replace=file1.obj -form=library
```

- モジュール mdl1 とライブラリ・ファイル lib1.lib 内のモジュール mdl1 を置換します。

```
>rlink -library=lib1.lib,lib2.lib -replace=lib1.lib(mdl1) -form=library
```

[備考]

- 本オプションは，-form={object|relocate|absolute|hexadecimal|stype|binary} オプション，および -extract，-strip オプションを指定した場合は無効となります。
- コンパイル・オプション -Xmerge_files と本オプションを組み合わせて使用した場合，動作は保証されません。

-EXTract

ライブラリ・モジュールを抽出します。

[指定形式]

```
-EXTract=module[,module]...
```

- 省略時解釈
  なし

[詳細説明]

- ライブラリ・モジュール *module* を -library オプションで指定したライブラリ・ファイルから抽出します。

[使用例]

- ライブラリ・ファイル lib1.lib からモジュール file1 を抽出し，オブジェクト・ファイル出力形式のファイルを出力します。

```
>rlink -library=lib1.lib -extract=file1 -form=obj
```

[備考]

- 本オプションは，-form={absolute|hexadecimal|stype|binary} オプション，および -strip オプションを指定した場合は無効となります。
- -form=library オプションを指定した場合は，ライブラリ・モジュールを削除することができます。
- -form={absolute|relocate|hexadecimal|stype|binary} オプションを指定した場合は，外部シンボルを削除することができます。

-STRip

ロード・モジュール・ファイル，ライブラリ・ファイルのデバッグ情報を削除します。

[指定形式]

```
-STRip
```

- 省略時解釈
  なし

[詳細説明]

- ロード・モジュール・ファイル，ライブラリ・ファイルのデバッグ情報を削除します。
- デバッグ情報を削除する前のファイルは，“ *ファイル名* .abk ” という名前のファイルにバックアップします。
- 本オプションを指定した場合，入力ファイルと出力ファイルは 1 対 1 対応になります。

[使用例]

- file1.abs，file2.abs，file3.abs のデバッグ情報を削除し，それぞれ file1.abs，file2.abs，file3.abs に出力します。デバッグ情報を削除する前のファイルは，file1.abk，file2.abk，file3.abk にバックアップします。
  ＜コマンド・ライン＞

```
>rlink -subcommand=sub.txt
```

＜サブコマンド・ファイル sub.txt ＞

```
-input=file1.abs file2.abs file3.abs
-strip
```

[備考]

- 本オプションは，-form={object|relocate|hexadecimal|stype|binary} オプションを指定した場合は無効となります。

-CHange_message

インフォメーション，ワーニング，エラーのメッセージ種別を変更します。

[指定形式]

```
-CHange_message=suboption [ , ...]
  suboption := level
             |  level=range[, ...]
  range := num
         |  num-num
```

- 省略時解釈
  なし

[詳細説明]

- インフォメーション，ワーニング，エラーのメッセージ種別 *level* を変更します。
- メッセージ出力時の処理継続／中断を変更することができます。
- *level* に指定可能なものを以下に示します。

Information	インフォメーション
Warning	ワーニング
Error	エラー

- メッセージ番号 *num* を指定すると，指定した番号のメッセージの種別を変更します。
  また，ハイフン（-）を使用して，メッセージ番号の範囲を指定することもできます。
- *num* には，コンポーネント番号（05），発生フェーズ（6）に続けて出力される 4 桁の数値（E0562310 の場合は 2310）を指定します。
- メッセージ番号の指定を省略した場合は，すべてのメッセージの種別を指定したものに変更します。

[使用例]

- E0561310 をワーニングに変更し，E0561310 出力時も処理を継続します。

```
>rlink a.obj b.obj -change_message=warning=1310
```

- すべてのインフォメーション，ワーニングをエラーに変更します。
  メッセージを 1 つでも出力すると，処理を中断します。

```
>rlink a.obj b.obj -change_message=error
```

-Hide

出力ファイル内のローカル・シンボル名情報を消去します。

[指定形式]

```
-Hide
```

- 省略時解釈
  なし

[詳細説明]

- 出力ファイル内のローカル・シンボル名情報を消去します。
- ローカル・シンボルに関する名前の情報が消去されるため，バイナリ・エディタなどでファイルを開いてもローカル・シンボル名を確認することができなくなります。
  なお，生成されるファイルの動作に対する影響は一切ありません。
- 本オプションは，ローカル・シンボル名を機密扱いにしたい場合などに指定してください。
- 秘匿対象となるシンボルの種類を以下に示します。
  - C ソース：static 型修飾子を指定した変数名，関数名など
  - C ソース：goto 文のラベル名
  - アセンブリ・ソース：外部定義（参照）シンボル宣言していないシンボル名
    なお，エントリ関数名は秘匿対象にはなりません。

[使用例]

- 出力ファイル内のローカル・シンボル名情報を消去します。

```
>rlink a.obj b.obj -hide
```

本オプションの機能が有効となる C ソース記述の例を以下に示します。

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

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

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

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

［備考］

- 本オプションは，-form={absolute|relocate|library} オプションを指定した場合のみ有効となります。
- コンパイル，アセンブル時に goptimize オプションを指定したファイルを入力する場合，出力ファイル形式が relocate，library の場合は本オプションを指定できません。
- 外部変数アクセス最適化を行う状況で本オプションを指定する場合は，一度目のリンク時には指定せず，二度目のリンク時にのみ指定してください。
- デバッグ情報内のシンボル名は，本オプションを指定しても削除されません。

-Total_size

リンク後のセクションの合計サイズを標準エラー出力に表示します。

[指定形式]

```
-Total_size
```

- 省略時解釈
  なし

[詳細説明]

- リンク後のセクションの合計サイズを標準エラー出力に表示します。
- 以下の 3 種類のセクションに分けて，合計サイズを表示します。
  - 実行可能なプログラム・セクション
  - プログラム・セクション以外の ROM 領域配置セクション
  - RAM 領域配置セクション
- 本オプションを指定することにより，ROM/RAM に配置する合計のセクション・サイズを容易に認識することができます。

[使用例]

- リンク後のセクションの合計サイズを標準エラー出力に表示します。

```
>rlink a.obj b.obj -total_size
```

[備考]

- リンク・マップ・ファイルに合計サイズを表示するためには，-show=total_size オプションを指定する必要があります。
- -rom オプションを使用する場合，転送元（ROM）と転送先（RAM）の両方で領域を使用するため，双方の合計サイズに対してセクション・サイズを加算します。

-VERBOSE 【V2.03.00 以降】

詳細情報を標準エラー出力に表示します。

[指定形式]

```
-VERBOSE=<sub>[, ...]
sub : CRC
```

- 省略時解釈
  なし

[詳細説明]

- サブオプションに指定した内容を標準エラー出力に表示します。
- サブオプションに指定可能なものを以下に示します。

CRC	CRC の演算結果，および出力位置アドレスを表示します。 crc オプションを指定したときに有効です。

[使用例]

- CRC の演算結果，および出力アドレスを標準エラー出力に表示します。

```
> rlink a.obj -form=stype -start=.SEC1/1000 -crc=2000=1000-10ff/CCITT -verbose=crc
```

-LOgo

コピーライトを出力します。

[指定形式]

```
-LOgo
```

- 省略時解釈
  コピーライトを出力します。

[詳細説明]

- コピーライトを出力します。

[使用例]

- コピーライトを出力します。

```
>rlink a.obj b.obj -logo
```

-NOLOgo

コピーライトの出力を抑止します。

[指定形式]

```
-NOLOgo
```

- 省略時解釈
  コピーライトを出力します（-logo オプションの指定と同じです）。

[詳細説明]

- コピーライトの出力を抑止します。

[使用例]

- コピーライトの出力を抑止します。

```
>rlink a.obj b.obj -nologo
```

-END

本オプションより前に指定したオプション列を実行します。

[指定形式]

```
-END
```

- 省略時解釈
  なし

[詳細説明]

- 本オプションより前に指定したオプション列を実行します。
  リンク処理の終了後に，本オプションより後に指定したオプション列の入力，リンク処理を継続します。

[注意]

- 本オプションは，サブコマンド・ファイル内のみで使用することができます。

[使用例]

- サブコマンド・ファイル test.sub を以下の内容で作成します。

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

コマンド・ライン上でサブコマンド・ファイル test.sub を指定します。

```
>rlink -subcommand=test.sub
```

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

-EXIt

オプション指定の終了を指定します。

[指定形式]

```
-EXIt
```

- 省略時解釈
  なし

[詳細説明]

- オプション指定の終了を指定します。

[注意]

- 本オプションは，サブコマンド・ファイル内のみで使用することができます。

[使用例]

- サブコマンド・ファイル test.sub を以下の内容で作成します。

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

コマンド・ライン上でサブコマンド・ファイル test.sub を指定します。

```
>rlink -subcommand=test.sub -nodebug
```

（1）～（3）の処理を実行し，a.abs を出力します。
本オプションの実行後にコマンド・ライン上で指定している -nodebug オプションは無効になります。

2.6 オプションの複数指定

ここでは，ccrh コマンドに対してオプションを同時に 2 つ以上指定した場合について説明します。

2.6.1 優先順序

以下のオプションは，ほかの特定のオプションを無効とします。

-V/-h	ほかのすべてのオプションは無効となります。 このとき，警告を出力しません。
-P	プリプロセス処理で終了するため，それ以降の処理に関するオプションは無効となります。 このとき，警告を出力します。
-S	コンパイル処理で終了するため，それ以降の処理に関するオプションは無効となります。 このとき，警告を出力します。
-c	アセンブル処理で終了するため，それ以降の処理に関するオプションは無効となります。 このとき，警告を出力します。
-Xcpu=g3k	-Xfloat オプションは無効となります。 このとき，警告を出力します。
-lang=c99	-Xmisra2004 は無効となります。 このとき，警告を出力します。

以下の組み合わせでオプションを指定した場合は，警告を出力して，最後に指定したものが有効となります。

- -P，-S，-c
- -D，-U（シンボル名が同じ場合）
- -Onothing，-Odefault，-Osize，-Ospeed

なお，オプションの指定順序により，以下のオプションは無効となります。

- -Onothing，-Odefault，-Osize，-Ospeed の前に指定した -O*item* 注

注 -O*item* については，「2.5.1 コンパイル・オプション “-O ”」を参照してください。ただし，-Omap，-Osmap は -O*level* の影響をうけません。

2.6.2 機能矛盾

以下の組み合わせでオプションを指定した場合は，エラーとなります。

- -Omap，-Xep=fix
- -Osmap，-Xep=fix
- -Omap と -Xsection=data=ep_disp16 もしくは ep_disp23
- -Osmap と -Xsection=data=ep_disp16 もしくは ep_disp23
- -Xmisra2004 と -Xmisra2012

2.6.3 依存関係

以下のオプションは，ほかの特定のオプションにより動作が変わります。

-Xpreprocess	-P オプションと同時に指定しない場合，無効となります。 このとき，警告を出力しません。
-o	-P/-S/-c オプションと同時に指定した場合，生成ファイルの種別は，それぞれプリプロセス処理済みファイル／アセンブリ・ソース・ファイル／オブジェクト・ファイルとなります。
-g	-O オプションと同時に指定した場合，デバッグ情報が正しくないことがあります。

-Oinline	-Xmerge_files オプションと同時に指定した場合，ファイル間インライン展開を行う場合があります。

2.6.4 #pragma 指定との関係

以下のオプションは，#pragma 指定との関係により動作が変わります。

- -Xep=callee，または -Xep の指定なし
  #pragma section で以下の属性指定文字を同時に指定した場合は，エラーとなります。
  edata，edata23，tdata，tdata4，tdata5，tdata7，tdata8，ep_auto，ep_disp4，ep_disp5，ep_disp7，ep_disp8，ep_disp16，ep_disp23

3. 出力ファイル

この章では，ビルドにより各コマンドが出力する各種リストのフォーマットなどについて説明します。

3.1 アセンブル・リスト・ファイル

ここでは，アセンブル・リスト・ファイルについて説明します。
アセンブル・リストとは，ソースをコンパイル，アセンブルして出力するコードをリスト形式にしたものです。
これにより，コンパイル，アセンブルした結果が，どのようなコードになっているかを確認することができます。

3.1.1 アセンブル・リストの構成

アセンブル・リストの構成と内容を以下に示します。

出力情報	説明
アセンブル・リスト	アセンブル時のロケーション・カウンタ値，コード，行番号，ソース・プログラム
セクション・リスト	セクションの種類，サイズ，名称
コマンド・ライン情報	アセンブラのコマンド・ライン文字列

3.1.2 アセンブル・リスト

アセンブル時のロケーション・カウンタ値，コード，行番号，ソース・プログラムを出力します。
アセンブル・リストの出力例を以下に示します。

```
(1)       (2)                 (3) (4)
OFFSET    CODE                NO  SOURCE STATEMENT

00000000                       1  #CC-RH Compiler RH850 Assembler Source
00000000                       2  #@      CC-RH Version : VX.XX.XXx [XX Xxx XXXX]
00000000                       3  #@      Command : main.c -Xcommon=rh850 -S
00000000                       4  #@      compiled at Xxx Xxx XX XX:XX:XX XXXX
00000000                       5          .cseg text
00000000                       6          ld.w $_data,r12
00000000  440E0000              -- movhi       0x0,gp,r1
00000004  21670100              -- ld.w        0x0[r1],r12
00000008                       7
00000000                       8          .dseg data
00000000  00000000             9  _data:  .dw 0
00000004                      10
```

項番	説明
(1)	ロケーション・カウンタ値 その行のソース・プログラムに対して生成したコードの先頭に対するロケーション・カウンタ値を出力します。
(2)	コード その行のソース・プログラムに対して生成したコード（機械語命令，またはデータ）を出力します。 1 バイトごとに 2 桁の 16 進数で表記します。
(3)	行番号 その行の行番号を出力します。 10 進数で表記します。

項番	説明
(4)	ソース・プログラム その行のソース・プログラムを出力します。 その行の命令に対して命令展開が生じた場合，その命令展開によって生成した機械語命令の命令列を逆アセンブルした結果を，“--” 以降に示します。 コンパイラ情報（1 ～ 4 行目）は，コンパイラが出力したアセンブリ・ソース・ファイルをアセンブルした場合のみ出力します。

3.1.3 セクション・リスト

セクションの種類，サイズ，名称を出力します。
セクション・リストの出力例を以下に示します。

```
Section List
(1)     (2)                 (3)
Attr    Size                Name

TEXT            8 (00000008) .text
DATA            4 (00000004) .data
```

項番	説明
(1)	セクションの種類 セクションの種類を再配置属性として出力します。
(2)	セクション・サイズ セクションのサイズを出力します（単位：バイト）。 10 進数，およびかっこ内に 16 進数で表記します。
(3)	セクション名 セクション名を出力します。

3.1.4 コマンド・ライン情報

アセンブラのコマンド・ライン文字列を出力します。
コマンド・ライン情報の出力例を以下に示します。

```
Command Line Parameter

a.asm -Xcommon=rh850 -Xprn_path      (1)
```

項番	説明
(1)	コマンド・ライン文字列 アセンブラに指定したコマンド・ライン文字列を出力します。

3.2 リンク・マップ・ファイル

ここでは，リンク・マップ・ファイルについて説明します。
リンク・マップとは，リンク結果の情報が書かれたもので，セクションの配置アドレスなどの情報を知ることができます。

3.2.1 リンク・マップの構成

リンク・マップの構成と内容を以下に示します。

出力情報	説明	-show オプション指定	-show オプション省略時
オプション情報	コマンド・ライン，サブコマンド・ファイルで指定したオプション列	―	出力する
エラー情報	エラー・メッセージ	―	出力する
リンク・マップ情報	セクション名，先頭／最終アドレス，サイズ，種別	―	出力する
	-show=relocation_attribute を指定した場合は，再配置属性を表示	-show=relocation_attribute	出力しない
合計セクション・サイズ	RAM，ROM，およびプログラム・セクションの合計サイズ	-show=total_size	出力しない
シンボル情報	静的定義シンボル名，アドレス，サイズ，種別（アドレス順），最適化実行の有無 -show=reference を指定した場合は，各シンボルの参照回数も出力します。 -show=struct を指定した場合は，構造体／共用体メンバのアドレスも出力します。	-show=symbol -show=reference -show=struct	出力しない
関数リスト情報	不正な間接関数呼び出し検出で用いる関数リスト情報	-show=cfi	出力しない
クロス・リファレンス情報	シンボルの参照情報	-show=xreference	出力しない
CRC 情報	CRC の演算結果および出力位置アドレス	―	CRC オプション指定時は常に出力

注意 -show オプションは，-list オプションを指定した場合に有効となります。
-show オプションについての詳細は，「-SHow」を参照してください。

3.2.2 オプション情報

コマンド・ライン，サブコマンド・ファイルで指定したオプション列を出力します。
コマンド・ライン，サブコマンド・ファイルで次のように指定した場合のオプション情報の出力例を以下に示します。

＜コマンド・ライン＞

```
>rlink -subcommand=test.sub -list -show
```

＜サブコマンド・ファイル test.sub ＞

```
input sample.obj
```

```
*** Options ***


-subcommand=test.sub    (1)

input sample.obj        (2)

-list                   (1)

-show                   (1)
```

項番	説明
(1)	コマンド・ラインで指定したオプション コマンド・ラインで指定したオプションを出力します（指定順）。
(2)	サブコマンド・ファイル内で指定したオプション サブコマンド・ファイル test.sub 内で指定したオプションを出力します。

3.2.3 エラー情報

エラー・メッセージを出力します。
エラー情報の出力例を以下に示します。

```
*** Error information ***

** E0562310 Undefined external symbol "_func_02" referenced in "sample.obj"     (1)
```

項番	説明
(1)	エラー・メッセージ エラー・メッセージを出力します。

3.2.4 リンク・マップ情報

各セクションの先頭／最終アドレス，サイズ，種別をアドレス順に出力します。
リンク・マップ情報の出力例を以下に示します。

```
*** Mapping List ***

(1)                             (2)       (3)         (4)     (5)
SECTION                         START     END         SIZE    ALIGN

.text
                                00000000  0000003b      3c    2
.data
                                fe600006  fe600003       4    4
.bss
                                fe600004  fe60000b       8    4
```

-show=relocation_attribute を指定した場合は，セクションに対応している再配置属性を表示します。再配置属性の出力例を以下に示します。

```
*** Mapping List ***

                                                                  (6)
SECTION                          START      END         SIZE    ALIGN    ATTRIBUTE

.text
                                 00000100   0000013b      3c    2        TEXT
.data
                                 000f0400   000f0403       4    4        DATA
.bss
                                 000f0404   000f040b       8    4        BSS
```

項番	説明
(1)	セクション名 セクション名を出力します。
(2)	先頭アドレス 先頭アドレスを出力します。 16 進数で表記します。
(3)	最終アドレス 最終アドレスを出力します。 16 進数で表記します。
(4)	セクション・サイズ セクション・サイズを出力します（単位：バイト）。 16 進数で表記します。
(5)	セクションのアライメント数 セクションのアライメント数を出力します。
(6)	セクションの再配置属性の種類 セクションの再配置属性を TEXT，CONST，DATA，BSS，OTHER の 5 種類に分類して出力します。

3.2.5 合計セクション・サイズ

-show=total_size オプションを指定した場合，ROM セクション，RAM セクション，およびプログラム・セクションの合計サイズを出力します。
合計セクション・サイズの出力例を以下に示します。

```
*** Total Section Size ***

RAMDATA SECTION:        00000660 Byte(s) (1)
ROMDATA SECTION:        00000174 Byte(s) (2)
PROGRAM SECTION:        000016d6 Byte(s) (3)
```

項番	説明
(1)	RAM データ・セクションの合計サイズ RAM データ・セクションの合計サイズを出力します。 16 進数で表記します。
(2)	ROM データ・セクションの合計サイズ ROM データ・セクションの合計サイズを出力します。 16 進数で表記します。
(3)	プログラム・セクションの合計サイズ プログラム・セクションの合計サイズを出力します。 16 進数で表記します。

3.2.6 シンボル情報

-show=symbol オプションを指定した場合は，外部定義シンボル，または静的内部定義シンボルのアドレス，サイズ，種別，最適化実行の有無をアドレス順で出力します。
また，-show=reference オプションを指定した場合は，各シンボルの参照回数も出力します。
シンボル情報の出力例を以下に示します。

```
*** Symbol List ***

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

SECTION=.text
FILE=sample.obj
                00000000    00000023        24
  _main
                00000000          0         func ,g            0
  _func_01
                00000018          0         func ,g            0
SECTION=.bss
FILE=sample.obj
                fe600004    fe60000b        8
  _gvall
                fe600004          4         data ,g            0
```

項番	説明
(1)	セクション名 セクション名を出力します。
(2)	ファイル名 ファイル名を出力します。

項番	説明
(3)	先頭アドレス (2) のファイルに含まれる該当セクションの先頭アドレスを出力します。 16 進数で表記します。
(4)	最終アドレス (2) のファイルに含まれる該当セクションの最終アドレスを出力します。 16 進数で表記します。
(5)	セクション・サイズ (2) のファイルに含まれる該当セクションのセクション・サイズを出力します（単位：バイト）。 16 進数で表記します。
(6)	シンボル名 シンボル名を出力します。
(7)	シンボル・アドレス シンボル・アドレスを出力します。 16 進数で表記します。
(8)	シンボル・サイズ シンボル・サイズを出力します（単位：バイト）。 16 進数で表記します。
(9)	シンボル種別 データ種別，および宣言種別を出力します。 - データ種別 func ： 関数名 data ： 変数名 entry ： エントリ関数名 none ： 未設定（ラベル，アセンブラ・シンボル） - 宣言種別 g ： 外部定義 w ： WEAK 外部定義 l ： 内部定義
(10)	シンボル参照回数 シンボル参照回数を出力します。 16 進数で表記します。 -show=reference オプションを指定した場合のみ出力します。 参照回数を出力しないときは，“*” を出力します。
(11)	最適化実行の有無 最適化実行の有無を出力します。 ch ： 最適化によって変更されたシンボル cr ： 最適化によって生成されたシンボル mv ： 最適化によって移動されたシンボル

-show=struct を指定した場合は，コンパイル時に -g を指定したファイル内で定義した構造体／共用体メンバの情報も出力します。構造体メンバ情報の出力例を以下に示します。

```
*** Symbol List ***
SECTION=(1)
FILE=(2)                                START     END       SIZE
                                         (3)       (4)       (5)
  SYMBOL                                  ADDR      SIZE      INFO    COUNTS OPT
   (6)                                    (7)       (8)       (9)      (10)  (11)
    STRUCT                                         SIZE
     (12)                                          (13)
      MEMBER                              ADDR      SIZE      INFO
       (14)                               (15)      (16)      (17)
SECTION=.bss
FILE=C:¥Users¥b1501079¥Desktop¥a.obj
                                        00001000 00001007         8
 _st
                                        00001000         8 data ,g      0
   struct {
                                                         8
      _st.mem1
                                        00001000         4 int
      _st.mem2
                                        00001004         2 short
      _st.stmem
                                        00001006         2
   struct {
                                                         2
      _st.stmem.mem3
                                        00001006         1 char
      _st.stmem.mem4
                                        00001007         1 char
}
}
```

項番	説明
(12)	構造体の場合は struct，共用体の場合は union
(13)	構造体または共用体全体のサイズ
(14)	メンバ名をシンボル名に “.”（ドット）で連結して表示
(15)	メンバアドレスを表示
(16)	メンバのサイズを表示
(17)	メンバの型を表示

3.2.7 関数リスト情報

-show=cfi を指定した場合，不正な間接関数呼び出し検出で用いる関数リストの内容を出力します。
出力例を以下に示します。

```
*** CFI Table List ***

SYMBOL/ADDRESS

_func     (1)
0000F100  (2)
```

項番	説明
(1)	関数シンボルを出力します。
(2)	関数シンボルが定義されていない場合，関数アドレスを出力します。

3.2.8 クロス・リファレンス情報

-show=xreference オプションを指定した場合，シンボルの参照情報（クロス・リファレンス情報）を出力します。
クロス・リファレンス情報の出力例を以下に示します。

```
*** Cross Reference List ***

(1)  (2)         (3)             (4)       (5)
No   Unit Name   Global.Symbol   Location  External Information
0001 sample1
     SECTION=.text
                 _main
                                 00000000
                 _func_01
                                 00000018
     SECTION=.data
                 _gval3
                                 fe600000  0003(00000032:.text)
                                           0003(00000038:.text)
     SECTION=.bss
                 _gval1
                                 fe600004  0001(0000001a:.text)
                                           0001(00000020:.text)
                 _gval2
                                 fe600008  0002(00000026:.text)
                                           0002(0000002c:.text)
0002 sample2
     SECTION=.text
                 _func02
                                 00000024  0001(0000000a:.text)
0003 sample3
     SECTION=.text
                 _func03
                                 00000030  0001(00000010:.text)
```

項番	説明
(1)	Unit 番号 オブジェクト単位の識別番号を出力します。
(2)	オブジェクト名 オブジェクト名をリンク時の入力指定順で出力します。

項番	説明
(3)	シンボル名 シンボル名をセクションごとに配置アドレスの昇順で出力します。
(4)	シンボルの配置アドレス シンボルの配置アドレスを出力します。 -form=relocate オプションを指定した場合は，セクションの先頭からの相対値となります。
(5)	参照している外部シンボルのアドレス 参照している外部シンボルのアドレスを以下の形式で出力します。 *Unit 番号 (アドレス，または セクション内オフセット：セクション名)*

3.2.9 CRC 情報

CRC オプション指定時に CRC の演算結果および出力位置アドレスを出力します。
出力例を以下に示します。

```
*** CRC Code ***

CODE: cb0b
(1)
ADDRESS: 00007ffe
(2)
```

項番	説明
(1)	CRC 演算結果。
(2)	CRC の演算結果の出力位置アドレス。

3.3 リンク・マップ・ファイル（オブジェクト結合時）

ここでは，マルチコア用プロジェクトにおいて，オブジェクト結合機能を使用する場合に出力するリンク・マップ・ファイルについて説明します。
リンク・マップ・ファイルを出力することにより，オブジェクト結合の詳細情報を知ることができます。

3.3.1 リンク・マップの構成

リンク・マップの構成と内容を以下に示します。

出力情報	説明
ヘッダ情報	最適化リンカのバージョン情報，およびリンク時刻
オプション情報	コマンド・ライン，サブコマンド・ファイルで指定したオプション列
エラー情報	エラー・メッセージ
エントリ情報	実行開始アドレス
結合アドレス情報	結合元ファイル，および連続範囲データの開始／終了アドレス，サイズ
アドレス重複情報	重複した結合元ファイル，および重複範囲データの開始／終了アドレス，サイズ

3.3.2 ヘッダ情報

バージョン情報，およびリンク時刻を出力します。
ヘッダ情報の出力例を以下に示します。

```
Renesas Optimizing Linker (VX.XX.XX)            XX-Xxx-XXXX XX:XX:XX    (1)
```

項番	説明
(1)	最適化リンカのバージョン情報，およびリンク時刻 最適化リンカのバージョン情報，およびリンク時刻を出力します。

3.3.3 オプション情報

コマンド・ライン，サブコマンド・ファイルで指定したオプション列を出力します。
コマンド・ライン，サブコマンド・ファイルで次のように指定した場合のオプション情報の出力例を以下に示します。

＜コマンド・ライン＞

```
>rlink -subcommand=test.sub -list
```

＜サブコマンド・ファイル test.sub ＞

```
input sample1.mot
input sample2.mot
form stype
output result
```

```
*** Options ***

-subcommand=test.sub    (1)
input sample1.mot       (2)
input sample2.mot       (2)
form stype              (2)
output result           (2)
-list                   (1)
```

項番	説明
(1)	コマンド・ラインで指定したオプション コマンド・ラインで指定したオプションを出力します（指定順）。
(2)	サブコマンド・ファイル内で指定したオプション サブコマンド・ファイル test.sub 内で指定したオプションを出力します。

3.3.4 エラー情報

エラー・メッセージを出力します。
エラー情報の出力例を以下に示します。

```
*** Error information ***
E0562420:"sample1.mot" overlap address "sample2.mot" : "00000100"       (1)
```

項番	説明
(1)	エラー・メッセージ エラー・メッセージを出力します。

3.3.5 エントリ情報

実行開始アドレスを出力します。
エントリ情報の出力例を以下に示します。

```
*** Entry address ***
00000100        (1)
```

項番	説明
(1)	実行開始アドレス 実行開始アドレスを出力します。 ただし，実行開始アドレスが “00000000” の場合は出力しません。

3.3.6 結合アドレス情報

結合元ファイル，および連続範囲データの開始／終了アドレス，サイズを出力します。
結合アドレス情報の出力例を以下に示します。

```
*** Combine information ***
(1)             (2)         (3)         (4)
FILE            START       END         SIZE
sample1.mot
                00000100    00000127    28
sample1.mot
                00000200    00000227    28
sample2.mot
                00000250    00000263    14
sample2.mot
                00000300    0000033b    3c
```

項番	説明
(1)	結合元ファイル名 結合元ファイル名を出力します。
(2)	連続範囲データの開始アドレス 連続範囲データの開始アドレスを出力します。 16 進数で表記します。
(3)	連続範囲データの終了アドレス 連続範囲データの終了アドレスを出力します。 16 進数で表記します。
(4)	連続範囲データのサイズ 連続範囲データのサイズを出力します（単位：バイト）。 16 進数で表記します。

3.3.7 アドレス重複情報

重複した結合元ファイル，および重複範囲データの開始／終了アドレス，サイズを出力します。
アドレス重複情報の出力例を以下に示します。

```
*** Conflict information ***
(1)             (2)         (3)         (4)
FILE            START       END         SIZE
Conflict 1
                00000200    00000213    14
sample1.mot
sample2.mot
```

項番	説明
(1)	重複した結合元ファイル名 重複した結合元ファイル名を出力します。
(2)	重複範囲データの開始アドレス 重複範囲データの開始アドレスを出力します。 16 進数で表記します。
(3)	重複範囲データの終了アドレス 重複範囲データの終了アドレスを出力します。 16 進数で表記します。
(4)	重複範囲データのサイズ 重複範囲データのサイズを出力します（単位：バイト）。 16 進数で表記します。

3.4 ライブラリ・リスト・ファイル

ここでは，ライブラリ・リスト・ファイルについて説明します。
ライブラリ・リストとは，ライブラリ作成結果の情報が書かれたものです。

3.4.1 ライブラリ・リストの構成

ライブラリ・リストの構成と内容を以下に示します。

出力情報	説明	-show オプション指定	-show オプション省略時
オプション情報	コマンド・ライン，サブコマンド・ファイルで指定したオプション列	－	出力する
エラー情報	エラー・メッセージ	－	出力する
ライブラリ情報	ライブラリ情報	－	出力する
ライブラリ内モジュール，セクション，シンボル情報	ライブラリ内モジュール	－	出力する
	モジュール内シンボル名	-show=symbol	出力しない
	各モジュール内セクション名，シンボル名	-show=section	出力しない

注意 -show オプションは，-list オプションを指定した場合に有効となります。
-show オプションについての詳細は，「-SHow」を参照してください。

3.4.2 オプション情報

コマンド・ライン，サブコマンド・ファイルで指定したオプション列を出力します。
コマンド・ライン，サブコマンド・ファイルで次のように指定した場合のオプション情報の出力例を以下に示します。

＜コマンド・ライン＞

```
>rlink -subcommand=test.sub -list -show
```

＜サブコマンド・ファイル test.sub ＞

```
form library
input extmod1
input extmod2
output usrlib.lib
```

```
*** Options ***

-subcommand=test.sub    (1)
form library            (2)
input extmod1           (2)
input extmod2           (2)
output usrlib.lib       (2)
-list                   (1)
-show                   (1)
```

項番	説明
(1)	コマンド・ラインで指定したオプション コマンド・ラインで指定したオプションを出力します（指定順）。

項番	説明
(2)	サブコマンド・ファイル内で指定したオプション サブコマンド・ファイル test.sub 内で指定したオプションを出力します。

3.4.3 エラー情報

エラー，ワーニングなどのメッセージを出力します。
エラー情報の出力例を以下に示します。

```
*** Error Information ***

** E0561200 Backed up file "sample1.lib" into "usrlib.lbk"      (1)
```

項番	説明
(1)	メッセージ メッセージを出力します。

3.4.4 ライブラリ情報

ライブラリの種別を出力します。
ライブラリ情報の出力例を以下に示します。

```
*** Library Information ***

LIBRARY NAME=usrlib.lib (1)
CPU=RH850               (2)
ENDIAN=Little           (3)
ATTRIBUTE=user          (4)
NUMBER OF MODULE=2      (5)
```

項番	説明
(1)	ライブラリ名 ライブラリ名を出力します。
(2)	マイコン名 マイコン名を出力します。
(3)	エンディアン種別 エンディアン種別を出力します。
(4)	ライブラリ・ファイルの属性 システム・ライブラリであるか，ユーザ・ライブラリであるかを出力します。
(5)	ライブラリ内モジュール数 ライブラリ内モジュール数を出力します。

3.4.5 ライブラリ内モジュール，セクション，シンボル情報

ライブラリ内のモジュールを出力します。
-show=symbol オプションを指定した場合は，モジュール内シンボル名を出力します。
また，-show=section オプションを指定した場合は，モジュール内セクション名も出力します。
ライブラリ内モジュール，セクション，シンボル情報の出力例を以下に示します。

```
*** Library List ***

(1)             (2)
MODULE          LAST UPDATE
  (3)
  SECTION
    (4)
    SYMBOL
extmod1
                12-Dec-2011 16:30:00
  .text
    _func_01
    _func_02
extmod2
                12-Dec-2011 16:30:10
  .text
    _func_03
    _func_04
```

項番	説明
(1)	モジュール名 モジュール名を出力します。
(2)	モジュールを登録した日付 モジュールを登録した日付を出力します。 モジュールが更新された場合は，最新の更新日付を出力します。
(3)	モジュール内セクション名 モジュール内セクション名を出力します。
(4)	セクション内シンボル名 セクション内シンボル名を出力します。

3.5 インテル拡張ヘキサ・ファイル

ここでは，インテル拡張ヘキサ・ファイルについて説明します。

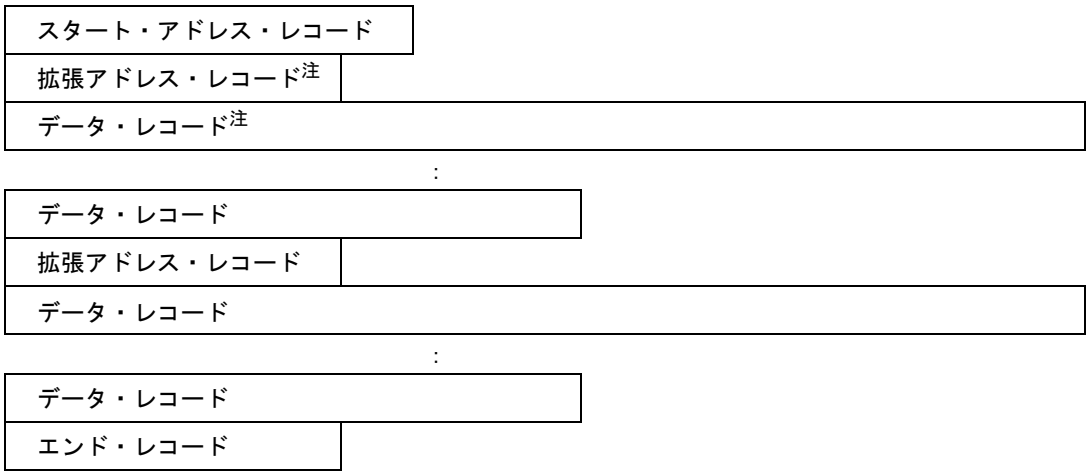
3.5.1 インテル拡張ヘキサ・ファイルの構成

インテル拡張ヘキサ・ファイル（20 ビット）は，スタート・アドレス・レコード，拡張アドレス・レコード，データ・レコード，およびエンド・レコードの 4 種類のレコード[注]により構成されます。
インテル拡張ヘキサ・ファイル（32 ビット）は，スタート・リニア・アドレス・レコード，拡張リニア・スタート・アドレス・レコード，スタート・アドレス・レコード，拡張アドレス・レコード，データ・レコード，およびエンド・レコードの 6 種類のレコード[注]により構成されます。

注 各レコードは，ASCII コードで出力します。

インテル拡張ヘキサ・ファイルの構成と内容を以下に示します。

図 3.1 インテル拡張ヘキサ・ファイルの構成



注 拡張アドレス・レコード，およびデータ・レコードは繰り返されます。

出力情報	説明
スタート・リニア・アドレス・レコード	リニア・アドレス
拡張リニア・アドレス・レコード	ビット 32 ～ビット 16 の上位 16 ビット・アドレス
スタート・アドレス・レコード	エントリ・ポイント・アドレス
拡張アドレス・レコード	ロード・アドレスのパラグラフ値
データ・レコード	コードの値
エンド・レコード	コードの終わり

各レコードは，各種フィールドにより以下の形で構成されます。

```
:   XX  XXXX  XX  DD......DD SS  NL
(1) (2) (3)   (4)     (5)    (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	バイト数 (5) の 2 桁ずつの 16 進数で表されるバイトのバイト数です。

項番	説明
(3)	ロケーション・アドレス
(4)	レコード・タイプ 05 : スタート・リニア・アドレス・レコード 04 : 拡張リニア・アドレス・レコード 03 : スタート・アドレス・レコード 02 : 拡張アドレス・レコード 00 : データ・レコード 01 : エンド・レコード
(5)	コード コードの 1 バイトごとを 2 桁の 16 進数で表したものです。
(6)	チェック・サム :，SS，NL を除くレコード内の各バイト値を 16 進数で加算した結果の 2 の補数です（2 桁）。
(7)	ニュー・ライン（¥n）

備考 インテル・ヘキサ・フォーマットのロケーション・アドレスは 2 バイト（16 ビット）です。
したがって，64K の空間しか直接指定はできません。
それを拡張するために，16 ビットの拡張アドレスを追加して 1M（20 ビット）の空間まで扱えるようにしたのがインテル拡張ヘキサ・フォーマットです。
具体的には，16 ビットの拡張アドレスを指定するレコード・タイプを追加しています。
この追加した拡張アドレスの 4 ビットをシフトしてロケーション・アドレスと加算することで，20 ビットのアドレスを表現できるようになっています。

3.5.2 スタート・リニア・アドレス・レコード

リニア・アドレスを示します。

```
:   04  0000  05  XXXXXXXX  SS  NL
(1) (2) (3)   (4)    (5)    (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	04 固定
(3)	0000 固定
(4)	レコード・タイプ（05 固定）
(5)	リニア・アドレス値
(6)	チェック・サム
(7)	ニュー・ライン

3.5.3 拡張リニア・アドレス・レコード

ビット 32 ～ビット 16 の上位 16 ビット・アドレスを示します。

```
:   02  0000  04  XXXX  SS  NL
(1) (2) (3)   (4)  (5)  (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	02 固定

項番	説明
(3)	0000 固定
(4)	レコード・タイプ（04 固定）
(5)	ビット 32 ～ビット 16 の上位 16 ビット・アドレス値
(6)	チェック・サム
(7)	ニュー・ライン

注 下位 16 ビットは，データ・レコードのロケーション・アドレスを用います。

3.5.4 スタート・アドレス・レコード

エントリ・ポイント・アドレスを示します。

```
:   04  0000  03  PPPP  XXXX  SS  NL
(1) (2) (3)   (4)  (5)   (6)  (7) (8)
```

項番	説明
(1)	レコード・マーク
(2)	04 固定
(3)	0000 固定
(4)	レコード・タイプ（03 固定）
(5)	エントリ・ポイント・アドレスのパラグラフ値[注]
(6)	エントリ・ポイント・アドレスのオフセット値
(7)	チェック・サム
(8)	ニュー・ライン

注 アドレスは（パラグラフ値 <<4）＋オフセット値で求められます。

3.5.5 拡張アドレス・レコード

ロード・アドレスのパラグラフ値を示します[注]。

注 （データ・レコードを出力する際）セグメントの先頭で，またはデータ・レコードのロード・アドレスのオフセット値が最大値 0xffff を越えてセグメントが新しくなる際，出力します。

```
:   02  0000  02  PPPP  SS  NL
(1) (2) (3)   (4)  (5)  (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	02 固定
(3)	0000 固定
(4)	レコード・タイプ（02 固定）
(5)	セグメントのパラグラフ値
(6)	チェック・サム

項番	説明
(7)	ニュー・ライン

3.5.6 データ・レコード

コードの値を示します。

```
:   XX  XXXX  00  DD......DD  SS  NL
(1) (2) (3)   (4)     (5)     (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	バイト数[注]
(3)	ロケーション・アドレス
(4)	レコード・タイプ（00 固定）
(5)	コード コードの 1 バイトごとを 2 桁の 16 進数で表したものです。
(6)	チェック・サム
(7)	ニュー・ライン

注 0x1 ～ 0xff の範囲に限られます（1 つのデータ・レコードで示されるコードのバイト数の最小値は 1 で最大値は 255 です）。

例

```
:   04  0100  00  3C58E01B  6C  NL
(1) (2) (3)   (4)    (5)    (6) (7)
```

項番	説明
(1)	レコード・マーク
(2)	3C58E01B の 2 桁ずつの 16 進数で表されるバイトのバイト数
(3)	ロケーション・アドレス
(4)	レコード・タイプ 00
(5)	コードの 1 バイトごとを 2 桁の 16 進数で表したもの
(6)	チェック・サム 04 ＋ 01 ＋ 00 ＋ 00 ＋ 3C ＋ 58 ＋ E0 ＋ 1B ＝ 194 の 2 の補数 E6C の下位 1 バイトを 2 桁の 16 進数で表したもの
(7)	ニュー・ライン（¥n）

3.5.7 エンド・レコード

コードの終わりを示します。

```
:   00  0000  01  FF  NL
(1) (2) (3)   (4) (5) (6)
```

項番	説明
(1)	レコード・マーク
(2)	00 固定
(3)	0000 固定
(4)	レコード・タイプ（01 固定）
(5)	FF 固定
(6)	ニュー・ライン

3.6 モトローラ・S タイプ・ファイル

ここでは，モトローラ・S タイプ・ファイルについて説明します。

3.6.1 モトローラ・S タイプ・ファイルの構成

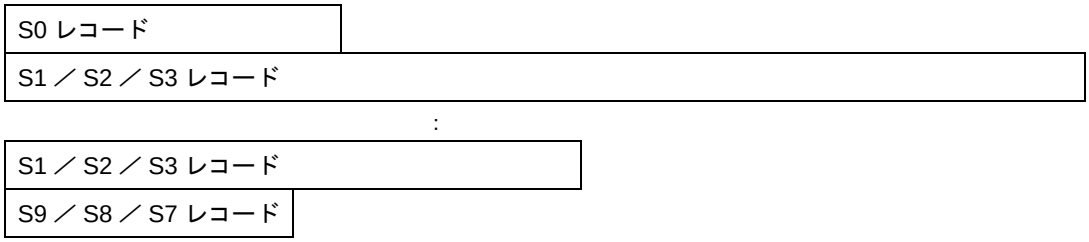
モトローラ・S タイプ・ファイルは，ヘッダ・レコードである S0 レコード，データ・レコードである S1 ／ S2 ／ S3 レコード，エンド・レコードである S9 ／ S8 ／ S7 レコードの 7 種類のレコード[注 1] により構成されます[注 2]。

注 1. 各レコードは，ASCII コードで出力します。

注 2. モトローラ・S タイプ・ファイルには，16 ビット・アドレスのもの，（24 ビット）スタンダード・アドレスのもの，32 ビット・アドレスのものが存在し，16 ビット・アドレスのフォーマットは S0，S1，および S9 レコード，スタンダード・アドレスのフォーマットは S0，S2，および S8 レコード，32 ビット・アドレスのフォーマットは S0，S3，および S7 レコードによって構成されます。

モトローラ・S タイプ・ファイルの構成と内容を以下に示します。

図 3.2 モトローラ・S タイプ・ファイルの構成



出力情報	説明
S0 レコード	ファイル名
S1 レコード	コードの値
S2 レコード	コードの値
S3 レコード	コードの値
S7 レコード	エントリ・ポイント・アドレス
S8 レコード	エントリ・ポイント・アドレス
S9 レコード	エントリ・ポイント・アドレス

各レコードは，各種フィールドにより以下の形で構成されます。

```
Sx  XX  YY......YY  SS  NL
(1) (2)    (3)     (4) (5)
```

項番	説明
(1)	レコード・タイプ S0 : S0 レコード S1 : S1 レコード S2 : S2 レコード S3 : S3 レコード S4 : S4 レコード S5 : S5 レコード S6 : S6 レコード S7 : S7 レコード S8 : S8 レコード S9 : S9 レコード

項番	説明
(2)	レコード長 (3) の 2 桁ずつの 16 進数で表されるバイトのバイト数＋ SS で表されるバイト数[注]
(3)	フィールド
(4)	チェック・サム Sx，SS，NL を除くレコード内の 2 桁ずつの 16 進数で表されるバイトの値を合計したものの 1 の補数を取り，その下位 1 バイトを 2 桁の 16 進数で表したもの
(5)	ニュー・ライン（¥n）

注 1 です。

3.6.2 S0 レコード

ファイル名を示します。

```
S0  0E  0000  XX......XX  SS  NL
(1) (2)  (3)      (4)     (5) (6)
```

項番	説明
(1)	S0 固定
(2)	0E 固定
(3)	0000 固定
(4)	主にファイル名（8 文字）＋ファイル形式（3 文字）
(5)	チェック・サム
(6)	ニュー・ライン

3.6.3 S1 レコード

コードの値を示します。

```
S1  XX  YYYY  ZZ......ZZ  SS  NL
(1) (2)  (3)      (4)     (5) (6)
```

項番	説明
(1)	S1 固定
(2)	レコード長
(3)	ロード・アドレス 16 ビット（0x0 ～ 0xFFFF）
(4)	コード コードの 1 バイトごとを 2 桁の 16 進数で表したもの
(5)	チェック・サム
(6)	ニュー・ライン

3.6.4 S2 レコード

コードの値を示します。

```
S2  XX  YYYYYY  ZZ......ZZ  SS  NL
(1) (2)  (3)       (4)     (5) (6)
```

項番	説明
(1)	S2 固定
(2)	レコード長
(3)	ロード・アドレス 24 ビット（0x0 ～ 0xFFFFFF）
(4)	コード コードの 1 バイトごとを 2 桁の 16 進数で表したもの
(5)	チェック・サム
(6)	ニュー・ライン

3.6.5 S3 レコード

コードの値を示します。

```
S3  XX  YYYYYYYY  ZZ......ZZ  SS  NL
(1) (2)  (3)         (4)     (5) (6)
```

項番	説明
(1)	S3 固定
(2)	レコード長
(3)	ロード・アドレス 32 ビット（0x0 ～ 0xFFFFFFFF）
(4)	コード コードの 1 バイトごとを 2 桁の 16 進数で表したもの
(5)	チェック・サム
(6)	ニュー・ライン

3.6.6 S7 レコード

エントリ・ポイント・アドレスを示します。

```
S7  XX  YYYYYYYY  SS  NL
(1) (2)  (3)     (4) (5)
```

項番	説明
(1)	S7 固定
(2)	レコード長
(3)	エントリ・ポイント・アドレス 32 ビット（0x0 ～ 0xFFFFFFFF）
(4)	チェック・サム
(5)	ニュー・ライン

3.6.7 S8 レコード

エントリ・ポイント・アドレスを示します。

```
S8  XX  YYYYYY  SS  NL
(1) (2)   (3)   (4) (5)
```

項番	説明
(1)	S8 固定
(2)	レコード長
(3)	エントリ・ポイント・アドレス 24 ビット（0x0 ～ 0xFFFFFF）
(4)	チェック・サム
(5)	ニュー・ライン

3.6.8 S9 レコード

エントリ・ポイント・アドレスを示します。

```
S9  XX  YYYY  SS  NL
(1) (2)  (3)  (4) (5)
```

項番	説明
(1)	S9 固定
(2)	レコード長
(3)	エントリ・ポイント・アドレス 16 ビット（0x0 ～ 0xFFFF）
(4)	チェック・サム
(5)	ニュー・ライン

4. コンパイラ言語仕様

この章では，CC-RH がサポートするコンパイラ言語仕様（基本言語仕様，拡張言語仕様など）について説明します。

4.1 基本言語仕様

本節では，C90 および C99 規格に対する CC-RH での処理系固有の定義について説明します。
なお，CC-RH で独自に追加されている拡張言語仕様については，「4.2 拡張言語仕様」を参照してください。

4.1.1 C90 の処理系定義

(1) どのような方法で診断メッセージを識別するか（5.1.1.3）。

「10. メッセージ」を参照してください。

(2) main 関数への実引数の意味（5.1.2.2.1）。

フリースタンディング環境であるため，規定しません。

(3) 対話型装置がどのようなもので構成されるか（5.1.2.3）。

対話型装置の構成については，特に規定しません。

(4) 外部給合でない識別子において（31 以上の）意味がある先頭の文字数（6.1.2）。

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。

(5) 外部結合である識別子において（6 以上の）意味がある先頭の文字数（6.1.2）。

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。

(6) 外部結合である識別子において英小文字と英大文字の区別に意味があるか否か（6.1.2）。

識別子内の英小文字と英大文字を区別します。

(7) ソース及び実行文字集合の要素で，この規格で明示的に規定しているもの以外の要素（5.2.1）。

ソースおよび実行文字集合の要素の値は，ASCII コード，EUC，SJIS，UTF-8，big5，gb2312 です。
コメントと文字列における日本語／中国語記述をサポートしています。

(8) 多バイト文字のコード化のために使用されるシフト状態（5.2.1.2）。

シフト状態はサポートしていません。

(9) 実行文字集合の文字におけるビット数（5.2.4.2.1）。

8 ビットとします。

(10) （文字定数内及び文字列リテラル内の）ソース文字集合の要素と実行文字集合の要素との対応付け（6.1.3.4）。

同一の値をもつ要素へ対応付けます。

(11) 基本実行文字集合で表現できない文字若しくは拡張表記を含む単純文字定数の値，又はワイド文字定数に対しては拡張文字集合で表現できない文字若しくは拡張表記を含むワイド文字定数の値（6.1.3.4）。

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。

拡張表記	値（ASCII）
¥a	0x07
¥b	0x08
¥f	0x0C
¥n	0x0A
¥r	0x0D
¥t	0x09
¥v	0x0B

(12) 2 文字以上の文字を含む単純文字定数又は 2 文字以上の多バイト文字を含むワイド文字定数の値（6.1.3.4）。

4 文字までの文字を含む単純文字定数は，末尾の文字を下位バイト，先頭の文字を上位バイトに持つ 4 バイトの値を持ちます。5 文字以上の文字を持つ文字定数はエラーとなります。基本的な実行環境文字集合で表現されない文字は，その値を持つ単純文字定数とみなします。不正な逆斜線表記は逆斜線を無視して次の文字を単純文字定数とみなします。

(13) ワイド文字定数に対して，多バイト文字を対応するワイド文字（コード）に変換するために使用されるロケール（6.1.3.4）。

ロケールはサポートしていません。

(14) " 単なる "char が signed char と同じ値の範囲をもつか，unsigned char と同じ値の範囲をもつか（6.2.1.1）。

char 型は，signed char 型と同じ値の範囲，同じ表現形式，同じ動作を持ちます。

(15) 整数の様々な型の表現方法及び値の集合（6.1.2.5）。

「4.1.3 データの内部表現と領域」を参照してください。

(16) 整数をより短い符号付き整数に変換した結果，又は符号無し整数を長さの等しい符号付き整数に変換した結果で，値が表現できない場合の変換結果（6.2.1.2）。

変換先の型の幅でマスクした（上位ビットを切り捨てた）ビット列とします。

(17) 符号付き整数に対してビット単位の演算を行った結果（6.3）。

シフト演算子の場合は算術シフトを行います。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。

(18) 整数除算における剰余の符号（6.3.5）。

“%” 演算子の結果の符号は第 1 オペランドの符号とします。

(19) 負の値をもつ符号付き汎整数型の右シフトの結果（6.3.7）。

算術シフトを行います。

(20) 浮動小数点数の様々な型の表現方法及び値の集合（6.1.2.5）。

「4.1.3 データの内部表現と領域」を参照してください。

(21) 汎整数の値を元の値に正確に表現することができない浮動小数点数に変換する場合の切捨ての方向（6.2.1.3）。

オプション -Xround の指定，およびマイコンの設定に従います。

(22) 浮動小数点数をより狭い浮動小数点数に変換する場合の切捨て又は丸めの方向（6.2.1.4）。

オプション -Xround の指定，およびマイコンの設定に従います。

(23) 配列の大きさの最大値を保持するために必要な整数の型。すなわち sizeof 演算子の型 size_t（6.3.3.4, 7.1.1）。

unsigned long 型です。

(24) ポインタを整数型へキャストした結果，及びその逆の場合の結果（6.3.4）。

整数からポインタへの変換結果
整数型のサイズがポインタ型のサイズより大きい場合は，整数型の下位バイトの値になります。整数型のサイズとポインタ型のサイズが等しい場合は，整数型のビットパターンがそのまま保持されます。整数型のサイズがポインタ型のサイズより小さい場合は，int 型に拡張した結果の値をそのまま保持します。

ポインタから整数への変換結果
ポインタ型のサイズが整数型のサイズより大きい場合は，ポインタ型の下位バイトの値になります。ポインタ型のサイズと整数型のサイズが等しい場合は，ポインタ型のビットパターンがそのまま保持されます。ポインタ型のサイズが整数型のサイズより小さい場合は，ポインタ型の値をゼロ拡張した値になります。

(25) 同じ配列内の，二つの要素へのポインタ間の差を保持するために必要な整数の型，すなわち ptrdiff_t の型（6.3.4，7.1.1）。

signed int 型です。

(26) register 記憶域クラス指定子を使用することによって実際にオブジェクトをレジスタに置くことができる範囲（6.5.1）。

register 指定子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。

(27) 共用体オブジェクトのメンバを異なる型のメンバを用いてアクセスする場合（6.3.2.3）。

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，整列条件にしたがって格納されるため，共用体オブジェクトのメンバを異なる型のメンバを用いてアクセスする場合，データの内部表現はアクセスする型に従います。

(28) 構造体のメンバの詰め物及び境界調整（6.5.2.1）。

「4.1.3 データの内部表現と領域」を参照してください。

(29) " 単なる "int 型のビットフィールドが，signed int のビットフィールドとして扱われるか，unsigned int のビットフィールドとして扱われるか（6.5.2.1）。

signed int 型として扱います。最上位ビットは符号ビットとして扱います。

(30) 単位内のビットフィールドの割付け順序（6.5.2.1）。

下位から割り付けます。オプション -Xbit_order または #pragma bit_order で選択が可能です。

(31) ビットフィールドを記憶域単位の境界にまたがって割り付けうるか否か（6.5.2.1）。

境界を跨がず，次の領域に割り付けます。

(32) 列挙型の値を表現するために選択される整数型（6.5.2.2）。

signed int 型です。ただし，オプション -Xenum_type=auto 指定時は，列挙値が収まる最小の型となります。

(33) volatile 修飾型のオブジェクトへのアクセスをどのように構成するか（6.5.3）。

アクセス幅，アクセス順序，アクセス回数は C ソース上の記述通りに実施しますが，対応するマイコンの命令がない型へのアクセスは，その限りではありません。

(34) 算術型 , 構造体型又は共用体型を修飾する宣言子の最大数（6.5.4）。

128 です。

(35) switch 文における case 値の最大数（6.6.4.2）。

2147483647 です。

(36) 条件付き取込みを制御する定数式中の単一文字からなる文字定数の値が実行文字集合中の同じ文字定数の値に一致するか否か。このような文字定数が負の値をもつことがあるか否か（6.8.1）。

条件付き取り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。
負の値を持つことがあります。

(37) 取込み可能なソース・ファイルを捜すための方法（6.8.2）。

次の順序で探索し，フォルダにある同名ファイルをヘッダと識別します。
1. フルパス指定の場合はそのパスが示すフォルダ
2. オプション -I で指定されたフォルダ
3. 標準インクルードファイルフォルダ

(38) 取込み可能なソース・ファイルに対する " で囲まれた名前の探索（6.8.2）。

次の順序で探索します。
1. フルパス指定の場合はそのパスが示すフォルダ
フルパス指定がない場合は
2. ソース・ファイルがあるフォルダ
3. インクルードオプションで指定されたフォルダ
4. 標準インクルードファイルフォルダ

(39) ソース・ファイル名と文字列との対応付け（6.8.2）。

#include に記述された文字列は，ソース文字集合として指定した文字コードとして解釈され，ヘッダ名または外部ソース・ファイル名に対応付けられます。

(40) 認識される #pragma 指令の動作（6.8.6）。

「4.2.6 拡張言語仕様の使用方法」を参照してください。

(41) 翻訳日付及び翻訳時刻がそれぞれ有効でない場合における __DATE 及び TIME__ の定義（6.8.8）。

日付や時刻が得られない場合はありません。

(42) マクロ NULL が展開する空ポインタ定数（7.1.6）。

(void*)0 とします。

(43) assert 関数によって表示される診断メッセージ及び assert 関数の終了時の動作（7.2）。

以下の通りです。
　Assertion failed：式，file ファイル名，line 行番号
終了時の動作は，abort 関数の実装に依存します。

(44) isalnum 関数，isalpha 関数，iscntrl 関数，islower 関数，isprint 関数及び isupper 関数によってテストされる文字集合（7.3.1）。

unsigned char 型（0 ～ 255）および EOF（-1）です。

(45) 数学関数に定義域エラーが発生した場合に返される値（7.5.1）。

「7.4.10 数学関数」を参照してください。

(46) アンダフロー値域エラーの場合に，数学関数がマクロ ERANGE の値を整数式 errno に設定するか否か（7.5.1）。

アンダーフロー発生時は errno に ERANGE を設定します。

(47) fmod 関数の第 2 引数が 0 の場合に，定義域エラーが発生するか又は 0 が返されるか（7.5.6.4）。

定義域エラーが発生します。詳細は fmod 関数群の説明を参照してください。

(48) signal 関数に対するシグナルの集合（7.7.1.1）。

シグナル操作関数はサポートしていません。

(49) signal 関数によって認識されるシグナルの意味（7.7.1.1）。

シグナル操作関数はサポートしていません。

(50) signal 関数によって認識されるシグナルに対する既定の処理及びプログラム開始時の処理（7.7.1.1）。

シグナル操作関数はサポートしていません。

(51) シグナル処理ルーチンの呼出しの前に signal(sig, SIG_DFL); と同等のことが実行されない場合のシグナルの遮断の処置（7.7.1.1）。

シグナル操作関数はサポートしていません。

(52) シグナル関数によって指定された処理ルーチンによって SIGILL シグナルが受け付けられる場合に既定の処理が再設定されるか否か（7.7.1.1）。

シグナル操作関数はサポートしていません。

(53) テキストストリームの最終行が，終了を示す改行文字を必要とするか否か（7.9.2）。

改行文字を必要としません。

(54) データが読み取られる時，改行文字の直前にテキストストリームに書き込まれた空白文字の並びが現れるか否か（7.9.2）。

データが読み取られる時に現れます。

(55) バイナリストリームに書き込むデータに付加してもよいナル文字の個数（7.9.2）。

0 個です。

(56) 追加モードのストリームのファイル位置表示子が，最初にファイルの先頭又は終わりのどちらに位置付けされるか（7.9.3）。

ファイル操作関数はサポートしていません。

(57) テキストストリームへの書込みが，結び付けられたファイルを最終書込み点の直後で切り捨てるか否か（7.9.3）。

ファイル操作関数はサポートしていません。

(58) ファイルバッファリングの特性（7.9.3）。

ファイル操作関数はサポートしていません。

(59) 長さ 0 のファイルが実際に存在するか否か（7.9.3）。

ファイル操作関数はサポートしていません。

(60) 正しいファイル名の規則（7.9.3）。

ファイル操作関数はサポートしていません。

(61) 同一ファイルを複数回オープンすることが可能か否か（7.9.3）。

ファイル操作関数はサポートしていません。

(62) オープンされているファイルに対する remove 関数の効果（7.9.4.1）。

ファイル操作関数はサポートしていません。

(63) rename 関数呼出し前に新しい名前をもつファイルが存在している場合の効果（7.9.4.2）。

ファイル操作関数はサポートしていません。

(64) fprintf 関数中の %p 変換による出力（7.9.6.1）。

10 進表記です。

(65) fscanf 関数中の %p 変換に対する入力（7.9.6.2）。

10 進整数です。

(66) fscanf 関数中の %[変換において文字 - が走査文字の並び中の最初の文字でも最後の文字でもない場合の解釈（7.9.6.2）。

「7.4.7 標準入出力関数」にある「sscanf」を参照してください。

(67) fgetpos 関数又は ftell 関数が失敗した場合に，マクロ errno に設定される値（7.9.9.1，7.9.9.4）。

ファイル操作関数はサポートしていません。

(68) perror 関数によって生成されるメッセージ（7.9.10.4）。

「7.4.7 標準入出力関数 perror」を参照してください。

(69) 要求された大きさが 0 の場合の calloc 関数，malloc 関数又は realloc 関数の動作（7.10.3）。

calloc 関数，malloc 関数は要求された大きさを 8 とみなし，確保したポインタを返します。realloc 関数は NULL を返します。

(70) オープンされているファイル及び一時ファイルに関しての abort 関数の動作（7.10.4.1）。

ファイル操作関数はサポートしていません。

(71) exit 関数の実引数の値が 0，EXIT_SUCCESS 又は EXIT FAILURE 以外の場合に返される状態（7.10.4.3）。

フリースタンディング環境であるため，規定しません。

(72) getenv 関数によって使われる環境の名前の集合及び環境の並びを変更する方法（7.10.4.4）。

getenv 関数はサポートしていません。

(73) system 関数による文字列の実行のモード及び内容（7.10.4.5）。

system 関数はサポートしていません。

(74) strerror 関数によって返されるエラーメッセージ文字列の内容（7.11.6.2）。

「7.4.3 文字列関数 strerror 関数」を参照してください。

(75) 地方時及び夏時間（7.12.1）。

time.h はサポートしていません。

(76) clock 関数のための時点（7.12.2.1）。

time.h はサポートしていません。

C90 規格の翻訳限界に対する CC-RH の仕様を示します。
「制限なし」となっている項目は，コンパイル環境の状態に依存します。

項目	限界値
条件付き取込みにおける入れ子のレベル数	制限なし
一つの宣言中の一つの算術型 , 構造体型，共用体型又は不完全型を修飾するポインタ，配列及び関数宣言子（の任意の組合せ）の個数	128
一つの完全宣言子における括弧で囲まれた宣言子の入れ子のレベル数	制限なし
一つの完全式における括弧で囲まれた式の入れ子のレベル数	制限なし
内部識別子又はマクロ名において意味がある先頭の文字数	制限なし
外部識別子において意味がある先頭の文字数	制限なし
一つの翻訳単位中における外部識別子数	制限なし
一つのブロック中におけるブロック有効範囲をもつ識別子数	制限なし
一つの翻訳単位中で同時に定義されうるマクロ識別子数	制限なし
一つの関数定義における仮引数の個数	制限なし
一つの関数呼出しにおける実引数の個数	制限なし
一つのマクロ定義における仮引数の個数	制限なし
一つのマクロ呼出しにおける実引数の個数	制限なし
一つの論理ソース行における文字数	制限なし
（連結後の）単純文字列リテラル又はワイド文字列リテラル中における文字数	制限なし
（ホスト環境の場合）一つのオブジェクトのバイト数	2147483647
#include で取り込まれるファイルの入れ子のレベル数	制限なし
一つの switch 文（入れ子になった switch 文を除く）中における case 名札の個数	2147483647
一つの構造体又は共用体のメンバ数	制限なし
一つの列挙体における列挙定数の個数	制限なし
一つのメンバ宣言並びにおける構造体又は共用体定義の入れ子のレベル数	制限なし

4.1.2 C99 の処理系定義

(1) 翻訳時の構文規則違反等に対する診断メッセージの出し方。（3.10，5.1.1.3）

「10. メッセージ」を参照してください。

(2) 翻訳フェーズ 3 において，改行文字を除く空白類文字の並びを保持するか一つの空白文字に置き換えるか。（5.1.1.2）

そのまま保持されます。

(3) 翻訳フェーズ 1 での，物理的なソース・ファイルの多バイト文字と対応するソース文字集合のマッピング方法。（5.1.1.2）

多バイト文字は，コンパイル・オプションにより対応するソース文字集合にマッピングします。

(4) フリースタンディング環境におけるプログラム開始時に呼び出される関数の名前と型。（5.1.2.1）

規定しません。スタート・アップの実装に依存します。

(5) 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。（5.1.2.1）

正常終了時はスタート・アップに依存します。プログラムを異常終了させる場合は abort 関数を使用します。

(6) main 関数を定義できる代替方法。（5.1.2.2.1）

フリースタンディング環境であるため，規定しません。

(7) main 関数の argv 実引数が指す文字列の値。（5.1.2.2.1）

フリースタンディング環境であるため，規定しません。

(8) ユーザーインタフェースとなる対話型装置がどのようなものであるか。（5.1.2.3）

対話型装置の構成については，特に規定しません。

(9) シグナル全体の集合，それぞれの意味および既定の操作。（7.14）

シグナル操作関数はサポートしていません。

(10) SIGFPE，SIGILL，SIGSEGV 以外の，計算例外に対応するシグナルの値。（7.14.1.1）

シグナル操作関数はサポートしていません。

(11) プログラム開始時に実行される，signal(sig, SIG_IGN); と同等なシグナル。（7.14.1.1）

シグナル操作関数はサポートしていません。

(12) getenv 関数で使用される，環境の並びに定義されている名前の集合および環境の並びを変更する方法。（7.20.4.5）

getenv 関数はサポートしていません。

(13) system 関数における，引数 string が指す文字列の実行方法。（7.20.4.6）

system 関数はサポートしていません。

(14) ソース基本文字集合の一部でない多バイト文字が識別子の中に現れることを許すかどうか，及び識別子に使用してよい多バイト文字およびその文字と国際文字名との対応。（6.4.2）

識別子として多バイト文字は使用できません。

(15) 識別子における，意味がある先頭の文字の個数。（5.2.4.1，6.4.2）

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。

(16) 1 バイトあたりのビット数。（3.6）

8 ビットとします。

(17) 実行文字集合の要素の値。（5.2.1）

実行文字集合の要素の値は，ASCII コード，EUC，SJIS，UTF-8，big5，gb2312 の値です。

(18) 標準の英字逆斜線表記のそれぞれに割り当てられた実行文字集合の要素の一意な値。（5.2.2）

逆斜線表記	値（ASCII）
¥a	0x07
¥b	0x08
¥f	0x0C
¥n	0x0A
¥r	0x0D
¥t	0x09
¥v	0x0B

(19) 実行基本文字集合の任意の要素以外の文字が格納された char 型のオブジェクトの値。（6.2.5）

char 型に型変換した値となります。

(20) signed char と unsigned char のどちらが，単なる char と同じ値の範囲，同じ表現形式，同じ動作を持つのか。（6.2.5，6.3.1.1）

char 型は，signed char 型と同じ値の範囲，同じ表現形式，同じ動作を持ちます。

(21) 文字定数と文字列リテラル中のソース文字集合の要素から実行文字集合の要素への対応付け方法。（6.4.4.4，5.1.1.2）

同一の値をもつ要素へ対応付けます。

(22) 2 文字以上を含む，または 1 バイトの実行文字で表現できない文字もしくは逆斜線表記を含む単純文字定数の値。（6.4.4.4）

4 文字までの文字を含む単純文字定数は，末尾の文字を下位バイト，先頭の文字を上位バイトに持つ 4 バイトの値を持ちます。5 文字以上の文字を持つ文字定数はエラーとなります。基本的な実行環境文字集合で表現されない文字は，その値を持つ単純文字定数とみなします。不正な逆斜線表記は逆斜線を無視して次の文字を単純文字定数とみなします。

(23) 2 文字以上の多バイト文字を含む，または実行拡張文字集合で表現できない多バイト文字もしくは逆斜線表記を含むワイド文字定数の値。（6.4.4.4）

多バイト文字としての左端 1 文字の値となります。

(24) 実行拡張文字集合の 1 つの文字に対応する単一の多バイト文字を含むワイド文字定数の値をワイド文字に対応させる，その時点のロケール。（6.4.4.4）

ロケールはサポートしていません。

(25) ワイド文字列リテラルから対応するワイド文字コードへの変換時に使用される，その時点のロケール。（6.4.5）

ロケールはサポートしていません。

(26) 実行文字集合で表現できない多バイト文字もしくは逆斜線表記を含む文字列リテラルの値。（6.4.5）

逆斜線表記は対応するバイトの値，多バイト文字はそれぞれのバイトの値になります。

(27) 処理系が提供する拡張整数型。（6.2.5）

拡張整数型は提供していません。

(28) 符号付き整数型が符号と大きさ，2 の補数，または 1 の補数を使用して表現されるかどうか，および異常値がトラップ表現か通常値かどうか。（6.2.6.2）

符号付き整数型は 2 の補数で表現します。トラップ表現はありません。

(29) 同じ精度を持つ別の拡張整数型に対する，整数拡張型の順位。（6.3.1.1）

拡張整数型は提供していません。

(30) 整数型の値を符号付き整数型に変換する際，値が変換先の型で表現できない場合の結果，あるいは生成されるシグナル。（6.3.1.3）

変換先の型の幅でマスクした（上位ビットを切り捨てた）ビット列とします。

(31) 符号付き整数に対するビット単位の操作の結果。（6.5）

シフト演算子の場合は算術シフトを行います。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。

(32) 浮動小数点演算の正確度，<math.h> および <complex.h> の中で定義される，浮動小数点型の結果を返却する，ライブラリ関数の正確度。（5.2.4.2.2）

不明です。

(33) FLT_ROUNDS に対する非標準の値において，特徴付けられた丸め動作。（5.2.4.2.2）

FLT_ROUNDS に対する非標準の値は定義しません。

(34) FLT_EVAL_METHOD に対する非標準の負の値によって特徴付けられる評価形式。（5.2.4.2.2）

FLT_EVAL_METHOD に対する非標準の値は定義しません。

(35) 整数が，元の値を正確に表現できない浮動小数点数に変換された時の丸めの方向。（6.3.1.4）

オプション -Xround の指定，およびマイコンの設定に従います。

(36) 浮動小数点数がより狭い浮動小数点数型に変換された時の丸めの方向。（6.3.1.5）

オプション -Xround の指定，およびマイコンの設定に従います。

(37) 浮動小数点定数を，最も近い表現可能な値とするか，それとも最も近い表現可能な値のすぐ隣（大きい値もしくは小さい値）で表現可能な値とするか。（6.4.4.2）

オプション -Xround の指定に従います。

(38) FP_CONTRACT プラグマがない場合，式が短縮されるかどうか。短縮されるならどのように短縮されるか。（6.5）

式の短縮は，各オプション指定に依存します。
FP_CONTRACT プラグマは機能しません。
#pragma STDC FP_CONTRACT 指定をしても無視します。

(39) FENV_ACCESS プラグマの既定の状態。（7.6.1）

FENV_ACCESS プラグマの既定の状態は ON になります。
ただし，#pragma STDC FENV_ACCESS 指定しても無視します。

(40) 付加的な浮動小数点例外，丸めモード，環境，分類，およびそれらのマクロ名。（7.6，7.12）

コンパイラが提供しているライブラリ math.h に準じます。付加的な定義はありません。

(41) FP_CONTRACT プラグマの既定の状態。（7.12.2）

FP_CONTRACT プラグマの既定の状態は ON になります。

(42) IEC 60559 に準拠した処理系で，丸めの結果が実際に数学的な結果と同等である時に，" 不正確結果 " 浮動小数点例外が生成されうるかどうか。（F.9）

浮動小数点例外をサポートしていません。
" 不正確結果 " 浮動小数点例外は生成されません。

(43) IEC 60559 に準拠した処理系で，結果が極めて小さいが不正確ではない時に，" アンダーフロー" 浮動小数点例外や " 不正確結果 " 浮動小数点例外が生成されうるかどうか。（F.9）

浮動小数点例外をサポートしていません。" アンダーフロー " 浮動小数点例外や " 不正確結果 " 浮動小数点例外は生成されません。

(44) ポインタから整数への変換の結果，またはその逆の結果。（6.3.2.3）

整数からポインタへの変換結果
整数型のサイズがポインタ型のサイズより大きい場合は，整数型の下位バイトの値になります。整数型のサイズとポインタ型のサイズが等しい場合は，整数型のビットパターンがそのまま保持されます。整数型のサイズがポインタ型のサイズより小さい場合は，int 型に拡張した結果の値をそのまま保持します。

ポインタから整数への変換結果
ポインタ型のサイズが整数型のサイズより大きい場合は，ポインタ型の下位バイトの値になります。ポインタ型のサイズと整数型のサイズが等しい場合は，ポインタ型のビットパターンがそのまま保持されます。ポインタ型のサイズが整数型のサイズより小さい場合は，ポインタ型の値をゼロ拡張した値になります。

(45) 同じ配列の要素への 2 つのポインタの減算の結果の大きさ。（6.5.6）

結果の型は signed int 型となります。

(46) register 記憶域クラス指定子の使用がどのくらい効果を持つか。（6.7.1）

register 指定子を無視して最適化します。

(47) inline 関数指定子の使用がどのくらい効果を持つか。（6.7.4）

常に展開を試みます。ただし，条件によっては展開を行わないことがあります。

(48) 単なる int ビットフィールドが signed int ビットフィールドとして扱われるか，それとも unsigned int ビットフィールドとして扱われるか。（6.7.2，6.7.2.1）

signed int 型として扱います。ビットフィールドの最上位ビットは符号ビットとして扱います。

(49) _Bool，signed int，unsigned int 以外で許されるビットフィールドの型。（6.7.2.1）

全ての整数型が許されています。

(50) ビットフィールドが記憶域単位の境界を跨ぐかどうか。（6.7.2.1）

構造体パッキング未指定時は，ビットフィールドは境界を跨がず，次の領域に割り付けます。構造体パッキング指定時は，ビットフィールドは境界を跨ぐことがあります。

(51) 記憶域単位内のビットフィールド割付けの順序。（6.7.2.1）

下位から割り付けます。オプション -Xbit_order または #pragma bit_order で選択が可能です。

(52) 構造体または共用体オブジェクトのビットフィールド以外の各メンバの境界の調整方法。（6.7.2.1）

「4.1.3 データの内部表現と領域」を参照してください。

(53) それぞれの列挙型が適合する整数型。（6.7.2.2）

signed int 型です。ただし，オプション -Xenum_type=auto 指定時は，列挙値が収まる最小の型となります。

(54) volatile 修飾型のオブジェクトへのアクセスをどのように構成するか。（6.7.3）

アクセス幅，アクセス順序，アクセス回数は C ソース上の記述通りに実施しますが，対応するマイコンの命令がない型へのアクセスは，その限りではありません。

(55) #include のヘッダ名の 2 つの形式中の文字列が，ヘッダまたは外部ソース・ファイルの名前に対応付けられる方法。（6.4.7）

#include ヘッダに記述された文字列は，ソース文字集合として指定した文字コードとして解釈され，ヘッダ名または外部ソース・ファイル名に対応付けられます。

(56) 条件付き取り込みを制御する定数式中の文字定数の値が実行文字集合の同じ文字定数の値と一致するかどうか。（6.10.1）

条件付き取り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。

(57) 条件付き取り込みを制御する定数式中の単一文字から成る文字定数が負の値を持ってもよいかどうか。（6.10.1）

負の値を持つことがあります。

(58) #include 指令で < と > で囲まれたヘッダが探索される場所，場所の指定方法，及びヘッダの識別方法。（6.10.2）

次の順序で探索し，フォルダにある同名ファイルをヘッダと識別します。なお，Windows では ¥ 文字をフォルダ区切りとして扱います。

1. フルパス指定の場合はそのパスが示すフォルダ
2. -I オプションで指定されたフォルダ
3. 標準インクルードファイルフォルダ（コンパイラが置かれた bin フォルダと同じフォルダ階層にある inc フォルダ）

(59) #include 指令で，2 つの " で囲まれたソース・ファイルの探索手順。（6.10.2）

次の順序で探索します。

1. フルパス指定の場合はそのパスが示すフォルダ
2. ソース・ファイルがあるフォルダ
3. -I オプションで指定されたフォルダ
4. 標準インクルードファイルフォルダ（コンパイラが置かれた bin フォルダからの相対パスでの ..¥inc フォルダ）

(60) #include 指令の前処理トークン（マクロ展開の可能性がある）をヘッダ名に結合する方法。（6.10.2）

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，単一のヘッダまたはソース・ファイル名の前処理字句として扱われます。

(61) #include 指令の入れ子の限界。（6.10.2）

制限はありません。

(62) # 演算子が，文字定数や文字列リテラル中の国際文字名の最初の ¥ 文字の前に ¥ 文字を挿入するかどうか。（6.10.3.2）

最初の ¥ 文字の前に ¥ 文字は挿入しません。

(63) 非 STDC のプラグマ指令の動作。（6.10.6）

「4.2.6 拡張言語仕様の使用方法」を参照してください。

(64) 翻訳の日付や時刻が得られない場合，__DATE__ マクロと __TIME__ マクロで得られる日付や時刻。（6.10.8）

日付や時刻が得られない場合はありません。

(65) ISO/IEC 9899:1999 の 4 章で要求される最低限必要なライブラリ機能以外で，フリースタンディング環境でプログラムが利用できるライブラリ機能。（5.1.2.1）

「7. ライブラリ関数仕様」を参照してください。

(66) assert マクロによって標準エラーストリームに書き込まれる診断機能の書式。（7.2.1.1）

以下の通りです。
Assertion failed：式，function 関数名，file ファイル名，line 行番号

(67) fegetexceptflag 関数によって格納される浮動小数点状態フラグの表現。（7.6.2.2）

fegetexceptflag 関数はサポートしていません。

(68) feraiseexcept 関数が，" オーバーフロー" 浮動小数点例外または " アンダーフロー" 浮動小数点例外を生成する場合はいつでも，それに加えて " 不正確結果 " 浮動小数点例外を生成するかどうか。（7.6.2.3）

feraiseexcept 関数はサポートしていません。

(69) setlocale 関数に第 2 引数として渡される文字列で "C" や "" 以外の文字列。（7.11.1.1）

setlocale 関数はサポートしていません。

(70) FLT_EVAL_METHOD マクロの値が 0 未満または 2 より大きい場合の float_t と double_t で定義される型。（7.12）

float_t は float 型，double_t は double 型になります。

(71) 数学関数における，この国際標準（C99）によって要求される以外の定義域エラー。（7.12.1）

atan2，cos，sin，tan，frexp，pow，lround，llround，fmod 関数群は定義域エラーとなる場合があります。

(72) 定義域エラー発生時の数学関数の返却値。（7.12.1）

「7.4.10 数学関数」を参照してください。

(73) 浮動小数点演算の結果がアンダーフローした場合の数学関数の返却値。アンダーフロー時，整数式 math_errhandling & MATH_ERRNO が 0 以外の値の場合，整数式 errno の値が ERANGE となるかどうか。アンダーフロー時，整数式 math_errhandling & MATH_ERREXCEPT が 0 以外の値の場合，" アンダーフロー" 浮動小数点例外を生成するかどうか。（7.12.1）

返却値は 0 です。ただし，exp，ldexp 関数群は 0 または非正規化数を返却します。アンダーフロー発生時は errno に ERANGE を設定します。アンダーフロー浮動小数点例外は生成しません。

(74) fmod 関数群の第 2 実引数が 0 の場合に，定義域エラーが発生するか，あるいは 0 が返されるかどうか。（7.12.10.1）

定義域エラーが発生します。詳細は fmod 関数群の説明を参照してください。

(75) remquo 関数群によって商を縮小する際に使用される法の，2 を底とする対数の値。（7.12.10.3）

remquo 関数群はサポートしていません。

(76) シグナル発生時，signal(sig, SIG_DFL); と同等のことが実行されるか，あるいは，その時点のシグナル処理ルーチンが完了するまで，シグナルの処理系定義の組に対してそれらの発生を遮るか。（7.14.1.1）

シグナル操作関数はサポートしていません。

(77) マクロ NULL 展開後の空ポインタ定数。（7.17）

(void*)0 とします。

(78) テキストストリームの最終行が終端を示す改行文字を必要とするかどうか。（7.19.2）

改行文字を必要としません。

(79) テキストストリームにおいて，改行文字の直前に書き込まれた空白文字の並びが，データが読み取られる時に現れるかどうか。（7.19.2）

データが読み取られる時に現れます。

(80) バイナリストリームの最後に付加される NULL 文字の数。（7.19.2）

0 個です。

(81) 追加モードでオープンされたファイルに対し，最初にファイル位置指定子がファイルの始めに位置付けられるか終わりに位置付けられるか。（7.19.3）

ファイル操作関数はサポートしていません。

(82) テキストストリームへの書込みが，結び付けられたファイルを最終書込み点の直後で切り捨てるかどうか。（7.19.3）

ファイル操作関数はサポートしていません。

(83) ファイルバッファリングの特性。（7.19.3）

ファイル操作関数はサポートしていません。

(84) 長さ 0 のファイルが実際に存在するかどうか。（7.19.3）

ファイル操作関数はサポートしていません。

(85) 正しいファイル名の規則。（7.19.3）

ファイル操作関数はサポートしていません。

(86) 同一のファイルを同時に複数回オープンできるかどうか。（7.19.3）

ファイル操作関数はサポートしていません。

(87) ファイル内の多バイト文字のために使用される表現形式の特性と選択。（7.19.3）

ファイル操作関数はサポートしていません。

(88) オープンされているファイルに対する remove 関数の効果。（7.19.4.1）

ファイル操作関数はサポートしていません。

(89) rename 関数を呼び出す前に，第 2 引数で指定された新しいファイル名のファイルが既に存在していた場合の rename 関数の動作。（7.19.4.2）

ファイル操作関数はサポートしていません。

(90) プログラムが異常終了した場合，オープン中の一時ファイルが削除されるかどうか。（7.19.4.3）

ファイル操作関数はサポートしていません。

(91) filename 引数が空ポインタの場合，どのようなモード変更を許すか，及びどのような状況での変更を許すか。（7.19.5.4）

ファイル操作関数はサポートしていません。

(92) 無限大や NaN を書き込む時の形式，及び NaN の書込みで使われるかもしれない n 文字列，n ワイド文字列の意味。（7.19.6.1，7.24.2.1）

正の無限大は（+INF），負の無限大は（-INF），非数は（NaN）を出力します。
NaN の書き込みにおける n 文字列，n ワイド文字列はサポートしていません。

(93) fprintf 関数および fwprintf 関数での %p 変換の出力。（7.19.6.1，7.24.2.1）

10 進表記です。
fwprintf 関数はサポートしていません。

(94) fscanf 関数および fwscanf 関数の %[変換において，文字 - が走査文字の並びに含まれ，かつ先頭の文字（先頭が文字 ^ のときは 2 文字目）でも最後の文字でもない場合の文字 - の解釈。（7.19.6.2，7.24.2.1）

「7.4.7 標準入出力関数」にある「sscanf」を参照してください。

(95) fscanf 関数および fwscanf 関数での %p 変換によって一致する文字の並びの集合と対応する入力項目の解釈。（7.19.6.2，7.24.2.2）

10 進整数です。
fwscanf 関数はサポートしていません。

(96) fgetpos 関数，fsetpos 関数，ftell 関数の失敗時に設定される errno マクロの値。（7.19.9.1，7.19.9.3，7.19.9.4）

ファイル操作関数はサポートしていません。

(97) strtod 関数，strtof 関数，strtold 関数，wcstod 関数，wcstof 関数，wcstold 関数によって変換された NaN を表現する文字列での n 文字あるいは n ワイド文字の並びの意味。（7.20.1.3，7.24.4.1.1）

strtod 関数，strtodf 関数は浮動小数点型の数値以外と解釈します。
strtof 関数，strtold 関数，wcstod 関数，wcstof 関数，wcstold 関数はサポートしていません。

(98) strtod 関数，strtof 関数，strtold 関数，wcstod 関数，wcstof 関数，wcstold 関数が，アンダーフロー発生時に errno に ERANGE 格納するかどうか。（7.20.1.3，7.24.4.1.1）

strtod 関数，strtodf 関数はグローバル変数 errno に ERANGE をセットします。
strtof 関数，strtold 関数，wcstod 関数，wcstof 関数，wcstold 関数はサポートしていません。

(99) calloc 関数，malloc 関数，realloc 関数が，要求された領域の大きさが 0 である時に，割り付けたオブジェクトへのポインタを返すか，それとも空ポインタを返すか。（7.20.3）

calloc 関数，malloc 関数は要求された大きさを 8 とみなし，確保したポインタを返します。realloc 関数は NULL を返します。

(100) abort 関数，_Exit 関数の呼び出し時に，書き出されていないバッファリングされたデータをもつオープンしているストリームをフラッシュするかどうか，オープンしているストリームをクローズするかどうか，一時ファイルを削除するかどうか。（7.20.4.1，7.20.4.4）

ファイル操作関数はサポートしていません。

(101) abort 関数，exit 関数，_Exit 関数によってホスト環境へ返される終了状態。（7.20.4.1，7.20.4.3，7.20.4.4）

フリースタンディング環境であるため，規定しません。

(102) system 関数に渡された実引数が空ポインタでない場合に，system 関数によって返される値。（7.20.4.6）

system 関数はサポートしていません。

(103) 地方時と夏時間。（7.23.1）

time.h はサポートしていません。

(104) clock_t と time_t で表現可能な時刻の範囲と精度。（7.23）

time.h はサポートしていません。

(105) clock 関数で計算されるプロセッサ時間の開始時点。（7.23.2.1）

time.h はサポートしていません。

(106) "C" ロケールでの，strftime 関数，wcsftime 関数の %Z 変換指定子に対応する置換文字列。（7.23.3.5，7.24.5.1）

time.h はサポートしていません。

(107) IEC 60559 準拠の処理系で，三角関数，双曲線関数，底が e の指数関数，底が e の対数関数，エラー関数，ログガンマ関数が，" 不正確結果 " 浮動小数点例外を生成するかどうか。生成する場合，いつ生成するか。（F.9）

" 不正確結果 " 浮動小数点例外を生成しません。

(108) IEC 60559 準拠の処理系で，<math.h> の関数が丸め方向モードを尊重するか。（F.9）

リンクするライブラリに依存します。
fesetround 関数はサポートしていません。

(109) <float.h>，<limits.h>，<stdint.h> の各ヘッダで指定されたマクロに割り当てられる値あるいは式。（5.2.4.2，7.18.2，7.18.3）

（）内は，sizeof(double)=sizeof(long double)=4 にするオプション（-Xdbl_size=4）指定時の値です。

float.h

名前	値	意味
FLT_ROUNDS	1（-Xround=nearest 指定時） 0（-Xround=zero 指定時）	浮動小数点加算に対する丸めのモード 1（最も近い方向へ丸める）とする。 0（ゼロ方向へ丸める）とする。
FLT_EVAL_METHOD	0	浮動小数点数の評価形式
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（+24）	
LDBL_MANT_DIG	+53（+24）	
DECIMAL_DIG	+17（+9）	基数 b の p 桁をもつ浮動小数点数を q 桁の 10 進数の浮動小数点数に丸めることができ，再び変更なしに基数 b の p 桁をもつ浮動小数点数に戻すことが可能な 10 進数の桁数（q）
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可能な 10 進数の桁数（q）
DBL_DIG	+15（+6）	
LDBL_DIG	+15（+6）	
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（-125）	
LDBL_MIN_EXP	-1021（-125）	
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（-37）	
LDBL_MIN_10_EXP	-307（-37）	
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（+128）	
LDBL_MAX_EXP	+1024（+128）	
FLT_MAX_10_EXP	+38	10 をその値でべき乗したとき，表現可能な有限浮動小数点数の範囲内になるような最大の整数 $\log_{10} ((1 - b^{-p}) * b^{e_{max}})$
DBL_MAX_10_EXP	+308（+38）	
LDBL_MAX_10_EXP	+308（+38）	
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 （3.40282347E+38F）	
LDBL_MAX	1.7976931348623158E+308 （3.40282347E+38F）	

名前	値	意味
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 （1.19209290E - 07F）	
LDBL_EPSILON	2.2204460492503131E-016 （1.19209290E - 07F）	
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 （1.17549435E - 38F）	
LDBL_MIN	2.2250738585072014E-308 （1.17549435E - 38F）	

half.h

名前	値	意味
HALF_MANT_DIG	+11	浮動小数点仮数部における FLT_RADIX を底とする数字の数（p）
HALF_DIG	+2	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可能な 10 進数の桁数（q）
HALF_MIN_EXP	-13	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}）
HALF_MIN_10_EXP	-4	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 $\log_{10} b^{e_{min-1}}$
HALF_MAX_EXP	+16	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}）
HALF_MAX_10_EXP	+4	10 をその値でべき乗したとき，表現可能な有限浮動小数点数の範囲内になるような最大の整数 $\log_{10} ((1 - b^{-p}) * b^{e_{max}})$
HALF_MAX	65504.0F	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
HALF_EPSILON	0.00097656F	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 $b^{1 - p}$
HALF_MIN	6.10352E-05F	正規化された正の浮動小数点数の最小値 $b^{e_{min - 1}}$

limits.h

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
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	-9223372036854775808	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 型の最大値

stdint.h

名前	値	意味
INT8_MIN	-0x7f-1	int8_t 型の最小値
INT16_MIN	-0x7fff-1	int16_t 型の最小値
INT32_MIN	-0x7fffffff-1	int32_t 型の最小値
INT64_MIN	-0x7fffffffffffffffLL-1	int64_t 型の最小値
INT8_MAX	0x7f	int8_t 型の最大値
INT16_MAX	0x7fff	int16_t 型の最大値
INT32_MAX	0x7fffffff	int32_t 型の最大値
INT64_MAX	0x7fffffffffffffffLL	int64_t 型の最大値
UINT8_MAX	0xff	uint8_t 型の最大値
UINT16_MAX	0xffff	uint16_t 型の最大値
UINT32_MAX	0xffffffff	uint32_t 型の最大値
UINT64_MAX	0xffffffffffffffffULL	uint64_t 型の最大値
INT_LEAST8_MIN	-0x7f-1	int_least8_t 型の最小値

名前	値	意味
INT_LEAST16_MIN	-0x7fff-1	int_least16_t 型の最小値
INT_LEAST32_MIN	-0x7fffffff-1	int_least32_t 型の最小値
INT_LEAST64_MIN	-0x7fffffffffffffffLL-1	int_least64_t 型の最小値
INT_LEAST8_MAX	0x7f	int_least8_t 型の最大値
INT_LEAST16_MAX	0x7fff	int_least16_t 型の最大値
INT_LEAST32_MAX	0x7fffffff	int_least32_t 型の最大値
INT_LEAST64_MAX	0x7fffffffffffffffLL	int_least64_t 型の最大値
UINT_LEAST8_MAX	0xff	uint_least8_t 型の最大値
UINT_LEAST16_MAX	0xffff	uint_least16_t 型の最大値
UINT_LEAST32_MAX	0xffffffffU	uint_least32_t 型の最大値
UINT_LEAST64_MAX	0xffffffffffffffffULL	uint_least64_t 型の最大値
INT_FAST8_MIN	-0x7fffffff-1	int_fast8_t 型の最小値
INT_FAST16_MIN	-0x7fffffff-1	int_fast16_t 型の最小値
INT_FAST32_MIN	-0x7fffffff-1	int_fast32_t 型の最小値
INT_FAST64_MIN	-0x7fffffffffffffffLL-1	int_fast64_t 型の最小値
INT_FAST8_MAX	0x7fffffff	int_fast8_t 型の最大値
INT_FAST16_MAX	0x7fffffff	int_fast16_t 型の最大値
INT_FAST32_MAX	0x7fffffff	int_fast32_t 型の最大値
INT_FAST64_MAX	0x7fffffffffffffffLL	int_fast64_t 型の最大値
UINT_FAST8_MAX	0xffffffffU	uint_fast8_t 型の最大値
UINT_FAST16_MAX	0xffffffffU	uint_fast16_t 型の最大値
UINT_FAST32_MAX	0xffffffffU	uint_fast32_t 型の最大値
UINT_FAST64_MAX	0xffffffffffffffffULL	uint_fast64_t 型の最大値
INTPTR_MIN	-0x7fffffff-1	intptr_t 型の最小値
INTPTR_MAX	0x7fffffff	intptr_t 型の最大値
UINTPTR_MAX	0xffffffffU	uintptr_t 型の最大値
INTMAX_MIN	-0x7fffffffffffffffLL-1	intmax_t 型の最小値
INTMAX_MAX	0x7fffffffffffffffLL	intmax_t 型の最大値
UINTMAX_MAX	0xffffffffffffffffULL	uintmax_t 型の最大値
PTRDIFF_MIN	-0x7fffffff-1	ptrdiff_t 型の最小値
PTRDIFF_MAX	0x7fffffff	ptrdiff_t 型の最大値
SIZE_MAX	0xffffffffU	size_t 型の最大値

(110) 明示的に規定されていない場合の，オブジェクトを構成するバイトの並びのバイト数，バイトの順序，表現方法。（6.2.6.1）

「4.1.3 データの内部表現と領域」を参照してください。

(111) sizeof 演算子の結果の値。（6.5.3.4）

「4.1.3 データの内部表現と領域」を参照してください。

翻訳限界
C99 規格の翻訳限界に対する CC-RH の仕様を示します。
「制限なし」となっている項目は，コンパイル環境の状態に依存します。

項目	限界値
ブロックの入れ子のレベル数	制限なし
条件付き取込みにおける入れ子のレベル数	制限なし
一つの宣言中の一つの算術型，構造体型，共用体型又は不完全型を修飾するポインタ，配列及び関数宣言子（の任意の組合せ）の個数	128
一つの完結宣言子における括弧で囲まれた宣言子の入れ子のレベル数	制限なし
一つの完結式における括弧で囲まれた式の入れ子のレベル数	制限なし
内部識別子又はマクロ名において意味がある先頭の文字数	制限なし
外部識別子において意味がある先頭の文字数	制限なし
一つの翻訳単位中における外部識別子数	制限なし
一つのブロックで宣言されるブロック有効範囲をもつ識別子数	制限なし
一つの前処理翻訳単位中で同時に定義されるマクロ識別子数	制限なし
一つの関数定義における仮引数の個数	制限なし
一つの関数呼出しにおける実引数の個数	制限なし
一つのマクロ定義における仮引数の個数	制限なし
一つのマクロ呼出しにおける実引数の個数	制限なし
一つの論理ソース行における文字数	制限なし
（連結後の）単純文字列リテラル又はワイド文字列リテラル中における文字数	制限なし
（ホスト環境の場合）一つのオブジェクトのバイト数	2147483647
#include で取り込まれるファイルの入れ子のレベル数	制限なし
一つの switch 文（入れ子になった switch 文を除く）中における case ラベルの個数	2147483647
一つの構造体又は共用体のメンバ数	制限なし
一つの列挙体における列挙定数の個数	制限なし
一つのメンバ宣言並びにおける構造体又は共用体定義の入れ子のレベル数	制限なし

4.1.3 データの内部表現と領域

この項では，CC-RH 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。

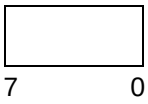
(1) 整数型

(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。

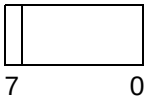
整数型の内部表現を以下に示します。

- _Bool

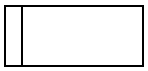


ゼロビット目のみが意味を持ちます。1 ～ 7 ビット目は不定値となります。
-lang=c オプションと -strict_std オプションを同時に指定している場合は，_Bool 型は C90 違反でエラーとなります。

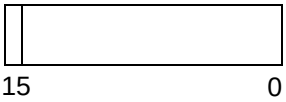
- char



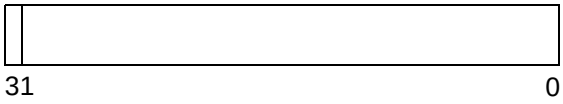
- signed char（unsigned では符号ビットなし）



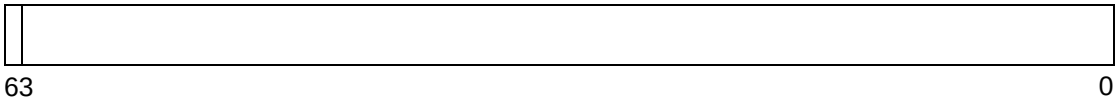
- short（unsigned では符号ビットなし）



- int，long（unsigned では符号ビットなし）



- long long（unsigned では符号ビットなし）



-lang=c オプションと -strict_std オプションを同時に指定している場合は，long long 型は C90 違反でエラーとなります。

(b) 値域

表 4.1 整数型の値域

型	値域
char	-128 ～ +127
short	-32768 ～ +32767
int	-2147483648 ～ +2147483647
long	-2147483648 ～ +2147483647
long long	-9223372036854775808 ～ +9223372036854775807
unsigned char	0 ～ 255
unsigned short	0 ～ 65535
unsigned int	0 ～ 4294967295
unsigned long	0 ～ 4294967295
unsigned long long	0 ～ 18446744073709551615

(c) 整数定数
整数定数の型は，次の並びのうちでその値を表現できる最初の型となります。

表 4.2 整数定数の型（-lang=c 指定あり，-strict_std 指定なし）

接尾語	10 進定数	8 進定数，または 16 進定数
なし	int long int unsigned long int 注 long long int unsigned long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u，または U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l，または L	long int unsigned long int 注 long long int unsigned long long int	long int unsigned long int long long int unsigned long long int
u，または U，および l，または L の両方	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll，または LL	long long int unsigned long long int	long long int unsigned long long int
u，または U，および ll，または LL の両方	unsigned long long int	unsigned long long int

注 C99 の仕様と異なります。

表 4.3 整数定数の型（-lang=c 指定あり，-strict_std 指定あり）

接尾語	10 進定数	8 進定数，または 16 進定数
なし	int long int unsigned long int	int unsigned int long int unsigned long int
u，または U	unsigned int unsigned long int	unsigned int unsigned long int
l，または L	long int unsigned long int	long int unsigned long int
u，または U，および l，または L の両方	unsigned long int	unsigned long int

表 4.4 整数定数の型（-lang=c99 指定あり）

接尾語	10 進定数	8 進定数，または 16 進定数
なし	int long int long long int unsigned long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u，または U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l，または L	long int long long int unsigned long long int	long int unsigned long int long long int unsigned long long int
u，または U，および l，または L の両方	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll，または LL	long long int unsigned long long int	long long int unsigned long long int
u，または U，および ll，または LL の両方	unsigned long long int	unsigned long long int

(2) 浮動小数点型

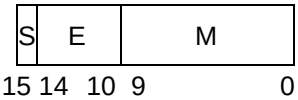
(a) 内部表現

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。

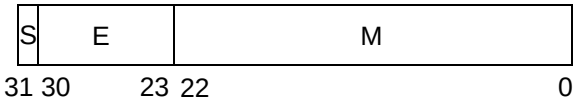
浮動小数点型の内部表現を以下に示します。

- __fp16



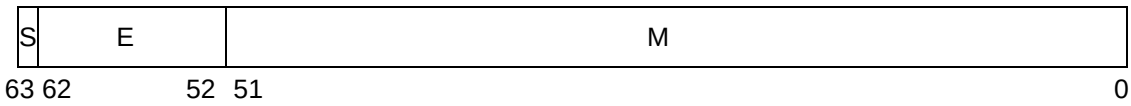
S： 仮数部の符号ビット
E： 指数部（5 ビット）
M： 仮数部（10 ビット）

- float



S： 仮数部の符号ビット
E： 指数部（8 ビット）
M： 仮数部（23 ビット）

- double, long double



S： 仮数部の符号ビット
E： 指数部（11 ビット）
M： 仮数部（52 ビット）

(b) 値域

表 4.5 浮動小数点型の値域

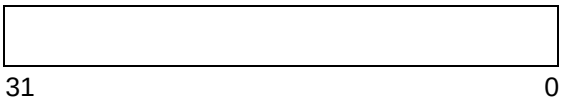
型	値域
__fp16	6.10352e-05F ～ 65504.0
float	1.17549435E-38F ～ 3.40282347E+38F
double	2.2250738585072014E-308 ～ 1.7976931348623158E+308
long double	2.2250738585072014E-308 ～ 1.7976931348623158E+308

-Xdbl_size=4 指定時は，double，long double 型は float 型と同じ内部表現，値域を持ちます。

(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。

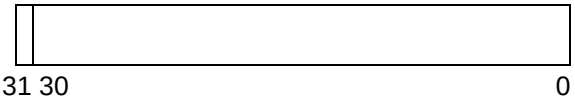
図 4.1 ポインタ型の内部表現



(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 4.2 列挙型の内部表現



-Xenum_type=auto オプション指定時には，「(32) 列挙型の値を表現するために選択される整数型（6.5.2.2）。」を参照してください。

(5) 配列型

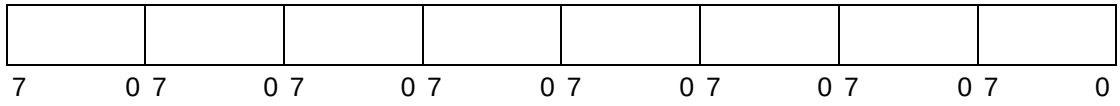
(a) 内部表現

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。

```
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 4.3 配列型の内部表現



(6) 構造体型

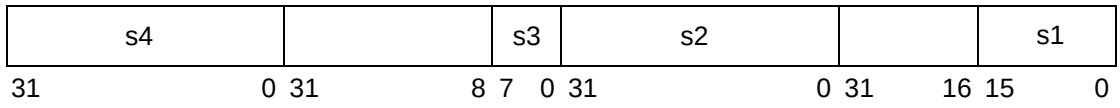
(a) 内部表現

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。

```
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 4.4 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，「4.2.6.8 構造体パッキング」を参照してください。

(7) 共用体型

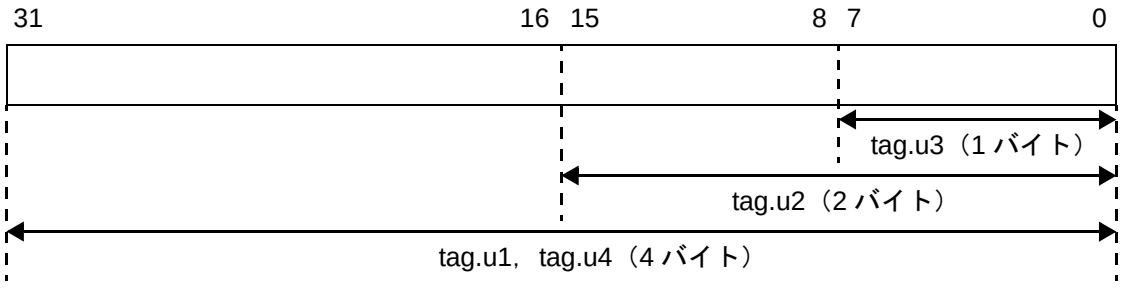
(a) 内部表現

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。

```
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 4.5 共用体型の内部表現



(8) ビット・フィールド

(a) 内部表現

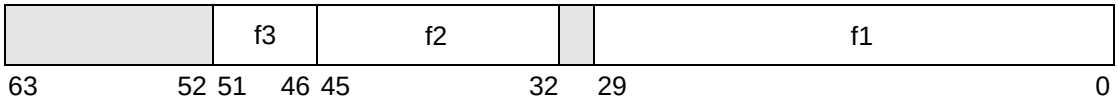
ビット・フィールドに対しては，宣言された数のビットを含む領域が取られます。符号付きの型として，または符号を明示せずに宣言されたビット・フィールドに対しては，最上位ビットは符号ビットとなります。
最初に宣言されたビット・フィールドは，ビット・フィールド宣言時の型のサイズの領域の最下位ビットから割り当てられます。ビット・フィールドに対し，その前のビット・フィールドに続けて領域を割り当てると，その領域がそのビット・フィールドの宣言において指定された型の整列条件を満たす境界を越えてしまう場合，そのビット・フィールドに対する領域はその整列条件を満たしている境界から割り当てられます。
-Xbit_order=left オプション，または #pragma bit_order left を指定することで，ビット・フィールド・メンバを上位ビット側から割り当てることも可能です。詳細は，「4.2.6.9 ビット・フィールドの割り付け」を参照してください。

例 1.

```
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 4.6 ビット・フィールドの内部表現

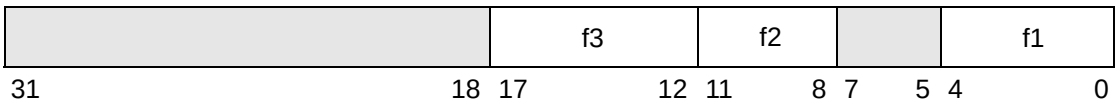


例 2.

```
struct {
    int             f1:5;
    char            f2:4;
    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 4.7 ビット・フィールドの内部表現



ビット・フィールドには _Bool，char，signed char，unsigned char，signed short，unsigned short，signed int，unsigned int，signed long，unsigned long，signed long long，unsigned long long，および列挙型を指定することができます。ただし，-lang=c オプションおよび -strict_std オプション指定時は，int 型と，unsigned int 型のみ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，「4.2.6.8 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，CC-RH 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 4.6 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 _Bool 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 (unsigned) long long とその配列型 double とその配列型	4 バイト境界
long double とその配列型	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，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。

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 */
```

例 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。

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 */
```

例 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 */
```

例 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 */
```

例 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 */
```

例 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・ファイルをリンクして実行可能なオブジェクト・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。

4.1.4 レジスタ・モード

CC-RH では，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，CC-RH 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，CC-RH 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。

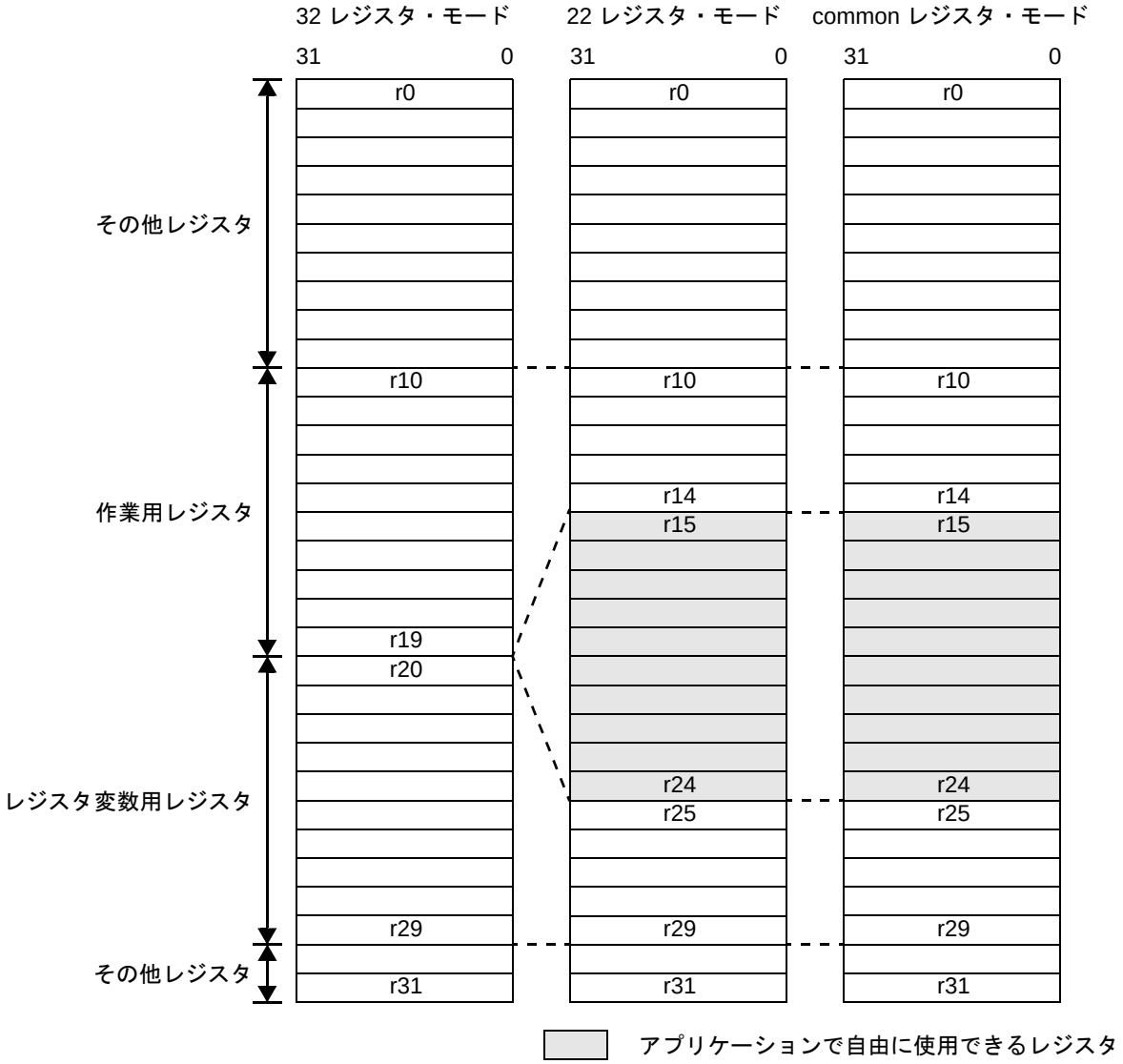
注意 CC-RH によるレジスタ割り付けの対象となる変数の多いアプリケーション・プログラムでは，レジスタ・モードの指定によって，それまでレジスタに割り付けられていた変数がメモリ・アクセスとなり，その分，処理速度が低下することがあります。

次表，および次図に，CC-RH 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 4.7 CC-RH 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
22 レジスタ・モード	r10 ～ r14	r25 ～ r29
common レジスタ・モード	r10 ～ r14	r25 ～ r29

図 4.8 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

```
> ccrh -Xreg_mode=22 file.c      ← 22 レジスタ・モードでコンパイル
```

4.2 拡張言語仕様

この節では，CC-RH で独自に追加されている拡張言語仕様について説明します。

4.2.1 予約語

CC-RH では 1 個の下線と大文字のアルファベットから始まる名前，また 2 個の下線を含む名前を，ラベルや変数，関数の識別子として使うことはできません。

4.2.2 マクロ

CC-RH であらかじめ定義されるマクロ名を次に示します。

表 4.8 サポートしているマクロ

定義する条件[注 1]	マクロ名	値
常に	__LINE__	その時点でのソース行の行番号（10 進数）
常に	__FILE__	ソース・ファイルの名前（文字列定数）
常に	__DATE__ [注 2]	ソース・ファイルの翻訳日付（文字列定数）[注 3]
常に	__TIME__ [注 2]	ソース・ファイルの翻訳時間（文字列定数）[注 4]
-strict_std 指定時	__STDC__	1
-lang=c99 指定時	__STDC_HOSTED__	0
常に	__STDC_VERSION__	199409L（-lang=c99 未指定時） 199901L（-lang=c99 指定時）
-lang=c99 指定時	__STDC_IEC_559__	1
常に	__RENESAS__	1
常に	__RENESAS_VERSION__	0xXXYYZZ00 [注 5]
常に	__CCRH__	1
	__CCRH	1
常に	__RH850__	1
	__RH850	1
常に	__v850e3v5__	1
	__v850e3v5	1
-Xdbl_size=4 指定時	__DBL4	1
	__DOUBLE_IS_32BITS__	1
-Xdbl_size=8 指定時	__DBL8	1
	__DOUBLE_IS_64BITS__	1
-Xround=nearest 指定時	__RON	1
-Xround=zero 指定時	__ROZ	1
-Xbit_order=left 指定時	__BITLEFT	1
-Xbit_order=right 指定時	__BITRIGHT	1
-Xenum_type=auto 指定時	__AUTO_ENUM	1
-Xfloat=fpu 指定時	__FPU	1

定義する条件[注 1]	マクロ名	値
常に	__CHAR_SIGNED__	1
-Xreg_mode=32 指定時	__reg32__	1
-Xreg_mode=22 指定時	__reg22__	1
-Xreg_mode=common 指定時	__reg_common__	1
常に	_LIT	1
常に	__MULTI_LEVEL__	*level* で指定した値（10 進数） （-Xmulti_level オプションが指定されていない場合，値は 0）
-pic 指定時	__PIC	1
-pirod 指定時	__PIROD	1
-pid 指定時	__PID	1

注 1. オプション指定の条件は，省略時解釈の場合も該当します。

注 2. 翻訳日付または翻訳時刻を得られない場合はなく，__DATE__，__TIME__ は常に定義を持ちます。

注 3. “Mmm dd yyyy” の形式をもつ文字列定数。ここで，月の名前（Mmm）は C 言語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。

注 4. 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。

注 5. バージョンが VXX.YY.ZZ の場合，0xXXYYZZ00 とします。
例）V1.02.03 → __RENESAS_VERSION__ = 0x01020300

4.2.3 C90 でサポートする C99 言語仕様

CC-RH では，C90 準拠時（-lang=c 指定時）でも，一部の C99 規格の仕様が有効です。

(1) // 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。

(2) ワイド文字列の結合
文字列定数とワイド文字列定数を結合した場合，ワイド文字列定数となります。

(3) _Bool 型
_Bool 型をサポートします。

注 -lang=c99 オプション指定時は，厳密な _Bool 型として扱います。-lang=c オプション指定時は，一部の式で signed char 型として扱います。

(4) 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。

(5) 整数拡張
_Bool 型と long long 型のサポートに伴って，整数拡張も C99 の仕様に従います。

(6) 既定の実引数拡張
_Bool 型と long long 型のサポートに伴って，既定の実引数拡張も C99 の仕様に従います。

- _Bool 型は int 型（4 バイト）に拡張して関数を呼び出します。
- (unsigned) long long 型は，8 バイトのまま関数を呼び出します。

(7) 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,};
```

(8) inline キーワード（inline 関数）
inline キーワードをサポートします。
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 ]... )注
```

注 外側のかっこは省略可能です。

なお，インライン展開についての詳細は「4.2.6.3 インライン展開」を参照してください。
-lang=c99 オプション指定時は，C99 の inline キーワードとして扱います。それ以外の場合は，#pragma inline 指令と同じ意味です。

(9) 整数定数の型
long long 型の追加に伴い，整数定数の型が変わります。詳細は「4.1.3 データの内部表現と領域」の「(c) 整数定数」を参照してください。

4.2.4 コンパイラ生成シンボル

次に，コンパイ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。
以下のシンボルと同名のシンボルは利用できません。

表 4.9 コンパイラ生成シンボル

対象	生成シンボル
extern 関数名	_ 関数名
static 関数名	_ 関数名 *.num* 注
extern 変数名	_ 変数名
ファイル内 static 変数	_ 変数名 *.num* 注
関数内 static 変数	_ 変数名 *.num* 注 . 関数ラベル
関数内ラベル	.BB.LABEL.*num1_num2* 注
文字列リテラル，自動変数の初期値	.STR.*num* 注
複合リテラル	_ _ _T *num1.num2* 注
switch テーブル	.SWITCH.LABEL.*num1_num2* 注 .SWITCH.LABEL.*num1_num2* 注 .END

注 *num*，*num1*，*num2* は任意の数字です。

4.2.5 #pragma 指令

次に，拡張言語仕様としてサポートしている #pragma 指令を示します。C99 言語における _Pragma 演算子でもこれらの拡張機能を使用できます。

表 4.10 サポートしている #pragma 指令

#pragma 指令	内容
#pragma section	「4.2.6.1 関数とデータのセクション割り当て」を参照してください。
#pragma inline_asm	「4.2.6.2 アセンブラ命令の記述」を参照してください。
#pragma inline #pragma noinline	「4.2.6.3 インライン展開」を参照してください。
#pragma interrupt	「4.2.4.5 (3) 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 block_interrupt	「4.2.4.6 (2) 関数全体の割り込みを禁止する方法」を参照してください。
#pragma pack	「4.2.6.8 構造体パッキング」を参照してください。
#pragma bit_order	「4.2.6.9 ビット・フィールドの割り付け」を参照してください。
#pragma pmodule	「4.2.6.10 コア番号指定（マルチコア用）」を参照してください。
#pragma align4	「4.2.6.11 分岐先アドレスのアライメント指定」を参照してください。
#pragma stack_protector #pragma no_stack_protector	「4.2.6.12 スタック破壊検出機能 【Professional 版のみ】」を参照してください。
#pragma register_group	「4.2.6.14 制御レジスタへの書き込みの検出，同期化処理挿入 【Professional 版のみ】【V1.06.00 以降】」を参照してください。

4.2.6 拡張言語仕様の使用方法

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。

- 関数とデータのセクション割り当て
- アセンブラ命令の記述
- インライン展開
- 割り込みレベルの制御
- 割り込み／例外処理ハンドラ
- マスカブル割り込みの禁止／許可
- 組み込み関数
- 構造体パッキング
- ビット・フィールドの割り付け
- コア番号指定（マルチコア用）
- 分岐先アドレスのアライメント指定
- スタック破壊検出機能 【Professional 版のみ】
- 半精度浮動小数点数 【Professional 版のみ】【V1.05.00 以降】
- 制御レジスタへの書き込みの検出，同期化処理挿入 【Professional 版のみ】【V1.06.00 以降】

4.2.6.1 関数とデータのセクション割り当て

CC-RH では，関数やデータをセクションに割り当てることによって，メモリ上の配置や，アクセス方法を制御します。

- 関数用セクション

表 4.11 関数用セクションの種類

セクション再配置属性	デフォルト・セクション名	アクセス方法	整列条件
text	.text	r0 からの 32 ビット長相対	2
pctext	.pctext	PC からの 32 ビット長相対	

- .text セクションがデフォルトで使用するセクションです。
  PIC 機能には使用できません。
- .pctext が PIC 機能で使用するためのセクションです。

- データ用 R0 相対セクション

表 4.12 データ用 R0 相対セクションの種類

セクション再配置属性	デフォルト・セクション名	割り当てる対象	アクセス方法	整列条件
zdata	.zdata	初期値あり変数	r0 からの 16 ビット長相対	4
zbss	.zbss	初期値なし変数		
zdata23	.zdata23	初期値あり変数	r0 からの 23 ビット長相対	
zbss23	.zbss23	初期値なし変数		
data	.data	初期値あり変数	r0 からの 32 ビット長相対	
bss	.bss	初期値なし変数		

- .data，.bss セクションがデフォルトで使用するセクションです。
- .zdata，.zbss，.zdata23，.zbss23 が短い命令長でアクセスする場合に使用するセクションです。
- どの R0 相対セクションも，PID 機能には使用できません。

- データ用 EP 相対セクション

表 4.13 データ用 EP 相対セクションの種類

セクション再配置属性	デフォルト・セクション名	割り当てる対象	アクセス方法	整列条件
tdata4	.tdata4	初期値あり変数	r30（EP）からの 4 ビット長相対 sld，sst 命令を使用可能	4
tbss4	.tbss4	初期値なし変数		
tdata5	.tdata5	初期値あり変数	r30（EP）からの 5 ビット長相対 sld，sst 命令を使用可能	
tbss5	.tbss5	初期値なし変数		
tdata7	.tdata7	初期値あり変数	r30（EP）からの 7 ビット長相対 sld，sst 命令を使用可能	
tbss7	.tbss7	初期値なし変数		
tdata8	.tdata8	初期値あり変数	r30（EP）からの 8 ビット長相対 sld，sst 命令を使用可能	
tbss8	.tbss8	初期値なし変数		
edata	.edata	初期値あり変数	r30（EP）からの 16 ビット長相対	
ebss	.ebss	初期値なし変数		
edata23	.edata23	初期値あり変数	r30（EP）からの 23 ビット長相対	
ebss23	.ebss23	初期値なし変数		
edata32	.edata32	初期値あり変数	r30（EP）からの 32 ビット長相対	
ebss32	.ebss32	初期値なし変数		

- .tdata*，.tbss* セクションは，短い命令長でアクセスする場合に使用するセクションです。
  より短い sld/sst 命令を使用することができます。
  PID 機能でも使用することができます。
- .edata，.ebss，.edata23，.ebss23 は，短い命令長でアクセスする場合に使用するセクションです。
  PID 機能でも使用することができます。
- .edata32，.ebss32 セクションは，PID 機能で使用するためのセクションです。

- データ用 GP 相対セクション

表 4.14 データ用 GP 相対セクションの種類

セクション再配置属性	デフォルト・セクション名	割り当てる対象	アクセス方法	整列条件
sdata	.sdata	初期値あり変数	r4（GP）からの 16 ビット長相対	4
sbss	.sbss	初期値なし変数		
sdata23	.sdata23	初期値あり変数	r4（GP）からの 23 ビット長相対	
sbss23	.sbss23	初期値なし変数		
sdata32	.sdata32	初期値あり変数	r4（GP）からの 32 ビット長相対	
sbss32	.sbss32	初期値なし変数		

- .sdata，.sbss，.sdata23，.sbss23 セクションは，短い命令長でアクセスする場合に使用するセクションです。PID 機能にも使用することができます。
- .sdata32，.sbss32 セクションは，PID 機能で使用するためのセクションです。

- 定数データ用セクション

表 4.15 定数データ用セクションの種類

セクション再配置属性	デフォルト・セクション名	割り当てる対象	アクセス方法	整列条件
zconst	.zconst	const 変数 文字列リテラル 自動変数の初期値注 1	r0 からの 16 ビット長相対	4
zconst23	.zconst23		r0 からの 23 ビット長相対	
const	.const		r0 からの 32 ビット長相対	
pcconst16	.pcconst16		__pc_data シンボルからの 16 ビット長相対	
pcconst23	.pcconst23		__pc_data シンボルからの 23 ビット長相対	
pcconst32	.pcconst32		__pc_data シンボルからの 32 ビット長相対	

注 1. 自動変数の初期値は，PIROD 機能を使用しない場合は .const セクション，PIROD 機能を使用する場合は .pcconst32 セクションであり，変更することはできません。

- .const セクションは，デフォルトで使用するセクションです。
  PIROD 機能には使用できません。
- .zconst，.zconst23 セクションは，短い命令長でアクセスするためのセクションです。
  PIROD 機能には使用できません。
- .pcconst16，.pcconst23，.pconst32 セクションは，PIROD 機能で使用するためのセクションです。

デフォルトの状態で，CC-RH が関数やデータを割り当てるセクションは，.text，.data，.bss，.const の 4 つです。
アクセスする時の相対距離が長いほど，長い命令長でのアクセスとなり，コード・サイズが増大します。

拡張機能を使用して，関数やデータを任意のセクションに割り当てることができます。これにより，それぞれのセクション再配置属性の性質を利用したコードを生成することができます。

(1) #pragma section 指令
#pragma section 指令は，次の書式で記述します。

```
#pragma section data データ用属性指定文字 " ユーザ定義名 "      【V2.03.00 以降】
または
#pragma section data データ用属性指定文字                       【V2.03.00 以降】
```

- データの配置先セクションを変更します。関数，定数データは影響を受けません。

```
#pragma section const 定数データ用属性指定文字 " ユーザ定義名 "  【V2.03.00 以降】
または
#pragma section const 定数データ用属性指定文字                   【V2.03.00 以降】
```

- 定数データの配置先セクションを変更します。関数，データは影響を受けません。

- 関数の配置先セクションを変更するには，次の書式で記述します。

```
#pragma section 属性指定文字  " ユーザ定義名 "
または
#pragma section 属性指定文字
```

- 関数，またはデータ，定数データの配置先セクションを変更します。
- データ用の属性指定文字を指定した場合は，定数データの配置先をデフォルト・セクションの .const に変更します。
- 定数データ用の属性指定文字を指定した場合は，データの配置先をデフォルト・セクションの .data, .bss に変更します。

```
#pragma section ユーザ定義名
```

- 関数，データ，定数データの再配置属性をデフォルトに変更し，セクション名をユーザ定義名に従って変更します。

```
#pragma section text default  【V2.03.00 以降】
```

- 関数の配置先をデフォルト・セクションの .text に変更します。データ，定数データは影響を受けません。

```
#pragma section data default  【V2.03.00 以降】
```

- データの配置先をデフォルト・セクションの .data, .bss に変更します。関数，定数データは影響を受けません。

```
#pragma section const default 【V2.03.00 以降】
```

- 定数データの配置先をデフォルト・セクションの .const に変更します。関数，データは影響を受けません。

```
#pragma section default
または
#pragma section
```

- 関数，データ，定数データの配置先をデフォルト・セクションの .text，.data，.bss，.const に変更します。

「属性指定文字」でセクション再配置属性を，「ユーザ定義名」でセクション名をそれぞれ指定します。「属性指定文字」は，オプション指定によって使用可能なものが変化します。
指定可能な「属性指定文字」，対応するセクション再配置属性，オプション指定条件の関係を表 4.16，表 4.17，表 4.18 に示します。
「属性指定文字」は，アルファベットの大文字，小文字を区別します。

表 4.16　関数用属性指定文字とセクション再配置属性の関係

属性指定文字	セクション再配置属性	指定可能な条件
text	text	-pic 非指定時
pctext 【V1.07.00 以降】	pctext	-pic 指定時

表 4.17　データ用属性指定文字とセクション再配置属性の関係

属性指定文字	セクション再配置属性	指定可能な条件
r0_disp16	zdata/zbss	-pid 非指定時
r0_disp23	zdata23/zbss23	
r0_disp32	data/bss	
ep_disp4	tdata4/tbss4	-Xep=fix 指定時
ep_disp5	tdata5/tbss5	
ep_disp7	tdata7/tbss7	
ep_disp8	tdata8/tbss8	
ep_disp16	edata/ebss	
ep_disp23	edata23/ebss23	
ep_disp32 【V1.07.00 以降】	edata32/ebss32	-Xep=fix かつ -pid 指定時

属性指定文字	セクション再配置属性	指定可能な条件
gp_disp16	sdata/sbss	-r4=fix 指定時
gp_disp23	sdata23/sbss23	
gp_disp32 【V1.07.00 以降】	sdata32/sbss32	-r4=fix かつ -pid 指定時

表 4.18 定数データ用属性指定文字とセクション再配置属性の関係

属性指定文字	セクション再配置属性	指定可能な条件
zconst	zconst	-pirod 非指定時
zconst23	zconst23	
const	const	
pcconst16 【V1.07.00 以降】	pcconst16	-pirod 指定時
pcconst23 【V1.07.00 以降】	pcconst23	
pcconst32 【V1.07.00 以降】	pcconst32	

「ユーザ定義名」に指定可能な文字は次の通りです。

- 0 ～ 9
- a ～ z，A ～ Z
- _
- @
- .

#pragma section 指令以降に記述された関数やデータは，次のルールに従って配置先セクションが決定されます。

1. 「属性指定文字」から再配置属性を決定する。
   (1) データの場合，初期値があるかどうかによって *data* または *bss* を自動的に選択する。
   (2) 「属性指定文字」が無い形式は，関数，データの両方に作用し，コンパイラのデフォルトの再配置属性を使用する。
2. 「ユーザ定義名」の後ろに，再配置属性を示す文字列を連結する[注]。
   (1) 「ユーザ定義名」が 0 ～ 9 から始まる場合，先頭に "_" を付加する。
   (2) 「ユーザ定義名」が無い場合，デフォルト・セクション名をそのまま使用する。
   注 異なるセクション再配置属性を持つセクションに，同じセクション名を持たせないためです。
3. #pragma section default または #pragma section の形式を指定した場合は，関数，データの両方に作用し，コンパイラのデフォルトの再配置属性と，デフォルト・セクション名を使用する。

#pragma section 指令は，記述された位置から，次の #pragma section 指令が現れる位置まで，またはソース・ファイルの終端まで有効です。

例

```
#pragma section gp_disp16 "foo"
int a = 1;        /* foo.sdata */
int b;            /* foo.sbss */

#pragma section zconst23 "bar"
const int c = 2;  /* bar.zconst23 */

#pragma section text "123"
void func() {}    /* _123.text */

#pragma section baz
int d = 3;        /* baz.data */
int e;            /* baz.bss */
const int f = 4;  /* baz.const */
void func2() {}   /* baz.text */

#pragma section default
int g = 3;        /* .data */
int h;            /* .bss */
const int i = 4;  /* .const */
void func3() {}   /* .text */
```

#pragma section 指令の効果は変数と関数で異なります。

- 変数
  ある変数に対する宣言，定義が複数存在し，それぞれに異なる #pragma section を指定した場合，最初に現れた #pragma section 指定が有効になります。

  例

```
int x = 1;    /* 変数 x は foo.data に配置される */
int y = 2;    /* 変数 y は bar.data に配置される */

#pragma section foo
extern int x;

#pragma section bar
extern int x;
extern int y;
```

- 関数
  -pic オプションを指定しない場合，#pragma section は関数定義に対してのみ有効です。
  -pic オプションを指定している場合，#pragma section は変数の場合と同様に，宣言，定義の両方に対して有効です。

  【V2.02.00 以降】#pragma section を関数内にも記述できます。
  このため，V2.01.00 以前では W0520609 を出力して無視していた関数内の #pragma section が有効になります。
  この仕様変更により，関数内 static 変数や文字列リテラルが複数ある場合に，それぞれを別のセクションに割り当てることなどが可能になります。

  例

```
void func() {
  #pragma section AAA
  static int aaa;         /* AAA.bss */
  #pragma section BBB
  static int bbb;         /* BBB.bss */
  #pragma section
     ;
}
static int ccc;           /* .bss */
```

4.2.6.2 アセンブラ命令の記述

CC-RH では，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。

(1) #pragma 指令
アセンブラ命令を埋め込む #pragma 指令には，#pragma inline_asm があります。
これは関数そのものをアセンブラ命令のみとみなして，呼び出し箇所にインライン展開します。

```
#pragma inline_asm ( 関数指定 [, 関数指定 ]... )注
  関数指定 : 関数名 [(size= 数値 )]
```

注　　外側のかっこは省略可能です。

#pragma inline_asm で宣言したアセンブリ記述関数をインライン展開します。
アセンブラ埋め込みインライン関数の呼び出し規則は通常関数の呼び出し規則と同様です。
（size= 数値）を指定しても，コンパイル結果には影響しません。

例

- C ソース

```
#pragma inline_asm      func_add
static int      func_add(int a, int b){
        add     r6, r7
        mov     r7, r10
}
void func(int *p){
        *p = func_add(10,20);
}
```

- 出力コード

```
_func:
prepare r20, 0
mov     r6, r20
movea   0x0014, r0, r7
mov     10, r6
add     r6, r7
mov     r7, r10
```

(2) #pragma inline_asm 使用時の注意事項

- #pragma inline_asm は，関数本体の定義の前に指定してください。
- #pragma inline_asm で指定した関数に対しても外部定義を生成します。
- アセンブラ埋め込みインライン関数内で関数の出入口で保証するレジスタを使用する場合は，アセンブラ埋め込みインライン関数の先頭と最後でこれらのレジスタの退避／復帰が必要です。
- コンパイラは #pragma inline_asm 内に書いた文字列を，チェックも変更もなしで，そのままアセンブラに渡します。
- （size= 数値）で指定する数値は，正の整数定数のみです。数値が浮動小数点数，または負の整数定数の場合，エラーとなります。
- static 関数に #pragma inline_asm を指定した場合，関数定義はインライン展開後に削除されます。
- アセンブリ記述は，プリプロセッサの処理対象となります。このため，アセンブリ言語で使用される命令やレジスタと同じ名前のマクロ（例："MOV" や "r5" など）を #define でマクロ定義する場合は注意してください。
- RH850 のアセンブリ言語は，# で始まるコメントを使用できますが，この # コメントを使うと，プリプロセッサは前処理指令と解釈するため，アセンブリ記述関数内では # コメントを使わないでください。
- #pragma inline_asm は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma noinline，#pragma interrupt，
  #pragma block_interrupt，#pragma stack_protector
- アセンブリ記述関数内にラベルを書くと，インライン展開の数だけ同一名のラベルが作られてしまいます。
  この場合は，次のいずれかの方法で対処してください。

- アセンブリ記述のローカル・ラベルを使ってください。ローカル・ラベルはアセンブリ・ソースでは同一名ですが，アセンブラが自動的に別名に変換します。
- ラベルは 1 箇所のみに展開されるように記述してください。
  - アセンブリ記述関数を同じソース・ファイル内から呼び出す場合は，アセンブリ記述関数定義に static を指定し，1 か所のみからアセンブリ記述関数を呼び出してください。また，アセンブリ記述関数のアドレスを取得しないでください。
  - アセンブリ記述関数を同じソース・ファイル内から呼び出さない場合は，アセンブリ記述関数を外部関数としてください。

出力コード例を次に示します。

- C ソース

```
#pragma inline_asm func1
static void func1(void) /* ラベル定義を含む inline_asm 指定関数を */
{                       /* 同一ファイルから呼び出す場合は，static 関数とする */
  .PUBLIC _label1
  add 1, r6
_label1:
  add -1, r6
}

void main(void) {
  func1();              /* ラベル定義を含む inline_asm 指定関数を呼び出す */
}

#pragma inline_asm func2
void func2(void)        /* ラベル定義を含む inline_asm 指定関数を */
{                       /* 同一ファイルから呼び出さない場合は，外部関数とする */
  .PUBLIC _label2
  add -1, r6
_label2:
  add 1, r6
}
```

- 出力アセンブリ・ソース

```
_main:
  .stack _main = 0
  ._line_top inline_asm
  .PUBLIC _label1
  add 1, r6
_label1:
  add -1, r6
  ._line_end inline_asm
  jmp [r31]

_func2:
  ._line_top inline_asm
  .PUBLIC _label2
  add -1, r6
_label2:
  add 1, r6
  ._line_end inline_asm
  jmp [r31]
```

4.2.6.3 インライン展開

CC-RH 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。

(1) 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 ]... ) 注
```

注　　外側のかっこは省略可能です。

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 */
}
```

(2) 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，CC-RH 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。

(a) 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，CC-RH 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。

(b) #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
```

(c) 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。

(d) 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。

(e) 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。

(f) “ 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。

(g) 展開対象関数のアドレスを介して呼び出しを行わない
展開対象関数のアドレスを介して呼び出しを行った場合，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。

(h) -Xmerge_files を指定した場合は，同一ファイル内に記述していなくても展開される場合がある

(3) インライン展開を抑止したい関数
-Oinline オプション使用時に特定関数のインライン展開を抑止したい場合，インライン展開を抑止したい関数を，#pragma noinline で指定します。

```
#pragma noinline ( 関数名 [, 関数名 ]...) 注
```

注 外側のかっこは省略可能です。

- #pragma inline は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma noinline，#pragma interrupt，
  #pragma block_interrupt，#pragma stack_protector

- #pragma noinline は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma noinline，#pragma interrupt，
  #pragma block_interrupt

(4) 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 ” は，次のようになります。

-Oinline=0	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。
-Oinline=1	インライン展開指定した関数をインライン展開します。
-Oinline=2	インライン展開指定していない関数でも自動的にインライン展開します。 ただし，インライン展開対象外関数指定した関数は，インライン展開しません。
-Oinline=3	インライン展開指定していない関数でも自動的にインライン展開します。 ただし，インライン展開対象外関数指定した関数は，インライン展開しません。

(5) インライン展開の例
インライン展開の例は，次のようになります。

- C ソース

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

- 展開イメージ

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

4.2.6.4 割り込みレベルの制御

CC-RH では，RH850 ファミリ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。

(1) 割り込み優先順位レベルの制御
“ 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_rh(long 割り込みの優先順位レベル , void* 割り込み制御レジスタのアドレス );
```

“ 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。RH850 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，“RH850 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。

(2) マスカブル割り込みの受け付けの許可／禁止
“ 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_rh(long マスカブル割り込みの許可／禁止 , void* 割り込み制御レジスタのアドレス );
```

“ マスカブル割り込みの許可／禁止 ” に設定できる値は “-3 ～ 0” の整数値です。

設定値	動作
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）
-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）
-2	割り込みベクタ方式を直接分岐方式 (標準仕様) にしたい場合
-3	割り込みベクタ方式をテーブル参照方式 (拡張仕様) にしたい場合

注意 本機能は，-Xcpu={g3m|g3k|g3mh|g3kh} オプション指定時のみ使用可能です。その他の場合は，本機能を使用せず，割り込み制御レジスタへ値を直接書き込んでください。

4.2.6.5 割り込み／例外処理ハンドラ

CC-RH 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。

(1) 割り込み／例外の発生
RH850 ファミリでは，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。
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，RH850 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。
具体的な記述方法は「(3) 割り込み／例外ハンドラの記述方法」で説明します。

(2) 割り込み／例外発生時に行う必要のある処理
関数実行時に割り込み／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に戻る必要があります。
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。

(3) 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，CC-RH 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。CC-RH 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 ( 関数指定 [, 関数指定 ]... ) 注
  関数指定 : 関数名 [( 割り込み仕様 [, 割り込み仕様 ]...)]
```

注　　外側のかっこは省略可能です。

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。
割り込み関数は，関数の出口コードが通常関数と異なるため，通常関数のように呼び出さないでください。

(a) 割り込み仕様
割り込み仕様には，以下のものを指定できます。

enable=	多重割込の可否を指定します。true/false/manual が記載できます。 - true ei/di を出力します。 eipc/eipsw の退避／復帰コードを出力します。 - false（デフォルト） ei/di を出力しません。 eipc/eipsw の退避／復帰コードを出力しません。 - manual ei/di を出力しません。 eipc/eipsw の退避／復帰コードを出力します。
priority= channel=	priority，または channel のどちらか一方のみ記述可能です（両方書いた場合は，コンパイル・エラーとなります）。 - priority= 例外発生要因を指定します。以下の字句から 1 個のみを書くことができます。 SYSERR/FETRAP/TRAP0/TRAP1/RIE/FPP 注 1/FPI 注 1/FPINT 注 2/FPE 注 3/ FXE 注 3/ UCPOP/MIP/MDP/PIE/MAE/FENMI/FEINT/EIINT_PRIORITYX（X は 0 から 15） - channel= 割込発生チャネルを指定します。割り込みの「拡張仕様」を使う場合に選択してください。 EI ハンドラと判断してコードを生成します。 priority，または channel を記載しなかった場合は，EIINT と判断します。

fpu=	fpu のコンテキスト fpepc/fpsr を退避／復帰するか指定します。注 4true/false/auto が記載できます。 - true fpepc/fpsr を退避／復帰します。 - false fpepc/fpsr を退避／復帰しません。 - auto（デフォルト） -Xfloat=fpu オプションが指定されている場合は，true を指定したとみなします。 -Xfloat=soft の場合は，false を指定したとみなします。
fxu= 注 3	fxu のコンテキスト fxsr/fxxp を退避／復帰するか指定します。true/false/auto を指定できます。 - true fxsr/fxxp を退避／復帰します。 - false fxsr/fxxp を退避／復帰しません。 - auto（デフォルト） -Xfxu=on オプションが指定されている場合は，fxsr/fxxp を退避／復帰します。 -Xfxu=off の場合は，fxsr/fxxp を退避／復帰しません。
callt=	callt のコンテキスト ctpc/ctpsw を退避／復帰するか指定します。true/false が記載できます。 - true（デフォルト） ctpc/ctpsw を退避／復帰します。 - false ctpc/ctpsw を退避／復帰しません。
resbank 注 3	関数の出口コードに resbank 命令を出力します。また，コンテキスト退避命令列の一部を出力しません。 以下の場合は警告を出力します。 - 割り込み仕様 fpu=false を同時に指定した場合 - -Xreg_mode=22 または -Xreg_mode=common を同時に指定した場合 - -Xreserve_r2 を同時に指定した場合 - -Xep=fix を同時に指定した場合 以下の場合はエラーになります。 - 割り込み仕様 priority= に，EIINT 以外を同時に指定した場合
param=	例外要因レジスタの値を仮引数で受け取る方法を指定します。 param=( ) の中に，仮引数の個数だけ例外要因レジスタ名を指定します。 例外要因レジスタ名は 1 ～ 4 個まで指定可能であり，カンマ (,) で区切って指定します。 指定可能な例外要因レジスタ名は以下の通りです。 eiic，feic，fpsr，fxsr 注 3，fxxc 注 3，fxxp 注 3 以下の場合はエラーになります。 - 指定した例外要因レジスタ名と，仮引数の個数が合わない場合 - 同じ例外要因レジスタ名を複数回指定した場合

注 1. -Xcpu=g3mh が指定されている場合はエラーになります。

注 2. -Xcpu=g3mh が指定されていない場合はエラーになります。

注 3. -Xcpu=g4mh が指定されていない場合はエラーになります。

注 4. インプレサイス例外をサポートしない CPU コアを指定している場合は，fpepc は退避 / 復帰の対象外です。インプレサイス例外についてはデバイスのユーザーズマニュアルを参照してください。

割り込み仕様のパラメータは省略できません。
たとえば，“enable=” だけを書くとコンパイル・エラーとなります。割り込み仕様のデフォルトとは，個々の割り込み仕様を書かない場合の動作を意味します。

(b) 割り込み関数定義
割り込み関数の戻り型は，常に void 型としてください。
割り込み関数の仮引数は，割り込み仕様 param= を指定した場合は 4 個まで，指定しない場合は 1 個まで記述できます。
仮引数の型は，常に unsigned long 型としてください。
param= を指定しない場合は，EI レベル例外であれば EIIC レジスタの値が，それ以外であれば FEIC レジスタの値が仮引数に格納されます。
param= を指定した場合は，指定した内容に従って，各例外要因レジスタの値が，対応する仮引数に格納されます。

例

```
#pragma interrupt handler1 (priority=EIINT)
void handler1(unsigned long a) {  /* a = EIIC; */
    :
}

#pragma interrupt handler2
void handler2(unsigned long a) {  /* a = FEIC */
    :
}

#pragma interrupt handler3 (param=(eiic,feic,fpsr))
void handler3(unsigned long a, unsigned long b, unsigned long c) {
    /* a = EIIC, b = FEIC, c = FPSR */
    :
}
```

(c) EI レベル例外の出力コード内容
EI レベル例外の割り込み関数に対して，コンパイラが入口／出口に挿入する命令列を以下に示します。主に，EIINT や FPI 等がこれに該当します。
ただし，これらをすべての割り込み関数に挿入するわけではなく，ユーザの #pragma 記述やコンパイル・オプション等に応じて必要な処理が出力されます。

<1> [割り込み関数の入口コード]
（1）コンテキストの退避に使用するスタック領域を確保
（2）割り込み関数中で使用する関数呼び出し前後で内容が保証されないレジスタを退避
（3）EIPC，EIPSW を退避
（4）仮引数を記述した関数の場合，EIIC を R6 に設定
（5）多重割り込みを許可
（6）CTPC，CTPSW を退避
（7）FPEPC，FPSR を退避

<2> [割り込み関数の出口コード]
（8）インプレサイス割り込み待ちを設定
（9）FPEPC，FPSR を復帰
（10）CTPC，CTPSW を復帰
（11）多重割り込みを禁止
（12）EIPC，EIPSW を復帰
（13）割り込み関数中で使用した関数呼び出し前後で内容が保証されないレジスタを復帰
（14）コンテキストの退避に使用したスタック領域を解放
（15）eiret

次に，具体的な出力コード例を示します。コード例中の番号（1）～（15）は，上記の各処理に記載された番号と対応しています。
なお，必ず出力コード例のとおりの命令が出力されるわけではありません。使用する命令や汎用レジスタ等は，コード例とは異なる場合があります。

例 1. EI レベル例外の出力例 1

```
#pragma interrupt func1(enable=true, callt=true, fpu=true)
void func1(unsigned long eiic)
{
        ユーザ記述の処理；
}
```

```
_func1:
    movea   -0x00000038, r3, r3     ; (1)
    st23.dw r6, 0x00000030[r3]      ; (2)
    stsr    0, r6                   ; (3)
    stsr    1, r7                   ; (3)
    st23.dw r6, 0x00000028[r3]      ; (3)
    stsr    13, r6                  ; (4)
    ei                              ; (5)
    st23.dw r4, 0x00000020[r3]      ; (2)
    st23.dw r8, 0x00000018[r3]      ; (2)
    st23.dw r10, 0x00000010[r3]     ; (2)
    stsr    16, r8                  ; (6)
    stsr    17, r9                  ; (6)
    st23.dw r8, 0x00000008[r3]      ; (6)
    stsr    7, r8                   ; (7)
    stsr    6, r9                   ; (7)
    st23.dw r8, 0x00000000[r3]      ; (7)
    prepare ....                    ; Callee-Save レジスタ退避
        :                           ; ユーザが記述した処理
    dispose ....                    ; Callee-Save レジスタ復帰
    synce                           ; (8)
    ld23.dw 0x00000000[r3], r8      ; (9)
    ldsr    r8, 7                   ; (9)
    ldsr    r9, 6                   ; (9)
    ld23.dw 0x00000008[r3], r8      ; (10)
    ldsr    r8, 16                  ; (10)
    ldsr    r9, 17                  ; (10)
    ld23.dw 0x00000010[r3], r10     ; (13)
    ld23.dw 0x00000018[r3], r8      ; (13)
    ld23.dw 0x00000020[r3], r4      ; (13)
    di                              ; (11)
    ld23.dw 0x00000028[r3], r6      ; (12)
    ldsr    r6, 0                   ; (12)
    ldsr    r7, 1                   ; (12)
    ld23.dw 0x00000030[r3], r6      ; (13)
    movea   0x00000038, r3, r3      ; (14)
    eiret                           ; (15)
```

割り込み関数先頭にコンパイラが埋め込む入口コード

ユーザが記述した割り込み処理

割り込み関数末尾にコンパイラが埋め込む出口コー

例 2. EI レベル例外の出力例 2
仮引数なし，割り込みの多重化は手動（enable=manual）の場合

```
#pragma interrupt func1(enable=true, callt=true, fpu=true)
void func1(unsigned long eiic)
 {
        ユーザ記述の処理；
}
```

```
_func1:
    movea   -0x00000038, r3, r3     ; (1)
    st23.dw r6, 0x00000030[r3]      ; (2)
    stsr    0, r6                   ; (3)
    stsr    1, r7                   ; (3)
    st23.dw r6, 0x00000028[r3]      ; (3)
    st23.dw r4, 0x00000020[r3]      ; (2)
    st23.dw r8, 0x00000018[r3]      ; (2)
    st23.dw r10, 0x00000010[r3]     ; (2)
    stsr    16, r8                  ; (6)
    stsr    17, r9                  ; (6)
    st23.dw r8, 0x00000008[r3]      ; (6)
    stsr    7, r8                   ; (7)
    stsr    6, r9                   ; (7)
    st23.dw r8, 0x00000000[r3]      ; (7)
    prepare ....                    ; Callee-Save レジスタ退避
        :                           ; ユーザが記述した処理
    dispose ....                    ; Callee-Save レジスタ復帰
    ld23.dw 0x00000000[r3], r8      ; (9)
    ldsr    r8, 7                   ; (9)
    ldsr    r9, 6                   ; (9)
    ld23.dw 0x00000008[r3], r8      ; (10)
    ldsr    r8, 16                  ; (10)
    ldsr    r9, 17                  ; (10)
    ld23.dw 0x00000010[r3], r10     ; (13)
    ld23.dw 0x00000018[r3], r8      ; (13)
    ld23.dw 0x00000020[r3], r4      ; (13)
    ld23.dw 0x00000028[r3], r6      ; (12)
    ldsr    r6, 0                   ; (12)
    ldsr    r7, 1                   ; (12)
    ld23.dw 0x00000030[r3], r6      ; (13)
    movea   0x00000038, r3, r3      ; (14)
    eiret                           ; (15)
```

割り込み関数先頭にコンパイラが埋め込む入口コード

ユーザが記述した割り込み処理

割り込み関数末尾にコンパイラが埋め込む出口コード

(d) FE レベル例外の出力コード内容

FE レベル例外の割り込み関数に対して，コンパイラが入口／出口に挿入する命令列を以下に示します。主に，FEINT や PIE 等がこれに該当します。

ただし，これらをすべての割り込み関数に挿入するわけではなく，ユーザの #pragma 記述やコンパイル・オプション等に応じて必要な処理が出力されます。

<1> [割り込み関数の入口コード]

（1）コンテキストの退避に使用するスタック領域を確保
（2）割り込み関数中で使用する関数呼び出し前後で内容が保証されないレジスタすべてを退避
（3）仮引数を記述した関数の場合，FEIC を R6 に設定
（4）CTPC，CTPSW を退避
（5）FPEPC，FPSR を退避

<2> [割り込み関数の出口コード]

（6）FPEPC，FPSR を復帰
（7）CTPC，CTPSW を復帰
（8）割り込み関数中で使用する関数呼び出し前後で内容が保証されないレジスタすべてを復帰
（9）コンテキストの退避に使用したスタック領域を解放
（10）feret

次に，具体的な出力コード例を示します。コード例中の番号（1）～（10）は，上記の各処理に記載された番号と対応しています。

なお，必ず出力コード例のとおりの命令が出力されるわけではありません。使用する命令や汎用レジスタ等は，コード例とは異なる場合があります。

例 FE レベル例外の出力例

```
#pragma interrupt func1(priority=feint, callt=true, fpu=true)
void func1(unsigned long feic)
{
        ユーザ記述の処理 ;
}
```

```
_func1:
    movea   -0x00000030, r3, r3     ; (1)
    st23.dw r4, 0x00000028[r3]      ; (2)
    st23.dw r6, 0x00000020[r3]      ; (2)
    st23.dw r8, 0x00000018[r3]      ; (2)
    st23.dw r10, 0x00000010[r3]     ; (2)
    stsr    14, r6                  ; (3)
    stsr    16, r8                  ; (4)
    stsr    17, r9                  ; (4)
    st23.dw r8, 0x00000008[r3]      ; (4)
    stsr    7, r8                   ; (5)
    stsr    6, r9                   ; (5)
    st23.dw r8, 0x00000000[r3]      ; (5)
    prepare ....                    ; Callee-Save レジスタ退避
        :                           ; ユーザが記述した処理
    dispose ....                    ; Callee-Save レジスタ復帰
    ld23.dw 0x00000000[r3], r8      ; (6)
    ldsr    r8, 7                   ; (6)
    ldsr    r9, 6                   ; (6)
    ld23.dw 0x00000008[r3], r8      ; (7)
    ldsr    r8, 16                  ; (7)
    ldsr    r9, 17                  ; (7)
    ld23.dw 0x00000010[r3], r10     ; (8)
    ld23.dw 0x00000018[r3], r8      ; (8)
    ld23.dw 0x00000020[r3], r6      ; (8)
    ld23.dw 0x00000028[r3], r4      ; (8)
    movea   0x00000030, r3, r3      ; (9)
    feret                           ; (10)
```

FEINT 先頭にコンパイラが埋め込む入口コード

ユーザが記述した FEINT 処理

FEINT 末尾にコンパイラが埋め込む出口コード

(e) FE レベル例外（復帰／回復不可）の出力コード内容

FE レベル例外（復帰／回復不可）の割り込み関数に対して，コンパイラが入口／出口に挿入する命令列を以下に示します。主に，FENMI や SYSERR 等がこれに該当します。

<1> [割り込み関数の入口コード]

（1）仮引数を記述した関数の場合，FEIC を R6 に設定
仮引数を記述しない場合は，何も出力されません。

<2> [割り込み関数の出口コード]

なし

備考 コンテキストの退避／復帰は一切出力されません。
関数呼び出し前後で内容が保証されるレジスタの退避／復帰も出力されません。
関数内で abort() を呼び出してプログラムを終了させるなど，ユーザ・プログラムで適切に処置してください。

次に，具体的な出力コード例を示します。コード例中の番号（1）は，上記の各処理に記載された番号と対応しています。

例 FE レベル例外（復帰／回復不可）の出力例

```
#pragma interrupt func1(priority=fenmi)
void func1(unsigned long feic)
{
        ユーザ記述の処理 ;
}
```

```
_func1:
    stsr    14, r6                  ; (1)

        :                           ; ユーザが記述した処理
        :
        :
```

FENMI 先頭に
コンパイラが
埋め込む入口コード

ユーザが記述した
FENMI 処理

末尾には，
出力されません

(f) resbank 指定をした EI レベル例外の出力コード内容

resbank 指定をした EI レベル例外の割り込み関数に対して，コンパイラが入口／出口に挿入する命令列を以下に示します。
通常の EI レベル例外の割り込み関数で出力する命令列とは，次の点が異なります。

- レジスタ・バンク機能が自動的に退避するコンテキストに対する，退避用の命令列がない。
- 同コンテキストに対する，復帰用の命令列のかわりに，resbank 命令を出力する。

なお，レジスタ・バンク機能が自動的に退避するコンテキストの情報は，-Xresbank_mode オプション指定から判断します。
また，これらをすべての割り込み関数に挿入するわけではなく，ユーザの #pragma 記述やコンパイル・オプション等に応じて必要な処理が出力されます。

resbank 未指定時	resbank 指定時（-Xresbank_mode=0）

_handler:
movea 0xFFFFFFA8, r3, r3
st.w r1, 0x14[r3] ; save r1
st.w r2, 0x18[r3] ; save r2
st.w r5, 0x1C[r3] ; save r5
st23.dw r6, 0x20[r3] ; save r6,r7
st23.dw r8, 0x28[r3] ; save r8,r9
st23.dw r10, 0x30[r3] ; save r10,r11
st23.dw r12, 0x38[r3] ; save r12,r13
st23.dw r14, 0x40[r3] ; save r14,r15
st23.dw r16, 0x48[r3] ; save r16,r17
st23.dw r18, 0x50[r3] ; save r18,r19
stsr 0x11, r9, 0x00
stsr 0x10, r8, 0x00
st23.dw r8, 0x00[r3] ; save CTPC,CTPSW
stsr 0x06, r8, 0x00
st.w r8, 0x08[r3] ; save FPSR
stsr 0x06, r8, 0x0A
stsr 0x0D, r9, 0x0A
st23.dw r8, 0x0C[r3] ; save FXSR,FXXP
prepare 0x01, 0x00 ; save r31
jarl _sub, r31
dispose 0x00, 0x01 ; restore r31
ld23.dw 0x0C[r3], r8
ldsr r9, 0x0D, 0x0A ; restore FXXP
ldsr r8, 0x06, 0x0A ; restore FXSR
ld.w 0x08[r3], r8
ldsr r8, 6 ; restore FPSR
ld23.dw 0x00[r3], r8
ldsr r8, 0x10, 0x00 ; restore CTPC
ldsr r9, 0x11, 0x00 ; restore CTPSW
ld23.dw 0x50[r3], r18 ; restore r18,r19
ld23.dw 0x48[r3], r16 ; restore r16,r17
ld23.dw 0x40[r3], r14 ; restore r14,r15
ld23.dw 0x38[r3], r12 ; restore r12,r13
ld23.dw 0x30[r3], r10 ; restore r10,r11
ld23.dw 0x28[r3], r8 ; restore r8,r9
ld23.dw 0x20[r3], r6 ; restore r6,r7
ld.w 0x1C[r3], r5 ; restore r5
ld.w 0x18[r3], r2 ; restore r2
ld.w 0x14[r3], r1 ; restore r1
movea 0x58, r3, r3
eiret
	_handler:
	movea 0xFFFFFFF0, r3, r3
	
	stsr 0x11, r9, 0x00
	stsr 0x10, r8, 0x00
	st23.dw r8, 0x00[r3] ; save CTPC,CTPSW
	
	stsr 0x06, r8, 0x0A
	stsr 0x0D, r9, 0x0A
	st23.dw r8, 0x08[r3] ; save FXSR,FXXP
	prepare 0x01, 0x00 ; save r31
	jarl _sub, r31
	dispose 0x00, 0x01 ; restore r31
	ld23.dw 0x08[r3], r8
	ldsr r9, 0x0D, 0x0A ; restore FXXP
	ldsr r8, 0x06, 0x0A ; restore FXSR
	
	ld23.dw 0x00[r3], r8
	ldsr r8, 0x10, 0x00 ; restore CTPC
	ldsr r9, 0x11, 0x00 ; restore CTPSW
	resbank ; restore r1-r19,r30,EIIC,FPSR
	
	eiret

［備考］
割り込み仕様 resbank を使用しても，RBCR0 へ値を設定するコードは生成されません。ユーザ・プログラムで直接設定してください。

(4) 割り込み／例外ハンドラの記述時の注意事項

- #pragma interrupt は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma noinline，#pragma interrupt，
  #pragma block_interrupt

4.2.6.6 マスカブル割り込みの禁止／許可

CC-RH 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

(1) 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，CC-RH 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 4.19 割り込み制御関数

割り込み制御関数	動作	CC-RH 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]
```

(2) 関数全体の割り込みを禁止する方法
CC-RH 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 ]... ) 注
```

注 外側のかっこは省略可能です。

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。「(1) 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，CC-RH 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]
```

(3) 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。

- 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は無効になります。
- 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。
- #pragma block_interrupt は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma noinline，#pragma interrupt，
  #pragma block_interrupt

4.2.6.7 組み込み関数

CC-RH 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，CC-RH で用意した関数の形式で記述します。

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，強制的に型変換して組み込み関数のコードを生成します。

また，ldsr/stsr 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の動作は，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。

以下に，関数として記述できる命令を示します。

表 4.20 アセンブラ命令 (1)

アセンブラ命令	機能	組み込み関数
di	割り込み制御	`void __DI(void);`
ei		`void __EI(void);`

-	割り込みレベルの制御[注1]	`void __set_il_rh(long NUM, void* ADDR[注2]);`
		
		- NUM：1 ～ 16
		`movhi highw1(ADDR), r0, rX`
		`ld.b loww(ADDR)[rX], rY`
		`andi 0x00F0, rY, rY`
		`ori ( 優先順位 - 1), rY, rY`
		`st.b loww(ADDR)[rX]`
		- NUM：0
		`movhi highw1(ADDR), r0, rX`
		`clr1 7, loww(ADDR)[rX]`
		- NUM：-1
		`movhi highw1(ADDR), r0, rX`
		`set1 7, loww(ADDR)[rX]`
		- NUM：-2
		`movhi highw1(ADDR), r0, rX`
		`clr1 6, loww(ADDR)[rX]`
		- NUM：-3
		`movhi highw1(ADDR), r0, rX`
		`set1 6, loww(ADDR)[rX]`
		- NUM：-4 以下，および 17 以上
		範囲外エラー

nop	ノー・オペレーション	`void __nop(void);`
halt	プロセッサの停止	`void __halt(void);`
satadd	飽和加算	`long __satadd(long a, long b);`
satsub	飽和減算	`long __satsub(long a, long b);`
bsh	ハーフワード・データのバイト・スワップ	`long __bsh(long a);`
bsw	ワード・データのバイト・スワップ	`long __bsw(long a);`
hsw	ワード・データのハーフワード・スワップ	`long __hsw(long a);`
mul	符号つき 64 ビット乗算結果の上位 32 ビットを変数に代入する命令	`long __mul32(long a, long b);`

アセンブラ命令	機能	組み込み関数
mulu	符号なし 64 ビット乗算結果の上位 32 ビットを変数に代入する命令	unsigned long __mul32u(unsigned long a, unsigned long b);
sch0l	MSB 側からのビット（0）検索	long __sch0l(long a);
sch0r	LSB 側からのビット（0）検索	long __sch0r(long a);
sch1l	MSB 側からのビット（1）検索	long __sch1l(long a);
sch1r	LSB 側からのビット（1）検索	long __sch1r(long a);
ldsr	システム・レジスタへのロード	void __ldsr(long regID 注3, unsigned long a);
ldsr	システム・レジスタへのロード	void __ldsr_rh(long regID 注3, long selID 注3, unsigned long a);
stsr	システム・レジスタの内容のストア	unsigned long __stsr(long regID 注3);
stsr	システム・レジスタの内容のストア	unsigned long __stsr_rh(long regID 注3, long selID 注3);
caxi	比較と交換	long __caxi(long *a, long b, long c);
clr1	ビット・クリア	void __clr1(unsigned char *a, long bit);
set1	ビット・セット	void __set1(unsigned char *a, long bit);
not1	ビット・ノット	void __not1(unsigned char *a, long bit);
ldl.w 注4	アトミック・ロード	long __ldlw(long *a);
stc.w 注4	ストア	void __stcw(long *a, long b);【V1.04.00 以前】 long __stcw(long *a, long b);【V1.05.00 以降】
synce	例外同期化	void __synce(void);
synci	命令パイプライン同期化	void __synci(void);
syncm	メモリ同期化	void __syncm(void);
syncp	パイプライン同期化	void __syncp(void);
dbcp	デバッグ・チェックポイント	void __dbcp(void);
dbpush	デバッグ・プッシュ	void __dbpush(long regID1, long regID2);
dbtag	デバッグ・タグ	void __dbtag(long a);

注 1. __set_il_rh 関数は，-Xcpu={g3m|g3k|g3mh|g3kh} オプション指定時のみ使用可能です。

注 2. ADDR には割り込み制御レジスタのアドレスを指定してください。

注 3. regID にはシステム・レジスタ番号（0 ～ 31）を，selID には 0 ～ 31 を指定してください。

注 4. -Xcpu=g3k オプション使用時は警告を出力します。

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。

-Xcpu=g4mh オプション指定時は，次の組み込み関数も使用できます。

表 4.21 アセンブラ命令 (2)

アセンブラ命令	機能	組み込み関数
clip.b	符号付きワード・データの飽和処理付きバイト・データ変換	long __clipb(long a);
clip.bu	符号無しワード・データの飽和処理付きバイト・データ変換	unsigned long __clipbu(unsigned long a);
clip.h	符号付きワード・データの飽和処理付きハーフワード・データ変換	long __cliph(long a);
clip.hu	符号無しワード・データの飽和処理付きハーフワード・データ変換	unsigned long __cliphu(unsigned long a);
ldl.bu	アトミックなバイト・データ操作を開始するロード	long __ldlbu(unsigned char* a);
ldl.hu	アトミックなハーフワード・データ操作を開始するロード	long __ldlhu(unsigned short* a);
stc.b	バイト・データ操作がアトミックに完了した場合にストアする条件付きストア	long __stcb(unsigned char* a, unsigned char b);
stc.h	ハーフワード・データ操作がアトミックに完了した場合にストアする条件付きストア	long __stch(unsigned short* a, unsigned short b);

4.2.6.8 構造体パッキング

CC-RH 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。

(1) 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 ({1|2|4| }) 注
```

注 外側のかっこは省略可能です。

#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 です。数値を指定しない場合，デフォルトのアライメントとなります。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

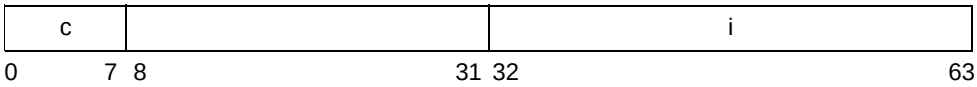
```
#pragma pack 1  /* 構造体メンバを 1 バイトのアライメントで整列 */
struct  TAG {
    char    c;
    int     i;
    short   s;
};
```

(2) 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。

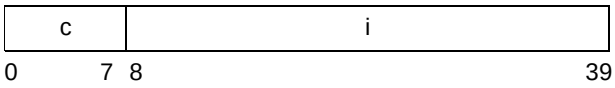
例

```
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 */
```

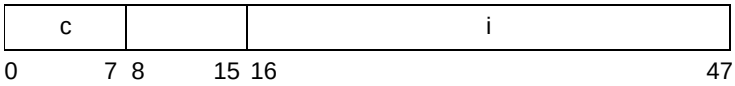
sobj



s1obj



s2obj



(3) 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 {
        char c;
        int i;
};

#pragma pack 1
union   U1 {
        char c;
        int i;
};

#pragma pack 2
union   U2 {
        char c;
        int i;
};

union   U   uobj;   /* サイズ 4 バイト */
union   U1  u1obj;  /* サイズ 4 バイト */
union   U2  u2obj;  /* サイズ 4 バイト */
```

例 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 */
```

(4) 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

(a) 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。

(b) 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

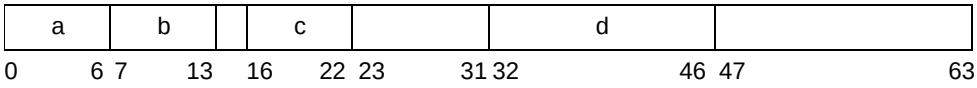
- 続けて割り当てるとその領域を含むバイト数が要素の型よりも大きくなる場合
構造体のパッキング値の整列条件を満たす形で割り当てます。

- それ以外の場合
続けて割り当てます。

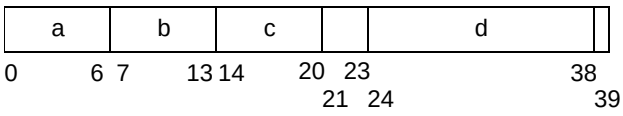
例

```
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;
```

sobj



s1obj



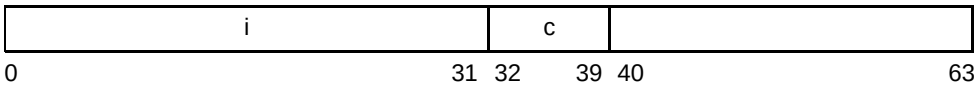
(5) 構造体オブジェクトの先頭の整列条件
構造体オブジェクトの先頭の整列条件は，構造体オブジェクトのアライメント値とパッキング値の小さい方です。

(6) 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。

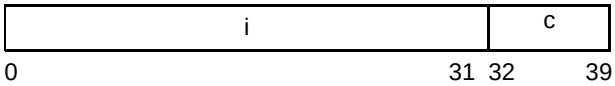
例 1.

```
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 */
```

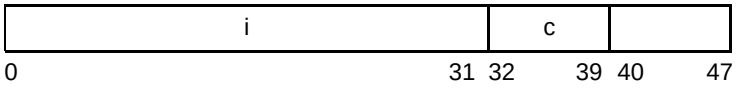
sobj



s1obj



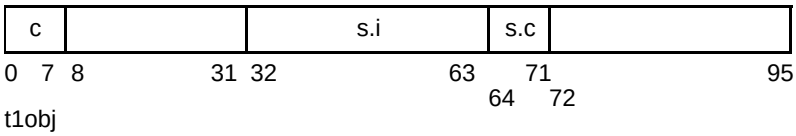
s2obj



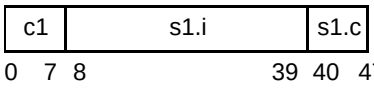
例 2.

```
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 */
```

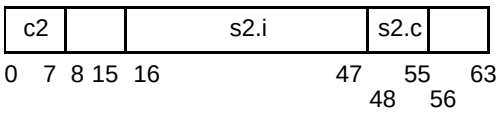
tobj



t1obj



t2obj

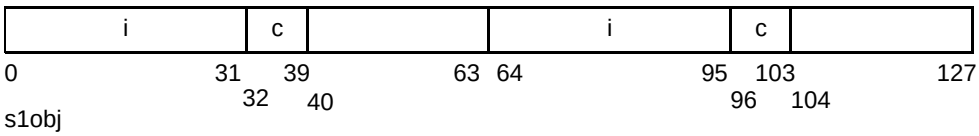


(7) 構造体配列のサイズ
構造体オブジェクトの配列のサイズは要素である構造体オブジェクトのサイズに要素数を乗算した値です。

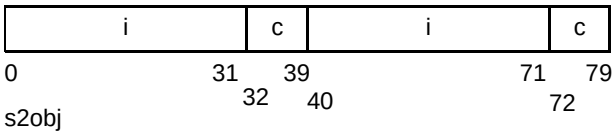
例

```
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 */
```

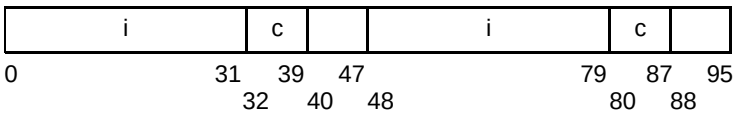
sobj



s1obj



s2obj

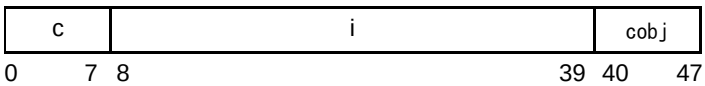


(8) オブジェクト間の領域
たとえば，次のソース・プログラムでは，sobj.c，sobj.i，cobj が隙間なく続いて配置される可能性があります（sobj，cobj の配置順は保証されません）。

例

```
#pragma pack 1
struct  S {
    char    c;
    int     i;
} sobj;
char    cobj;
```

sobj，cobj



(9) 構造体パッキング機能の注意点

(a) -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。
その後に #pragma pack（値なし）を書くと，#pragma pack（値なし）以降の行は，オプション指定値が適用されます。

例 （-Xpack=4 を指定した場合）

```
struct  S2 {...};       /* オプションでパッキング値 4 を指定している
                           -Xpack=4 オプションが有効：パッキング値 4*/
#pragma pack 2          /*#pragma 指令でパッキング 2 を指定している
struct  S1 {...};         pragma pack(2) が有効：パッキング値 2*/
#pragma pack            /*#pragma 指令でパッキング値に指定なし
struct  S2_2 {...};       -Xpack=4 オプションが有効：パッキング値 4*/
```

(b) 構造体のパッキング値とメンバの持つアライメント値
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 long 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 */
        long    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 */
        long    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 */
        long    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) #pragma pack の入れ子指定
異なる #pragma pack 指定された構造体オブジェクトの入れ子は，次のように定義します。
アライメントが異なるメンバを持つ構造体・共用体メンバに対しては，警告を出します。
警告対象のメンバは，ソース・コードに書かれたとおりの #pragma pack に従います。

例

```
#pragma pack 1
struct ST1
{
        char    c;
#pragma pack 4
        struct ST4              //size=8, align=4 ( 型としては 4)
        {
                char    c;      //offset=1
                short   s;      //offset=3
                int     i;      //offset=5
        } st4;                  //size=8, align=1 (ST1 のメンバなので 1)
                                // メンバ st4 の位置で警告
        int     i;
} st1;                          //size=13, align=1
```

[注意事項]

-Xpack=1，2 もしくは #pragma pack 1，2 を指定した構造体 , 共用体のメンバはポインタを用いてアクセスすることはできません。

```
例
#pragma   pack  1
struct st {
    char  x;
    int  y;
} ST;
int *p = &ST.y; /* ST.y のアドレスが奇数になる場合があります */
void func(void){
    ST.y =1;    /* 正しくアクセスできます */
    *p = 1;     /* 正しくアクセスできない場合があります */
}
```

4.2.6.9 ビット・フィールドの割り付け

CC-RH は，ビット・フィールドの並び順の切り替えを指定できます。

(1) ビット・フィールドの割り付け指定の形式
ビット・フィールドの割り付けは次の形式で指定します。

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

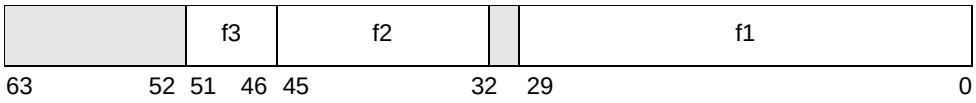
left を指定した場合は上位ビット側から，right を指定した場合は下位ビット側から，それぞれメンバが割り付けられます。
デフォルトは right となります。

例 1.

```
#pragma bit_order right
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 4.9 ビット・フィールドの内部表現

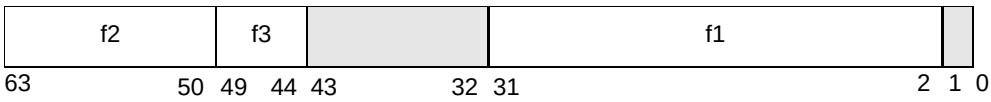


例 2.

```
#pragma bit_order left
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 4.10 ビット・フィールドの内部表現

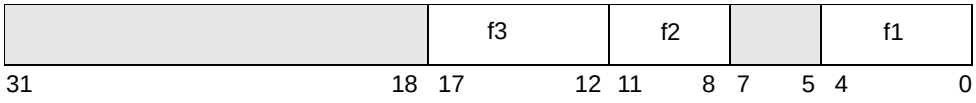


例 3.

```
#pragma bit_order right
struct {
    int             f1:5;
    char            f2:4;
    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 4.11 ビット・フィールドの内部表現

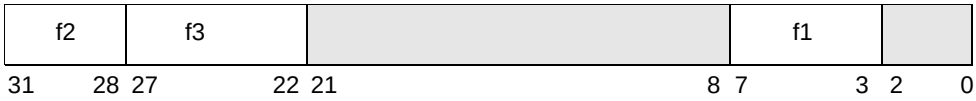


例 4.

```
#pragma bit_order left
struct {
    int             f1:5;
    char            f2:4;
    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 4.12 ビット・フィールドの内部表現



4.2.6.10 コア番号指定（マルチコア用）

コア番号指定機能とは，マルチコア利用時に，指定したコアのローカル・メモリや全コア共通のグローバル・メモリ上にデータを配置したり，指定したコアや任意のコアで関数を実行可能にする機能です。
ここで記述する #pragma とリンク・オプションを組み合わせて実現します。

例えば，ある変数 x（データ・セクションに配置されると仮定）をコア番号 1 のローカル・メモリに配置したい場合は次のようにします。

(a) 変数 x がファイル中，最も最初に定義・宣言されるより前に以下の pragma を置きます：

```
#pragma pmodule pm1
```

すると，コンパイラとアセンブラは変数 x をセクション .data.pm1 に配置します。

(b) ユーザは以下のリンク・オプションを指定します：

```
-start=.data.pm1/fe8f0000
```

すると，最適化リンカはセクション .data.pm1 をコア番号 1 のローカル・メモリ内に配置します（0xfe8f0000 はコア番号 1 のローカル・メモリ内のアドレスと仮定します）。

変数や関数にコア番号を指定することによって以下の利点が得られます。

- セクションにコア番号が付くことで，どの変数をどのコアのローカル・メモリに割り付けたか，どの関数をどのコアで実行するのか，といった対応付けを整理することができます。
  この情報は CS+ で表示することも可能です。
- デフォルト・セクションを含めたすべてのセクションにコア番号が付与されるので，コア毎にセクション名を変更する必要がなくなります。

(1) コア番号指定の形式
マルチコア利用時のコア指定は次の形式で指定します。

```
#pragma pmodule pm 指定
```

本 pragma は，-Xmulti_level=1 オプションを指定した時のみ有効です。-Xmulti_level=1 オプションを指定しない場合は，警告メッセージを出力して，コア番号指定を無視します。

pm 指定に書くことが可能な書式とその時の配置セクション名は次の表のとおりです。
pm 指定には pm1 ～ pm255，または cmn のみ書くことが可能です。pm 指定された変数および関数の配置セクション名の末尾には「.」と pm 指定した文字列が付加されます。

-Xmulti_level の値	pm 指定の値	意味	配置先セクション名
1	#pragma pmodule の指定なし	デフォルト（cmn）を設定	***.cmn
	cmn	- データの場合 全コア共通のグローバル共有メモリに配置 - 関数の場合 どのコアで実行してもよい	***.cmn
	pm*N*	コア *N* 用のデータ，または関数	***.pm*N*
0	警告メッセージを出力して，コア番号指定を無視		*** （セクション末尾に文字列を付加しない。）

- 以下に示した指定以外の pm 指定を行った場合は，コンパイル・エラーとなります。
  - cmn，pm1 ～ pm255【V1.07.00 以前】
  - cmn，pm0 ～ pm255【V2.00.00 以降】
- #pragma pmodule は，その宣言行以降に出現したすべての「静的変数の宣言」，「関数宣言」，「文字列リテラル」に作用します。注

  注　構造体，共用体，配列型自動変数の初期値データには作用しません。
- #pragma pmodule は，デフォルト・セクション名，およびユーザ指定セクション名の双方に，上に示した文字列を付加します。

  例　.data　→ .data.pm1
  mydata.data→ mydata.data.pm1
- 同一翻訳単位内に，cmn 以外の異なる #pragma pmodule 指定された変数や関数が混在する場合はエラーになります。
- pm 指定された変数，関数間の参照可能な組み合わせは次の通りです。

		参照先オブジェクト			
		pmN 指定関数	cmn 指定関数	pmN 指定関数	cmn 指定関数
参照元オブジェクト	pmN 指定関数	呼び出し可能	呼び出し可能	参照可能	参照可能
	cmn 指定関数	呼び出し不可能	呼び出し可能	R0 相対参照のみ可能	参照可能

- cmn 指定変数は，r0 相対セクションに配置する必要があります。そのため，#pragma section 指令と組み合わせて使う場合は次の属性指定文字のみ指定可能です。
  r0_disp16，r0_disp23，r0_disp32，const，zconst，zconst23，default
  cmn 指定関数には配置先セクションの制約はありません。
  pmN 指定変数，pmN 指定関数には配置先セクションの制約はありません。
- -pic，-pirod，-pid のいずれかのオプションを指定している場合，cmn 指定の変数や関数を定義するとエラーとなります。

指定例を示します。
以下はすべて，-Xmulti_level=1 オプションを指定した時の #pragma 指定例です。

例 1.

```
#pragma section r0_disp16
int     i;                      //.zbss.cmn
--------------------------------------------------------------------------------
#pragma pmodule pm2
int     i;                      //.bss.pm2
int     j = 5;                  //.data.pm2
const int       k = 10;         //.const.pm2
void func(void)                 //.text.pm2
{
        func2("abcde");         //"abcde" は .const.pm2
}
--------------------------------------------------------------------------------
#pragma pmodule pm2
#pragma section r0_disp16
int     i;                      //.zbss.pm2
--------------------------------------------------------------------------------
#pragma section r0_disp16
#pragma pmodule pm2
int     i;                      //.zbss.pm2
```

例 2.

```
extern  int     i;
#pragma pmodule pm2
int     i;                      //.bss.pm2（警告なし）
```

例 3.

```
#pragma pmodule pm2
extern  int     i;
int     i;                      //.bss.pm2（警告なし）
```

例 4.

```
#pragma pmodule cmn
extern  int     i;
#pragma pmodule pm2
int     i;                      //.bss.cmn（警告なし）
```

4.2.6.11 分岐先アドレスのアライメント指定

関数の先頭および分岐先アドレスのアライメントを 4 にします。

```
#pragma align4 ( 関数指定 [, 関数指定 ]... ) 注
  関数指定 : 関数名 [ ( 仕様 ) ]
```

注 外側のかっこは省略可能です。

- 仕様には以下のいずれかを指定します。

仕様	内容
function	関数の先頭アドレスのアライメントを 4 にします。
loop	関数の先頭とすべてのループの先頭アドレスのアライメントを 4 にします。
innermostloop	関数の先頭と最内側ループの先頭アドレスのアライメントを 4 にします。
all	関数の先頭とすべての分岐先アドレスのアライメントを 4 にします。

仕様にこれら以外を指定した場合はエラーとします。
仕様を省略した場合は function が指定されたものとみなします。

- -Xalign4 オプションと同時に指定した場合は #pragma align4 の指定が有効になります。
- #pragma align4 は，一つの関数に対して一回のみ指定できます。複数回指定するとエラーになります。
- #pragma align4 を指定しない関数を含む場合や，-Xalign4 オプションを使用しないでコンパイルして生成したオブジェクト・モジュール・ファイルや標準ライブラリとリンクした場合，リンク時に W0561322 の警告が出力される場合がありますが，動作は問題ありません。

4.2.6.12 スタック破壊検出機能 【Professional 版のみ】

スタック破壊検出機能は，-Xstack_protector オプション，-Xstack_protector_all オプションまたは本節で説明する #pragma で実現します。

```
#pragma stack_protector ( 関数指定 [, 関数指定 ]... ) 注
  関数指定 : 関数名 [ (num= 整数値 ) ]
#pragma no_stack_protector ( 関数名 [, 関数名 ]... ) 注
```

注 外側のかっこは省略可能です。

- 関数の入口・出口にスタック破壊検出コードを生成します。スタック破壊検出コードとは次に示す 3 つの処理を実行するための命令を指します。
  (1) 関数の入口で，ローカル変数領域の直前（0xFFFFFFFF 番地に向かう方向）に 4 バイトの領域を確保し，< 数値 > で指定した値を確保した領域に格納します。
  (2) 関数の出口で，< 数値 > を格納した 4 バイトの領域が書き換わっていないことをチェックします。
  (3) (2) で書き換わっている場合には，スタックが破壊されたとして __stack_chk_fail 関数を呼び出します。
- < 数値 > には 0 から 4294967295 までの整数値を指定します。< 数値 > の指定を省略した場合には，コンパイラが自動的に数値を指定します。
- __stack_chk_fail 関数はユーザが定義する必要があり，スタックの破壊検出時に実行する処理を記述します。
  __stack_chk_fail 関数を定義する際には，次の項目に注意してください。
  - 返却値の型は void 型のみであり，仮引数を持たない関数です。
  - static 指定をしないでください。
  - 通常の関数のように呼び出すことは禁止します。
  - __stack_chk_fail 関数は，オプション -Xstack_protector，-Xstack_protector_all と #pragma stack_protector によるスタック破壊検出コードの生成の対象にはなりません。
  - 関数内では abort() を呼び出してプログラムを終了させるなど，呼び出し元であるスタックの破壊を検出した関数にリターンしないようにしてください。
  - __stack_chk_fail 関数内で関数を呼び出す場合は，呼び出した先の関数内で再帰的にスタックの破壊を検出しないように注意してください。
  - PIC（「8.6 PIC/PID 機能」を参照してください）である関数に対して本機能を使用する場合，__stack_chk_fail 関数も PIC 対象としてください。
- #pragma no_stack_protector が指定された関数は -Xstack_protector オプション，-Xstack_protector_all オプションに関わらず，スタック破壊検出コードを生成しません。
- #pragma stack_protector と -Xstack_protector オプション，-Xstack_protector_all オプションが同時に使用された場合は，#pragma 指定が有効になります。
- #pragma stack_protector は，次の #pragma 指令とは同時に指定できません。
  #pragma inline_asm，#pragma inline，#pragma stack_protector，#pragma no_stack_protector
- #pragma no_stack_protector は，次の #pragma 指令とは同時に指定できません。
  #pragma stack_protector，#pragma no_stack_protector

例

- ＜C ソース＞

```
#include <stdio.h>
#include <stdlib.h>

#pragma stack_protector f1(num=1234)
void f1()   // スタックが破壊されるプログラムの例
{
    volatile char str[10];
    int i;
    for (i = 0; i <= 10; i++){
      str[i] = i; // i=10 の場合にスタックが破壊される
    }
}

void __stack_chk_fail(void)
{
    printf("stack is broken!");
    abort();
}
```

- ＜出力コード＞

```
_f1:
        .stack _f1 = 16
        add 0xFFFFFFF0, r3
        movea 0x000004D2, r0, r1  ; 指定した＜数値＞ 1234 をスタックの領域へ格納する
        st.w r1, 0x0000000C[r3]
        mov 0x00000000, r2
        br9 .BB.LABEL.1_2
.BB.LABEL.1_1:  ; bb
        movea 0x00000002, r3, r5
        add r2, r5
        st.b r2, 0x00000000[r5]
        add 0x00000001, r2
.BB.LABEL.1_2:  ; bb7
        cmp 0x0000000B, r2
        blt9 .BB.LABEL.1_1
.BB.LABEL.1_3:  ; return
        ld.w 0x0000000C[r3], r1   ; 関数の入口で＜数値＞を格納した位置からロードし，
        movea 0x000004D2, r0, r12 ; 指定した＜数値＞1234 と，
        cmp r12, r1               ; 比較する
        bnz9 .BB.LABEL.1_5        ; 異なっている場合には，分岐する
.BB.LABEL.1_4:  ; return
        dispose 0x00000010, 0x00000000, [r31]
.BB.LABEL.1_5:  ; return
        br9 ___stack_chk_fail     ; __stack_chk_fail を呼び出す

___stack_chk_fail:
        .stack ___stack_chk_fail = 4
        prepare 0x00000001, 0x00000000
        mov #.STR.1, r6
        jarl _printf, r31
        jarl _abort, r31
        dispose 0x00000000, 0x00000001, [r31]
```

4.2.6.13 半精度浮動小数点数 【Professional 版のみ】 【V1.05.00 以降】

半精度浮動小数点型（half-precision floating-point type）を使用することができます。
半精度浮動小数点型は次の特徴を持ちます。

- 型名を __fp16 とします。
- サイズは 2 バイトです。整列条件も 2 バイトです。
- データの内部表現は IEEE754-2008 の binary16 に準拠します。
  - 符号部 1 ビット，指数部 5 ビット，仮数部 10 ビット（隠れビットを含む場合は 11 ビット）
  - 指数部のバイアスは 0xf（例 1.0 は 16 進数ビット表現での 0x3c00）
- 演算は，__fp16 型同士の代入，__fp16 型から float 型への変換，float 型から __fp16 型への変換のみをサポートします。その他の演算は，float 型へ変換してから行い，その結果は float 型に対する同演算と同じ型を持ちます。__fp16 型から double 型への変換も，一度 float 型に変換してから行います。
- float 型から __fp16 型への変換時は非正規化数をサポートせず，丸めモードに従った正規化数にフラッシュします。
- 丸めモードは -Xround=nearest しかサポートしません。
- 浮動小数点定数に対する接尾語はありません。
- 関数仮引数型，関数返却型に指定できません。__fp16 型の値を関数間で受け渡す場合は，float など別の型にキャストして受け渡すか，ポインタで受け渡すか，__fp16 型をメンバに持つ構造体引数を使って受け渡します。
- 呼び出し先に仮引数型が無い場合[注]，既定の実引数拡張により float 型へ変換した後，さらに double 型に変換してから受け渡します。

  注 プロトタイプ宣言がない，仮引数並びがない，可変個数実引数，のいずれかの場合です。
- 実引数に指定した場合は，仮引数型に変換してから受け渡します。仮引数型が無い場合，既定の実引数拡張により float 型へ変換後，double 型に変換してから受け渡します。
- 構造体メンバ，共用体メンバ，配列要素に指定でます。ビットフィールドメンバには指定できません。

例

```
extern __fp16 hpvar1, hpvar2, hpvar3;
extern float fvar;
extern double dvar;
extern int ivar;

/* 外部変数定義 */
__fp16 hpvar = 1.0;

void fun() {
  /* 定数代入 */
  hpvar = 1.0;

  /* __fp16 同士の代入 */
  hpvar1 = hpvar2;

  /* 単精度浮動小数への型変換 */
  fvar = hpvar;                                    /* fvar = (float)hpvar; と同義 */

  /* 倍精度浮動小数への型変換 */
  dvar = hpvar;                                 /* dvar = (double)(float)hpvar; と同義 */

  /* 倍精度浮動小数からの型変換 */
  hpvar = dvar;                                 /* hpvar = (__fp16)(float)dvar; と同義 */

  /* 整数への型変換 */
  ivar = hpvar;                                    /* ivar = (int)(float)hpvar; と同義 */

  /* 整数からの型変換 */
  hpvar = ivar;                                 /* hpvar = (__fp16)(float)ivar; と同義 */

  /* 算術演算 */
  hpvar3 = hpvar1 + hpvar2;   /* hpvar3 = (__fp16)((float)hpvar1 + (float)hpvar2;) と
同義 */
}
```

4.2.6.14 制御レジスタへの書き込みの検出，同期化処理挿入 【Professional 版のみ】【V1.06.00 以降】

RH850 のストア命令によって複数の制御レジスタを連続して更新するとき，制御レジスタの更新順序がソース・ファイルの記述順と一致しない場合があります。更新する順序を制御したい場合は，同期化処理を挿入する必要があります。
CC-RH では，制御レジスタへの書き込みを検出して，書き込み処理の情報を表示させることや，定型の同期化処理を挿入することができます。

制御レジスタへのアクセスを示す記述は，以下の条件をすべて満たすものを対象とします。

- 1 つの整数定数を volatile 修飾型へのポインタにキャストし，単項 * 演算子や -> 演算子で間接参照している式。

```
*(volatile int*)0xffff0000 = x;
((volailte struct ST*)0xffff0004)->member = y;
```

このとき，整数定数が制御レジスタのアドレスを示します。
整数定数ではなく，変数を含む式や，複数のキャストを含む式である場合は対象にならない場合があります。

- 上記の整数定数が #pragma register_group で指定された範囲内である。

制御レジスタのアドレス範囲とグループの情報を，#pragma register_group で指定します。
#pragma register_group は次の形式で指定します。

```
#pragma register_group 開始アドレス , 終端アドレス [, id=" グループ ID"]
```

- 開始アドレス，終端アドレスは符号なし整数で指定します。8 進，10 進，16 進整数を使用できます。
  終端アドレスは，その番地がグループに所属するものとして扱います。

  例　　0x100 番地から始まる 16 バイトの領域は次のように指定します。

```
#pragma register_group 0x100, 0x10f
```

- グループ ID は，制御レジスタが所属するグループを指定するための識別子です。
  グループ ID に使用できる文字は，アルファベット（a-z，A-Z，大文字小文字を区別します），数字（0-9），アンダースコア（_）のみです。
  グループ ID の長さに制限はありません。
- アドレス空間上で連続していない同一のグループを指定するために，複数の #pragma register_group に同じグループ ID を指定することができます。
- グループ ID は省略することができます。省略した場合，その領域への書き込みは他のどの書き込みとも連続しないものとして扱います。
- 次の場合はエラーを出力します。
  - 開始アドレスが終了アドレスより大きい場合
  - グループ ID に使用できない文字を使用している場合
  - 複数の #pragma register_group で指定したアドレス範囲が重複している場合

以下の入力ソース例をもとに，使用方法を説明します。
入力ソース例

```
#pragma register_group 0xfedf0000, 0xfedfffff, id="CPU"
#pragma register_group 0xfee00000, 0xfee0ffff, id="0"

#define REG1 (*(volatile unsigned char*)0xfedf0000)    /* CPU グループの制御レジスタ */
#define REG2 (*(volatile unsigned char*)0xfedf0001)    /* CPU グループの制御レジスタ */
#define REG_Z (*(volatile unsigned short*)0xfee00000)  /* 0 グループの制御レジスタ */

void func(void) {
   REG1  = 0;
   REG2  = 1;
   REG_Z = 2;
}
```

(a) 制御レジスタへの書き込みを検出する方法
上記の例を -store_reg=list オプションを指定してコンパイルすると，次のように判定し，書き込み処理があることのメッセージを標準エラー出力へ出力します。

- REG1 と REG2 は同じグループなので，REG1 への書き込みに対する同期化処理は不要である。
- REG2 と REG_Z は異なるグループなので，REG2 への書き込みに対する同期化処理は必要である。
- REG_Z への書き込みの後，同じグループへの書き込みがあるかどうか不明なので，REG_Z への書き込みに対する同期化処理は必要である。

```
src.c(10):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0001)
src.c(11):M0536001: 制御レジスタを更新します。(id=0, 0xfee00000)
```

なお，ある制御レジスタへの書き込みの後に，関数呼び出しや，書き込み先アドレスが不明なメモリアクセスなど，制御レジスタへの書き込みと判定できない記述が現れた場合も，同期化処理が必要である，と判定します。

(b) 制御レジスタへの全ての書き込みを検出する方法
同じ例を -store_reg=list_all オプションを指定してコンパイルすると，同じグループへの連続する書き込みかどうかを判断せず，#pragma register_group で指定したすべての制御レジスタへの書き込みに対してメッセージを出力します。

```
src.c(9):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0000)
src.c(10):M0536001: 制御レジスタを更新します。(id=CPU, 0xfedf0001)
src.c(11):M0536001: 制御レジスタを更新します。(id=0, 0xfee00000)
```

(c) 制御レジスタへの書き込みの後に，同期化処理を挿入する方法
同じ例を -store_reg=sync オプションを指定してコンパイルすると，(a) と同じ判定を行い，メッセージを出力するかわりに，出力コード中に同期化処理を挿入します。同期化処理には，同じ制御レジスタからのロードと，syncp 命令を組み合わせて出力します。
出力例

```
_func:
        .stack _func = 0
        movhi 0x0000FEDF, r0, r2
        st.b r0, 0x00000000[r2]
         ; 同じグループへの書き込みが後ろにあるので，同期化処理を挿入しない
        movhi 0x0000FEDF, r0, r2
        mov 0x00000001, r5
        st.b r5, 0x00000001[r2]
        ld.bu 0x00000001[r2], r10  ; 同期化処理を挿入する
        syncp                      ;

        movhi 0x0000FEE0, r0, r2
        mov 0x00000002, r5
        st.h r5, 0x00000000[r2]
        ld.hu 0x00000002[r2], r10  ; 同期化処理を挿入する
        syncp                      ;
        jmp [r31]
```

［注意事項］

- -Xmerge_file オプションと同時に #pragma registr_group を使用する場合，ソース・ファイル間で矛盾する #pragma register_group を指定してもエラーとならず，意図しない結果になる可能性があります。#pragma register_group はインクルード・ファイルに記述して，各ソース・ファイル間で共有することを推奨します。
- 制御レジスタへの書き込みと同期化処理の間に例外が発生する可能性がある場合は，必要に応じて，例外ハンドラ内に手動で同期化処理を記述してください。
- コンパイル状況により，同じグループに属する制御レジスタへの書き込みが連続していることを検出できない場合があります。この場合，冗長に検出メッセージを出力したり，不要な同期化処理を挿入します。

4.2.7 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 RH850 ファミリに則したもので，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 CC-RH への移植と，CC-RH 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 CC-RH ＞

- #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 C90 および C99 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。

＜ CC-RH から他の C コンパイラ＞

- CC-RH 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

```
#if !defined(__CCRH__)
#define inline          /* inline 関数を通常の関数にします */
#endif
```

例 2. 他の型に変更する

```
#if !defined(__CCRH__)
#define _Bool char      /* _Bool 型変数を char 型変数にします */
#endif
```

5. アセンブラ言語仕様

この章では，CC-RH 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。

5.1 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。

5.1.1 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。

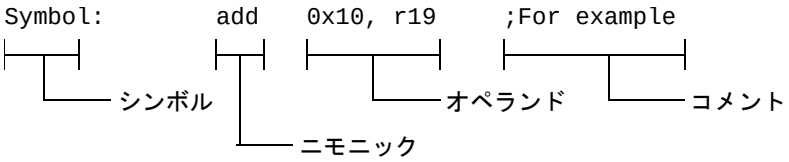
以下の箇所には，1 つ以上の空白があってもなくても問題ありません。

- 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 5.1 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 5.1 言語文字とその用途

文字	用途
数字	識別子，および定数の構成
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）	ニモニック，識別子，および定数の構成
@	識別子の構成
_（アンダースコア）	識別子の構成

文字	用途
.（ピリオド）	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
<	比較演算子
>	比較演算子
( )	演算順序の指定
$	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 識別子の構成
=	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）	各欄の区切り記号
~	コンカティネート記号（マクロボディ内で使用）
&	論理積演算子
#	ラベルの絶対アドレス参照，およびコメントの開始（行頭の場合）
[ ]	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 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。

(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
ネーム	ソース中で，数値データやアドレスとして使用	シンボル定義疑似命令のシンボル欄に記述します。
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	外部参照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	セクション定義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，「5.5 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               ; “CODE01” はセクション名
VAR01   .set    0x10        ; “VAR01” はネーム
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    0x3         ; 先頭文字に数字は使用できません。
LAB     mov     1, r10      ; “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。
```

(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。
正しい例を以下に示します。

```
mov     1, r10
```

誤った例を以下に示します。

```
mov1, r10       ; ニモニック欄とオペランド欄の間に，空白がありません。
mo v    1, r10  ; ニモニック中に空白があります。
MOVE            ; 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，各オペランドをカンマ（ , ）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
- 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。

<1> 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	数値の前に “0b” または “0B” を記述	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” または “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
```

<2> 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

```
'A'             ; 0x00000041
' '             ; 0x00000020（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 5.2 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。

<3> 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。
例を以下に示します。

```
"ab"            ; 0x6162
"A"             ; 0x41
" "             ; 0x20（空白 1 個）
```

(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。

- 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

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。

備考 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，本アセンブラでは，PC をオペランドに指定する方法はありません。

(c) 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。

(d) 式

式は，定数，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，「5.1.2 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, r10
```

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, r10” 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。

5.1.2 式と演算子

式とは，シンボル，定数，および，前述の 2 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 5.3 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 5.4 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 5.3 演算子の種類

演算子の種類	演算子
算術演算子	+，-，*，/，%，+ 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
セクション演算子	STARTOF，SIZEOF
その他の演算子	( )

上記の演算子は，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
2 項演算子	+，-，*，/，MOD（%），&，|，^，==，!=，>，>=，<，<=，>>，<<，&&，||

表 5.4 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，!，HIGH，LOW，HIGHW，HIGHW1，LOWW，STARTOF，SIZEOF
	2	*，/，%，>>，<<
	3	&，|，^
	4	+，-
低い	5	==，!=，>，>=，<，<=
	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。ただし，乗算，除算，剰余算，および，論理シフトの第 2 項は符号付き 32 ビットとして扱います。
  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，下位 32 ビットを有効として計算されます。
- 除算では，小数部分を切り捨てます。
  除算がゼロの場合は，エラーとなります。
- 負の値は，2 の補数形式となります。
- 相対値式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）。

表 5.5 式評価の例

式	評価値
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0xFFFFFFFF
EXT 注 + 1	0

注 EXT：外部参照記号

5.1.3 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
%	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:  jr      START + 6       ; (1)
```

(1) jr 命令により，「“START” 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，START ラベルが 0x100 番地の場合，“0x100 + 0x6 = 0x106” へジャンプします。

```
-
```

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:   jr      BACK - 6        ; (1)
```

(1) jr 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。

```
*
```

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, r10    ; (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, r10” 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     256 / 50, r10           ; (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 5, r10” 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，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。

［使用例］

```
mov     256 % 50, r10           ; (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, r10” 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。

5.1.4 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。

［使用例］

```
mov     !0x3, r10               ; (1)
```

(1) “0x3” の論理否定をとります。
よって，0xFFFFFFFC を返します。

!)	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。

［使用例］

```
mov     0x6FA & 0xF, r10        ; （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 “mov 0xA, r10” 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。

［使用例］

```
mov     0xA | 0b1101, r10       ; （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 “mov 0xF, r10” 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。

［使用例］

```
mov32   0x9A ^ 0x9D, r10        ; （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 “mov 0x7, r10” 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

5.1.5 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 1（真），等しくないときに 0（偽）を返します。

!=

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 1（真），等しいときに 0（偽）を返します。

```
>
```

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 1（真），等しいか小さいときに 0（偽）を返します。

>=

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 1（真），小さいときに 0（偽）を返します。

<

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 1（真），等しいか大きいときに 0（偽）を返します。

<=

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 1（真），大きいときに 0（偽）を返します。

&&

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。

5.1.6 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。

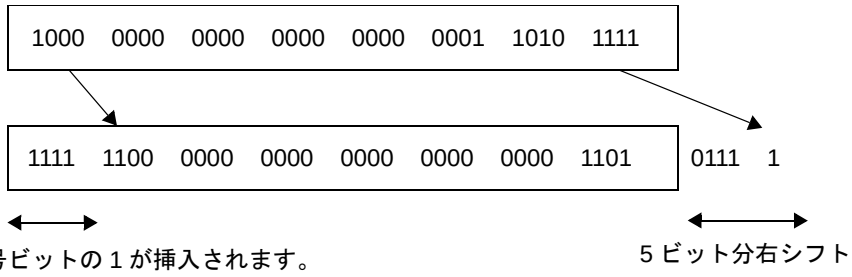
［機能］

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。

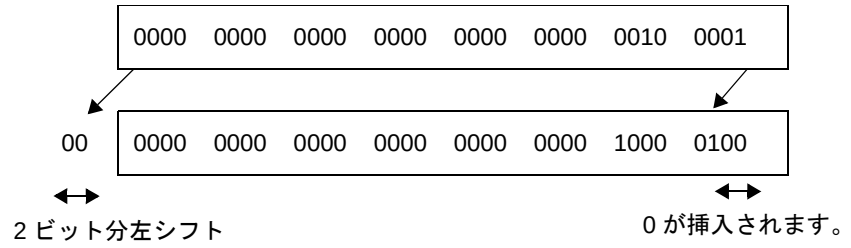
［機能］

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］

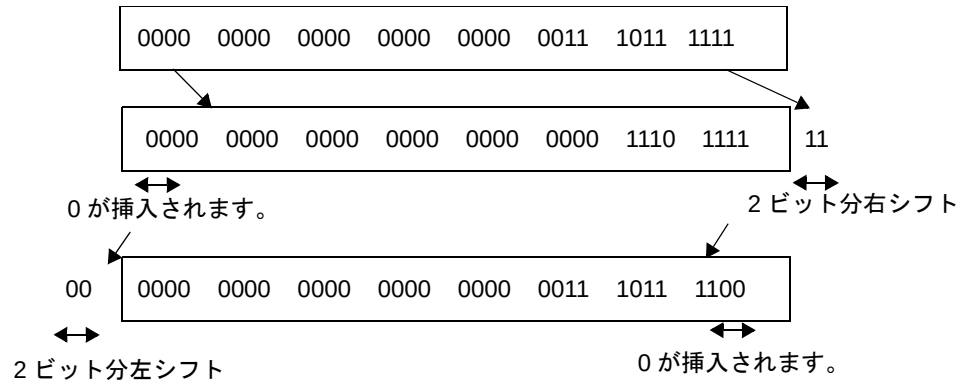
```
mov32   0x21 << 2,r20           ; (1)
```

(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。



5.1.7 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。

［使用例］

```
mov     HIGH(0xC08), r10        ; （1）
```

(1) mov 命令実行により，0xC08 の上位 8 ビット値 0xC を返します。
したがって，（1）は “mov 0xC, r10” 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。

［使用例］

```
mov     LOW(0xC08), r10         ; （1）
```

(1) mov 命令実行により，0xC08 の下位 8 ビット値 0x8 を返します。
したがって，（1）は “mov 0x8, r10” と記述することもできます。

5.1.8 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。

［使用例］

```
movea   HIGHW(0x12345678), R0, r10      ; (1)
```

(1) movea 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，“movea 0x1234, R0, r10” 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。

［使用例］

```
movea   LOWW(0x12345678), R0, r10       ; (1)
```

(1) movea 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，“movea 0x5678,R0, r10” 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。
上位 16 ビット値が 0xffff で，かつ第 15 ビットの値が 1 の場合，HIGHW1 は 0 を返します。

［使用例］

```
movhi   HIGHW1(0x12348765), R0, r10     ; (1)
```

(1) movhi 命令実行により，0x12348765 の上位 16 ビット値 0x1234 に，0x12348765 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，“movhi 0x1235, R0, r10” と記述することもできます。

5.1.9 セクション演算子

セクション演算子には，次のものがあります。

演算子	概要
STARTOF	項のセクションのリンク後の先頭アドレスを返す
SIZEOF	項のセクションのリンク後のサイズを返す

STARTOF

項のセクションのリンク後の先頭アドレスを返します。

[機能]

項のセクションのリンク後の先頭アドレスを返します。

[使用例]

```
.DW         STARTOF(.text)          ; (1)
```

(1) 4 バイト領域を確保し，.text セクションの先頭アドレスで初期化します。

SIZEOF 演算子と組み合わせて指定する場合

```
.DW         STARTOF(.data) + SIZEOF(.data)
```

[注意事項]

- STARTOF は，データ定義疑似命令 .dw のオペランドにのみ記述できます。
- 2 項演算子 + を使用して，SIZEOF 演算子と組み合わせて使用することができます。
  ただし，1 つのオペランドに STARTOF，および SIZEOF を複数記述すること，STARTOF，SIZEOF 以外の式を記述することはできません。

SIZEOF

項のセクションのリンク後のサイズを返します。

[機能]

項のセクションのリンク後のサイズを返します。

[使用例]

```
.DW     SIZEOF(.text)           ; (1)
```

(1) 4 バイト領域を確保し，.text セクションのサイズで初期化します。

STARTOF 演算子と組み合わせて指定する場合

```
.DW         STARTOF(.data) + SIZEOF(.data)
```

[注意事項]

- SIZEOF は，データ定義疑似命令 .dw のオペランドにのみ記述できます。
- 2 項演算子 + を使用して，STARTOF 演算子と組み合わせて使用することができます。
  ただし，1 つのオペランドに STARTOF，および SIZEOF を複数記述すること，STARTOF，SIZEOF 以外の式を記述することはできません。

5.1.10 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     (4 + 3) * 2, r10
```

(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 5.4 演算子の優先順位」を参照してください。

5.1.11 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，同じファイル内でシンボル定義疑似命令によって定数として定義されたもので，かつ，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。

.DW 疑似命令においては，「2 つのラベル参照」が絶対アドレス参照であれば，別ファイルで異なるセクションに定義を持っていても，アセンブル可能とします。

(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。

注 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

```
SIZE    .set    0x10
        add     #label1, r10
        add     #label1 + 0x10, r10
        add     #label2 - SIZE, r10
        add     #label2 - SIZE + 0x10, r10
```

5.1.12 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，「5.5 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

5.2 疑似命令

この節では，疑似命令について説明します。
疑似命令とは，アセンブラが一連の処理を行う際に必要な各種の指示を行うものです。

5.2.1 概　　要

インストラクションは，アセンブルの結果，機械語に変換されますが，疑似命令は，原則として機械語に変換されません。
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，最適化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 5.6　　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，.section，.org，.offset
シンボル定義疑似命令	.set，.equ
コンパイラ出力疑似命令	.file，.line，.stack，._line_top，._line_end，.dbl_size【V2.04.00 以降】
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，.weak【V2.07.00 以降】
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。

5.2.2 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，最適化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ます。
セクションは分割記述を行うことができます。つまり，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，最適化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 5.7 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・セクションの開始を指示
.dseg	アセンブラにデータ・セクションの開始を指示
.section	アセンブラにセクションの開始を指示
.org	アセンブラに絶対アドレス形式セクションの開始を指示
.offset	セクション先頭からのオフセットを設定

.cseg

アセンブラにコード・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*]	.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が現れるまでコード・セクションに属します。

[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]

- コード・セクションの開始アドレスは，.org 疑似命令により指定できます。
- 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 5.8 .cseg の再配置属性

再配置属性	説明	デフォルト・セクション名	整列条件のデフォルト値
TEXT	プログラムを配置します。	.text	2
PCTEXT【V1.07.00 以降】	位置独立のプログラムを配置します。	.pctext	2
ZCONST	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。	.zconst	4
ZCONST23	定数データ用（読み出し専用）のセクションで，r0 と 23 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 4M バイト）に配置します。	.zconst23	4
CONST	定数データ用（読み出し専用）のセクションで，r0 と 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲（r0 からプラス方向に最大 4G バイト）に配置します。	.const	4
PCCONST16【V1.07.00 以降】	位置独立の定数データ用（読み出し専用）のセクションで，__pc_data シンボルから 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（__pc_data から最大 ±32K バイト）へ配置します。	.pcconst16	4

再配置属性	説明	デフォルト・セクション名	整列条件のデフォルト値
PCCONST23【V1.07.00 以降】	位置独立の定数データ用（読み出し専用）のセクションで，__pc_data シンボルから 23 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（__pc_data から最大 ±4M バイト）へ配置します。	.pcconst23	4
PCCONST32【V1.07.00 以降】	位置独立の定数データ用（読み出し専用）のセクションで，__pc_data シンボルから 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲へ配置します。	.pcconst32	4

- 次の場合はエラーとします。
  - 「表 5.8 .cseg の再配置属性」以外の再配置属性を指定した場合
  - -pic オプション指定時に，TEXT を指定した場合
  - -pic オプションを指定せずに，PCTEXT を指定した場合
  - -pirod オプション指定時に，CONST，ZCONST，ZCONST23 のいずれかを指定した場合
  - -pirod オプションを指定せずに，PCCONST16，PCCONST23，PCCONST32 のいずれかを指定した場合
- セクション定義疑似命令を記述していない場合の，命令やデータの配置先セクションは，-pic オプションを指定していない場合は “.text” セクション，-pic オプションを指定している場合は “.pctext” セクションです。
- .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。
- デフォルトのセクション名は上記の再配置属性を持つものとし，別の再配置属性を指定することはできません。

例

```
test        .cseg   text
            nop
            nop
```

- セクション名に指定可能な文字は下記です。
  - 英数字（0 ～ 9，a ～ z，A ～ Z）
  - 英字相当文字（@，_，.）

.dseg

アセンブラにデータ・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*]	.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が現れるまで，データ・セクションに属します。

[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。

[詳細説明]

- データ・セクションの開始アドレスは，.org 疑似命令により指定できます。
- 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 5.9 .dseg の再配置属性

再配置属性	説明	デフォルト・セクション名	整列条件のデフォルト値
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（SBSS セクションとあわせて最大 64K バイト）に配置します。	.sdata	4
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。	.sbss	4
SDATA23	初期値を持ち，gp と 23 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（SBSS23 セクションとあわせて最大 8M バイト）に配置します。	.sdata23	4
SBSS23	初期値を持たず，gp と 23 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（SDATA23 セクションとあわせて最大 8M バイト）に配置します。	.sbss23	4
SDATA32 【V1.07.00 以降】	初期値を持ち，gp と 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲に配置します。	.sdata32	4
SBSS32 【V1.07.00 以降】	初期値を持たず，gp と 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲に配置します。	.sbss32	4
TDATA	初期値を持ち，ep 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 256 バイト）に配置します。[注]	.tdata	4
TDATA4	初期値を持ち，ep と 4 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 16 バイト）に配置します。	.tdata4	4

再配置属性	説明	デフォルト・セクション名	整列条件のデフォルト値
TBSS4	初期値を持たず，ep と 4 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 16 バイト）に配置します。	.tbss4	4
TDATA5	初期値を持ち，ep と 5 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 32 バイト）に配置します。	.tdata5	4
TBSS5	初期値を持たず，ep と 5 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 32 バイト）に配置します。	.tbss5	4
TDATA7	初期値を持ち，ep と 7 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 128 バイト）に配置します。	.tdata7	4
TBSS7	初期値を持たず，ep と 7 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 128 バイト）に配置します。	.tbss7	4
TDATA8	初期値を持ち，ep と 8 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 256 バイト）に配置します。	.tdata8	4
TBSS8	初期値を持たず，ep と 8 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 256 バイト）に配置します。	.tbss8	4
EDATA	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（EBSS セクションとあわせて最大 64K バイト）に配置します。	.edata	4
EBSS	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（EDATA セクションとあわせて最大 64K バイト）に配置します。	.ebss	4
EDATA23	初期値を持ち，ep と 23 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（EBSS23 セクションとあわせて最大 8M バイト）に配置します。	.edata23	4
EBSS23	初期値を持たず，ep と 23 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（EDATA23 セクションとあわせて最大 8M バイト）に配置します。	.ebss23	4
EDATA32 【V1.07.00 以降】	初期値を持ち，ep と 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲に配置します。	.edata32	4
EBSS32 【V1.07.00 以降】	初期値を持たず，ep と 32 ビットのディスプレースメントを用いて 2 命令で参照されるメモリ範囲に配置します。	.ebss32	4
ZDATA	初期値を持ち，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 からマイナス方向に ZBSS セクションとあわせて最大 32K バイト）に配置します。	.zdata	4
ZBSS	初期値を持たず，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 からマイナス方向に ZDATA セクションとあわせて最大 32K バイト）に配置します。	.zbss	4
ZDATA23	初期値を持ち，r0 と 23 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 からマイナス方向に ZBSS23 セクションとあわせて最大 4M バイト）に配置します。	.zdata23	4

再配置属性	説明	デフォルト・セクション名	整列条件のデフォルト値
ZBSS23	初期値を持たず，r0 と 23 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 からマイナス方向に ZDATA23 セクションとあわせて最大 4M バイト）に配置します。	.zbss23	4
DATA	初期値を持ち，r0 と 2 命令で参照されるメモリ範囲（r0 からマイナス方向に BSS セクションとあわせて最大 4G バイト）に配置します。	.data	4
BSS	初期値を持たず，r0 と 2 命令で参照されるメモリ範囲（r0 からマイナス方向に DATA セクションとあわせて最大 4G バイト）に配置します。	.bss	4

注 TDATA 再配置属性のセクションを複数のソース・プログラムに定義した場合，リンク時にエラーになります。

注 TDATA 再配置属性のセクション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。

- 次の場合はエラーとします。
  - 「表 5.9 .dseg の再配置属性」以外の再配置属性を指定した場合
  - 再配置属性に “BSS” の付くセクション中に，機械語命令やデータ定義疑似命令を記述した場合
  - -pid オプションを指定せずに，SDATA32，SBSS32，EDATA32，EBSS32 のいずれかを指定した場合
  - -pid オプション指定時に，DATA,BSS，ZDATA，ZBSS，ZDATA23，ZBSS23 のいずれかを指定した場合
- .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。
- デフォルトのセクション名は上記の再配置属性を持つものとし，別の再配置属性を指定することはできません。

例

```
test        .dseg   data
            .dw     0x1234
            .dw     0x5678
```

- セクション名に指定可能な文字は下記です。
  - 英数字（0 ～ 9，a ～ z，A ～ Z）
  - 英字相当文字（@，_，.）

.section

アセンブラに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
	.section	*セクション名*, *再配置属性* [, align= *絶対値式*]	[; *コメント*]

[機能]

- .section 疑似命令は，アセンブラにセクション（コード，データの区別なし）の開始を指示します。

[用途]

- コード・セクション，データ・セクションにより，.cseg 疑似命令，.dseg 疑似命令を使い分けずに，.cseg 疑似命令，または .dseg 疑似命令で定義可能なすべてのセクションを .section 疑似命令をで定義することができます。
- align パラメータを指定すると，整列条件のデフォルト値を変更することができます。【V2.03.00 以降】
- align パラメータに指定した値より大きな値を .align 疑似命令に指定した場合，エラーになります。
- align パラメータに指定できる値は，1,2,4 のいずれかです。
- align パラメータをコード・セクションに指定した場合，エラーになります。

.org

絶対アドレス形式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
	.org	*絶対式*	[; *コメント*]

[機能]

- 絶対アドレス形式セクションの開始を指示します。
- .org 疑似命令以降，再びセクション定義疑似命令が現れるまで，有効です。
- .org 疑似命令を記述した行以降，再びセクション定義疑似命令が現れるまで，絶対アドレス形式セクションとします。
- 絶対アドレス形式セクションのセクション名は，“.org 疑似命令を記述したセクション ” ＋ “.AT” ＋ “ 指定アドレス ” となり，再配置属性は .org 疑似命令を記述したセクションの属性になります。
- ソース・プログラムの先頭でセクション定義疑似命令が出現する前に，.org 疑似命令が記述された場合は，セクション名は “.text.AT”+“ 指定アドレス ”，再配置属性は “TEXT” となります。

[使用例]

セクション定義疑似命令の直後に .org 疑似命令を記述した場合は，絶対アドレス形式セクションのみ生成します。

```
.section    "My_text", text
.org        0x12                    ;"My_text.AT12" を 0x12 に配置する
mov         r10, r11
.org        0x30                    ;"My_text.AT30" を 0x30 に配置する
mov         r11, r12
```

記述が直後でない場合は，.org 疑似命令の記述行以降を絶対アドレス形式セクションとして生成します。

```
.section    "My_text", text
nop                                 ;"My_text" に配置する
.org        0x50
mov         r10, r11                ;"My_text.AT50" に配置する
```

[注意事項]

- オペランドの値は「絶対値式」に従います。値として不正な記述をした場合はエラーとし，処理を終了します。
- 1 つのセクション定義中に，複数回記述できます。ただし，.org 疑似命令で指定したセクションのアドレスが，同一ファイル内で他の絶対アドレス形式セクションの配置範囲である場合エラーとなります。
- 次の場合はエラーとします。
  - TDATA，PCTEXT，PCCONST16，PCCONST23，PCCONST32 属性のセクションに .org 疑似命令を記述した場合
  - -pid オプション指定時に，GP 相対セクション，EP 相対セクションに .org 疑似命令を記述した場合

.offset

セクション先頭からのオフセットを設定します。

[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
	.offset	*絶対式*	[; *コメント*]

[機能]

- .offset 疑似命令を記述した行以降の命令コードが格納されるセクション先頭からのオフセットを設定します。
- .offset 疑似命令以降，再びセクション定義疑似命令が現れるまで，有効です。
- ソース・プログラムの先頭でセクション定義疑似命令が出現する前に，.offset 疑似命令が記述された場合は，セクション名は “.text”，再配置属性は “TEXT” となります。
- セクション名はダブルクォーテーション（"）で囲んで指定することもできます。

例

```
.section    "My_data", data
.offset     0x12
mov         r10, r11            ;オフセットは 0x12 になる
```

[注意事項]

- オペランドの値は「絶対値式」に従います。値として不正な記述をした場合はエラーとし，処理を終了します。
- 1 つのセクション定義中に，複数回記述できます。ただし，.offset 疑似命令記述行のオフセットよりも小さい値を設定した場合はエラーとなります。
- .offset 疑似命令のオペランドに記述可能な値は，0x0 ～ 0x7fffffff までの範囲の絶対値式です。
  ただし，実際には動作するホスト・マシンのメモリ量に制限されます。
- .offset 疑似命令の行以降，指定されたオフセットまでの間の領域の初期値は 0x0 となります。
- .offset 疑似命令を，再配置属性に “BSS” が付くセクションに記述した場合は，エラーとなります。

5.2.3 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 5.10 シンボル定義疑似命令

疑似命令	概要
.set	ネームの定義
.equ	シンボルの定義

.set

ネームを定義します。

[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
ネーム	.set	*絶対式*	[; *コメント*]

[機能]

シンボル欄に指定したネーム名と，オペランド欄に指定した絶対式の値を持つネームを定義します。

[用途]

- ソース・プログラム中で使用する数値データをネームとして定義し，機械語命令や疑似命令のオペランドに数値データの代わりに記述できます。
- ソース・プログラム中で頻繁に使用する数値データはネームとして定義しておくと，ソース・プログラム中のあるデータ値を変更しなければならない場合に，そのネームのオペランド値を変更するだけですみます。

[詳細説明]

- オペランドの記述形式にエラーがある場合，処理を終了します。
- .set 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。
- ネームは再定義可能なシンボルです。
- ネームを外部定義することはできません。

[使用例]

ネーム sym1 の値を 0x10 として定義する。

```
sym1    .set 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，エラーになります。
- ラベル名，.macro 疑似命令で定義済みのマクロ名と同名のシンボルを指定した場合，エラーになります。

.equ

シンボルを定義します。

[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
シンボル	.equ	*式*	[; *コメント*]

[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絶対式または相対式 の値を持つシンボルを定義します。

[用途]

- ソース・プログラム中で使用する数値データをシンボルとして定義し，機械語命令や疑似命令のオペランドに数値データの代わりに記述できます。

[詳細説明]

- オペランドの記述形式にエラーがある場合，処理を終了します。
- .equ 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。
- すでに定義されたシンボルは再定義することはできません。
- .equ 疑似命令で生成したシンボルは .public 疑似命令で外部定義することができます。
    - .public 疑似命令で外部定義した場合は，オペランドに他のモジュールで定義したシンボルを記述できません。

5.2.4 コンパイラ出力疑似命令

コンパイラ出力疑似命令は，コンパイラのデバッグ情報などコンパイラが出力する情報をアセンブラに知らせるものです。
コンパイラ出力疑似命令には，次のものがあります。

表 5.11 コンパイラ出力疑似命令

疑似命令	概要
.file	シンボル・テーブル・エントリを生成
.line	C ソース・プログラムの行番号情報
.stack	シンボルに対するスタック使用量を定義
._line_top	#pragma inline_asm 指定情報
._line_end	#pragma inline_asm 指定情報
.dbl_size 【V2.04.00 以降】	-Xdbl_size オプションの情報をオブジェクトに埋め込む

.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]

- .file 疑似命令はコンパイラのデバッグ情報です。

[詳細説明]

- 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。

.line

C ソース・プログラムの行番号情報です。

[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
	.line	["*ファイル名*",] *行番号*	[; *コメント*]

[機能]

- .line 疑似命令はコンパイラのデバッグ情報です。

[詳細説明]

- デバッグ時に参照する行番号とファイル名を変更します。
- ソース・プログラム内の最初の .line 疑似命令 以降，次の .line 疑似命令まで行番号とファイル名を更新しません。
- ファイル名を省略した場合は，行番号だけを変更します。
- .line 疑似命令は，コンパイラが出力する C ソース・プログラムの行番号情報です。アセンブリ･ソース・ファイル上で変更しても無効です。

.stack

シンボルに対するスタック使用量を定義します。

[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
	.stack	*シンボル名 = 値*	[; *コメント*]

[機能]

- シンボルに対して，Call Walker で表示するスタック使用量を定義します。

[詳細説明]

- シンボルに対して，Call Walker で表示するスタック使用量を定義します。
- 1 つのシンボルに対して定義できるスタック使用量は 1 度のみとし，以降の定義は無視されます。
- 指定可能なスタック使用量は，0x0 ～ 0xFFFFFFFC の範囲の 4 の倍数のみとし，それ以外を指定した場合はその定義は無効となります。

[備考]

- Call Walker については，CS+ のユーザーズマニュアルを参照してください。

._line_top

コンパイラの #pragma inline_asm 指定情報です。

[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
	._line_top	inline_asm	[; コメント]

[機能]

- ._line_top 疑似命令はコンパイラの #pragma inline_asm 指定情報です。

[詳細説明]

- ._line_top 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 #pragma inline_asm 指定情報です。
- ._line_top 疑似命令は，inline_asm 指定関数における命令列の開始を意味します。

[注意事項]

- inline_asm 指定関数では，$MACRO/$NOMACRO/$WARNING/$NOWARNING 以外のアセンブラ制御命令の記述を許可しません。また疑似命令については，以下に記載した命令以外の記述はできません。下記以外の命令を記述した場合はエラーとなります。
  - データ定義疑似命令（.db/.db2/.dhw/.db4/.dw/.db8/.ddw/.dshw/.ds/.float/.double）
  - マクロ疑似命令（.macro/.irp/.rept/.local/.endm）
  - 外部定義疑似命令（.PUBLIC）【V1.05.00 以降】
- inline_asm 指定関数内の .PUBLIC 疑似命令には，inline_asm 指定関数内で定義したラベルのみを指定可能です。それ以外のものを指定した場合はエラーとなります。
- inline_asm 指定関数では命令展開は無効ですが，$MACRO 制御命令により，命令展開を有効とすることができます。なお，._line_end 疑似命令が現れた時点で，$MACRO および $NOWARNING 制御命令の効果は無効になります。

._line_end

コンパイラの #pragma inline_asm 指定情報です。

[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
	._line_end	inline_asm	[; コメント]

[機能]

- ._line_end 疑似命令はコンパイラの #pragma inline_asm 指定情報です。

[詳細説明]

- ._line_end 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 #pragma inline_asm 指定情報です。
- ._line_end 疑似命令は，inline_asm 指定関数における命令列の終了を意味します。

.dbl_size 【V2.04.00 以降】

コンパイラの -Xdbl_size オプションの情報をオブジェクトに埋め込みます。

[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
	.dbl_size	*サイズ値*	[; *コメント*]

[機能]

- .dbl_size 疑似命令は，コンパイラの -Xdbl_size オプションの情報をオブジェクトに埋め込みます。

[詳細説明]

- コンパイラの -Xdbl_size オプションの情報をオブジェクトに埋め込みます。
- 本疑似命令は，.float, .double 疑似命令の動作に影響を与えません。

5.2.5 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，機械語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 5.12 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	{ *式* |" *文字列定数* "}[, ...]	[; *コメント*]

[機能]

- 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]

- バイト領域を初期化します。

  (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
  オペランドが '“' 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，必要なバイト数分が確保されます。

- 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。
- .db 疑似命令を記述するセクションの再配置属性に “BSS” が付く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]

```
                .dseg   data
MASSAG:         .db     "ABCDEF"                ; (1)
DATA1:          .db     0xA, 0xB, 0xC           ; (2)
DATA3:          .db     "AB" + 1                ; (3)   ←エラー
```

(1) 6 バイトの領域を文字列 “ABCDEF” で初期化します。

(2) 3 バイトの領域を 0xA，0xB，0xC で初期化します。

(3) 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	*式* [, ...]	[; *コメント*]
[*ラベル* :]	.dhw	*式* [, ...]	[; *コメント*]

[機能]

- 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]

- 2 バイト領域を初期化します。

  (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 疑似命令を記述するセクションの再配置属性に “BSS” が付く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 疑似命令はセクションの再配置属性に “BSS” が付く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	*式* [, ...]	[; *コメント*]
[*ラベル* :]	.dw	*式* [, ...]	[; *コメント*]

[機能]

- 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]

- 4 バイト領域を初期化します。

  (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。

- 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に “BSS” が付く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*]
[*ラベル* :]	.ddw	*絶対式* [, ...]	[; *コメント*]

[機能]

- 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]

- 8 バイト領域を初期化します。

  (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 疑似命令はセクションの再配置属性に “BSS” が付く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 -3.40282347e+38 ～ 3.40282347e+38 の間でなければなりません。それ以外の場合は，同じ符号を持つ無限大数として確保されます。
- .float 疑似命令はセクションの再配置属性に “BSS” が付く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 -1.7976931348623157e+308 ～ 1.7976931348623157e+308 の間でなければなりません。それ以外の場合は，同じ符号を持つ無限大数として確保されます。
- .double 疑似命令はセクションの再配置属性に “BSS” が付く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*]

[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]

- 本命令を記述するセクションの再配置属性に “BSS” が付く場合には，オペランドで指定されたバイト数分の領域を確保します。その他のセクションでは，オペランドで指定されたバイト数分の領域を 0 で初期化して確保します。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
- サイズには絶対式が記述可能です。サイズの記述が不正な場合，エラーとなります。

.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。
- 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。
- 再配置属性に "BSS" が付くセクションに本疑似命令を記述し，絶対式を指定した場合は，エラーとなります。

5.2.6 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 5.13 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シンボルを宣言
.weak【V2.07.00 以降】	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

.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* [, *絶対式*]	[; *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。ただし，サイズの指定は無視されます (CX との互換性のため，記述は可能)。

  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が，同一モジュール内で定義されていない場合はワーニングを出力します。どのファイルにも定義がない場合，リンク時にエラーとなります。
- 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。
  (a) .set 疑似命令で定義したシンボル
  (b) セクション名
  (c) マクロ名

[使用例]

- モジュール 1

```
        .public A1              ; (a)
        .extern B1

A1:
        .db2    0x10

        .cseg   text
        jr      B1
```

- モジュール 2

```
        .public B1              ; (b)
        .extern A1
        .cseg   text
B1:
        mov     A1, r12
```

(a) 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。

(b) 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* [, *絶対式*]	[; *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，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。ただし，サイズの指定は無視されます (CX との互換性のため，記述は可能)。

  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。
- 第 1 オペランドには，次のものを記述することはできません。
  (a) .set 疑似命令で定義したシンボル
  (b) セクション名
  (c) マクロ名

.weak【V2.07.00 以降】

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* :]	.weak	[*シンボル名*]	[; *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。

[詳細説明]

- 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。
- .public 疑似命令と次の点で異なります。
  - 異なるモジュールに存在する同名シンボルに対して，それぞれに .public 疑似命令を指定した場合，リンク時にエラーになります。
  - 異なるモジュールに存在する同名シンボルに対して，一つに .public 疑似命令を，他に .weak 疑似命令を指定した場合，エラーにならず，.public 疑似命令を指定したモジュールがリンクされます。
- .weak 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。
- 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。

(a) .set 疑似命令で定義したシンボル

(b) セクション名

(c) マクロ名

5.2.7 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 5.14 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 : *マクロ・ボディ* : .endm	[*仮パラメータ* [, ...]]	[; *コメント*] [; *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。
- マクロを呼び出す場合は，ニモニック欄に定義済みのマクロ名を記述します。マクロ定義より先に参照を行った場合，エラーが出力されます。
- 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。
- 仮パラメータと実パラメータの個数が一致しない場合はエラーが出力されます。
- 仮パラメータの最大数は利用可能なメモリ量に依存します。
- 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。
- マクロ定義内で現在定義中のマクロの呼び出しを行った場合，エラーが出力されます。
- 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。
- 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]

- 指定したシンボル名をアセンブラ特有のシンボル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]

- シンボル名の最大数は利用可能なメモリ量に依存します。
- ラベル，またはシンボル定義疑似命令で定義したシンボルに対して，その定義名をマクロ呼び出しごとの固有な名前に置き換えます。
- シンボル名には，マクロ・ボディ内で本疑似命令以降に記述されたラベル，またはシンボル定義疑似命令で定義されたシンボルのみ指定できます。
- .local 疑似命令は，マクロ・ボディ内，REPT-ENDM ブロック内，IRP-ENDM ブロック内，および C ソースの inline_asm 関数内にのみ記述可能で，それ以外の場合はエラーを出力します。
- 1 つのブロック内に同名のローカル・シンボルを複数宣言した場合は，エラーを出力します。異なるブロックや，入れ子のブロックの内外であれば同名で宣言可能です。

[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 : .endm	*絶対式*	[; *コメント*] [; *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，その位置で展開を中止します。
- 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とエラーを出力します。
- 値は，32 ビットの符号付き整数として評価されます。
- 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。

[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 が現れると，その位置で展開を終了します。
- IRP-ENDM ブロック内に，マクロを定義を記述するとエラーを出力します。
- 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。
- 実パラメータの最大数は利用可能なメモリ量に依存します。

[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     PARA1, r12
                add     PARA2, r12
        .endm
```

(2) REPT-ENDM

```
.rept   3
        add     1, r15
        sub     r15, r16
.endm
```

(3) IRP-ENDM

```
.irp    PARA    1, 2, 3
        add     PARA, r10
        st.w    r10, [r20]
.endm
```

5.3 制御命令

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。

5.3.1 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，機械語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 5.15　　制御命令一覧

制御命令の種類	制御命令
アセンブラ制御命令	REG_MODE，NOMACRO，MACRO，DATA，SDATA，NOWARNING，WARNING
ファイル入力制御命令	INCLUDE，BINCLUDE
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。

5.3.2 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 5.16 アセンブラ制御命令

制御命令	概要
REG_MODE	レジスタ・モード情報を出力
NOMACRO	命令展開の抑制
MACRO	NOMACRO による指定を解除
DATA	シンボル名の外部データが sdata 再配置属性，および sbss 再配置属性のセクションに割り当てられていない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

REG_MODE

レジスタ・モード情報を出力します。

[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・ファイル中に，レジスタ・モード情報を出力します。

[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 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・ファイルに設定されます。
- 本制御命令でのレジスタ・モード指定と，オプションによるレジスタ・モード指定が異なる場合，CC-RH はワーニングを出力し，オプションによるレジスタ・モード指定を無視します。
- 本制御命令でのレジスタ・モード指定が複数行あり，それらのレジスタ・モード指定が異なる場合，最初のレジスタ・モード指定が有効となります。また，異なるレジスタ・モード指定に対して，CC-RH はワーニングを出力してレジスタ・モード指定を無視します。

NOMACRO

命令展開を行いません。

［指定形式］

```
[ △ ]$[ △ ]NOMACRO[ △ ][; 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

NOMACRO 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，NOMACRO による指定を解除します。

DATA

シンボル名の外部データを参照する命令に対して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を参照する命令に対して，gp を用いた 2 命令に命令展開を行います。

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，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。

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。

5.3.3 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，アセンブリ・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 5.17 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。
  (a) オプション（-I）で指定されたフォルダ
  (b) 標準インクルード・ファイル・フォルダ
  (c) ソース・ファイルのあるフォルダ
  (d) （元の）C ソース・ファイルのあるフォルダ
  (e) 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とします。

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。
  (a) オプション（-I）で指定されたフォルダ
  (b) 標準インクルード・ファイル・フォルダ
  (c) ソース・ファイルのあるフォルダ
  (d) （元の）C ソース・ファイルのあるフォルダ
  (e) 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，または，本制御命令を再配置属性に "BSS" が付くセクションに記述した場合，メッセージが出力され，アセンブルが中止されます。

5.3.4 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したがって，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 5.18 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。

- 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，ELSE 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］

- 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，ELSE 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］

- 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，ELSE 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。

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，ELSE 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。

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，ELSE 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。

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，ELSE 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。

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。

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。

5.4 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。

5.4.1 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。

5.4.2 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]
```

5.4.3 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「数字，英字相当文字からなる列」同士を連結します。
  マクロ展開時には，コンカティネート記号の左右の「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，仮パラメータを実パラメータに置換してから連結します。
- 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。

例 1.

```
abc     .macro  x
        abc~x:  mov     r10, r20
                sub     def~x, r20
.endm
abc STU
```

【展開結果】

```
abcSTU: mov     r10, r20
        sub     defSTU, r20
```

例 2.

```
abc     .macro  x, xy
        a_~xy:  mov     r10, r20
        a_~x~y: mov     r20, r10
.endm
abc stu, STU
```

【展開結果】

```
a_STU:  mov     r10, r20
a_stuy: mov     r20, r10
```

例 3.

```
abc     .macro  x, xy
        ~ab:    mov r10, r20
.endm
abc stu, STU
```

【展開結果】

```
ab:     mov     r10, r20
```

5.5 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
- デフォルト・セクション名
- GHS の予約セクション（“_GHS” と “.ghs”，および “.__ghs” で始まるセクション名）

5.6 あらかじめ定義されたマクロ名

アセンブラでは以下のマクロを定義します。

定義済みマクロ名	備考
__RENESAS__	値は 1
__ASRH__	値は 1
__ASRH	値は 1
__RH850__	値は 1
__RH850	値は 1
__PIC	値は 1。-pic オプション指定時に定義する。
__PIROD	値は 1。-pirod オプション指定時に定義する。
__PID	値は 1。-pid オプション指定時に定義する。

5.7 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。
以下のシンボルと同名のシンボルは利用できません。
アセンブラは，“.” で始まるシンボルを内部処理用のシンボルとして，オブジェクト・ファイルへは出力しません。ただし，予約セクション名は除きます。

表 5.19 アセンブラ生成シンボル

シンボル名	説明
.??00000000 ～ .??FFFFFFFF	.local 疑似命令生成ローカル・シンボル
.LMn_n （n：0 ～ 4294967294 ）	例 .LM0_1

5.8 命令セット

この節では，CC-RH 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 5.20 記号の意味

記号	意味
CMD	命令
CMDi	命令（addi, mulhi, satsubi, 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
$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	ラベルの ep オフセット参照（命令展開なし）
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
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）
selID	グループ番号（0 ～ 31）

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，機械語命令がオペランドとして r0 を指定することを禁止している場合，メッセージを出力して，アセンブルが中止されます。

```
mov     0x10, r0
    ↓
E0550240:RH850 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。
```

(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（-Xno_warning）を指定することにより抑止できます。

```
mov     0x10, r1
    ↓
W0550013 : r1 がレジスタとしてオペランドに指定されています。
```

命令展開を行う際に r1 を使用する命令を次に示します。

ld.b，ld.h，ld.w，ld.bu，ld.hu，st.b，st.h，st.w，add，addi，sub，subr，mulh，mulhi，mul，mulu，divh，div，divhu，divu，cmp，movea，cmov，satadd，satsub，satsubi，satsubr，or，ori，xor，xori，and，andi，not，tst，set1，clr1，not1，tst1，prepare，dispose

(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，.float 疑似命令および .double 疑似命令のオペランド指定には浮動小数点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 5.21 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，ld23，st23，mov，jmp 命令命令以外は，命令展開されます。
	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，命令展開されます。 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，ld23，st23，mov 命令以外は，命令展開されます。
	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）。
	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	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，命令展開されます。 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 1. リンク後のオブジェクト・ファイルにおけるアドレス 0 からのオフセットです。

注 2. リンク後のオブジェクト・ファイルにおいて，ラベル label 定義の存在する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。

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 5.22 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，ld23，st23 命令以外は，命令展開されます。#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，ld23，st23 命令以外は，命令展開されます。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。
$label[reg]	ラベル label の gp オフセットがディスプレースメントとして扱われます。ラベル label の定義されたセクションにより，32，または 16 ビットの値を持ち，命令展開のパターンが変化します注。16 ビットの値を持つ命令展開が行われた場合，ラベル label の定義したアドレスより算出されたオフセットが 16 ビットで表現できる範囲でない場合，リンク時にエラーになります。$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 5.23 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，命令展開されます。
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，命令展開されます。
$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。
!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に指定した場合，命令展開は行われません。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に指定した場合，命令展開は行われません。 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。

表 5.24 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，命令展開されます。
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 5.25 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。
label	.db4/.db2/.db 疑似命令において，ラベル label 定義の定義されたセクション内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。
%label	.db4/.db2/.db 疑似命令において，ラベル label の ep 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。
$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。

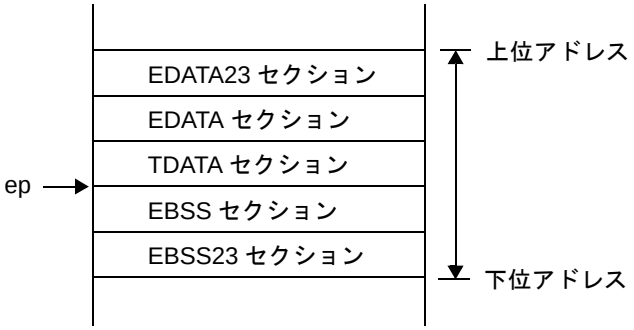
(g) ep オフセット参照

ここでは，ep オフセット参照について説明します。CC-RH では，以下に示す再配置属性のセクション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

- TDATA/TDATA4/TBSS4/TDATA5/TBSS5/TDATA7/TBSS7/TDATA8/TBSS8 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）
- EDATA/EBSS 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）
- EDATA23/EBSS23 セクション（コード・サイズが大きいメモリ参照命令（ld23/st23）で参照するデータ）

図 5.2 ep オフセット参照セクションのメモリ配置イメージ



<1> データの割り当て

ep オフセット参照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，“ep_” で始まる属性指定文字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，再配置属性 tdata，tdata4，tbss4，tdata5，tbss5，tdata7，tbss7，tdata8，tbss8，edata，ebss，edata23，または ebss23 のセクションへデータを割り当てます。

<2> データの参照

%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列が生成されます。

例

```
        .dseg   EDATA
sdata:  .db2    0xFFF0
        .dseg   DATA
data:   .db2    0xFFF0
        .cseg   TEXT
        ld.h    %sdata, 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），（2）の両方とも ep オフセット参照とみなし，機械語の命令列を生成します。
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。

(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。CC-RH では，以下に示す再配置属性のセクション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。

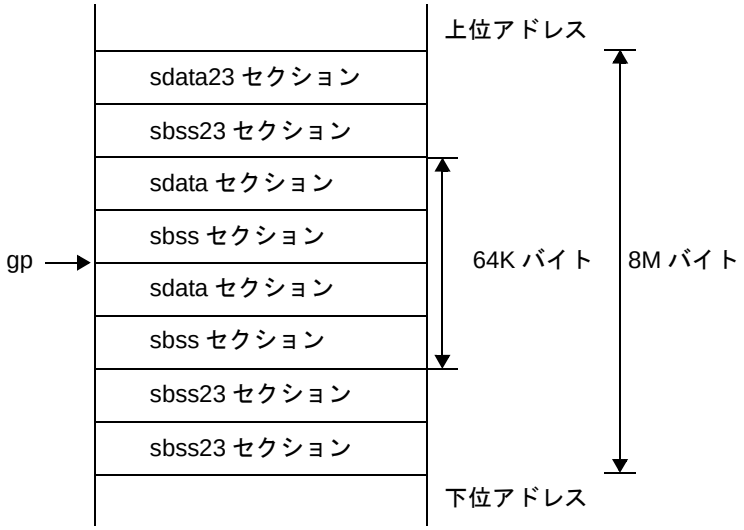
グローバル・ポインタ（gp）の指すアドレスからのオフセットによって参照する

- SDATA/SBSS セクション（16 ビットのディスプレースメントをとるメモリ参照命令 (ld/st）で参照するデータ）
- SDATA23/SBSS23 セクション（23 ビットのディスプレースメントをとるメモリ参照命令（ld23/st23）で参照するデータ）

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。

<1> データの割り当て

図 5.3 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 セクションと sbss セクションをあわせて 64K バイトです。gp は sdata セクションと sbss セクションの中心の位置です。

- 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/sdata23/sbss23 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。

<2> データの参照

- 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 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 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。
- 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
sdata/sbss セクションに割り当てるデータである（gp オフセット参照されているラベルが sdata/sbss セクションに定義を持つ）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) HIGH/LOW/HIGHW/LOWW/HIGHW1 演算子について

<1> 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
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）

<2> 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 5.26 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 ($L1), r0, r10    ;L1 の gp オフセットの値の上位 16 ビットを
                                        ;r10 の上位 16 ビットに格納し，
                                        ; 位 16 ビットに 0 を格納する。
        movea   LOWW ($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。
            :
```

5.9 アセンブリ言語の拡張

アセンブラがサポートするアセンブラ命令，および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。

本節では，デバイスの仕様から変更，または拡張している命令仕様について説明します。

ld, st

- デバイスの ld，st 命令には次のものがあり，それぞれ disp16，disp23 のオペランドを持ちます。
  - (1) LD.B, LD.BU, LD.H, LD.HU, LD.W
  - (2) ST.B, ST.H, ST.W

  asrh では，これらの命令に対して disp23 を明示的に指定したい場合，次のニモニックを指定する必要があります。
  - (1) ld23.b, ld23.bu, ld23.h, ld23.hu, ld23.w
  - (2) st23.b, st23.h, st23.w

- また，デバイスの ld，st 命令には次のものがあります。
  - (3) LD.DW
  - (4) ST.DW

  これらの命令に対しては，asrh では，次のどちらの形式も指定できます。意味は同じです。
  - (3) ld.dw, ld23.dw
  - (4) st.dw, st23.dw

- disp16 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 の範囲の範囲を越え -4194304 ～ 4194303 の範囲の絶対値式

指定形式	アセンブル結果
`ld.w disp[reg1], reg2`	`ld.w disp23[reg1], reg2`

(b) -4194304 ～ +4194303 の範囲を越える絶対値式

指定形式	アセンブル結果
`ld.w disp[reg1], reg2`	`movhi HIGHW1(disp), reg1, r1` `ld.w LOWW(disp)[r1], reg2`

(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

指定形式	アセンブル結果
`ld.w #label[reg1], reg2`	`movhi HIGHW1(#label), reg1, r1` `ld.w LOWW(#label)[r1], reg2`
`ld.w label[reg1], reg2`	`movhi HIGHW1(label), reg1, r1` `ld.w LOWW(label)[r1], reg2`
`ld.w $label[reg1], reg2`	`movhi HIGHW1($label), reg1, r1` `ld.w 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 に絶対値式，!label を持つ相対値式，#label を持つ相対値式，または #label を持つ相対値式に LOWW 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 LOWW 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 %label を持つ相対値式を指定した場合，その後ろの [reg1] の部分を省略できます。だだし，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。

sld, sst

- デバイスの sld，sst 命令は次の形式です。
  - SLD.* dispN [ep] , reg2
  - SST.* reg2, dispN [ep]

- asrh では，dispN に相対値式を指定する場合，%label の形式で指定してください。
- また，[ep] を省略することができます。

```
sld.b  %_sym+2, r10          ;  sld.b  %_sym+2[ep], r10 と同じ
```

add, mulh

- デバイスの add，mulh 命令は次の形式です。
  - ADD　reg1, reg2
  - ADD　imm5, reg2
  - MULH　reg1, reg2
  - MULH　imm5, reg2

- “add imm, reg2”，“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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`add imm, reg`	`movhi HIGHW(imm), r0, r1` `add r1, reg`

上記以外の場合

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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`

addi, mulhi

- デバイスの addi，mulhi 命令は次の形式です。
  - ADDI imm16, reg1, reg2
  - MULHI imm16, reg1, reg2

- 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

<1> imm の値の下位 16 ビットがすべて 0 の場合
reg2 が r0 の場合（addi でのみ指定できます。mulhi ではエラーになります。）

指定形式	アセンブル結果
`addi imm, reg1, r0`	`movhi HIGHW(imm), r0, r1` `add reg1, r1`

reg1=reg2 の場合

指定形式	アセンブル結果
`addi imm, reg1, reg2`	`movhi HIGHW(imm), r0, r1` `add r1, reg2`

上記以外の場合

指定形式	アセンブル結果
`addi imm, reg1, reg2`	`movhi HIGHW(imm), r0, reg2` `add reg1, reg2`

<2> 上記以外の場合
reg2 が r0 の場合（addi でのみ指定できます。mulhi ではエラーになります。）

指定形式	アセンブル結果
`addi imm, reg1, r0`	`mov imm, r1` `add reg1, r1`

reg1=reg2 の場合

指定形式	アセンブル結果
`addi imm, reg1, reg2`	`mov imm, r1` `add r1, reg2`

上記以外の場合

指定形式	アセンブル結果
`addi imm, reg1, reg2`	`mov imm, reg2` `add 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
reg2 が r0 の場合（addi でのみ指定できます。mulhi ではエラーになります。）

指定形式	アセンブル結果
`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`

reg1=reg2 の場合

指定形式	アセンブル結果
`addi #label, reg1, reg2`	`mov #label, r1` `add r1, reg2`
`addi label, reg1, reg2`	`mov label, r1` `add r1, reg2`
`addi $label, reg1, reg2`	`mov $label, r1` `add r1, reg2`

上記以外の場合

指定形式	アセンブル結果
`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`

adf, sbf, sasf, setf

- デバイスの adf，sbf，sasf，setf 命令は次の形式です。
  - ADF　cccc, reg1, reg2, reg3
  - SBF　cccc, reg1, reg2, reg3
  - SASF　cccc, reg2
  - SETF　cccc, reg2

- asrh では，上記に加えて，次の形式も指定できます。
  - adf*cond*　reg1, reg2, reg3
  - sbf*cond*　reg1, reg2, reg3
  - sasf*cond*　reg2
  - setf*cond*　reg2

- setf 命令を例として，*cond* に指定可能な記述と，意味を「表 5.27　setfcond 命令」に示します。

表 5.27　setf*cond* 命令

命令	フラグ状態	フラグ状態の意味	アセンブル結果
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
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 注

注　adf，sbf 命令では，sa（0xD）は指定できません。エラーになります。

mul

- デバイスの mul 命令は次の形式です。
  - MUL reg1, reg2, reg3
  - MUL imm9, reg2, reg3

- “mul imm9, 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) -256 ～ +255 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

指定形式	アセンブル結果
`mul imm16, reg2, reg3`	`movea imm16, r0, r1` `mul 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

指定形式	アセンブル結果
`mul imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mul r1, reg2, reg3`

上記以外の場合

指定形式	アセンブル結果
`mul imm, reg2, reg3`	`mov imm, r1` `mul 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つラベルの $label を持つ相対値式

指定形式	アセンブル結果
`mul !label, reg2, reg3`	`movea !label, r0, r1` `mul r1, reg2, reg3`
`mul %label, reg2, reg3`	`movea %label, r0, r1` `mul r1, reg2, reg3`
`mul $label, reg2, reg3`	`movea $label, r0, r1` `mul 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たないラベルの $label を持つ相対値式

指定形式	アセンブル結果
`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`

mulu

- デバイスの mulu 命令は次の形式です。
  - MULU reg1, reg2, reg3
  - MULU imm9, reg2, reg3

- “mulu imm9, 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 ～ -1 の範囲の絶対値式

指定形式	アセンブル結果
`mulu imm5, reg2, reg3`	`mov imm5, r1` `mulu r1, reg2, reg3`

(b) -16 ～ 511 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

指定形式	アセンブル結果
`mulu imm16, reg2, reg3`	`movea imm16, r0, r1` `mulu r1, reg2, reg3`

(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

指定形式	アセンブル結果
`mulu imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `mulu r1, reg2, reg3`

上記以外の場合

指定形式	アセンブル結果
`mulu imm, reg2, reg3`	`mov imm, r1` `mulu 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

指定形式	アセンブル結果
`mulu !label, reg2, reg3`	`movea !label, r0, r1` `mulu r1, reg2, reg3`
`mulu %label, reg2, reg3`	`movea %label, r0, r1` `mulu r1, reg2, reg3`
`mulu $label, reg2, reg3`	`movea $label, r0, r1` `mulu r1, reg2, reg3`

(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

指定形式	アセンブル結果
`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`

divh

- デバイスの divh 命令は次の形式です。
  - DIVH reg1, reg2
  - DIVH reg1, reg2, reg3

- asrh では，上記に加えて，次の形式も指定できます。
  - divh imm, reg2
  - divh imm, reg2, reg3

- “divh 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) 0 以外の -16 ～ +15 の範囲の絶対値式

指定形式	アセンブル結果
`divh imm5, reg`	`mov imm5, r1` `divh r1, reg`

(b) -16 ～ +15 の範囲を越え，-32768 ～ +32767 の範囲の絶対値式

指定形式	アセンブル結果
`divh imm16, reg`	`movea imm16, r0, r1` `divh r1, reg`

(c) 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

指定形式	アセンブル結果
`divh imm, reg`	`movhi HIGHW(imm), r0, r1` `divh r1, reg`

上記以外の場合

指定形式	アセンブル結果
`divh imm, reg`	`mov imm, r1` `divh 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つラベルの $label を持つ相対値式

指定形式	アセンブル結果
`divh !label, reg2`	`movea !label, r0, r1` `divh r1, reg`
`divh %label, reg2`	`movea %label, r0, r1` `divh r1, reg`
`divh $label, reg2`	`movea $label, r0, r1` `divh r1, 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

指定形式	アセンブル結果
`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`

- “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

指定形式	アセンブル結果
`divh 0, reg2, reg3`	`divh r0, reg2, reg3`

(b) 0 以外の -16 ～ +15 の範囲の絶対値式

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`divh imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `divh r1, reg2, reg3`

上記以外の場合

指定形式	アセンブル結果
`divh imm, reg2, reg3`	`mov imm, r1` `divh r1, reg2, reg3`

(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つラベルの $label を持つ相対値式

指定形式	アセンブル結果
`divh !label, reg2, reg3`	`movea !label, r0, r1` `divh r1, reg2, reg3`
`divh %label, reg2, reg3`	`movea %label, r0, r1` `divh r1, reg2, reg3`
`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

指定形式	アセンブル結果
`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`

div, divhu, divu

- デバイスの div，divhu，divu 命令は次の形式です。
  - DIV reg1, reg2, reg3
  - DIVHU reg1, reg2, reg3
  - DIVU reg1, reg2, reg3

- asrh では，上記に加えて，次の形式も指定できます。
  - div imm, reg2, reg3
  - divhu imm, reg2, reg3
  - divu imm, reg2, reg3

- “div imm, reg2, reg3”，“divhu imm, reg2, reg3”，“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。

(a) 0

指定形式	アセンブル結果
`div 0, reg2, reg3`	`div r0, reg2, reg3`

(b) 0 以外の -16 ～ +15 の範囲の絶対値式

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`div imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `div r1, reg2, reg3`

上記以外の場合

指定形式	アセンブル結果
`div imm, reg2, reg3`	`mov imm, r1` `div r1, reg2, reg3`

(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つラベルの $label を持つ相対値式

指定形式	アセンブル結果
`div !label, reg2, reg3`	`movea !label, r0, r1` `div r1, reg2, reg3`
`div %label, reg2, reg3`	`movea %label, r0, r1` `div r1, reg2, reg3`
`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

指定形式	アセンブル結果
`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`

cmp

- デバイスの cmp 命令は次の形式です。
  - CMP reg1, reg2
  - CMP imm5, reg2

- “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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`cmp imm, reg`	`movhi HIGHW(imm), r0, r1` `cmp r1, reg`

上記以外の場合

指定形式	アセンブル結果
`cmp imm, reg`	`mov imm, r1` `cmp 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

指定形式	アセンブル結果
`cmp !label, reg`	`movea !label, r0, r1` `cmp r1, reg`
`cmp %label, reg`	`movea %label, r0, r1` `cmp r1, reg`
`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

指定形式	アセンブル結果
`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`

mov

- デバイスの mov 命令は次の形式です。
  - MOV reg1, reg2
  - MOV imm5, reg2
  - MOV imm32, reg1

- asrh では，上記に加えて，次の形式も指定できます。
  - mov32 imm32, reg1

  32 ビット長の imm を持つ 48 ビット長命令を，明示的に使用したい場合に指定します。

- “mov imm, reg2” の形式で reg2 に r0 を指定した場合，アセンブラでは，機械語命令の 48 ビット長の mov 命令が 1 つ生成されます。

- “mov imm, reg2” の形式で reg2 に r0 以外を指定し，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 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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`mov imm, reg`	`movhi HIGHW(imm), r0, reg`

上記以外の場合[注]

指定形式	アセンブル結果
`mov imm, reg`	`mov imm32, 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

指定形式	アセンブル結果
`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]

指定形式	アセンブル結果
`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。

movea

- デバイスの movea 命令は次の形式です。
  - MOVEA imm16, reg1, reg2

- 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

指定形式	アセンブル結果
`movea imm, reg1, reg2`	`movhi HIGHW(imm), reg1, reg2`

上記以外の場合

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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`

cmov

- デバイスの cmov 命令は次の形式です。
  - CMOV cccc, reg1, reg2, reg3
  - CMOV cccc, imm5, reg2, reg3

- asrh では，上記に加えて，次の形式も指定できます。
  - cmov*cond* reg1, reg2, reg3
  - cmov*cond* imm, reg2, reg3

  cond に指定できる記述，および意味は setf 命令と同じです。詳細は「表 5.27 setfcond 命令」を参照してください。

- “cmov cccc, imm, reg2, reg3”，または “cmov*cond* 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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`cmov imm4, imm, reg2, reg3`	`movhi HIGHW(imm), r0, r1` `cmov imm4, r1, reg2, reg3`

上記以外の場合

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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`

satadd

- デバイスの satadd 命令は次の形式です。
  - SATADD reg1, reg2
  - SATADD imm5, reg2
  - SATADD reg1, reg2, reg3

- “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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`satadd imm, reg`	`movhi HIGHW(imm), r0, r1` `satadd r1, reg`

上記以外の場合

指定形式	アセンブル結果
`satadd imm, reg`	`mov imm, r1` `satadd 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

指定形式	アセンブル結果
`satadd !label, reg`	`movea !label, r0, r1` `satadd r1, reg`
`satadd %label, reg`	`movea %label, r0, r1` `satadd r1, reg`
`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

指定形式	アセンブル結果
`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`

satsub

- デバイスの satsub 命令は次の形式です。
  - SATSUB reg1, reg2
  - SATSUB reg1, reg2, reg3

- asrh では，上記に加えて，次の形式も指定できます。
  - satsub imm, reg2

- “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。

(a) 0

指定形式	アセンブル結果
`satsub 0, reg`	`satsub r0, reg`

(b) -32768 ～ +32767 の範囲の絶対値式

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`satsub imm, reg`	`movhi HIGHW(imm), r0, r1` `satsub r1, reg`

上記以外の場合

指定形式	アセンブル結果
`satsub imm, reg`	`mov imm, r1` `satsub r1, reg`

(d) !label，または %label を持つ相対値式，および 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

指定形式	アセンブル結果
`satsub !label, reg`	`satsubi !label, reg, reg`
`satsub %label, reg`	`satsubi %label, reg, reg`
`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

指定形式	アセンブル結果
`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`

satsubi

- デバイスの satsubi 命令は次の形式です。
  - SATSUBI imm16, reg1, reg2

- 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

<1> imm の値の下位 16 ビットがすべて 0 の場合
reg1=reg2 の場合

指定形式	アセンブル結果
satsubi imm, reg1, reg2	movhi HIGHW(imm), r0, r1 satsub r1, r2

reg1 ≠ reg2 の場合

指定形式	アセンブル結果
satsubi imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 satsubr reg1, reg2

<2> 上記以外
reg1=reg2 の場合

指定形式	アセンブル結果
satsubi imm, reg1, reg2	mov imm, r1 satsub r1, reg2

reg1 ≠ reg2 の場合

指定形式	アセンブル結果
satsubi imm, reg1, reg2	mov imm, reg2 satsubr 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
reg1=reg2 の場合

指定形式	アセンブル結果
satsubi #label, reg1, reg2	mov #label, r1 satsub r1, reg2
satsubi label, reg1, reg2	mov label, r1 satsub r1, reg2
satsubi $label, reg1, reg2	mov $label, r1 satsub r1, reg2

eg1 ≠ reg2 の場合

指定形式	アセンブル結果
`satsubi #label, reg1, reg2`	`mov #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`

and, or, xor

- デバイスの and，or，xor 命令は次の形式です。
  - AND reg1, reg2
  - OR reg1, reg2
  - XOR reg1, reg2

- asrh では，上記に加えて，次の形式も指定できます。
  - and imm, reg2
  - or imm, reg2
  - xor imm, reg2

- “and imm, reg2”，“or imm, reg2”，“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。

(a) 0

指定形式	アセンブル結果
`and 0, reg`	`and r0, reg`

(b) 1 ～ 65535 の範囲の絶対値式

指定形式	アセンブル結果
`and imm16, reg`	`andi imm16, reg, reg`

(c) -16 ～ -1 の範囲の絶対値式

指定形式	アセンブル結果
`and imm5, reg`	`mov imm5, r1` `and r1, reg`

(d) -32768 ～ -17 の範囲の絶対値式

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`and imm, reg`	`movhi HIGHW(imm), r0, r1` `and r1, reg`

上記以外の場合

指定形式	アセンブル結果
`and imm, reg`	`mov imm, r1` `and r1, reg`

(f) !label，または %label を持つ相対値式

指定形式	アセンブル結果
`and !label, reg`	`andi !label, reg, reg`
`and %label, reg`	`andi %label, reg, reg`

(g) 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

指定形式	アセンブル結果
`and $label, reg`	`movea $label, r0, r1` `and r1, reg`

(h) #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

指定形式	アセンブル結果
`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`

andi, ori, xori

- デバイスの andi，ori，xori 命令は次の形式です。
  - ANDI imm16, reg1, reg2
  - ORI imm16, reg1, reg2
  - XORI imm16, reg1, reg2

- 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

指定形式	アセンブル結果
`andi imm5, reg1, r0`	`mov imm5, r1` `and reg1, r1`

reg1=reg2 の場合

指定形式	アセンブル結果
`andi imm5, reg1, reg2`	`mov imm5, r1` `and r1, reg2`

上記以外の場合

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`andi imm16, reg1, r0`	`movea imm16, r0, r1` `and reg1, r1`

reg1=reg2 の場合

指定形式	アセンブル結果
`andi imm16, reg1, reg2`	`movea imm16, r0, r1` `and r1, reg2`

上記以外の場合

指定形式	アセンブル結果
`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

<1> imm の値の下位 16 ビットがすべて 0 の場合
reg2 が r0 の場合

指定形式	アセンブル結果
andi imm, reg1, r0	movhi HIGHW(imm), r0, r1 and eg1, r1

reg1=reg2 の場合

指定形式	アセンブル結果
andi imm, reg1, reg2	movhi HIGHW(imm), r0, r1 and r1, reg2

上記以外の場合

指定形式	アセンブル結果
andi imm, reg1, reg2	movhi HIGHW(imm), r0, reg2 and reg1, reg2

<2> 上記以外の場合
reg2 が r0 の場合

指定形式	アセンブル結果
andi imm, reg1, r0	mov imm, r1 and reg1, r1

reg1=reg2 の場合

指定形式	アセンブル結果
andi imm, reg1, reg2	mov imm, r1 and r1, reg2

上記以外の場合

指定形式	アセンブル結果
andi imm, reg1, reg2	mov imm, reg2 and reg1, reg2

(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

指定形式	アセンブル結果
andi $label, reg1, r0	movea $label, r0, r1 and reg1, r1

reg1=reg2 の場合

指定形式	アセンブル結果
andi $label, reg1, reg2	movea $label, r0, r1 and r1, reg2

上記以外の場合

指定形式	アセンブル結果
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

指定形式	アセンブル結果
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

reg1=reg2 の場合

指定形式	アセンブル結果
andi #label, reg1, reg2	mov #label, r1 and r1, reg2
andi label, reg1, reg2	mov label, r1 and r1, reg2
andi $label, reg1, reg2	mov $label, r1 and r1, reg2

上記以外の場合

指定形式	アセンブル結果
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

not, satsubr, sub, subr, tst

- デバイスの not，satsubr，sub，subr，tst 命令は次の形式です。
  - NOT reg1, reg2
  - SATSUBR reg1, reg2
  - SUB reg1, reg2
  - SUBR reg1, reg2
  - TST reg1, rg2

- asrh では，上記に加えて，次の形式も指定できます。
  - not imm, reg2
  - satsubr imm, reg2
  - sub imm, reg2
  - subr imm, reg2
  - tst imm, reg2

- “not imm, reg2”，“satsubr imm, reg2”，“sub imm, reg2”，“subr imm, reg2”，“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。

(a) 0

指定形式	アセンブル結果
`not 0, reg`	`not r0, reg`

(b) 0 以外の -16 ～ +15 の範囲の絶対値式

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`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

指定形式	アセンブル結果
`not imm, reg`	`movhi HIGHW(imm), r0, r1` `not r1, reg`

上記以外の場合

指定形式	アセンブル結果
`not imm, reg`	`mov imm, r1` `not r1, 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つラベルの $label を持つ相対値式

指定形式	アセンブル結果
`not !label, reg`	`movea !label, r0, r1` `not r1, reg`
`not %label, reg`	`movea %label, r0, r1` `not r1, reg`
`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

指定形式	アセンブル結果
`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`

b*cond*

- デバイスの b*cond* 命令は次の形式です。
  - B*cond* disp9
  - B*cond* disp17

- asrh では，上記に加えて，次の形式も指定できます。
  - b*cond*9 disp9
  - b*cond*17 disp17
  - j*cond* disp9
  - j*cond*9 disp9
  - j*cond*17 disp17
  - jbr disp

- b*cond*9，b*cond*17 は，常に固定の disp 幅で出力したい場合に指定します。これらは命令展開の対象になりません。
- j*cond*，j*cond9*，j*cond*17 は，b*cond*，b*cond9*，b*cond*17 と同じ意味です。
- jbr は，br と同じ意味です。
- *cond* に指定可能な記述を「表 5.28 bcond 命令」に示します。

表 5.28 b*cond* 命令

命令	フラグ状態	フラグ状態の意味
bgt	((S xor OV) or Z) = 0	Greater than（signed）
bge	(S xor OV) = 0	Greater than or equal（signed）
blt	(S xor OV) = 1	Less than（signed）
ble	((S xor OV) or Z) = 1	Less than or equal（signed）
bh	(CY or Z) = 0	Higher（Greater than）
bnl	CY = 0	Not lower（Greater than or equal）
bl	CY = 1	Lower（Less than）
bnh	(CY or Z) = 1	Not higher（Less than or equal）
be	Z = 1	Equal
bne	Z = 0	Not equal
bv	OV = 1	Overflow
bnv	OV = 0	No overflow
bn	S = 1	Negative
bp	S = 0	Positive
bc	CY = 1	Carry
bnc	CY = 0	No carry
bz	Z = 1	Zero
bnz	Z = 0	Not zero
br	―	Always（無条件）

命令	フラグ状態	フラグ状態の意味
bsa	SAT = 1	Saturated

- asrh では，bt，bf 命令は指定できません。
- 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，-65536 ～ +65535 の範囲の絶対値式，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，-256 ～ +255 の範囲を越え，-65536 ～ +65535 の範囲の相対値式

指定形式	アセンブル結果
`br disp17`	`jr disp17`
`bcond disp17`	`bcond disp17`

(b) -65536 ～ +65535 の範囲を越え，-2097150 ～ +2097153 の範囲[注 1] の絶対値式，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 -65536 ～ +65535 の範囲を越える範囲の相対値式，または本命令と同じファイル内に定義を持っていないか同じセクションに定義を持っていないラベルの PC オフセット参照を持つ相対値式

指定形式	アセンブル結果
`br disp22`	`jr disp22`

`bsa disp22`	`bsa Label1`
	`br Label2`
	`Label1:`
	`jr disp22 - 4`
	`Label2:`

`bcond disp22`	`bncond Label` [注 2]
	`jr disp22 - 2`
	`Label:`

注 1. -2097150 ～ +2097153 の範囲は br および bsa 命令以外の命令の場合の範囲で，br 命令の場合は -2097152 ～ +2097151，bsa 命令の場合は -2097148 ～ +2097155 となります。

注 2. b*ncond* 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。

jmp

- デバイスの jmp 命令は次の形式です。
  - JMP [reg1]
  - JMP disp32 [reg1]

- asrh では，上記に加えて，次の形式も指定できます。

指定形式	意味
`jmp disp32`	`jmp disp32[r0]`
`jmp32 [reg1]`	`jmp [reg1]`
`jmp32 disp32`	`jmp disp32[r0]`
`jmp32 disp32[reg1]`	`jmp disp32[reg1]`

jr

- デバイスの jr 命令は次の形式です。
  - JR　disp22
  - JR　disp32

- asrh では，上記に加えて，次の形式も指定できます。
  - jr22　disp22
  - jr32　disp32

- jr は，-Xasm_far_jump オプションを指定しない場合は disp22，指定した場合は disp32 と解釈されます。
- jr22，jr32 は，常に固定の disp 幅で出力したい場合に指定します。これらは -Xasm_far_jump オプションの影響をうけません。

jarl

- デバイスの jarl 命令は，次の形式です。
  - JARL　disp22, reg2
  - JARL　disp32, reg1
  - JARL　[reg1], reg3

- asrh では，上記に加えて，次の形式も指定できます。
  - jarl22　disp, reg2
  - jarl32　disp, reg1

- jarl は，-Xasm_far_jump オプションを指定しない場合は disp22，指定した場合は disp32 と解釈されます。
- jarl22，jarl32 は，常に固定の disp 幅で出力したい場合に指定します。これらは -Xasm_far_jump オプションの影響をうけません。

set1, clr1, not1, tst1

- デバイスの set1，clr1，not1，tst1 命令は，次の形式です。
  - set1 bit#3, disp16 [reg1]
  - clr1 bit#3, disp16 [reg1]
  - not1 bit#3, disp16 [reg1]
  - tst1 bit#3, disp16 [reg1]
  - set1 reg2, [reg1]
  - clr1 reg2, [reg1]
  - not1 reg2, [reg1]
  - tst1 reg2, [reg1]

- このうち，“*op* bit#3, disp16 [reg1]” の形式に対して，asrh では次の解釈を行います。
  - disp16 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。
    - -32768 ～ +32767 の範囲を越える絶対値式

指定形式	アセンブル結果
`set1 bit#3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `set1 bit#3, LOWW(disp)[r1]`

    - #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

指定形式	アセンブル結果
`set1 bit#3, disp[reg1]`	`movhi HIGHW1(disp), reg1, r1` `set1 bit#3, LOWW(disp)[r1]`

  - disp を省略した場合，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 LOWW 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。その場合，[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 LOWW 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。その場合，[gp] 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 LOWW 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。その場合，[ep] が指定されたものとみなされます。

push, pushm, pop, popm

- push，pushm，pop，popm 命令は，デバイスには存在しない疑似命令です。
  スタック領域へのレジスタのプッシュ，ポップを行います。

- asrh では，次の形式で指定します。
  - push reg1
  - pushm reg1, reg2, ..., regN
  - pop reg1
  - popm reg1, reg2, ..., regN

- それぞれ次のように命令展開が行われ，複数個の機械語命令が生成されます。

指定形式	アセンブル結果
`push reg1`	`add -4, sp` `st.w reg1, 0[sp]`

`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]`

`pop reg1`	`ld.w 0[sp], reg1` `add 4, sp`

`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`

注意 add/addi 命令を使用するため，PSW の内容が不定になります。
デバイスが提供している pushsp，popsp 命令の使用を推奨します。

prepare, dispose

- デバイスの prepare，dispose 命令は，次の形式です。
  - PREPARE list12, imm5
  - PREPARE list12, imm5, sp/imm
  - DISPOSE imm5, list12
  - DISPOSE imm5, list12, [reg1]

  asrh では，次の形式で指定する必要があります。
  - prepare list, imm1
  - prepare list, imm1, imm2
  - prepare list, imm1, sp
  - dispose imm1, list
  - dispose imm1, list, [reg1]

- list は，prepare，dispose 命令で操作可能な 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絶対値式
    12 ビットと 12 本のレジスタとの対応は次のとおりです。

    ビット 11 ビット 0

r30	r24	r25	r26	r27	r20	r21	r22	r23	r28	r29	r31

  ［記述例］

```
prepare r26, r29, r31, 0x10                ; prepare 0x103, 0x10 と同じ
```

- imm1 には，sp の増分値を 4 の倍数で指定してください。アセンブラがその値を 2 ビット右シフトして機械語命令に格納し，デバイスが実行時に 2 ビット左シフトして解釈します。

  ［記述例］
  スタック領域を 16 バイト確保したい場合，0x4 ではなく 0x10 と指定します。

```
prepare 0, 0x10
```

- 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。ただし，“prepare list, imm1, sp” の形式の命令に対して，imm1 に 0 ～ 127 の範囲を越える絶対値式を指定することはできません。

(a) 0 ～ 127 の範囲を越え，0 ～ 32767 の範囲の絶対値式

指定形式	アセンブル結果
`prepare list, imm1`	`prepare list, 0` `movea -imm1, sp, sp`
`prepare list, imm1, imm2`	`prepare list, 0, imm2` `movea -imm1, sp, sp`
`dispose imm1, list`	`movea imm1, sp, sp` `dispose 0, list`
`dispose imm1, list, [reg1]`	`movea imm1, sp, sp` `dispose 0, list, [reg1]`

(b) 0 ～ 32767 の範囲を越える絶対値式

指定形式	アセンブル結果

`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`

`dispose imm1, list`	`mov imm1, r1`
	`add r1, sp`
	`dispose 0, list, [reg1]`

`dispose imm1, list, [reg1]`	`mov imm1, r1`
	`add r1, sp`
	`dispose 0, list, [reg1]`

［注意事項］

- “prepare list, imm1, sp” の形式の命令に対して，imm1 に 0 ～ 127 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。

- list に操作可能ではないレジスタを指定した場合，次のメッセージが出力され，そのレジスタを無視したコードが生成されます。

W0550015:prepare/dispose 命令のレジスタ・リストに指定したレジスタが不正です。

- list に 0 ～ 4095 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，list を 0xfff でマスクしたコードが生成されます。

W0550014:prepare/dispose 命令のレジスタ・リストに指定した値が不正です。

- imm に 4 の倍数でない絶対値式を指定した場合，次のメッセージが出力され，imm の下位 2 ビットを無視したコードが生成されます。

W0550019: オペランドに指定した値は 4 の倍数である必要があります。

pushsp, popsp

- デバイスの pushsp，popsp 命令は，“pushsp rh-rt”，“popsp rh-rt” の形式です。
- asrh では，“pushsp rh, rt”，“popsp rh, rt” の形式で指定してください。

cmpf.d, cmpf.s

- デバイスの cmpf.d，cmpf.s 命令は，“cmpf.* fcond, reg2, reg1, fcbit” の形式です。
  asrh では，“cmpf*fcond*.* reg2, reg1, fcbit” の形式も指定できます。cmpf*fcond* に指定可能な記述と，意味を表 5.29 に示します。

表 5.29 cmpf*fcond* 命令一覧

命令	定義	説明	アセンブル結果
cmpff.*	FALSE	常に偽	cmpf.* 0x0
cmpfun.*	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.* 0x1
cmpfeq.*	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.* 0x2
cmpfueq.*	reg2 ？＝ reg1	少なくとも一方が非数か，等しい	cmpf.* 0x3
cmpfolt.*	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.* 0x4
cmpfult.*	reg2 ？＜ reg1	少なくとも一方が非数か，より小さい	cmpf.* 0x5
cmpfole.*	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.* 0x6
cmpfule.*	reg2 ？≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.* 0x7
cmpfsf.*	FALSE	常に偽	cmpf.* 0x8
cmpfngle.*	Unordered	reg1, reg2 の少なくとも一方が非数	cmpf.* 0x9
cmpfseq.*	reg2 ＝ reg1	いずれも非数ではなく，かつ等しい	cmpf.* 0xA
cmpfngl.*	reg2 ？＝ reg1	少なくとも一方が非数か，等しい	cmpf.* 0xB
cmpflt.*	reg2 ＜ reg1	いずれも非数ではなく，かつより小さい	cmpf.* 0xC
cmpfnge.*	reg2 ？＜ reg1	少なくとも一方が非数か，より小さい	cmpf.* 0xD
cmpfle.*	reg2 ≦ reg1	いずれも非数ではなく，かつより小さいか，等しい	cmpf.* 0xE
cmpfngt.*	reg2 ？≦ reg1	少なくとも一方が非数か，より小さいか，等しい	cmpf.* 0xF

備考 ？：Unordered（比較不能）

［記述例］

```
cmpfeq.s r10, r11, 0                    ; cmpf.s 0x2, r10, r11, 0 と同じ
```

6. セクション仕様

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，最適化リンカに指示する必要があります。

6.1 セクション

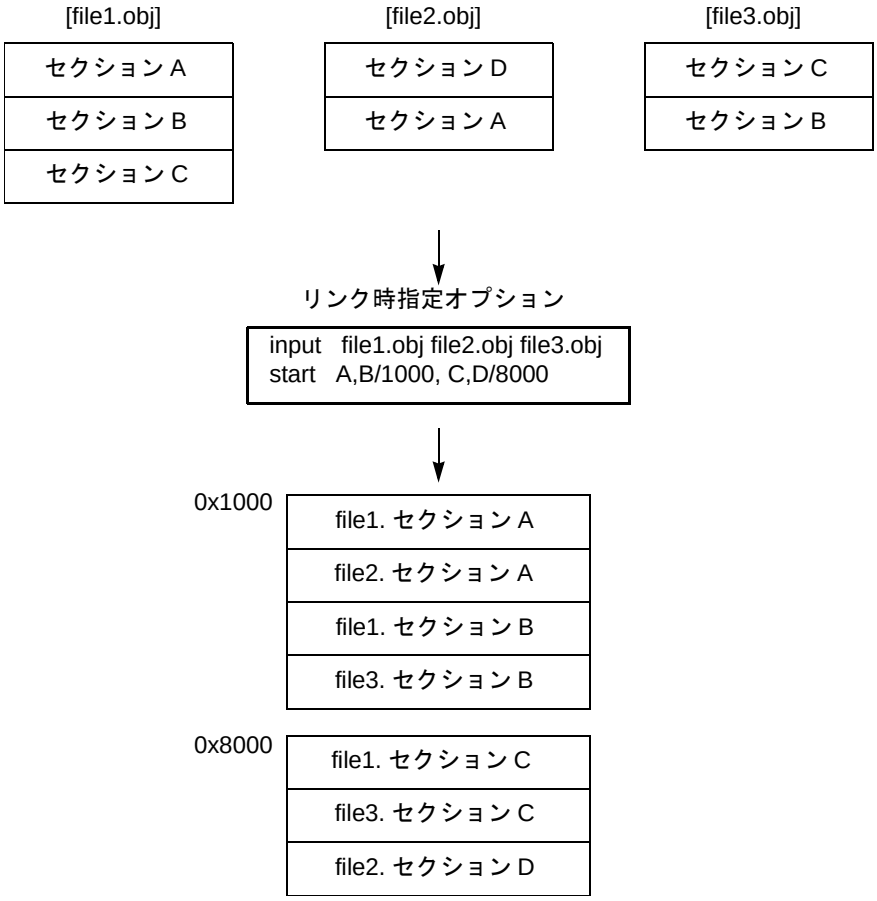
セクションとは，プログラムを構成する基本的な単位（プログラムやデータが配置される領域）です。たとえば，プログラム・コードは text 属性セクションへ，初期値を持つ変数は data 属性セクションへ，というように，セクションごとに分けて配置することになります。
セクション名はアプリケーション内で指定できます。C 言語では “#pragma section 指令 ”，アセンブリ言語では “ セクション定義疑似命令 ” によって指定できます。
ただし #pragma 指令でセクションの指定をしない場合でも，コンパイラはプログラム・コードやデータ（変数）にデフォルトとして決められたセクションを割り当てるようにしています。

6.1.1 セクションの結合

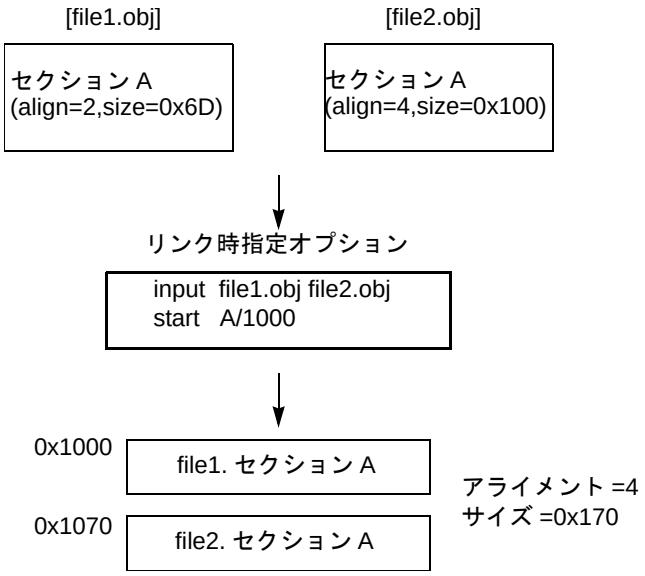
最適化リンカ（以降，“rlink” と略します）では，入力リロケータブルファイル内の同一セクションを結合し，-start オプションによって指定されたアドレスに割り付けます。

(1) -start オプションによるセクション配置

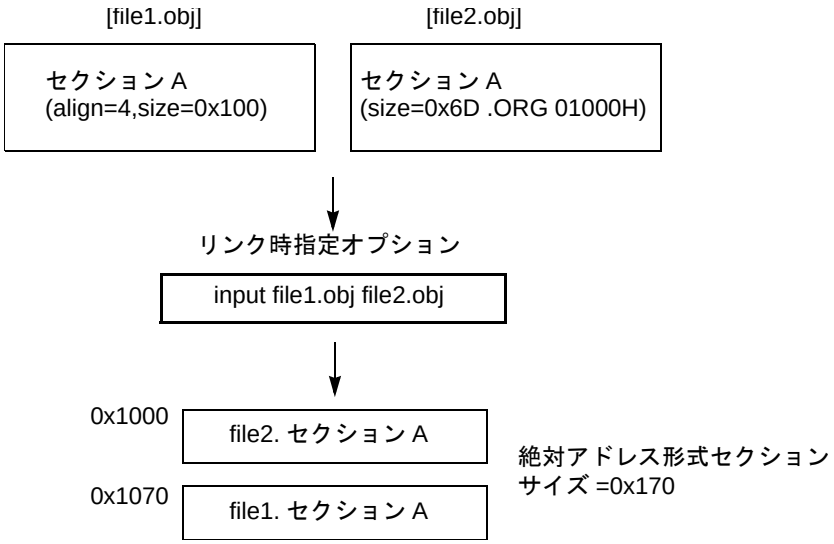
(a) 異なるファイルの同名セクションは，ファイルの入力順に連続して割り付けます。



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

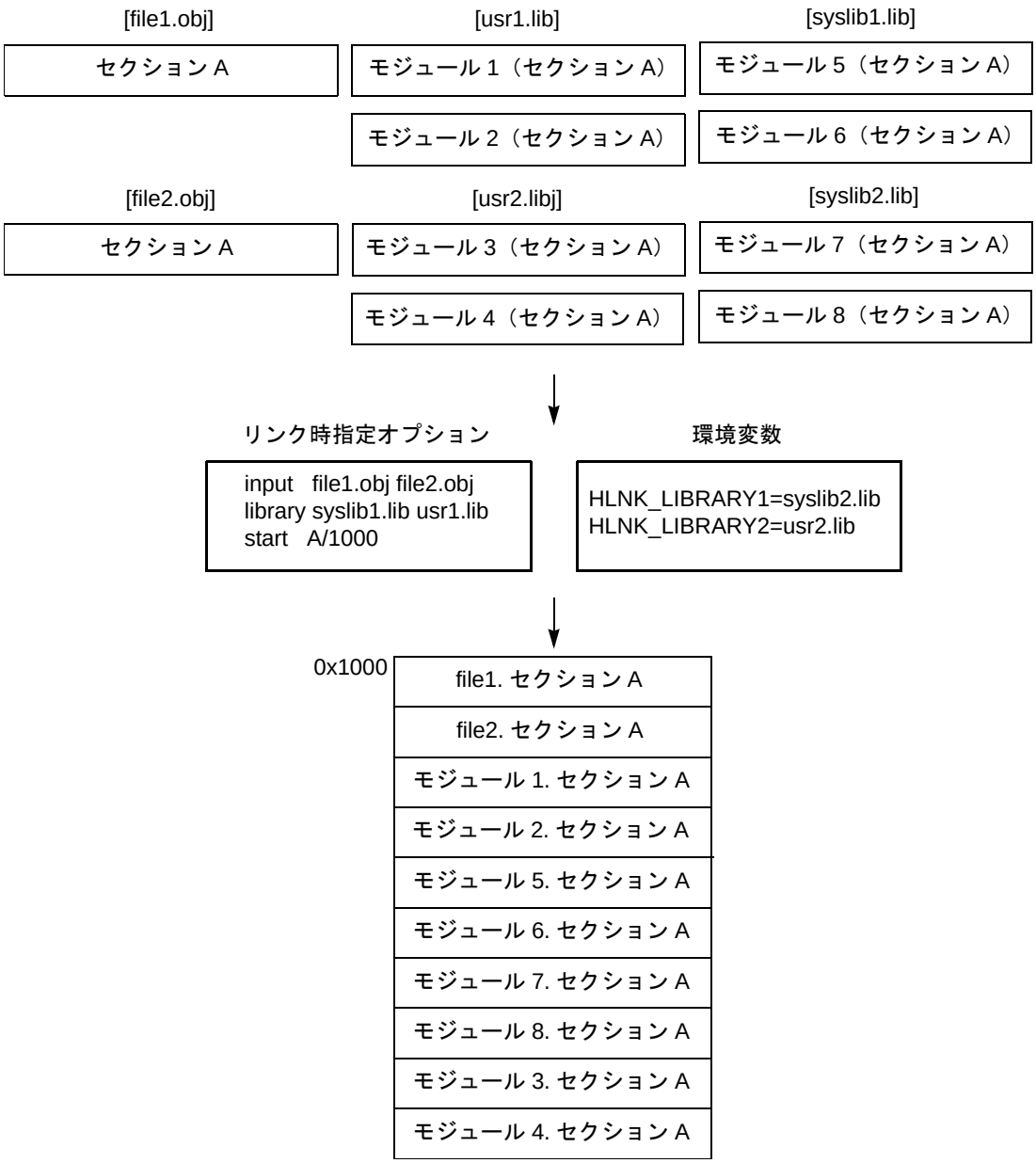


(c) 同名セクションに絶対アドレス形式と相対アドレス形式が含まれている場合，絶対アドレス形式セクションの後に相対アドレス形式セクションを結合します。



(d) 同名セクションの結合順序に関する規則は，優先度の高い順に以下のとおりです。

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



6.2 特殊シンボル

リンクの処理において，各出力セクションの先頭アドレス，各出力セクションの終端を越える最初のアドレス値を値として持つ予約シンボルを生成します。ユーザがこれらの予約シンボルと同名のシンボルを定義した場合，最適化リンカは定義されたシンボルを用い，独自に生成することはしません。

セクションの先頭アドレス値を値として持つ予約シンボルとして，次の 2 種類のシンボルが用いられます。

- 出力セクション名の先頭に “__s” を付加したシンボル
- 出力セクション名の中にある ”@”，“.” の文字を ”_” に置き換え，先頭に “__S” を付加したシンボル 【V1.06.00 以降】

セクションの終端を越える最初のアドレス値を値として持つ予約シンボルとして，次の 2 種類のシンボルが用いられます。

- 出力セクション名の先頭に “__e” を付加したシンボル
- 出力セクション名の中にある ”@”，“.” の文字を ”_” に置き換え，先頭に “__E” を付加したシンボル 【V1.06.00 以降】

例えば，実行形式ファイル中に .text セクションと，FOO.const セクションの 2 つが存在する場合，次の 8 個の特殊シンボルを生成します。

__s.text
__S_text
__e.text
__E_text
__sFOO.const
__SFOO_const
__eFOO.const
__EFOO_const

このうち，__S_text, __E_text, __SFOO_const, __EFOO_const の 4 つは次の手順で C ソースから参照できます。

- シンボル名の先頭にある '_' を 1 つ除いて，char 型の変数，または char 配列型の変数として extern 宣言する。

例

```
extern char _S_text;
char* get_S_text(void) {
  return &_S_text;
}

extern char _E_text[];
char* get_E_text(void) {
  return _E_text;
}
```

7. ライブラリ関数仕様

この章では，CC-RH が提供するライブラリ関数について説明します。

7.1 提供ライブラリ

CC-RH で提供しているライブラリは，次のとおりです。

表 7.1 提供ライブラリ
【V1.02.00 以降】

提供ライブラリ	ライブラリ名	使用する条件
標準ライブラリ （プログラム診断機能関数 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 周辺装置の初期化関数 RAM セクション領域初期化関数 演算用ランタイム関数 間接関数呼び出しチェック関数） 数学ライブラリ （数学関数（倍精度，単精度））	lib ¥ v850e3v5 ¥ rhf8n.lib	-Xfloat=fpu，-Xdbl_size=8， -Xround=nearest 指定時
	lib ¥ v850e3v5 ¥ rhs8n.lib	-Xfloat=soft，-Xdbl_size=8， -Xround=nearest 指定時
	lib ¥ v850e3v5 ¥ rhf4n.lib	-Xfloat=fpu，-Xdbl_size=4， -Xround=nearest 指定時
	lib ¥ v850e3v5 ¥ rhs4n.lib	-Xfloat=soft，-Xdbl_size=4， -Xround=nearest 指定時
	lib ¥ v850e3v5 ¥ rhf8z.lib	-Xfloat=fpu，-Xdbl_size=8，-Xround=zero 指定時
	lib ¥ v850e3v5 ¥ rhf4z.lib	-Xfloat=fpu，-Xdbl_size=4，-Xround=zero 指定時
標準ライブラリ （動的メモリ管理関数） 【V1.04.00 以降】	lib ¥ v850e3v5 ¥ libmalloc.lib	常に使用可能
	lib¥v850e3v5¥secure¥libmalloc.lib	セキュリティ機能使用時 【Professional 版のみ】
標準ライブラリ （非局所分岐関数）	lib ¥ v850e3v5 ¥ libsetjmp.lib	-Xep=callee 指定時
	lib¥v850e3v5¥ep¥fix¥libsetjmp.lib	-Xep=fix 指定時
標準ライブラリ （プログラム診断機能関数 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 周辺装置の初期化関数 RAM セクション領域初期化関数 演算用ランタイム関数）	lib ¥ v850e3v5 ¥ libc.lib	常に使用可能（V1.01.00 以前の後方互換用）
FPU 使用数学ライブラリ （数学関数（倍精度））	lib ¥ v850e3v5 ¥ libm.lib	-Xfloat=fpu 指定時（V1.01.00 以前の後方互換用）
FPU 使用数学ライブラリ （数学関数（単精度））	lib ¥ v850e3v5 ¥ libmf.lib	
FPU 不使用数学ライブラリ （数学関数（倍精度））	lib ¥ v850e3v5 ¥ softfloat ¥ libm.lib	-Xfloat=soft 指定時（V1.01.00 以前の後方互換用）
FPU 不使用数学ライブラリ （数学関数（単精度））	lib ¥ v850e3v5 ¥ softfloat ¥ libmf.lib	

【V1.01.00 以前】

提供ライブラリ	ライブラリ名	使用する条件
標準ライブラリ （プログラム診断機能関数 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 周辺装置の初期化関数 RAM セクション領域初期化関数 演算用ランタイム関数）	lib ¥ v850e3v5 ¥ libc.lib	常に使用可能
FPU 使用数学ライブラリ （数学関数（倍精度））	lib ¥ v850e3v5 ¥ libm.lib	-Xfloat=fpu 指定時
FPU 使用数学ライブラリ （数学関数（単精度））	lib ¥ v850e3v5 ¥ libmf.lib	
FPU 不使用数学ライブラリ （数学関数（倍精度））	lib ¥ v850e3v5 ¥ softfloat ¥ libm.lib	-Xfloat=soft 指定時
FPU 不使用数学ライブラリ （数学関数（単精度））	lib ¥ v850e3v5 ¥ softfloat ¥ libmf.lib	
標準ライブラリ （非局所分岐関数）	lib ¥ v850e3v5 ¥ libsetjmp.lib	-Xep=callee 指定時
	lib¥v850e3v5¥ep¥fix¥libsetjmp.lib	-Xep=fix 指定時

表 7.2 提供ライブラリ（-Xcpu=g3kh 向け）

提供ライブラリ	ライブラリ名	使用する条件
標準ライブラリ （プログラム診断機能関数 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 周辺装置の初期化関数 RAM セクション領域初期化関数 演算用ランタイム関数 間接関数呼び出しチェック関数） 数学ライブラリ （数学関数（倍精度，単精度））	lib \ v850e3v5 \ rhs8n.lib	-Xfloat=fpu，-Xdbl_size=8, -Xround=nearest 指定時
	lib\v850e3v5\rhs8n.lib	-Xdbl_size=8，-Xround=nearest 指定時
	lib\v850e3v5\rhf4n.lib	-Xfloat=fpu，-Xdbl_size=4, -Xround=nearest 指定時
	lib\v850e3v5\rhs4n.lib	-Xfloat=soft，-Xdbl_size=4, -Xround=nearest 指定時
	lib \ v850e3v5 \ rhs8n.lib	-Xfloat=fpu，-Xdbl_size=8, -Xround=zero 指定時
	lib\v850e3v5\rhf4z.lib	-Xfloat=fpu，-Xdbl_size=4, -Xround=zero 指定時
標準ライブラリ （動的メモリ管理関数） 【V1.04.00 以降】	lib\v850e3v5\libmalloc.lib	常に使用可能
	lib\v850e3v5\secure\libmalloc.lib	セキュリティ機能使用時【Professional 版のみ】
標準ライブラリ （非局所分岐関数）	lib\v850e3v5\libsetjmp.lib	-Xep=callee 指定時
	lib\v850e3v5\ep\fix\libsetjmp.lib	-Xep=fix 指定時

提供ライブラリ	ライブラリ名	使用する条件
標準ライブラリ （プログラム診断機能関数 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 周辺装置の初期化関数 RAM セクション領域初期化関数 演算用ランタイム関数 間接関数呼び出しチェック関数）	lib\v850e3v5\libc.lib	常に使用可能（V1.01.00 以前の後方互換用）
数学ライブラリ （数学関数（倍精度））	lib\v850e3v5\softfloat\libm.lib	常に使用可能（V1.01.00 以前の後方互換用）
FPU 使用数学ライブラリ （数学関数（単精度））	lib\v850e3v5\libmf.lib	-Xfloat=fpu 指定時（V1.01.00 以前の後方互換用）
FPU 不使用数学ライブラリ （数学関数（単精度））	lib\v850e3v5\softfloat\libmf.lib	-Xfloat=soft 指定時（V1.01.00 以前の後方互換用）

- 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。
  ただし，プログラム診断機能関数，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。
- CS+ 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，CC-RH 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 ” 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。
- rh***.lib は libc.lib, libm.lib, libmf.lib の内容を含む（libmalloc.lib，libsetjmp.lib は含まない）ので，rh***.lib 指定時は libc.lib，libm.lib，libmf.lib を同時に指定しないでください。また rh***.lib は 1 つだけ指定してください。それ以外の指定をした場合の動作は保証しません。
- 単精度 FPU しか持たないデバイスの場合，-Xfloat=soft または -Xdbl_size=4 を指定したうえで，それぞれに対応するライブラリを使用してください。
- secure ¥ libmalloc.lib は Professional 版ライセンスを持たない場合はリンク時エラーになります。
- libmalloc.lib をリンクする場合は標準ライブラリ（rh***.lib）も同時にリンクする必要があります。
- 旧版との互換用ライブラリ（libc.lib）の使用時は，セキュリティ機能用ライブラリは使用できません。

7.2 ヘッダ・ファイル

CC-RH 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 7.3 ヘッダ・ファイル

ファイル名	概要
assert.h	プログラム診断機能のためのヘッダ・ファイル
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
half.h 【V1.05.00 以降】	半精度浮動小数点表現のためのヘッダ・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
mathf.h	数学計算のためのヘッダ・ファイル（C90 規格規定外の単精度形式の数学関数の宣言とマクロの定義)
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
iso646.h 【V1.07.00 以降】	代替つづりマクロのためのヘッダ・ファイル
stdbool.h 【V1.07.00 以降】	論理型と論理値のためのヘッダ・ファイル
stdint.h 【V1.07.00 以降】	指定した幅の整数型のためのヘッダ・ファイル
_h_c_lib.h	初期設定用ルーチンのためのヘッダ・ファイル

7.3 リエントラント性

“ リエントラント（再入可能）” とは，ある処理を複数の処理から同時に並行して呼び出すことが可能であるという意味です。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を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。複数の関数でデータ変数や RAM を共有している場合，それらの関数はリエントラント性を持たないことがあります。

7.4 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。

7.4.1 プログラム診断機能関数

プログラム診断機能関数として，以下のものがあります。

表 7.4 プログラム診断機能関数

関数／マクロ名	概要
assert	プログラム中に診断機能を付加

assert

プログラム中に診断機能を付け加えます。

[所属]

標準ライブラリ

[指定形式]

#include <asssert.h>
assert(int *expression*);

[詳細説明]

expression が真の時は値を返さずに処理を終了します。expression が偽の時は，診断情報をコンパイラによって定義された書式で標準エラー・ファイルに出力し，その後 abort 関数[注]を呼び出します。
診断情報の中には，パラメータのプログラム・テキスト，ソース・ファイル名，ソース行番号が含まれています。

assert マクロを無効にする場合，assert.h を取り込む前に #define NDEBUG を定義してください。

注 assert マクロを使用する場合，ユーザ・システムに合わせて abort 関数を作成する必要があります。

[使用例]

```
#include    <assert.h>
int func(void);
int main() {
    int ret;
    ret = func();
    assert(ret == 0);       /* ret が 0 以外のとき，abort() が呼ばれる */
    return 0;
}
```

7.4.2 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 7.5 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。

7.4.3 文字列関数

文字列関数として，以下のものがあります。

表 7.6 文字列関数

関数／マクロ名	概要
strpbrk	文字列検索（最初の位置）
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

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。

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。

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]

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。

出力するメッセージは，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）

7.4.4 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 7.7 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
memcpy	メモリ・コピー
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);
}
```

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。

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。

7.4.5 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 7.8 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）
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）

toupper

英小文字から英大文字変換（引数が英小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 c = 'a';
int func() {
    int i;
    i = toupper(c); /*c の英小文字 'a' を英大文字 'A' に変換 */
    return(i);
}
```

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。

7.4.6 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 7.9 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。

7.4.7 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 7.10 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 と同じ）
fputc	ストリームへの文字書き込み（putc と同じ）
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　　CS+ 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　　CS+ 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　　CS+ 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 CS+ 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 CS+ 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。（fputc と同じ）

備考　　CS+ 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。（putc と同じ）

備考　　CS+ 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 CS+ 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　　CS+ 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　　CS+ 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 CS+ 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　　CS+ 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。

注 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.ddde±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	ポインタを出力します。CC-RH 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 CS+ 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]

stdout には 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 stdout 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;

extern  FILE*   _REL_stdin();
extern  FILE*   _REL_stdout();
extern  FILE*   _REL_stderr();
#define stdin   (_REL_stdin())
#define stdout  (_REL_stdout())
#define stderr  (_REL_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】

```
stdout->handle = 0xfffff000;
stderr->handle = 0x00fff000;
stdin->handle  = 0xfffff002;
```

［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 *format, 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　　CS+ 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;

extern  FILE*   _REL_stdin();
extern  FILE*   _REL_stdout();
extern  FILE*   _REL_stderr();
#define stdin   (_REL_stdin())
#define stdout  (_REL_stdout())
#define stderr  (_REL_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】

```
stdout->handle = 0xfffff000;
stderr->handle = 0x00fff000;
stdin->handle  = 0xfffff002;
```

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 CS+ 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 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;

extern  FILE*   _REL_stdin();
extern  FILE*   _REL_stdout();
extern  FILE*   _REL_stderr();
#define stdin   (_REL_stdin())
#define stdout  (_REL_stdout())
#define stderr  (_REL_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】

```
stdout->handle = 0xfffff000;
stderr->handle = 0x00fff000;
stdin->handle  = 0xfffff002;
```

［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 CS+ 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;

extern  FILE*   _REL_stdin();
extern  FILE*   _REL_stdout();
extern  FILE*   _REL_stderr();
#define stdin   (_REL_stdin())
#define stdout  (_REL_stdout())
#define stderr  (_REL_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】

```
stdout->handle = 0xfffff000;
stderr->handle = 0x00fff000;
stdin->handle  = 0xfffff002;
```

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。 CC-RH 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 CS+ 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　　CS+ 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　　CS+ 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　　CS+ 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>
#include    <errno.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 */
}
```

7.4.8 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 7.11 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）
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）
rand	疑似乱数列生成
srand	疑似乱数列の種類を設定
abort	プログラムを異常終了する

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。

［注意事項］

-lang=c オプション指定，かつ -strict_std オプション指定時は使用できません。

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(const void *x, const void *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(const void *x, const void *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。

［注意事項］

-lang=c オプション指定，かつ -strict_std オプション指定時は使用できません。

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。

［注意事項］

-lang=c オプション指定，かつ -strict_std オプション指定時は使用できません。

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 long long 型になること以外，strtol と同じです。

［注意事項］

-lang=c オプション指定，かつ -strict_std オプション指定時は使用できません。

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。

［注意事項］

-lang=c オプション指定，かつ -strict_std オプション指定時は使用できません。

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。

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。

abort

プログラムを異常終了します。

[所属]

標準ライブラリ

[指定形式]

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

[詳細説明]

abort(void) は，プログラムを異常終了します。使用する場合，ユーザ・システムに合わせて abort 関数を作成する必要があります。

[使用例]

```
#include    <stdlib.h>
int func(void);
int main() {
    int ret;
    ret = func();
    if (ret == 0) {
        abort();                /* ret が 0 以外のとき，abort() が呼ばれる */
    }
    return 0;
}
```

7.4.9 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 7.12 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);
```

[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。

【setjmp.h における jmp_buf 型の定義】

```
typedef int     jmp_buf[14];
```

[注意事項]

本関数が退避／復帰するのはコンパイラで予約したレジスタのみです。
22 レジスタ・モード，または共通レジスタ・モードの関数内で setjmp を呼び出し，32 レジスタ・モードの関数内で r20 ～ r24 を破壊した後に本関数を呼び出した場合，r20 ～ r24 の値は回復しません。本関数を呼び出す前に，r20 ～ r24 を回復しておく必要があります。

[使用例]

```
#include    <setjmp.h>
#define ERR_XXX1    1
#define ERR_XXX2    2

jmp_buf jmp_env;

void main(void) {
    for(;;) {
        switch(setjmp(jmp_env)) {
            case    ERR_XXX1:
                /*error XXX1 の終結処理 */
                break;
            case    ERR_XXX2:
                /*error XXX2 の終結処理 */
                break;
            case    0:          /* 非局所分岐ではない */
            default:
                break;
        }
    }
}

void funcXXX(void) {
    longjmp(jmp_env, ERR_XXX1); /*error XXX1 の発生により非局所分岐 */
    longjmp(jmp_env, ERR_XXX2); /*error XXX2 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]

setjmp からの戻りの場合 0 を返します。longjmp によって非局所分岐の場合，longjmp の第 2 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]

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。

【setjmp.h における jmp_buf 型の定義】

```
typedef int     jmp_buf[14];
```

[注意事項]

本関数が保存，回復するのはコンパイラで予約したレジスタのみです。
22 レジスタ・モード，または共通レジスタ・モードの関数内で本関数を呼び出し，32 レジスタ・モードの関数内で r20 ～ r24 を破壊した後に longjmp を呼び出した場合，r20 ～ r24 の値は回復しません。longjmp を呼び出す前に，r20 ～ r24 を回復しておく必要があります。
setjmp 関数へのポインタを使った間接呼び出しはしないでください。

7.4.10 数学関数

数学関数として，以下のものがあります。

表 7.13 数学関数

関数／マクロ名	概要
acos 関数群	逆余弦
asin 関数群	逆正弦
atan 関数群	逆正接
atan2 関数群	逆正接（y / x）
cos 関数群	余弦
sin 関数群	正弦
tan 関数群	正接
cosh 関数群	双曲線余弦
sinh 関数群	双曲線正弦
tanh 関数群	双曲線正接
exp 関数群	指数関数
frexp 関数群	浮動小数点数を仮数部とべき乗に分割
ldexp 関数群	浮動小数点数をべき乗に変換
log 関数群	対数関数（自然対数）
log10 関数群	対数関数（底 = 10）
modf 関数群	浮動小数点数を整数部と小数部に分割
fabs 関数群	絶対値関数
pow 関数群	べき乗関数
sqrt 関数群	平方根関数
ceil 関数群	ceiling 関数
floor 関数群	floor 関数
round 関数群 【V2.01.00 以降】	浮動小数点形式の最も近い整数値への丸め
lround 関数群及び llround 関数群 【V2.01.00 以降】	最も近い整数値への丸め
trunc 関数群 【V2.01.00 以降】	浮動小数点形式の最も近い整数値への丸め
fmod 関数群	剰余関数
copysign 関数群 【V2.00.00 以降】	符号と絶対値の合成
fmax 関数群 【V2.00.00 以降】	大きい方の値の選択
fmin 関数群 【V2.00.00 以降】	小さい方の値の選択

acos 関数群

［指定形式］

double acos(double x)
float acosf(float x)
long double acosl(long double x) 【V2.01.00 以降】

［詳細説明］

x の逆余弦を計算します。

［特殊ケース］

条件	返却値	例外
|x|>1	NaN	EDOM

asin 関数群

［指定形式］

double asin(double x)
float asinf(float x)
long double asinl(long double x) 【V2.01.00 以降】

［詳細説明］

x の逆正弦を計算します。

［特殊ケース］

条件	返却値	例外
|x|>1	NaN	EDOM

atan 関数群

［指定形式］

double atan(double x)
float atanf(float x)
long double atanl(long double x) 【V2.01.00 以降】

［詳細説明］

x の逆正接を計算します。

［特殊ケース］

条件	返却値	例外
アンダーフロー時	-	ERANGE

atan2 関数群

［指定形式］

double atan2(double y, double x)
float atan2f(float y, float x)
long double atan2l(long double y, long double x) 【V2.01.00 以降】

［詳細説明］

y/x の逆正接を計算します。

［特殊ケース］

条件	返却値	例外
x==0，y==0	NaN	EDOM
x==± ∞，y==± ∞	NaN	EDOM
アンダーフロー時	±0	ERANGE
x<0，y==0	π	-
x==0	±π/2	-

cos 関数群

［指定形式］

double cos(double x)
float conf(float x)
long double cosl(long double x) 【V2.01.00 以降】

［詳細説明］

x（ラジアン値）の余弦を計算します。

［特殊ケース］

条件	返却値	例外
x==± ∞	NaN	EDOM

sin 関数群

［指定形式］

double sin(double x)
float sinf(float x)
long double sinl(long double x) 【V2.01.00 以降】

［詳細説明］

x（ラジアン値）の正弦を計算します。

［特殊ケース］

条件	返却値	例外
x==± ∞	NaN	EDOM
アンダーフロー時	-	ERANGE

tan 関数群

［指定形式］

double tan(double x)
float tanf(float x)
long double tanl(long double x) 【V2.01.00 以降】

［詳細説明］

x（ラジアン値）の正接を計算します。

［特殊ケース］

条件	返却値	例外
x==± ∞	NaN	EDOM
アンダーフロー時	-	ERANGE

cosh 関数群

［指定形式］

double cosh(double x)
float coshf(float x)
long double coshl(long double x) 【V2.01.00 以降】

［詳細説明］

x の双曲線余弦を計算します。

［特殊ケース］

条件	返却値	例外
オーバーフロー時	HUGE_VAL	ERANGE

sinh 関数群

［指定形式］

double sinh(double x)
float sinhf(float x)
long double sinhl(long double x) 【V2.01.00 以降】

［詳細説明］

x の双曲線正弦を計算します。

［特殊ケース］

条件	返却値	例外
オーバーフロー時	HUGE_VAL	ERANGE

tanh 関数群

［指定形式］

double tanh(double x)
float tanhf(float x)
long double tanhl(long double x) 【V2.01.00 以降】

［詳細説明］

x の双曲線正接を計算します。

［特殊ケース］

条件	返却値	例外
アンダーフロー時	-	ERANGE

exp 関数群

［指定形式］

double exp(double x)
float expf(float x)
long double expl(long double x) 【V2.01.00 以降】

［詳細説明］

自然対数の底 e の x 乗を計算します。

［特殊ケース］

条件	返却値	例外
アンダーフロー時	-	ERANGE
オーバーフロー時	HUGE_VAL	ERANGE

frexp 関数群

［指定形式］

double frexp(double x, int *exp)
float frexpf(float x, int *exp)
long double frexpl(long double x, int *exp) 【V2.01.00 以降】

［詳細説明］

x を，正規化した数と 2 の整数べき乗とに分割します。正規化した数を返却し，整数を exp が指す int 型のオブジェクトに格納します。
返却値を ret とした場合，次の条件を満たします。

- 0.5 <= |ret| < 1
- $x = ret * 2^{exp}$

［特殊ケース］

条件	返却値	例外
x==0	0，*exp=0	-
x==± ∞	NaN，*exp=0	EDOM
x==NaN	NaN，*exp=0	-

ldexp 関数群

［指定形式］

double ldexp(double x)
float ldexpf(float x)
long double ldexpl(long double x) 【V2.01.00 以降】

［詳細説明］

浮動小数点数と 2 の整数べき乗を乗算します。

［特殊ケース］

条件	返却値	例外
オーバーフロー時	±HUGE_VAL	ERANGE
アンダーフロー時	非正規化数	ERANGE

log 関数群

［指定形式］

double log(double x)
float logf(float x)
long double logl(long double x) 【V2.01.00 以降】

［詳細説明］

x の e を底とする自然対数を計算します。

［特殊ケース］

条件	返却値	例外
x<0	NaN	EDOM
x==0	- ∞	ERANGE

log10 関数群

［指定形式］

double log10(double x)
float log10f(float x)
long double log10l(long double x) 【V2.01.00 以降】

［詳細説明］

x の 10 を底とする常用対数を計算する。

［特殊ケース］

条件	返却値	例外
x<0	NaN	EDOM
x==0	- ∞	ERANGE

modf 関数群

［指定形式］

double modf(double x, double *iptr)
float modff(float x, float *iptr)
long double modfl(long double x, long double *iptr) 【V2.01.00 以降】

［詳細説明］

x を，整数部と小数部とに分割します。小数部を返却値とし，整数部を iptr が指すオブジェクトに格納します。
整数部及び小数部の符号は，x と同一です。

fabs 関数群

［指定形式］

double fabs(double x)
float fabsf(float x)
long double fabsl(long double x) 【V2.01.00 以降】

［詳細説明］

x の絶対値を計算します。

pow 関数群

［指定形式］

double pow(double x)
float powf(float x)
long double powl(long double x) 【V2.01.00 以降】

［詳細説明］

x の y 乗を計算します。

［特殊ケース］

条件	返却値	例外
x<0，y が非整数	NaN	EDOM
x<0，y== ∞	NaN	EDOM
x==± ∞，y==0	NaN	EDOM
x==0，y==0	NaN	EDOM
x==0，y<0	+HUGE_VAL	ERANGE
オーバーフロー時	±HUGE_VAL	ERANGE
アンダーフロー時	0	ERANGE

sqrt 関数群

［指定形式］

double sqrt(double x)
float sqrtf(float x)
long double sqrtl(long double x) 【V2.01.00 以降】

［詳細説明］

x の平方根を計算します。

［特殊ケース］

条件	返却値	例外
x<0	NaN	EDOM

ceil 関数群

［指定形式］

double ceil(double x)
float ceilf(float x)
long double ceill(long double x) 【V2.01.00 以降】

［詳細説明］

x 以上の最小の整数値を計算します。

floor 関数群

［指定形式］

double floor(double x)
float floorf(float x)
long double floorl(long double x) 【V2.01.00 以降】

［詳細説明］

x 以下の最大の整数値を計算します。

round 関数群 【V2.01.00 以降】

［指定形式］

```
double round(double x)
float roundf(float x)
long double roundl(long double x)
```

［詳細説明］

x を最も近い整数値に丸めます。x がちょうど中間にある場合は，その時点の丸め方向にかかわらず，0 から遠い値を選びます。

lround 関数群及び llround 関数群 【V2.01.00 以降】

［指定形式］

long int lround(double x)
long int lroundf(float x)
long int lroundl(long double x)
long long int llround(double x)
long long int llroundf(float x)
long long int llroundl(long double x)

［詳細説明］

x を最も近い整数値に丸めます。x がちょうど中間にある場合は，その時点の丸め方向にかかわらず，0 から遠い方向を選びます。

［特殊ケース］

条件	返却値	例外
x==NaN	0	EDOM
x==± ∞	0	EDOM

trunc 関数群 【V2.01.00 以降】

［指定形式］

double trunc(double x)
float truncf(float x)
long double truncl(long double x)

［詳細説明］

x を最も近い整数値（ただし，その絶対値が x の絶対値より大きくない値）に丸めます。

fmod 関数群

［指定形式］

double fmod(double x, double y)
float fmodf(float x, float y)
long double fmodl(long double x, long double y) 【V2.01.00 以降】

［詳細説明］

y が 0 以外の場合，ある整数 n に対して x-ny の値を返します。その結果は，x と同じ符号をもち，y の絶対値より小さい絶対値を持ちます。

［特殊ケース］

条件	返却値	例外
x==± ∞	NaN	EDOM
y==± ∞	x	-
y==0	NaN	EDOM

copysign 関数群 【V2.00.00 以降】

[指定形式]

double copysign(double x, double y)
float copysignf(float x, float y)
long double copysignl(long double x, long double y) 【V2.01.00 以降】

[詳細説明]

x の絶対値をもち，かつ y の符号をもつ値を返します。

fmax 関数群 【V2.00.00 以降】

［指定形式］

double fmax(double x, double y)
float fmaxf(float x, float y)
long double fmaxl(long double x, long double y) 【V2.01.00 以降】

［詳細説明］

x，y のうち，大きい方の値を返します。x または y の一方が NaN である場合，他方の値を返します。

fmin 関数群 【V2.00.00 以降】

[指定形式]

double fmin(double x, double y)
float fminf(float x, float y)
long double fminl(long double x, long double y) 【V2.01.00 以降】

[詳細説明]

x，y のうち，小さい方の値を返します。x または y の一方が NaN である場合，他方の値を返します。

7.4.11 RAM セクション領域初期化関数

RAM セクション領域初期化関数として，以下のものがあります。

表 7.14 RAM セクション領域初期化関数

関数／マクロ名	概要
_INITSCT_RH	RAM 領域セクションの初期値コピーとゼロ・クリア

_INITSCT_RH

RAM 領域セクションの初期値コピーとゼロ・クリアを行います。

[所属]

標準ライブラリ

[指定形式]

```
#include <_h_c_lib.h>
void _INITSCT_RH(void * datatbl_start, void * datatbl_end, void * bsstbl_start, void * bsstbl_end)
```

[引数／戻り値]

引数	戻り値
datatbl_start ： 　data 属性セクション初期化テーブルの先頭アドレス datatbl_end ： 　data 属性セクション初期化テーブルの末尾アドレス bsstbl_start： 　bss 属性セクション初期化テーブルの先頭アドレス bsstbl_end ： 　bss 属性セクション初期化テーブルの末尾アドレス	なし

[詳細説明]

RAM 領域セクションのうち，data 属性セクションの初期値を ROM 領域からコピーし，bss 属性セクションのゼロ・クリアを行います。
第 1，第 2 引数は，data 属性セクションの初期化テーブルの先頭，末尾アドレスを渡します。
第 3，第 4 引数は，bss 属性セクションの初期化テーブルの先頭，末尾アドレスを渡します。

第 1 引数＞＝第 2 引数の場合，data 属性セクションの初期化を行いません。
第 3 引数＞＝第 4 引数の場合，bss 属性セクションのゼロ・クリアを行いません。

[使用例]

```
struct {
        void *rom_s;    //data 属性セクションの ROM 上の先頭アドレス
        void *rom_e;    //data 属性セクションの ROM 上の末尾アドレス
        void *ram_s;    //data 属性セクションの RAM 上の先頭アドレス
} _C_DSEC[M];

struct {
        void *bss_s;    //bss 属性セクションの RAM 上の先頭アドレス
        void *bss_e;    //bss 属性セクションの RAM 上の末尾アドレス
} _C_BSEC[N];

_INITSCT_RH(_C_DSEC, _C_DSEC + M, _C_BSEC, _C_BSEC + N);
```

備考　.bss セクションの開始アドレスが 0x100，サイズが 0x50 バイトであるとき，0 クリアされるメモリは 0x100，0x101，...，0x14e，0x14f 番地ですが，初期化テーブルには 0x100, 0x150 と指定します。

7.4.12 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 7.15 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]

標準ライブラリ

[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。

7.4.13 演算用ランタイム関数

演算用ランタイム関数として，以下のものがあります。

表 7.16 演算用ランタイム関数

分類	関数名	概要
float 型演算関数	_COM_fadd	単精度浮動小数点の加算
	_COM_fsub	単精度浮動小数点の減算
	_COM_fmul	単精度浮動小数点の乗算
	_COM_fdiv	単精度浮動小数点の除算
double 型演算関数	_COM_dadd	倍精度浮動小数点型の加算
	_COM_dsub	倍精度浮動小数点型の減算
	_COM_dmul	倍精度浮動小数点型の乗算
	_COM_ddiv	倍精度浮動小数点型の除算
long long 型演算関数	_COM_mul64	64 ビット整数の乗算
	_COM_div64	符号付き 64 ビット整数の除算
	_COM_udiv64	符号なし 64 ビット整数の除算
	_COM_rem64	符号付き 64 ビット整数の剰余算
	_COM_urem64	符号なし 64 ビット整数の剰余算
	_COM_shll_64_32	64 ビット整数の論理左シフト演算
	_COM_shrl_64_32	64 ビット整数の論理右シフト演算
	_COM_shra_64_32	64 ビット整数の算術右シフト演算
	_COM_neg64	符号反転

分類	関数名	概要
型変換関数	_COM_itof	32 ビット整数から単精度浮動小数点への変換
	_COM_itod	32 ビット整数から倍精度浮動小数点への変換
	_COM_utof	符号なし 32 ビット整数から単精度浮動小数点への変換
	_COM_utod	符号なし 32 ビット整数から倍精度浮動小数点への変換
	_COM_i64tof	64 ビット整数から単精度浮動小数点への変換
	_COM_i64tod	64 ビット整数から倍精度浮動小数点への変換
	_COM_u64tof	符号なし 64 ビット整数から単精度浮動小数点への変換
	_COM_u64tod	符号なし 64 ビット整数から倍精度浮動小数点への変換
	_COM_ftoi	単精度浮動小数点数から 32 ビット整数への変換
	_COM_dtoi	倍精度浮動小数点数から 32 ビット整数への変換
	_COM_ftou	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
	_COM_dtou	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
	_COM_ftoi64	単精度浮動小数点数から 64 ビット整数への変換
	_COM_dtoi64	倍精度浮動小数点数から 64 ビット整数への変換
	_COM_ftou64	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
	_COM_dtou64	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
	_COM_ftod	単精度浮動小数点数から倍精度浮動小数点への変換
	_COM_dtof	倍精度浮動小数点数から単精度浮動小数点への変換
浮動小数点比較演算関数	_COM_fgt	比較
	_COM_fge	比較
	_COM_feq	比較
	_COM_fne	比較
	_COM_flt	比較
	_COM_fle	比較
	_COM_funord	順序付け不可
	_COM_dgt	比較
	_COM_dge	比較
	_COM_deq	比較
	_COM_dne	比較
	_COM_dlt	比較
	_COM_dle	比較
	_COM_dunord	順序付け不可

7.4.14 間接関数呼び出しチェック関数

間接関数呼び出しチェック関数として，以下のものがあります。

表 7.17 間接関数呼び出しチェック関数

関数／マクロ名	概要
__control_flow_integrity	間接関数呼び出しチェック

7.4.15 動的メモリ管理関数

動的メモリ管理関数として，以下のものがあります。

表 7.18 動的メモリ管理関数

関数／マクロ名	概要
calloc	動的メモリの割り当て
free	動的メモリの開放
malloc	動的メモリの割り当て
realloc	動的メモリの再割り当て

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】

```
#include    <stddef.h>
#define     SIZEOF_HEAP 0x2000
int     _REL_sysheap[SIZEOF_HEAP >> 2];
size_t  _REL_sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数 “_REL_sysheap” 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。

備考 2. 変数 “_REL_sizeof_sysheap” 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。

[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);
}
```

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;
    if((buf = calloc(40, sizeof(s_data))) == NULL)  /*s_data40 個のための領域を割り付け */
        return(1);

    free(buf);                                      /* 領域を開放 */
    return(0);
}
```

セキュリティ機能用ライブラリを使用する場合，次の操作を行った場合に __heap_chk_fail 関数を呼び出します。

- calloc，malloc，realloc で確保した領域以外のポインタを free，realloc に渡す。
- free で開放した後のポインタを再度 free，realloc に渡す。
- calloc，malloc，realloc で割り当てた領域の外側（前後それぞれ 4 バイト以内）に何らかの値を書き込んだ後，割り当てた領域を指すポインタを free，realloc に渡す。

__heap_chk_fail 関数はユーザが定義する必要があり，動的メモリ管理の異常時に実行する処理を記述します。
__heap_chk_fail 関数を定義する際には，次の項目に注意してください。

- 返却値の型は void 型のみであり，仮引数を持たない関数です。
- static 指定をしないでください。
- __heap_chk_fail 関数内で再帰的にヒープ・メモリ領域の破壊を検出しないように注意してください。
- __heap_chk_fail 関数を，PIC（「8.6 PIC/PID 機能」を参照してください）対象としないでください。

セキュリティ機能用の calloc，malloc，および realloc は，領域外への書き込みを検出するために前後 4 バイト余分に領域を割り当てます。このため，通常より多くヒープ・メモリ領域を消費します。

```
#include <stdlib.h>

void sub(int *ip) {
    ...
    free(ip);
}

int func(void) {
    int *ip;
    if ((ip = malloc(40 * sizeof(int))) == NULL)
        if ((ip = malloc(10 * sizeof(int))) == NULL) return(1);
        else sub(ip); /* 1 回目の free */
    else
        ...
    free(ip); /* 2 回目の free */
    return(0);
}

void __heap_chk_fail(void) {
    /* ヒープ・メモリ領域破壊を検出したときの処理 */
}
```

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】

```
#include    <stddef.h>
#define     SIZEOF_HEAP 0x2000
int     _REL_sysheap[SIZEOF_HEAP >> 2];
size_t  _REL_sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数 “_REL_sysheap” 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。

備考 2. 変数 “_REL_sizeof_sysheap” 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。

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】

```
#include    <stddef.h>
#define     SIZEOF_HEAP 0x2000
int     _REL_sysheap[SIZEOF_HEAP >> 2];
size_t  _REL_sizeof_sysheap = SIZEOF_HEAP;
```

備考 1. 変数 “_REL_sysheap” 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。

備考 2. 変数 “_REL_sizeof_sysheap” 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。

7.5 データ用セクションの使用，リエントラント性一覧

この節では，ライブラリに含まれている各種関数の定数データセクション（.const）の使用有無，初期値ありデータ用セクション（.data）の使用有無，初期値なしデータ用セクション（.bss）の使用有無，リエントラント性について説明します。

関数名	.const 使用	.data 使用	.bss 使用	リエントラント性	備考（格納先のライブラリ，非リエントラント性の要因）
strpbrk	×	×	×	○	
strrchr	×	×	×	○	
strchr	×	×	×	○	
strstr	×	×	×	○	
strspn	×	×	×	○	
strcspn	×	×	×	○	
strcmp	×	×	×	○	
strncmp	×	×	×	○	
strcpy	×	×	×	○	
strncpy	×	×	×	○	
strcat	×	×	×	○	
strncat	×	×	×	○	
strtok	×	×	○	×	内部管理データ
strlen	×	×	×	○	
strerror	○	○	×	×	内部管理データ
memchr	×	×	×	○	
memcmp	×	×	×	○	
memcpy	×	×	×	○	
memmove	×	×	×	○	
memset	×	×	×	○	
toupper	○	×	×	○	
tolower	○	×	×	○	
isalnum	○	×	×	○	
isalpha	○	×	×	○	
isascii	×	×	×	○	
isupper	○	×	×	○	
islower	○	×	×	○	
isdigit	○	×	×	○	
isxdigit	○	×	×	○	
iscntrl	○	×	×	○	
ispunct	○	×	×	○	

関数名	.const 使用	.data 使用	.bss 使用	リエントラント性	備考（格納先のライブラリ，非リエントラント性の要因）
isspace	○	×	×	○	
isprint	○	×	×	○	
isgraph	○	×	×	○	
fread	×	×	×	○	
getc	×	×	×	○	
fgetc	×	×	×	○	
fgets	×	×	×	○	
fwrite	×	×	×	○	
putc	×	×	×	○	
fputc	×	×	×	○	
fputs	×	×	×	○	
getchar	×	○	×	×	stdin
gets	×	○	×	×	stdin
putchar	×	○	×	×	stdout
puts	×	○	×	×	stdout
sprintf	○	×	○	×	errno
fprintf	○	×	○	×	errno
vsprintf	○	×	○	×	errno
printf	○	○	○	×	errno, stdout
vfprintf	○	×	○	×	errno
vprintf	○	○	○	×	errno, stdout
sscanf	○	×	×	○	
fscanf	○	×	×	○	
scanf	○	○	×	×	stdin
ungetc	×	×	×	○	
rewind	×	×	×	○	
perror	○	○	○	×	errno,stderr
abs	×	×	×	○	
labs	×	×	×	○	
llabs	×	×	×	○	
bsearch	×	×	×	○	
qsort	×	×	×	○	
div	×	×	×	○	
ldiv	×	×	×	○	

関数名	.const 使用	.data 使用	.bss 使用	リエント ラント性	備考 （格納先のライブラリ， 非リエントラント性の要因）
lldiv	×	×	×	○	
lldiv	○	×	×	○	(libc.lib)
atoi	○	×	○	×	errno
atol	○	×	○	×	errno
atoll	○	×	○	×	errno
strtol	○	×	○	×	errno
strtoul	○	×	○	×	errno
strtoll	○	×	○	×	errno
strtoull	○	×	○	×	errno
atoff	○	×	○	×	errno
atof	○	×	○	×	errno
strtodf	○	×	○	×	errno
strtod	○	×	○	×	errno
rand	×	○	×	×	内部管理データ
srand	×	○	×	×	内部管理データ
abort	×	×	×	-	処理が戻らないため
longjmp	×	×	×	×	SP
setjmp	×	×	×	○	
expf	○	×	○	×	errno
exp	○	×	○	×	errno
expl	○	×	○	×	errno
logf	○	×	○	×	errno
log	×	×	○	×	errno
log	○	×	○	×	(libm.lib, softfloat ¥ libm.lib) errno
logl	○	×	○	×	errno
logl	×	×	○	×	(rhf8n.lib, rhf8z.lib, libm.lib) errno
log10f	×	×	○	×	errno
log10f	○	×	○	×	(libmf.lib, softfloat ¥ libmf.lib) errno
log10	×	×	○	×	errno
log10	○	×	○	×	(libm.lib, softfloat ¥ libm.lib) errno
log10l	○	×	○	×	errno

関数名	.const 使用	.data 使用	.bss 使用	リエントラント性	備考（格納先のライブラリ，非リエントラント性の要因）
log10l	×	×	○	×	(rhf8n.lib, rhf8z.lib, libm.lib) errno
powf	○	×	○	×	errno
pow	○	×	○	×	errno
powl	○	×	○	×	errno
sqrtf	×	×	○	×	errno
sqrtf	○	×	○	×	(libmf.lib, softfloat ¥ libmf.lib) errno
sqrt	×	×	○	×	errno
sqrt	○	×	○	×	(libm.lib, softfloat ¥ libm.lib) errno
sqrtl	×	×	○	×	errno
sqrtl	○	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
ceilf	○	×	×	○	
ceilf	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libmf.lib)
ceil	○	×	×	○	
ceil	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)
ceill	○	×	×	○	
ceill	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat\libm.lib)
fabsf	×	×	×	○	
fabs	×	×	×	○	
fabsl	×	×	×	○	
floorf	○	×	×	○	
floorf	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libmf.lib)
floor	○	×	×	○	
floor	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)
floorl	○	×	×	○	
floorl	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)
roundf	○	×	×	○	
round	○	×	×	○	
roundl	○	×	×	○	
lroundf	○	○	○	×	errno
lround	○	○	○	×	errno
lroundl	○	○	○	×	errno
llroundf	○	○	○	×	errno

関数名	.const 使用	.data 使用	.bss 使用	リエント ラント性	備考 （格納先のライブラリ， 非リエントラント性の要因）
llround	○	○	○	×	errno
llroundl	○	○	○	×	errno
truncf	○	×	×	○	
trunc	○	×	×	○	
truncl	○	×	×	○	
fmodf	×	×	○	×	errno
fmodf	○	×	○	×	(softfloat ¥ libmf.lib) errno
fmod	×	×	○	×	errno
fmod	○	×	○	×	(softfloat ¥ libmf.lib) errno
fmodl	×	×	○	×	errno
copysignf	×	×	×	○	
copysign	×	×	×	○	
copysignl	×	×	×	○	
frexpf	×	×	○	×	errno
frexpf	○	×	○	×	(softfloat ¥ libmf.lib) errno
frexp	×	×	○	×	errno
frexp	○	×	○	×	(softfloat ¥ libmf.lib) errno
frexpl	×	×	○	×	errno
ldexpf	○	×	○	×	errno
ldexpf	×	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
ldexp	○	×	○	×	errno
ldexp	×	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
ldexpl	○	×	○	×	errno
ldexpl	×	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
modff	○	×	×	○	
modff	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)
modf	○	×	×	○	
modf	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)
modfl	○	×	×	○	
modfl	×	×	×	○	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib)

関数名	.const 使用	.data 使用	.bss 使用	リエント ラント性	備考 （格納先のライブラリ， 非リエントラント性の要因）
cosf	×	×	○	×	errno
cosf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
cos	○	×	○	×	errno
cos	×	×	○	×	(rhf4n.lib, rhf4z.lib) errno
cosl	○	×	○	×	errno
cosl	×	×	○	×	(rhf4n.lib, rhfnz.lib) errno
sinf	×	×	○	×	errno
sinf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
sin	○	×	○	×	errno
sin	×	×	○	×	(rhf4n.lib, rhf4z.lib) errno
sinl	○	×	○	×	errno
sinl	×	×	○	×	(rhf4n.lib, rhfnz.lib) errno
tanf	×	×	○	×	errno
tanf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
tan	×	×	○	×	errno
tan	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
tanl	×	×	○	×	errno
tanl	○	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
acosf	×	×	○	×	errno
acosf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
acos	×	×	○	×	errno
acos	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
acosl	×	×	○	×	errno
acosl	○	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno

関数名	.const 使用	.data 使用	.bss 使用	リエントラント性	備考 （格納先のライブラリ, 非リエントラント性の要因）
asinf	×	×	○	×	errno
asinf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
asin	×	×	○	×	errno
asin	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
asinl	×	×	○	×	errno
asinl	○	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
atanf	×	×	○	×	errno
atanf	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
atan	×	×	○	×	errno
atan	○	×	○	×	(rhs8n.lib, rhs4n.lib, libmf.lib, softfloat ¥ libmf.lib) errno
atanl	×	×	○	×	errno
atanl	○	×	○	×	(rhs8n.lib, rhs4n.lib, softfloat ¥ libm.lib) errno
atan2f	○	×	○	×	errno
atan2	○	×	○	×	errno
atan2l	○	×	○	×	errno
coshf	○	×	○	×	errno
cosh	○	×	○	×	errno
coshl	○	×	○	×	errno
sinhf	○	×	○	×	errno
sinh	○	×	○	×	errno
sinhl	○	×	○	×	errno
tanhf	○	×	○	×	errno
tanh	○	×	○	×	errno
tanhl	○	×	○	×	errno
fmax	×	×	×	○	
fmaxf	×	×	×	○	
fmaxl	×	×	×	○	
fmin	×	×	×	○	

関数名	.const 使用	.data 使用	.bss 使用	リエント ラント性	備考 （格納先のライブラリ， 非リエントラント性の要因）
fminf	×	×	×	○	
fminl	×	×	×	○	
calloc	×	○	○	×	内部管理データ
free	×	○	○	×	内部管理データ
malloc	×	○	○	×	内部管理データ
realloc	×	○	○	×	内部管理データ
_INITSCT_RH	×	×	×	○	
hdwinit	×	×	×	○	
_COM_fadd	×	×	×	○	
_COM_fsub	×	×	×	○	
_COM_fmul	×	×	×	○	
_COM_fdiv	×	×	×	○	
_COM_dadd	×	×	×	○	
_COM_dsub	×	×	×	○	
_COM_dmul	×	×	×	○	
_COM_ddiv	×	×	×	○	
_COM_mul64	×	×	×	○	
_COM_div64	×	×	×	○	
_COM_udiv64	×	×	×	○	
_COM_rem64	×	×	×	○	
_COM_urem64	×	×	×	○	
_COM_shll_64_32	×	×	×	○	
_COM_shrl_64_32	×	×	×	○	
_COM_shra_64_32	×	×	×	○	
_COM_neg64	×	×	×	○	
_COM_itof	×	×	×	○	
_COM_itod	×	×	×	○	
_COM_utof	×	×	×	○	
_COM_utod	×	×	×	○	
_COM_i64tof	×	×	×	○	
_COM_i64tod	×	×	×	○	
_COM_u64tof	×	×	×	○	
_COM_u64tod	×	×	×	○	
_COM_ftoi	×	×	×	○	

関数名	.const 使用	.data 使用	.bss 使用	リエント ラント性	備考 （格納先のライブラリ， 非リエントラント性の要因）
_COM_dtoi	×	×	×	○	
_COM_ftou	×	×	×	○	
_COM_dtou	×	×	×	○	
_COM_ftoi64	×	×	×	○	
_COM_dtoi64	×	×	×	○	
_COM_ftou64	×	×	×	○	
_COM_dtou64	×	×	×	○	
_COM_ftod	×	×	×	○	
_COM_dtof	×	×	×	○	
_COM_fgt	×	×	×	○	
_COM_fge	×	×	×	○	
_COM_feq	×	×	×	○	
_COM_fne	×	×	×	○	
_COM_flt	×	×	×	○	
_COM_fle	×	×	×	○	
_COM_funord	×	×	×	○	
_COM_dgt	×	×	×	○	
_COM_dge	×	×	×	○	
_COM_deq	×	×	×	○	
_COM_dne	×	×	×	○	
_COM_dlt	×	×	×	○	
_COM_dle	×	×	×	○	
_COM_dunord	×	×	×	○	
__control_flow_integrity	○	×	×	○	関数リスト

8. スタートアップ

この章では，スタートアップについて説明します。

8.1 概　　要

スタートアップは，C 言語により記述されたユーザ・アプリケーションをシステムへ組み込むためのセクションの初期化処理，main 関数の起動などを行うための処理です。
ここではコアを 1 つのみ使用する「シングルコア用のプログラム」，コアを複数使用する「マルチコア用のプログラム」の 2 種類を想定して説明します。
これらのプログラムを動作させるための基本的なスタートアップ・ルーチンの構成例を示します。

注意　　デバイスの仕様・注意事項により本節の記載以外の追加処理が必要になります。詳細はデバイスのユーザーズマニュアルを参照してください。

8.2 スタートアップ・ルーチン

スタートアップ・ルーチンとは，マイクロコントローラ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指します。
サンプルでは，本ルーチンを次の 2 つの構成に分けています。

- ハードウェア向けの初期化ルーチン
- ユーザ・プログラム向けの初期化ルーチン

8.2.1 ハードウェア向けの初期化ルーチン

ハードウェア向けの初期化処理は次の要素で構成されます。

- RESET ベクタ
- 割り込みハンドラ・テーブル
- 例外ハンドラ・ルーチン
- エントリ・ポイント
- 汎用レジスタの初期化
- 各 PE 用の初期化処理への分岐
- __exit ルーチン
- 各 PE 用の初期化処理
- ハードウェア初期化処理
- 例外ハンドラ･アドレスの拡張仕様（テーブル参照方式）を使用するための設定処理

サンプルでは boot.asm に配置しています。

(1) RESET ベクタ
マイクロコントローラのリセット時に各 PE（プロセッサ・エレメント）のプログラム・カウンタが分岐してくるアドレスに，PE ごとのエントリ・ポイント・アドレスへの分岐命令を配置します。RESET ベクタのアドレスを保持する RBASE レジスタが 512 バイト単位で値を保持するため，RESET ベクタの先頭は 512 バイトに整列します。
サンプルでは RESET_PE*n* セクションに配置しています。

```
    .section "RESET_PE1", text
    .align   512
    jr32     __start ; RESET
    .align   16
    jr32     _Dummy  ; SYSERR
      :
    .align   16
    jr32     _Dummy_EI ; INTn(priority15)
```

セクション名は任意に変更可能ですが，最適化リンカの -start オプションと連動して変更する必要があります。

```
-start=RESET_PE1/01000000
```

注意 RESET ベクタのアドレスは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 PE のリセット時の分岐先が同一のアドレスである場合，1 つの RESET_PE*n* セクションを各 PE で共有して使用します。

(2) 割り込みハンドラ・テーブル
拡張仕様（テーブル参照方式）の例外ハンドラ・アドレスを使用する場合，使用する例外ハンドラ・ルーチンのアドレスを本テーブルの対応する要素位置に配置します。テーブル・アドレスを保持する INTBP レジスタが 512 バイト単位で値を保持するため，テーブルの先頭は 512 バイトに整列します。
サンプルでは EIINTTBL_PE*n* セクションに配置しています。

```
    .section "EIINTTBL_PE1", const
    .align   512
    .dw      #_Dummy_EI ; INT0
    .dw      #_Dummy_EI ; INT1
    .dw      #_Dummy_EI ; INT2
    .rept    512 - 3
    .dw      #_Dummy_EI ; INTn
    .endm
```

注意 テーブルの最大要素数はデバイスのユーザーズマニュアルを参照してください。

セクション名は任意に変更可能ですが，スタートアップ内の INTBP レジスタ設定処理と連動して変更する必要があります。

```
    mov     #__sEIINTTBL_PE1, r10
    ldsr    r10, 4, 1       ; set INTBP
```

(3) 例外ハンドラ・ルーチン
FE，EI 各レベルの例外ハンドラ・ルーチンのサンプルです。何もせず自分自身への分岐を繰り返します。
通常は，C ソース記述で #pragma interrupt を用いて用意します。

```
    .align   2
_Dummy:
    br       _Dummy
_Dummy_EI:
    br       _Dummy_EI
```

(4) エントリ・ポイント
リセット時に RESET ベクタから分岐してくるラベル（アドレス）です。

```
    .align   2
    .public  __start
__start:
```

(5) 汎用レジスタの初期化
ロックステップ機能を使用する準備として，各 PE の汎用レジスタと，EIPC，CTPC，FPEPC レジスタを初期化します。

```
    $nowarning
    mov     r0, r1
    $warning
    mov     r0, r2
    mov     r0, r3
     :
    mov     r0, r31
    ldsr    r0, 0, 0        ;  EIPC
    ldsr    r0, 16, 0       ;  CTPC
```

FPEPC は FPU を有効にした後でなければ初期化できないため，後で初期化します。詳細は「FPU の初期設定」を参照してください。

(6) 各 PE 用の初期化処理への分岐
各 PE が共通に実行する処理です。自分自身の PEID（プロセッサ・エレメント番号）値を読み出し，その値から各 PE ごとに用意された初期化処理へ分岐します。

```
    stsr    0, r10, 2       ; get HTCFG0
    shr     16, r10         ; get PEID
    cmp     1, r10
    bz      .L.entry_PE1
    cmp     2, r10
    bz      .L.entry_PE2
     :
    cmp     7, r10
    bz      .L.entry_PE7
```

シングルコア用のプログラムではこの処理は不要です。

(7) __exit ルーチン
使用しない PE を待機させておくための，自分自身への分岐を繰り返すのみのルーチンです。

```
__exit:
    br      __exit
```

(8) 各 PE 用の初期化処理
各 PE 用に用意したハードウェア初期化処理（_hdwinit_PE*n*）と，例外ハンドラ･アドレスの拡張仕様（テーブル参照方式）を使用するための設定処理（_set_table_reference_method）を呼び出し，ユーザ･プログラム向けの初期化ルーチン (_cstart_pm*n*) へ分岐します。

```
.L.entry_PE1:
    jarl    _hdwinit_PE1, lp        ; initialize hardware
    mov     #__sEIINTTBL_PE1, r6
    jar     _set_table_reference_method, lp ; set table reference method
    jr32    __cstart_pm1
```

(9) ハードウェア初期化処理
サンプルでは ECC 機能を使用するための準備として，RAM 領域を初期化します。PE1 用の初期化処理で Global RAM と Local RAM（PE1 用）を，PE2 用の初期化処理で Local RAM（PE2 用）を初期化します。

```
    .align  2
_hdwinit_PE1:
    mov     lp, r29                 ; save return address
    ; clear Global RAM
    mov     GLOBAL_RAM_ADDR, r6
    mov     GLOBAL_RAM_END, r7
    jarl    _zeroclr4, lp
    ; clear Local RAM PE1
    mov     LOCAL_RAM_PE1_ADDR, r6
    mov     LOCAL_RAM_PE1_END, r7
    jarl    _zeroclr4, lp
    mov     r29, lp
    jmp     [lp]

    .align  2
_hdwinit_PE2:
    mov     lp, r29                 ; save return address
    ; clear Local RAM PE2
    mov     LOCAL_RAM_PE2_ADDR, r6
    mov     LOCAL_RAM_PE2_END, r7
    jarl    _zeroclr4, lp
    mov     r29, lp
    jmp     [lp]

    .align  2
_zeroclr4:
    br      .L.zeroclr4.2
.L.zeroclr4.1:
    st.w    r0, [r6]
    add     4, r6
.L.zeroclr4.2:
    cmp     r6, r7
    bh      .L.zeroclr4.1
    jmp     [lp]
```

注意 サンプルでは無効なアドレスをマクロで指定しています。初期化対象の RAM アドレスはデバイスのユーザーズマニュアルを参照してください。

(10) 例外ハンドラ･アドレスの拡張仕様（テーブル参照方式）を使用するための設定処理
INTBP レジスタへの割り込みハンドラ・テーブルのアドレスの設定と，割り込み制御レジスタの設定を行います。

```
    .align  2
_set_table_reference_method:
    ldsr    r6, 4, 1        ; set INTBP
    mov     ICBASE, r10     ; get interrupt control register address
    set1    6, 0[r10]       ; set INT0 as table reference
    set1    6, 2[r10]       ; set INT1 as table reference
    set1    6, 4[r10]       ; set INT2 as table reference
    jmp     [lp]
```

8.2.2 ユーザ・プログラム向けの初期化ルーチン

ユーザ・プログラム向けの初期化処理は次の要素で構成されます。

- スタック領域
- エントリ・ポイント
- ベース・レジスタ初期化

- RAM セクションの初期化
- FPU の初期設定
- 例外処理の初期設定
- main 関数への分岐

サンプルでは cstart.asm に配置しています。

(1) スタック領域

コンパイラが生成するコードが使用するスタック領域です。CC-RH は，スタック・ポインタ（sp）の値が 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生成し，スタック領域をアドレスの 0x0 番地方向に成長させます。そのため，スタック・ポインタ（sp）には，.stack.bss セクションの 0xffffffff 番地側の 4 バイト境界に整列されたアドレスを指定する必要があります。

```
STACKSIZE    .set    0x200
    .section ".stack.bss", bss
    .align   4
    .ds      (STACKSIZE)
    .align   4
_stacktop:
```

(2) エントリ・ポイント

ハードウェア向け初期化ルーチンから分岐してくるラベル（アドレス）です。

```
    .public __cstart_pm1
    .align  2
__cstart_pm1:
```

(3) ベース・レジスタ初期化

スタック・ポインタ，gp レジスタ，ep レジスタの３つを初期化します。

```
    mov     #_stacktop, sp      ;  set sp register
    mov     #__gp_data, gp      ;  set gp register
    mov     #__ep_data, ep      ;  set ep register
```

__gp_data, __ep_data の詳細は，8.4 シンボルを参照してください。

(4) RAM セクションの初期化

C ソースやアセンブリ・ソース上で定義した変数領域を初期化します。初期化する対象のセクションのアドレスを格納したテーブルを用意し，ライブラリ関数 _INITSCT_RH にテーブルのアドレスを渡して呼び出します。
初期値ありデータセクションの初期化テーブルは次の書式で記述します。

```
    .section ".INIT_DSEC.const", const
    .align   4
    .dw    #__s セクション名 1, #__e セクション名 1,
           #__s セクション名 1 の初期化対象の RAM セクション名
    .dw    #__s セクション名 2, #__e セクション名 2,
           #__s セクション名 2 の初期化対象の RAM セクション名
     :
    .dw    #__s セクション名 n, #__e セクション名 n,
           #__s セクション名 n の初期化対象の RAM セクション名
```

初期化対象の RAM セクションは，最適化リンカの -rom オプションで指定します。

```
-rom=.data=.data.R
-rom=.sdata=.sdata.R
```

この場合は次のような初期化テーブルの記述になります。

```
    .section ".INIT_DSEC.const", const
    .align   4
    .dw      #__s.data,  #__e.data,  #__s.data.R
    .dw      #__s.sdata, #__e.sdata, #__s.sdata.R
```

初期値なしデータセクションの初期化テーブルは次の書式で記述します。

```
    .section ".INIT_BSEC.const", const
    .align   4
    .dw      #__s セクション名1, #__e セクション名1
    .dw      #__s セクション名2, #__e セクション名2
     :
    .dw      #__s セクション名n, #__e セクション名n
```

初期値データを配置する ROM セクションと，初期化する対象の RAM セクションのアドレスは，最適化リンカの -start オプションで指定します。

```
-start=.data,.sdata/00008000
-start=.data.R,.sdata.R/fedf0000
-start=.bss,.sbss/fedf8000
```

注意 ROM，RAM セクションの配置先アドレスは，デバイスのメモリ・マップに依存します。デバイスのユーザーズ・マニュアルを参照して決定してください。

初期化テーブルの先頭アドレス，末尾アドレスを _INITSCT_RH の引数で渡し，初期化を実行します。
それぞれの初期化テーブルの先頭は 4 バイトに整列している必要があります。

```
    mov     #__s.INIT_DSEC.const, r6
    mov     #__e.INIT_DSEC.const, r7
    mov     #__s.INIT_BSEC.const, r8
    mov     #__e.INIT_BSEC.const, r9
    jarl32  __INITSCT_RH, lp     ; initialize RAM area
```

_INITSCT_RH の使用方法は 7.4.11 RAM セクション領域初期化関数を参照してください。

(5) FPU の初期設定
PID レジスタを参照して，FPU を搭載しているかどうかを確認します。
搭載している場合は，各種の初期設定を行います。
PSW.CU0 ビットに 1 を設定して，FPU を使用可能に設定します。
FPSR レジスタに対して FPU の動作モードの設定を行います。
ロックステップ機能を使用する準備として，FPEPC レジスタの初期化を行います。

```
    stsr    6, r10, 1        ; r10 <- PID
    shl     21, r10
    shr     30, r10
    bz      .L1              ; detect FPU
    stsr    5, r10, 0        ; r10 <- PSW
    movhi   0x0001, r0, r11
    or      r11, r10
    ldsr    r10, 5, 0        ; enable FPU
    movhi   0x0002, r0, r11
    ldsr    r11, 6, 0        ; initialize FPSR
    ldsr    r0, 7, 0         ; initialize FPEPC
.L1:
```

FPU を使用しないプログラムでは，これらの記述を削除してください。

(6) 例外処理の初期設定
PSW.ID ビットに 0 を設定して，例外発生を許可します。
PSW.UM ビットに 1 を設定して，ユーザ・モードへ遷移します。

main 関数への分岐と同時に設定を反映させるために，PSW ではなく FEPSW へ書き込み，feret 命令で FEPSW から PSW へ設定を反映させます。

```
    stsr    5, r10, 0        ; r10 <- PSW
    xori    0x0020, r10, r10 ; enable interrupt
    movhi   0x4000, r0, r11
    or      r11, r10         ; supervisor mode -> user mode
    ldsr    r10, 3, 0        ; FEPSW <- r10
```

(7) main 関数への分岐

FEPC レジスタに main 関数のアドレスを，r31 レジスタに main 関数の実行が終わった後に実行したい _exit 関数のアドレスを設定して，feret 命令を実行することで main 関数へ分岐します。

```
    mov     #_exit, lp       ; lp <- #_exit
    mov     #_main, r10
    ldsr    r10, 2, 0        ; FEPC <- #_main
    feret
```

8.2.3 プロジェクト間の情報の受け渡し

マルチコア用のプログラムを，1 つのブート・ローダ・プロジェクトと複数のアプリケーション・プロジェクトに分けて構成する場合，ブート・ローダ・プログラムからアプリケーション・プログラムの情報を参照するために，最適化リンカの -fsymbol オプションを使用します。

最適化リンカに -fsymbol オプションを指定してアプリケーション・プロジェクトをリンクすると，オプションに指定したセクション内に存在する public ラベルの名前とアドレス値を，シンボル・アドレス・ファイル（.fsy ファイル）に出力します。各アプリケーション・プロジェクトから出力したシンボル・アドレス・ファイルをブート・ローダ・プロジェクトの入力とすることで，情報の参照が可能になります。

受け渡しの例を次に示します。

```
;; cstart.asm
    .section  ".text.cmn", text
    .public   __cstart_pm1       ; .fsy に出力するために public 指定する
__cstart_pm1:
```

アプリケーション・プロジェクトのリンク時に，-fsymbol オプションで __cstart_pm1 ラベルの存在する .text.cmn セクションを指定します。この場合，pm1.fsy ファイルを生成します。

```
> rlink cstart.obj -output=pm1.abs -fsymbol=.text.cmn
```

ブート・ローダ・プロジェクトで，アプリケーション・プロジェクト側のラベルを参照します。

```
;; boot.asm
    jr32     #__cstart_pm1
```

ブート・ローダ・プロジェクトのコンパイル時に，各アプリケーション・プロジェクトから生成した .fsy ファイルを一緒に入力することで，.fsy ファイル内のアドレス値でラベルの参照解決を行います。

```
> ccrh boot.asm pm1.fsy pm2.fsy -oboot.abs
```

注意 .fsy ファイルに出力したラベルは，ブート・ローダ・プロジェクトでも public ラベルの定義として扱います。このため，複数のアプリケーション・プロジェクトから同名のラベルを .fsy ファイルに出力していたり，ブート・ローダ・プロジェクトで同名のラベルを定義していると，ブート・ローダ・プロジェクトのリンク時に多重定義エラーになります。

8.3 コーディング例

boot.asm，cstart.asm の例を次に示します。
boot.asm

```
    ; if using eiint as table reference method,
    ; enable next line's macro.

    ;USE_TABLE_REFERENCE_METHOD .set 1

;---------------------------------------------------------------------------
;   exception vector table
;---------------------------------------------------------------------------
    .section "RESET_PE1", text
    .align   512
    jr32     __start   ; RESET

    .align   16
    jr32     _Dummy    ; SYSERR

    .align   16
    jr32     _Dummy

    .align   16
    jr32     _Dummy    ; FETRAP

    .align   16
    jr32     _Dummy_EI ; TRAP0

    .align   16
    jr32     _Dummy_EI ; TRAP1

    .align   16
    jr32     _Dummy    ; RIE

    .align   16
    jr32     _Dummy_EI ; FPP/FPI

    .align   16
    jr32     _Dummy    ; UCPOP

    .align   16
    jr32     _Dummy    ; MIP/MDP

    .align   16
    jr32     _Dummy    ; PIE

    .align   16
    jr32     _Dummy

    .align   16
    jr32     _Dummy    ; MAE

    .align   16
    jr32     _Dummy

    .align   16
    jr32     _Dummy    ; FENMI

    .align   16
    jr32     _Dummy    ; FEINT
```

```
    .align   16
    jr32     _Dummy_EI ; INTn(priority0)

    .align   16
    jr32     _Dummy_EI ; INTn(priority1)

    .align   16
    jr32     _Dummy_EI ; INTn(priority2)

    .align   16
    jr32     _Dummy_EI ; INTn(priority3)

    .align   16
    jr32     _Dummy_EI ; INTn(priority4)

    .align   16
    jr32     _Dummy_EI ; INTn(priority5)

    .align   16
    jr32     _Dummy_EI ; INTn(priority6)

    .align   16
    jr32     _Dummy_EI ; INTn(priority7)

    .align   16
    jr32     _Dummy_EI ; INTn(priority8)

    .align   16
    jr32     _Dummy_EI ; INTn(priority9)

    .align   16
    jr32     _Dummy_EI ; INTn(priority10)

    .align   16
    jr32     _Dummy_EI ; INTn(priority11)

    .align   16
    jr32     _Dummy_EI ; INTn(priority12)

    .align   16
    jr32     _Dummy_EI ; INTn(priority13)

    .align   16
    jr32     _Dummy_EI ; INTn(priority14)

    .align   16
    jr32     _Dummy_EI ; INTn(priority15)

    .section "EIINTTBL_PE1", const
    .align   512
    .dw      #_Dummy_EI ; INT0
    .dw      #_Dummy_EI ; INT1
    .dw      #_Dummy_EI ; INT2
    .rept    512 - 3
    .dw      #_Dummy_EI ; INTn
    .endm
```

```
    .section ".text", text
    .align   2
_Dummy:
    br       _Dummy

_Dummy_EI:
    br       _Dummy_EI

;-------------------------------------------------------------------------------
;   startup
;-------------------------------------------------------------------------------
    .section ".text", text
    .align   2
    .public  __start
__start:
$if 1    ; initialize register
    $nowarning
    mov      r0, r1
    $warning
    mov      r0, r2
    mov      r0, r3
    mov      r0, r4
    mov      r0, r5
    mov      r0, r6
    mov      r0, r7
    mov      r0, r8
    mov      r0, r9
    mov      r0, r10
    mov      r0, r11
    mov      r0, r12
    mov      r0, r13
    mov      r0, r14
    mov      r0, r15
    mov      r0, r16
    mov      r0, r17
    mov      r0, r18
    mov      r0, r19
    mov      r0, r20
    mov      r0, r21
    mov      r0, r22
    mov      r0, r23
    mov      r0, r24
    mov      r0, r25
    mov      r0, r26
    mov      r0, r27
    mov      r0, r28
    mov      r0, r29
    mov      r0, r30
    mov      r0, r31
    ldsr     r0, 0, 0       ;  EIPC
    ldsr     r0, 16, 0      ;  CTPC
$endif
```

```
$if 1
    ; jump to entry point of each PE
    stsr     0, r10, 2      ; get HTCFG0
    shr      16, r10        ; get PEID

    cmp      1, r10
    bz       .L.entry_PE1
    cmp      2, r10
    bz       .L.entry_PE2
    cmp      3, r10
    bz       .L.entry_PE3
    cmp      4, r10
    bz       .L.entry_PE4
    cmp      5, r10
    bz       .L.entry_PE5
    cmp      6, r10
    bz       .L.entry_PE6
    cmp      7, r10
    bz       .L.entry_PE7
__exit:
    br       __exit

.L.entry_PE1:
    jarl     _hdwinit_PE1, lp ; initialize hardware
$ifdef USE_TABLE_REFERENCE_METHOD
    mov       #__sEIINTTBL_PE1, r6
    jarl      _set_table_reference_method, lp ; set table reference method
$endif

    jr32      __cstart_pm1

.L.entry_PE2:
    jarl      _hdwinit_PE2, lp ; initialize hardware
;$ifdef USE_TABLE_REFERENCE_METHOD
;   mov       #__sEIINTTBL_PE2, r6
;   jarl      _set_table_reference_method, lp ; set table reference method
;$endif
;   jr32      __cstart_pm2
    br        __exit

.L.entry_PE3:
    br        __exit
.L.entry_PE4:
    br        __exit
.L.entry_PE5:
    br        __exit
.L.entry_PE6:
    br        __exit
.L.entry_PE7:
    br        __exit
$endif
```

```
;-------------------------------------------------------------------------------
; hdwinit_PE1
; Specify RAM addresses suitable to your system if needed.
;-------------------------------------------------------------------------------
    GLOBAL_RAM_ADDR     .set    0
    GLOBAL_RAM_END      .set    0
    LOCAL_RAM_PE1_ADDR  .set    0
    LOCAL_RAM_PE1_END   .set    0

    .align    2
_hdwinit_PE1:
    mov       lp, r29           ; save return address

    ; clear Global RAM
    mov       GLOBAL_RAM_ADDR, r6
    mov       GLOBAL_RAM_END, r7
    jarl      _zeroclr4, lp

    ; clear Local RAM PE1
    mov       LOCAL_RAM_PE1_ADDR, r6
    mov       LOCAL_RAM_PE1_END, r7
    jarl      _zeroclr4, lp

    mov       r29, lp
    jmp       [lp]

;-------------------------------------------------------------------------------
; hdwinit_PE2
; Specify RAM addresses suitable to your system if needed.
;-------------------------------------------------------------------------------
    LOCAL_RAM_PE2_ADDR  .set    0
    LOCAL_RAM_PE2_END   .set    0

    .align    2
_hdwinit_PE2:
    mov       lp, r14           ; save return address

    ; clear Local RAM PE2
    mov       LOCAL_RAM_PE2_ADDR, r6
    mov       LOCAL_RAM_PE2_END, r7
    jarl      _zeroclr4, lp

    mov       r14, lp
    jmp       [lp]
```

```
;-------------------------------------------------------------------------------
;   zeroclr4
;-------------------------------------------------------------------------------
    .align    2
_zeroclr4:
    br        .L.zeroclr4.2
.L.zeroclr4.1:
    st.w      r0, [r6]
    add       4, r6
.L.zeroclr4.2:
    cmp       r6, r7
    bh        .L.zeroclr4.1
    jmp       [lp]

$ifdef USE_TABLE_REFERENCE_METHOD
;-------------------------------------------------------------------------------
;   set table reference method
;-------------------------------------------------------------------------------
    ; interrupt control register address
    ICBASE    .set 0xfffeea00

    .align    2
_set_table_reference_method:
    ldsr      r6, 4, 1      ; set INTBP

    ; Some interrupt channels use the table reference method.
    mov       ICBASE, r10   ; get interrupt control register address
    set1      6, 0[r10]     ; set INT0 as table reference
    set1      6, 2[r10]     ; set INT1 as table reference
    set1      6, 4[r10]     ; set INT2 as table reference

    jmp       [lp]
$endif
;-------------------- end of start up module -------------------;
```

cstart.asm

```
;-------------------------------------------------------------------------------
;   system stack
;-------------------------------------------------------------------------------
STACKSIZE    .set    0x200
    .section ".stack.bss", bss
    .align   4
    .ds      (STACKSIZE)
    .align   4
_stacktop:

;-------------------------------------------------------------------------------
;   section initialize table
;-------------------------------------------------------------------------------
    .section ".INIT_DSEC.const", const
    .align   4
    .dw      #__s.data, #__e.data, #__s.data.R

    .section ".INIT_BSEC.const", const
    .align   4
    .dw      #__s.bss, #__e.bss
```

```
;-------------------------------------------------------------------------------
;   startup
;-------------------------------------------------------------------------------
    .section ".text.cmn", text
    .public  __cstart_pm1
    .align   2
__cstart_pm1:
    mov      #_stacktop, sp     ;  set sp register
    mov      #__gp_data, gp     ;  set gp register
    mov      #__ep_data, ep     ;  set ep register

    mov      #__s.INIT_DSEC.const, r6
    mov      #__e.INIT_DSEC.const, r7
    mov      #__s.INIT_BSEC.const, r8
    mov      #__e.INIT_BSEC.const, r9
    jarl32   __INITSCT_RH, lp   ;  initialize RAM area

    ; set various flags to PSW via FEPSW

    stsr     5, r10, 0          ; r10 <- PSW

    movhi    0x0001, r0, r11
    or       r11, r10
    ldsr     r10, 5, 0          ; enable FPU

    movhi    0x0002, r0, r11
    ldsr     r11, 6, 0          ; initialize FPSR
    ldsr     r0, 7, 0           ; initialize FPEPC

    stsr     5, r10, 0          ; r10 <- PSW

    ;xori    0x0020, r10, r10   ; enable interrupt

    ;movhi   0x4000, r0, r11
    ;or      r11, r10           ; supervisor mode -> user mode

    ldsr     r10, 3, 0          ; FEPSW <- r10

    mov      #_exit, lp         ; lp <- #_exit
    mov      #_main, r10
    ldsr     r10, 2, 0          ; FEPC <- #_main

    ; apply PSW and PC to start user mode
    feret

_exit:
    br       _exit              ;  end of program

;-------------------------------------------------------------------------------
;   dummy section
;-------------------------------------------------------------------------------
    .section    ".data", data
.L.dummy.data:
    .section    ".bss", bss
.L.dummy.bss:
    .section    ".const", const
.L.dummy.const:
    .section    ".text", text
.L.dummy.text:
;------------------- end of start up module ------------------;
```

8.4 シンボル

CC-RH では，必要に応じて次のシンボルを使用します。

- __gp_data シンボル
  グローバル・ポインタ・レジスタ（r4）に設定する値です。
  sdata 属性セクションや sdata23 属性セクションに配置した変数に対して，短い命令で参照するために使用します。
  また，PID（Position Independent Data）機能でも使用します。
- __ep_data シンボル
  エレメント・ポインタ・レジスタ（r30）に設定する値です。
  tdata 属性セクション，edata 属性セクションや edata23 属性セクションに配置した変数に対して，短い命令で参照するために使用します。また，PID（Position Independent Data）機能でも使用します。
- __pc_data シンボル
  pcconst16，pcconst32，pcconst32 属性セクションに配置した変数を参照するために使用します。

ここでは，各シンボル値の決め方について説明します。

8.4.1 __gp_data

__gp_data の値は次の優先順位で決定します。

(1) アプリケーション中で __gp_data を定義している場合は，その値で決定します。

注意 コンパイラは，__gp_data が 2 バイト整列されている前提でコードを生成します。このため，アプリケーション中で __gp_data を定義する場合は，値が 2 の倍数になるようにしてください。

(2) アプリケーション中に __gp_data の参照だけがある場合，最適化リンカ（rlink）が，次の優先順位で自動的に，__gp_data の値を決定します。
  (2-1) sdata または sbss 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
  (2-2) sdata23 または sbss23 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
  (2-3) sdata32 または sbss32 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
  (2-4) 上記のいずれのセクションも存在せず，__gp_data の参照だけがある場合，ゼロ

  ただし，定義しようとした値が奇数値である場合，その値に 1 を加えます。
  また，コンパイラは __gp_data を直接参照するコードを生成しないので，通常，__gp_data の参照とはスタートアップ・ルーチン内で，__gp_data の値をグローバル・ポインタ・レジスタ（r4）に設定する処理を指します。

(3) アプリケーション中に __gp_data の定義や参照が無い場合，最適化リンカ（rlink）は __gp_data を生成しません。この状態で GP 相対セクションへの参照コードがあると，リンク時エラーになります。

```
Undefined external symbol "GP-symbol (__gp_data)" referenced in "FILE"
```

8.4.2 __ep_data

__ep_data の値は次の優先順位で決定します。

(1) アプリケーション中で __ep_data を定義している場合は，その値で決定します。

注意 コンパイラは，__ep_data が 2 または 4 バイト整列されている前提でコードを生成します。このため，アプリケーション中で __ep_data を定義する場合は，値が 4 の倍数になるようにしてください。

(2) アプリケーション中に __ep_data の参照だけがある場合，最適化リンカ（rlink）が，次の優先順位で自動的に，__ep_data の値を決定します。
  (2-1) tdata，tdata4，tbss4，tdata5，tbss5，tdata7，tbss7，tdata8，tbss8 のいずれかの属性のセクションが存在する場合，次の優先順位で，その属性を持つ全てのセクションの最小アドレス
    (a) tdata 属性のセクション
    (b) tdata4 または tbss4 属性のセクション

(c) tdata5 または tbss5 属性のセクション
(d) tdata7 または tbss7 属性のセクション
(e) tdata8 または tbss8 属性のセクション

注意 上記の属性のセクションを使用する場合は，上記の順序で配置してください。sld，sst 命令は符号なしのオフセットを持ちます。このため，低優先度のセクションを，高優先度のセクションより小さいアドレスに配置すると，低優先度のセクションを sld，sst 命令で参照できません。この場合，リンク時エラーになります。

(2-2) edata または ebss 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
(2-3) edata23 または ebss23 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
(2-4) edata32 または ebss32 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
(2-5) 上記のいずれのセクションも存在せず，__ep_data の参照だけがある場合，ゼロ

ただし，定義しようとした値が奇数値である場合，その値に 1 を加えます。
また，コンパイラは __ep_data を直接参照するコードを生成しないので，通常，__ep_data の参照とはスタートアップ・ルーチン内で，__ep_data の値をエレメント・ポインタ・レジスタ（r30）に設定する処理を指します。

(3) アプリケーション中に __ep_data の定義や参照が無い場合，最適化リンカ（rlink）は __ep_data を生成しません。この状態で EP 相対セクションへの参照コードがあると，リンク時エラーになります。

```
Undefined external symbol "EP-symbol (__ep_data)" referenced in "FILE"
```

8.4.3 __pc_data

__pc_data の値は次の優先順位で決定します。

(1) アプリケーション中で __pc_data を定義している場合は，その値で決定します。

注意 コンパイラは，__pc_data が 2 バイト整列されている前提でコードを生成します。このため，アプリケーション中で __pc_data を定義する場合は，値が 2 の倍数になるようにしてください。

(2) アプリケーション中に __pc_data の参照だけがある場合，最適化リンカ（rlink）が，次の優先順位で自動的に，__pc_data の値を決定します。
(2-1) pcconst16 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
(2-2) pcconst23 属性のセクションが存在する場合，それら全てのセクションの最小アドレスと最大アドレスの中間値
(2-3) pcconst32 属性のセクションが存在する場合，それら全てのセクションの最小アドレス
(2-4) 上記のいずれのセクションも存在せず，__pc_data の参照だけがある場合，ゼロ

ただし，定義しようとした値が奇数値である場合，その値に 1 を加えます。

8.5 ROM イメージの作成

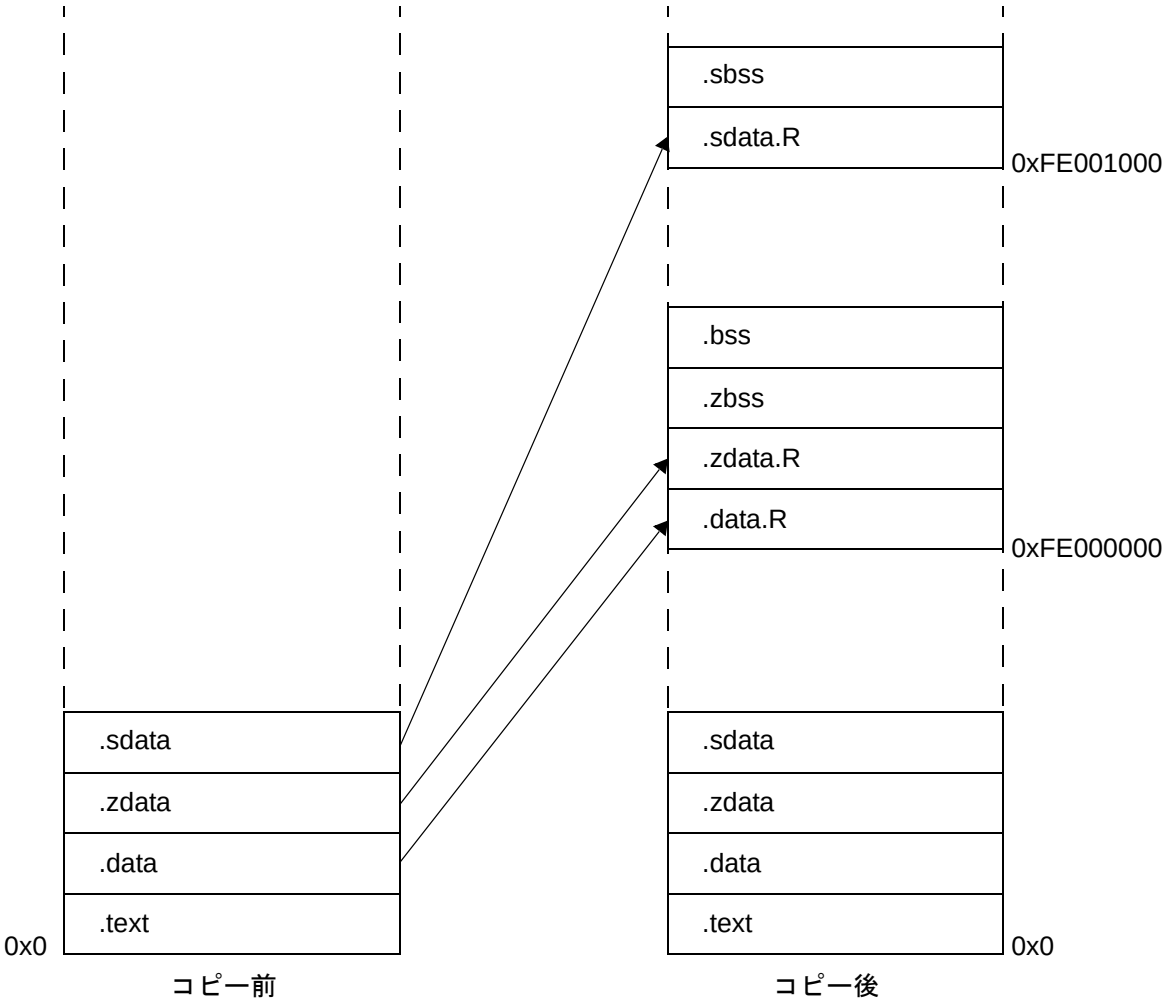
この節では，組み込み向けアプリケーションで必要となる，ROM イメージ作成について説明します。

アプリケーション中で，外部変数や静的変数を定義すると，それらの変数は RAM 上のセクションに配置されます。変数が初期値を持つ場合，アプリケーションの開始時には，RAM に初期値が存在している必要があります。
一方で，ハードウェアの起動時や，リセット時は，RAM の値は不定です。従って，リセットからアプリケーションの起動までの間に，RAM に変数の初期値を格納する必要があります。
CC-RH では，プログラム本体と，初期値データを ROM に配置した，ROM だけのプログラムイメージを作成することができます。このプログラムを実行すると，スタートアップ・ルーチン内で，ROM に準備した初期値データを RAM にコピーして，RAM 側の初期化を行います。

ROM イメージ作成の手順と，RAM へのコピー処理の詳細は，「8.2.2 ユーザ・プログラム向けの初期化ルーチン」の「RAM セクションの初期化」の項を参照してください。

ROM に用意した初期値データと，RAM 上にコピーした初期値のイメージは，次のようになります。

図 8.1 コピー前後のイメージ



備考

- 変数の初期値だけでなく，RAM 上で実行したいプログラムがある場合にも，同様の方法で対応できます。
- 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，実行可能なロード・モジュールを ROM や RAM に直接ダウンロードして実行できます。このような場合には，実行時の初期値のコピー処理は不要です。

8.6 PIC/PID 機能

本節では，メモリ上の任意の位置に関数やデータを配置できるプログラム（位置独立プログラム）の作成方法について説明します。

CC-RH では，配置対象と機能を，次の 3 つに分類します。

- 任意の位置にコード（関数）を配置し，実行できる機能を，PIC（Position Independent Code）機能と呼びます。
- 任意の位置に定数データ（const 変数）を配置し，参照できる機能を，PIROD（Position Independent Read Only Data）機能と呼びます。
- 任意の位置にデータ（変数）を配置し，参照できる機能を，PID（Position Independent Data）機能と呼びます。

8.6.1 PIC

CC-RH では，コードの参照を PC 相対で行うことにより，位置独立を実現します。

-pic オプションを指定すると，コードの配置先のデフォルト・セクションが .pctext セクションになります。.pctext セクションに配置した関数への呼び出しや，アドレス参照を，PC 相対で行うことにより，任意の位置での実行を実現します。

8.6.2 PIROD

CC-RH では，定数データの参照を PC 相対で行うことにより，位置独立を実現します。コードと同じ相対参照方法であるため，PIROD 機能は PIC 機能と同時に有効にしたうえで，コード領域と定数データ領域の距離がリンク時と同じになるように，実行したい位置に適切に配置する必要があります。

-pirod オプションを指定すると，定数データの配置先のデフォルト・セクションが .pcconst32 セクションになります。そして，.pcconst32 セクションに配置した定数データへの参照を，PC 相対参照で行うことにより，任意の位置への配置と参照をを実現します。#pragma section 指令や -Xsection オプションを使用することで，配置先セクションを .pcconst16 セクションや .pcconst23 セクションに変更できます。これにより，より短い命令長で定数データを参照できます。

8.6.3 PID

CC-RH では，データの参照を GP 相対または EP 相対で行うことにより，位置独立を実現します。

-pid オプションを指定すると，データの配置先のデフォルト・セクションが，.sdata32 セクションと .sbss32 セクションになります。そして，これらのセクションに配置したデータへの参照コードを，GP 相対で出力します。PID 機能を使用しない場合と同様に，#pragma section 指令や -Xsection オプションを使用することで，配置先セクションを .sdata セクションや .sdata23 セクションに変更できます。

8.6.4 位置独立プログラムから位置独立でないプログラムへの参照

特定の手順によって，既存資産などの位置独立ではないプログラムを，位置独立プログラムから参照することができます。この場合の位置独立ではない側のプログラムを，共有部と呼びます。

(1) 事前の条件として，位置独立プログラムを作成する前に，共有部を作成済みである必要があります。
共有部の実行形式の作成時に，最適化リンカの -fsymbol オプションを使用して，位置独立プログラムから参照したい関数や変数のアドレスを，.fsy ファイルに出力しておきます。

- ライブラリ関数を共有部内では参照しないが，位置独立プログラムから共有部として参照したい場合は，次のような疑似的な参照コードを記述し，共有部にライブラリ関数をリンクさせます。

```
#include <string.h>
void* const dummy_libcall[] = {&memcpy, &memcmp, &strcpy};
```

次の操作で，common.fsy ファイルが出力されます。

```
>ccrh dummy_libcall.c -ocommon.abs -Xlk_option=-fsymbol=.text
```

すでに作成済みの共有部の実行形式を位置独立プログラムから参照したい場合は，共有部のリンク・マップ・ファイルなどで参照したい関数や変数のアドレスを確認し，.fsy ファイルに記述します。

例　　リンク・マップ・ファイルに次のようにアドレスが表示されている場合：

```
FILE=memcmp
                                00002000  00002023        24
  _memcmp
                                00002000          0   none ,g         *
FILE=memcpy
                                00002024  0000203b        18
  _memcpy
                                00002024          0   none ,g         *
FILE=strcpy
                                0000203c  0000204f        14
  _strcpy
                                0000203c          0   none ,g         *
```

common.fsy ファイルに記述する内容：

```
        .public _memcmp
_memcmp .equ 0x2000
        .public _memcpy
_memcpy .equ 0x2024
        .public _strcpy
_strcpy .equ 0x203c
```

(2) 位置独立プログラムの作成時には，参照したい共有部の関数や変数の宣言と，参照する処理を記述します。このとき，宣言の所属するセクションを，共有部側の関数，変数定義のセクションと合わせておきます。参照する側の PIC 関数は，PIC 用のセクションに定義してください。

注 -pid，-pirod，-pid オプション指定時でも，#pragma section 指令に，text，const，r0_disp16 等の位置独立でないセクション再配置属性を指定することは可能です。この場合，関数や変数の宣言のみを記述できます。関数や変数の定義を記述するとエラーになります。

```
#pragma section text
extern  void *memcpy(void *, const void *, unsigned long);
extern  int memcmp(const void *, const void *, size_t);
extern  char *strcpy(char *, const char *);

#pragma section pctext  /* PIC 関数を定義するときは，セクション再配置属性を PIC 用に戻すこと
*/

void pic_func(char* a, char* b, unsigned long c) {
  memcpy(a, b, c);
}
```

共有部の関数，変数を参照するコードを記述し，位置独立プログラムをビルドします。このとき，共有部から作成した .fsy ファイを一緒にビルドすることで，共有部側にある関数や変数への参照を絶対アドレスで解決します。

```
>ccrh pic.c common.fsy
```

(3) 位置独立プログラムから参照できる共有部の変数や関数には，参照方法に制約があります。
位置独立プログラム同士，また位置独立プログラムと共有部との間の，参照可能な関係と参照方法を次の表に示します。

		参照先					
		PIC 関数	非 PIC 関数	PIROD 変数	非 PIROD 変数	PID 変数[注 2]	非 PID 変数
参照元	PIC 関数	PC 相対	R0 相対	PC 相対	R0 相対	GP,EP 相対	GP,EP 相対 R0 相対
	非 PIC 関数	不可[注 1]	PC 相対 R0 相対	不可[注 1]	R0 相対	GP,EP 相対	GP,EP 相対 R0 相対

注 1. 非 PIC 関数のリンク時には，リンカが PIC 関数，PIROD 変数の実行時のアドレスを特定できないため，あらゆる直接参照ができません。実行時にポインタを受け取って，ポインタ経由で参照することは可能です。

注 2. PID 変数とは，GP 相対，EP 相対セクションに配置している変数全般ではなく，-pid オプションを指定してコンパイルした変数を指します。

8.6.5 PIC/PID 機能の制約

(1) 位置独立であるコードやデータは，リンク時とは異なるアドレスで動作します。そのため，位置独立であるコードやデータのアドレスを，静的変数の初期化子に指定することはできません。

(2) GP 相対，EP 相対セクションは，位置独立データと，位置独立でないデータの両方で使用可能です。ただし，GP，EP レジスタを共有しているため，PID 機能を使用するために GP，EP レジスタの値を変更すると，位置独立でないデータの参照アドレスも変更されます。GP，EP レジスタはそれぞれについて，位置独立データ用に使うか，位置独立でないデータ用に使うかを，プログラム全体で統一することを推奨します。

(3) 標準ライブラリは，PIC，PIROD，PID 機能に対応していません。共有部に配置して使用してください。

(4) PIC/PID 機能を使用するとセクション名が変わるので，リンカの -start オプション指定も合わせて変更する必要があります。

(5) PIC/PID 機能を使用する場合，標準のスタートアップ・ルーチンは使用できません。「8.6.6 スタートアップ・ルーチン」を参照して，スタートアップ・ルーチンを作成してください。

8.6.6 スタートアップ・ルーチン

PIC，PIROD 機能や PID 機能を使用する場合，スタートアップ・ルーチン内の次の処理を変更する必要があります。

- リセット・ベクタ
- ベース・レジスタ初期化
- RAM セクションの初期化
- main 関数への分岐

(1) リセット・ベクタ
プログラム全体を位置独立として構成する場合は，リセット・ベクタから任意の位置に分岐させる必要があります。このため，例えば，特定の RAM 領域や，データ・フラッシュ領域に，分岐先アドレスを書き込んでおきます。または，マイコンの電源を遮断しないでプログラムを再起動する場合は，特定のレジスタに分岐先アドレスを格納しておきます。

リセット・ベクタでは，分岐したいアドレスを取得して，レジスタ間接分岐を実行します。

```
    cstart_address .set 0xXXXXXXXX    ; 実行したい分岐先アドレスを格納しているアドレス

    .section "RESET", text
    .align   512
    mov cstart_address, r10
    ld.w 0[r10], r10
    jmp [r10]
```

(2) ベース・レジスタ初期化
PID 機能を使用する場合は，まず事前に，リンク時に指定した RAM セクションの開始アドレスから，実行時にどれだけずらして配置するか，のオフセット情報（以降 RAM オフセット値と呼びます）を，受け渡す手段を決めておきます。例えば，特定の RAM 領域や，データ・フラッシュ領域に RAM オフセット値を書き込んでおきます。注

注 この場合，その特定の領域は絶対アドレスで参照する必要があるため，PID，PIROD 機能の対象外になります。

または，マイコンの電源を遮断しないでプログラムを再起動する場合は，特定のレジスタに RAM オフセット値を格納しておきます。
受け取った RAM オフセット値を，ベース・レジスタに加算して，実行時のベース・アドレスとして使用します。

```
    mov     0xfedf0000, r28          ; 受け渡し用のメモリのアドレス
    ld.w    0[r28], r28              ; リンク時のデータ配置と，実行時のデータ配置の間の
                                     ; オフセット（RAM オフセット）

    mov     #_stacktop, sp           ; set sp register
    mov     #__gp_data, gp           ; set gp register
    mov     #__ep_data, ep           ; set ep register
    add     r28, sp
    add     r28, gp
    add     r28, ep
```

(3) RAM セクションの初期化

_INITSCT_RH() 関数はセクション情報のテーブルが入力であるため，PID 機能使用時のセクション初期化には使用できません。そこで，スタートアップ・ルーチン内で直接初期値をコピーします。
事前準備として，コード領域，定数データ領域の，リンク時と実行時の配置オフセットを求めます。以降 ROM オフセット値と呼びます。

```
    jarl    .pic_base, r29   ; r29 に実行時の .pic_base ラベルのアドレスを格納する
.pic_base:
    mov     #.pic_base, r10  ; r10 にリンク時の .pic_base ラベルのアドレスを格納する
    sub     r10, r29         ; r29 - r10 の値が，ROM オフセット値になる
```

次に，初期値ありセクションの初期化を行います。
初期化の対象となるセクションの，初期値のコピー元の先頭アドレス，終端アドレス，コピー先アドレスを r6，r7，r8 レジスタに格納します。

```
    mov     #__s.sdata32, r6
    mov     #__e.sdata32, r7
    mov     #__s.sdata32.R, r8
```

PIROD 機能を使用している場合，初期値のコピー元の先頭アドレス，終端アドレス（r6,r7 レジスタ）に，ROM オフセット値を加算します。

```
    add     r29, r6
    add     r29, r7
```

PID 機能を使用している場合，データのコピー先アドレス（r8 レジスタ）に RAM オフセット値を加算します。

```
    add     r28, r8
```

ここまででコピーの準備ができたので，コピールーチンを呼び出します。

```
    jarl    _copy4, lp
....
    ; r6: source begin  (4-byte aligned)
    ; r7: source end  (r6 <= r7)
    ; r8: destination begin  (4-byte aligned)
    .align  2
_copy4:
    sub     r6, r7
.copy4.1:
    cmp     4, r7
    bl      .copy4.2
    ld.w    0[r6], r10
    st.w    r10, 0[r8]
    add     4, r6
    add     4, r8
    add     -4, r7
    br      .copy4.1
.copy4.2:
    cmp     2, r7
    bl      .copy4.3
    ld.h    0[r6], r10
    st.h    r10, 0[r8]
    add     2, r6
    add     2, r8
    add     -2, r7
.copy4.3:
    cmp     0, r7
    bz      .copy4.4
    ld.b    0[r6], r10
    st.b    r10, 0[r8]
.copy4.4:
    jmp     [lp]
```

ここまでの処理を，初期値が必要なセクションの数だけ繰り返します。

次に，初期値無しセクションを 0 で初期化します。対象セクションの先頭アドレス，終端アドレスを，r6，r7 レジスタに格納します。

```
    mov     #__s.sbss32, r6
    mov     #__e.sbss32, r7
```

PID 機能を使用している場合，先頭アドレス，終端アドレス（r6，r7 レジスタ）に RAM オフセット値を加算します。

```
    add     r28, r6
    add     r28, r7
```

初期化ルーチンを呼び出して，対象セクションを 0 で初期化します。

```
    jarl    _clear4, lp
....
    ; r6: destination begin (4-byte aligned)
    ; r7: destination end  (r6 <= r7)
    .align  2
_clear4:
    sub     r6, r7
.clear4.1:
    cmp     4, r7
    bl      .clear4.2
    st.w    r0, 0[r6]
    add     4, r6
    add     -4, r7
    br      .clear4.1
.clear4.2:
    cmp     2, r7
    bl      .clear4.3
    st.h    r0, 0[r6]
    add     2, r6
    add     -2, r7
.clear4.3:
    cmp     0, r7
    bz      .clear4.4
    st.b    r0, 0[r6]
.clear4.4:
    jmp     [lp]
```

ここまでの処理を，初期化が必要なセクションの数だけ繰り返します。

(4) main 関数への分岐

PIC 機能を使用していて，FERET 命令で main 関数へ分岐する場合，FEPC に格納する値に ROM オフセット値を加算します。

```
    mov     #_exit, lp          ; lp <- #_exit
    mov     #_main, r10

    add     r29, lp             ; ROM オフセット値を加算
    add     r29, r10            ; ROM オフセット値を加算

    ldsr    r10, 2, 0           ; FEPC <- #_main

            ; apply PSW and PC to start user mode
    feret
```

コーディング例

PIC，PIROD，PID 機能使用向けのコーディング例を次に示します。

```
$ifdef __PIC
  .TEXT .macro
    .section .pctext, pctext
  .endm
$else
  .TEXT .macro
    .section .text, text
  .endm
$endif
```

```
$ifdef __PID
  .STACK_BSS .macro
    .section .stack.bss, sbss32
  .endm
$else
  .STACK_BSS .macro
    .section .stack.bss, bss
  .endm
$endif

;------------------------------------------------------------------------------
;   system stack
;------------------------------------------------------------------------------
STACKSIZE    .set    0x200
    .STACK_BSS
    .align   4
    .ds      (STACKSIZE)
    .align   4
_stacktop:

;------------------------------------------------------------------------------
;    startup
;------------------------------------------------------------------------------
    .TEXT
    .public __cstart
    .align  2
__cstart:

$ifdef __PIC
    jarl    .pic_base, r29
.pic_base:
    mov     #.pic_base, r10
    sub     r10, r29
$endif

$ifdef __PID
    mov     0xfedf0000, r28          ; 受け渡し用のメモリのアドレス
    ld.w    0[r28], r28              ; リンク時のデータ配置と，実行時のデータ配置の間の
                                     ; オフセット（RAM オフセット）
$endif

    mov     #_stacktop, sp           ; set sp register
    mov     #__gp_data, gp           ; set gp register
    mov     #__ep_data, ep           ; set ep register
$ifdef __PID
    add    r28, sp
    add    r28, gp
    add    r28, ep
$endif

    ; initialize 1 data section
$ifdef __PID
  $ifdef __PIROD
    mov     #__s.sdata32, r6
    add     r29, r6
    mov     #__e.sdata32, r7
    add     r29, r7
    mov     #__s.sdata32.R, r8
    add     r28, r8
```

```
  $else
    mov     #__s.sdata32, r6
    mov     #__e.sdata32, r7
    mov     #__s.sdata32.R, r8
    add     r28, r8
  $endif
$else
  $ifdef __PIROD
    mov    #__s.data, r6
    add    r29, r6
    mov    #__e.data, r7
    add    r29, r7
    mov    #__s.data.R, r8
  $else
    mov    #__s.data, r6
    mov    #__e.data, r7
    mov    #__s.data.R, r8
  $endif
$endif
    jarl   _copy4, lp

    ; initialize 1 bss section
$ifdef __PID
    mov    #__s.sbss32, r6
    mov    #__e.sbss32, r7
    add    r28, r6
    add    r28, r7
$else
    mov    #__s.bss, r6
    mov    #__e.bss, r7
$endif
    jarl   _clear4, lp

    ; enable FPU
$if 1 ; disable this block when not using FPU
    stsr   6, r10, 1                ; r10 <- PID
    shl    21, r10
    shr    30, r10
    bz     .L1                      ; detecting FPU
    stsr   5, r10, 0                ; r10 <- PSW
    movhi  0x0001, r0, r11
    or     r11, r10
    ldsr   r10, 5, 0                ; enable FPU

    movhi  0x0002, r0, r11
    ldsr   r11, 6, 0                ; initialize FPSR
    ldsr   r0, 7, 0                 ; initialize FPEPC
.L1:
$endif

    ; set various flags to PSW via FEPSW

    stsr   5, r10, 0                ; r10 <- PSW
    ;xori  0x0020, r10, r10         ; enable interrupt
    ;movhi 0x4000, r0, r11
    ;or    r11, r10                 ; supervisor mode -> user mode
    ldsr   r10, 3, 0                ; FEPSW <- r10
    mov    #_exit, lp               ; lp <- #_exit
    mov    #_main, r10
```

```
$ifdef __PIC
    add    r29, lp
    add    r29, r10
$endif
    ldsr   r10, 2, 0                ; FEPC <- #_main

    ; apply PSW and PC to start user mode
    feret

_exit:
    br     _exit                    ;  end of program

;-------------------------------------------------------------------------------
;   copy routine
;-------------------------------------------------------------------------------
    ; r6: source begin  (4-byte aligned)
    ; r7: source end  (r6 <= r7)
    ; r8: destination begin  (4-byte aligned)
    .align 2
_copy4:
    sub    r6, r7
.copy4.1:
    cmp    4, r7
    bl     .copy4.2
    ld.w   0[r6], r10
    st.w   r10, 0[r8]
    add    4, r6
    add    4, r8
    add    -4, r7
    br     .copy4.1
.copy4.2:
    cmp    2, r7
    bl     .copy4.3
    ld.h   0[r6], r10
    st.h   r10, 0[r8]
    add    2, r6
    add    2, r8
    add    -2, r7
.copy4.3:
    cmp    0, r7
    bz     .copy4.4
    ld.b   0[r6], r10
    st.b   r10, 0[r8]
.copy4.4:
    jmp    [lp]

;-------------------------------------------------------------------------------
;   clear routine
;-------------------------------------------------------------------------------
    ; r6: destination begin (4-byte aligned)
    ; r7: destination end  (r6 <= r7)
    .align 2
_clear4:
    sub    r6, r7
.clear4.1:
    cmp    4, r7
    bl     .clear4.2
    st.w   r0, 0[r6]
    add    4, r6
    add    -4, r7
    br     .clear4.1
```

```
.clear4.2:
    cmp    2, r7
    bl     .clear4.3
    st.h   r0, 0[r6]
    add    2, r6
    add    -2, r7
.clear4.3:
    cmp    0, r7
    bz     .clear4.4
    st.b   r0, 0[r6]
.clear4.4:
    jmp    [lp]

;-------------------------------------------------------------------------------
;   dummy section
;-------------------------------------------------------------------------------
$ifdef __PID
    .section .sdata32, sdata32
.L.dummy.sdata32:
    .section .sbss32, sbss32
.L.dummy.sbss32:
$else
    .section .data, data
.L.dummy.data:
    .section .bss, bss
.L.dummy.bss:
$endif

$ifdef __PIROD
    .section .pcconst32, pcconst32
.L.dummy.pcconst32
$else
    .section .const, const
.L.dummy.const:
$endif
;-------------------- end of start up module -------------------;
```

9. 関数呼び出し仕様

この章では，CC-RH 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 関数呼び出しインタフェース

この節では，CC-RH 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。

9.1.1 関数呼び出し前後で保証される汎用レジスタ

関数呼び出しの前後で，レジスタの内容が同一であることを保証する汎用レジスタと，保証しない汎用レジスタがあります。汎用レジスタの保証規則を以下に示します。

(1) 関数呼び出し前後で内容が保証されるレジスタ（Callee-Save レジスタ）
これらの汎用レジスタは，呼び出し先の関数において退避／復帰が必要なものです。そのため，呼び出し元では，関数呼び出しの前後でレジスタの中身が同一であることが保証されます。
r20，r21，r22，r23，r24，r25，r26，r27，r28，r29，r30[注]，r31

注 1. r30（EP）は，プログラム全体で固定して使用する場合があります。固定して使用する場合は，プログラム全体で汎用レジスタの内容を変更しないため，呼び出し先での退避／復帰は不要となります。

(2) 関数呼び出し前後で内容が保証されないレジスタ（Caller-Save レジスタ）
上記の関数呼び出し前後で内容が保証されるレジスタ以外の汎用レジスタは，呼び出し先の関数において内容が書き換わる可能性があります。そのため，呼び出し元では，関数呼び出しの前後でレジスタの中身が同一であることが保証されません。

備考 1. r1 はアセンブラが使用する場合があるレジスタなので，書き換えはユーザ責任となります。

備考 2. r2 は OS 予約となる場合があります。予約レジスタは，コンパイラは汎用レジスタとしては使用しないので，ここに書いた規則には当てはまりません。書き換えはユーザ責任となります。

備考 3. r3 はスタック・ポインタであり，汎用レジスタの用途としては使わないのでここに書いた規則には当てはまりません。書き換えはユーザ責任となります。

備考 4. r2，r4，r30 はオプションによって使用方法を指定することがきます。

9.1.2 引数と戻り値の設定方法，および参照方法

(1) 引数の受け渡し方法
引数には，レジスタで渡すものとスタックで渡すものがあります。各引数をどちらで渡すのかは，以下の手順を実施することにより決定されます。

(a) 各引数をスタック上に割り付けたメモリ・イメージを作成する

<1> 2 バイト以下のスカラ型は，4 バイトに整数拡張して格納されます。

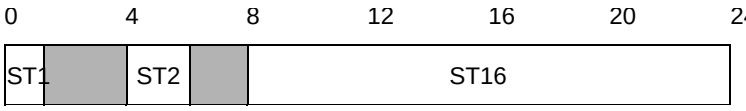
<2> 各引数は，基本的にはすべて 4 バイト境界へと配置されます。

<3> 戻り値が構造体や共用体の場合，メモリ・イメージの先頭に，戻り値データを書き込むアドレスを設定されます。

<4> 関数原型が不明な場合には，各スカラ型の引数は以下のように格納されます。

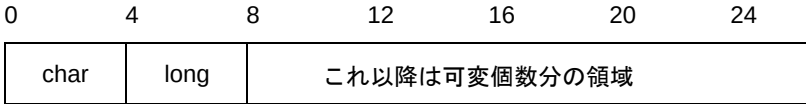
- 1 バイトのスカラ型整数 → 4 バイトに整数拡張して格納
- 2 バイトのスカラ型整数 → 4 バイトに整数拡張して格納
- 4 バイトのスカラ型整数 → そのまま格納
- 8 バイトのスカラ型整数 → そのまま格納
- 4 バイトのスカラ型浮動小数点数 → 8 バイトの浮動小数点数に拡張して格納
- 8 バイトのスカラ型浮動小数点数→ そのまま格納

例 1. 関数原型 : f(ST1, ST2, ST16)
STx は，サイズが x[byte] の構造体であることを表している場合の例



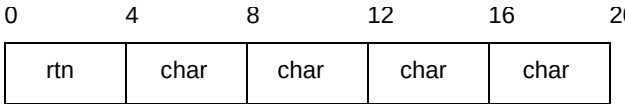
サイズが 4 の倍数でない構造体や共用体の場合，引数同士の間にパディング（図のグレー部分）ができ，この部分は内容が不定となります。

例 2. 関数原型 : f(char, long, ...)
可変個数の実引数を受け取る場合の例



例の「これ以降は可変個数分の領域」では，実引数で設定する個数分だけメモリを消費します。

例 3. 関数原型 : ST4 f(char, char, char, char)



rtn で，ST4 の戻り値を書き込む場所のアドレスを渡します。

(b) 作成したメモリ・イメージの先頭 4 ワード（16 バイト）をレジスタ r6 ～ r9 で，4 ワードに収まらない分をスタックで渡す

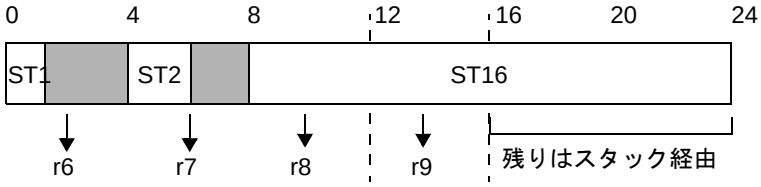
<1> レジスタ渡しの場合，各レジスタ（r6 ～ r9）へは各々ワード単位にロードされます。
バイト単位やハーフワード単位のロードはされません。

<2> スタックで渡す引数は，呼び出し側関数のスタック・フレーム内に設定されます。

<3> スタックで渡す引数をスタックへ設定する場合は，メモリ・イメージの右側から左側にかけてという順番でスタックへ格納されます。そのため，メモリ・イメージの 16 バイト・ オフセット位置のワードデータが，最も 0 に近い位置へ配置されます。

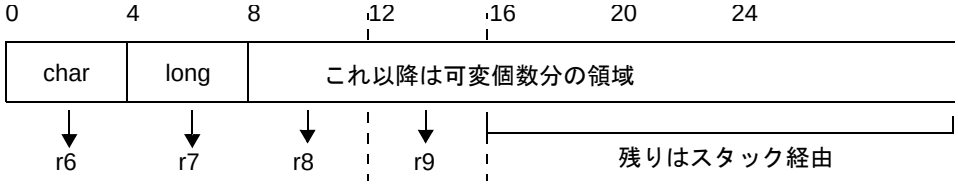
備考 スタックへの配置の方法は，「9.1.4 スタック・フレーム」を参照してください。

例 1. 関数原型 : f(ST1, ST2, ST16)
STx は，サイズが x[byte] の構造体であることを表している場合の例



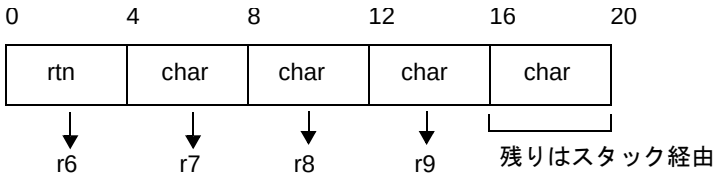
構造体（ここでは “ST16”）が一部分しかレジスタに設定できなくても，かまわずレジスタで渡します。

例 2. 関数原型 : f(char, long, ...)
可変個数の実引数を受け取る場合の例



可変個数であっても，レジスタが使用できる場合はレジスタで受け渡しします。

例 3. 関数原型 : ST4 f(char, char, char, char)



char 型の引数 4 個を渡すだけでも，戻り値によっては，4 番目の引数がスタック渡しとなる場合があります。

(2) 戻り値の受け渡し方法
戻り値の受け渡しの方法には，次の 3 通りがあります。

(a) 4 バイト以下のスカラ型の場合
r10 で戻り値を呼び出し側へ返却します。
4 バイト未満のサイズのスカラ型の場合，4 バイトに拡張したデータを r10 へと設定します。
戻り値の型が , 符号なしであればゼロ拡張 , 符号有りであれば符号拡張します。

(b) 8 バイトのスカラ型の場合
r10，r11 で戻り値を呼び出し側へ返却します。
r10 には下位 32 ビット，r11 には上位 32 ビットを設定します。

(c) 構造体や共用体の場合
戻り値が構造体や共用体の場合，呼び出し側は関数呼び出し時に，引数レジスタ r6 へ戻り値データを書き込む領域のアドレスを設定します。呼び出される側は，パラメータ・レジスタ r6 で指示されるアドレス位置に戻り値を設定し，呼び出し側関数へと返却します。
返却した時点では，呼び出し側の関数にとっては，r6 も r10 も不定（他の関数呼び出し前後で内容が保証されないレジスタと同じ) となります。
構造体や共用体は，サイズによらずすべて同じ返却方法となります。構造体や共用体のデータ自体をレジスタに設定して返却することはしません。

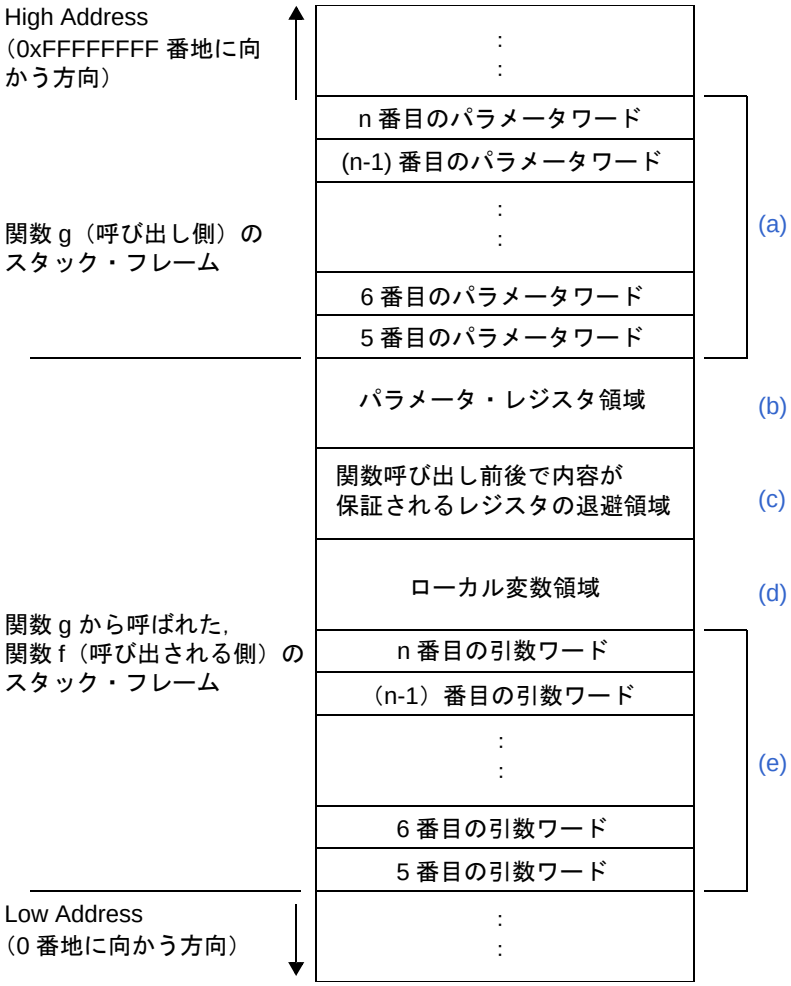
9.1.3 スタック・ポインタが指すアドレス

スタック・ポインタには，4 の倍数のアドレスが設定されます。
スタック・ポインタが指すアドレスは，4 の倍数ですが，スタックに格納するデータをすべて 4 バイト境界に整合する必要があるわけではありません。各データのアライメントに応じて，スタック内に格納される位置が決まります。たとえば，char 型データであれば，データのアライメント数は 1 なので，スタック内でも 1 バイト境界に配置できます。

9.1.4 スタック・フレーム

(1) スタック・フレーム内の構成
関数 f が，関数 g から呼び出された場合の，関数 f から見た双方のスタック・フレームの内容を以下に示します。

図 9.1 スタック・フレームの内容

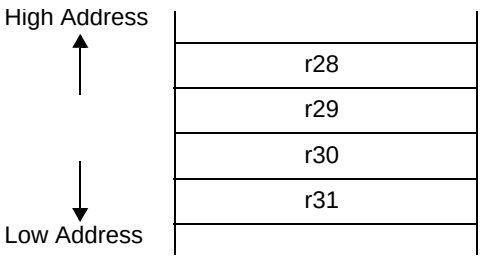


関数 f が参照，または設定できる範囲の領域の内容は以下です。

(a) 5 ～ n 番目のパラメータ・ワード
関数 f が 4 ワード（16 バイト）より大きいパラメータ・サイズを持つ場合に，4 ワードを越えた分のパラメータを設定している領域です。パラメータ・サイズが 4 ワード以下であれば，この領域はサイズ 0 となります。

(b) パラメータ・レジスタ領域
パラメータを受け取るレジスタ（r6 ～ r9）を設定するための領域です。16 バイト固定ではなく，不要であればサイズ 0 となります。
パラメータ・レジスタ領域の詳細は，「(2) パラメータ・レジスタ領域」を参照してください。

(c) 関数呼び出し前後で内容が保証されるレジスタの退避領域
関数 f 内で使用する関数呼び出し前後で内容が保証されるレジスタを退避するための領域です。退避する必要があるレジスタ本数分だけ，この領域のサイズが必要となります。
レジスタの退避と復帰は，基本的に prepare/dispose 命令を使用することになるので，この退避領域へはレジスタ番号昇順にストアされます。
たとえば，r28 ～ r31 を退避する場合は，以下の並び順となります。



(d) ローカル変数領域
ローカル変数として使用するためのスタック領域です。

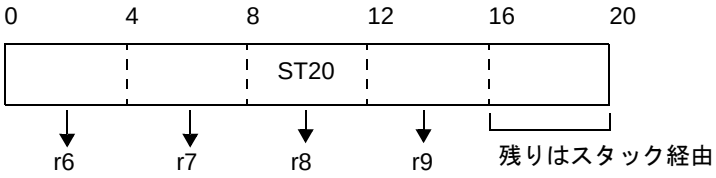
(e) 5 ～ n 番目の引数ワード

関数 f が他の関数を呼び出す際に，4 ワードより大きいサイズの引数を設定するための領域です。他の関数を呼び出す際に必要となる引数は，関数 f のスタック・フレーム内に領域を確保して設定します。
呼び出しに要する引数サイズが 4 ワード以下の場合，この領域はサイズ 0 となります。

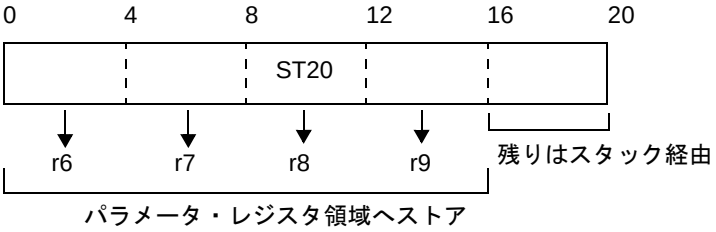
(2) パラメータ・レジスタ領域

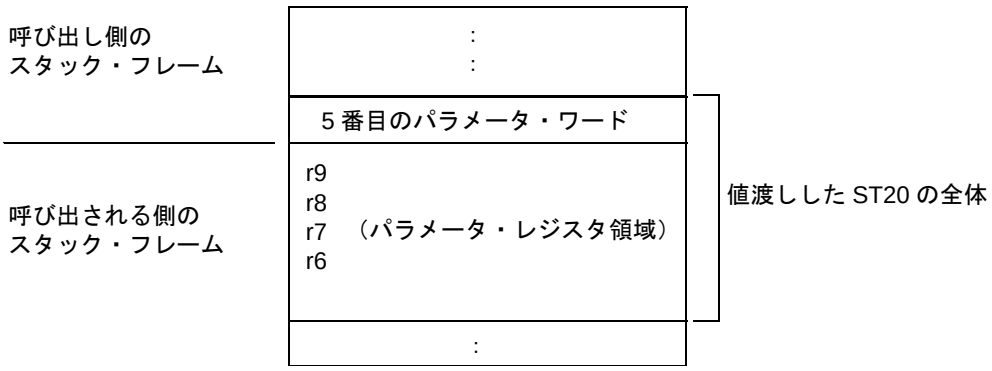
パラメータ・レジスタ領域は，パラメータ・サイズが 4 ワード（16 バイト）を越える場合に，必要に応じて確保する領域です。パラメータ・レジスタ（r6 ～ r9）を必要に応じてこの領域にストアするため，この領域のサイズは 0，4，8，12，16 バイトのいずれかです。
この領域は，パラメータ・レジスタの内容とスタック内のパラメータとを，連続した配置関係で参照する必要がある場合に，パラメータ・レジスタをストアするためのものです。
たとえば，サイズが 20 バイトの構造体引数を値渡しする場合，16 バイトは r6 ～ r9 で，残りの 4 バイトはスタック（5 番目のパラメータ・ワード）として渡されます。

例 関数原型 : f(ST20)



値渡しした構造体全体を参照するためには，メモリ上に全体を連続した位置関係で配置する必要がありますが，関数呼び出し直後は，レジスタとメモリに分割された状態になっています。
この場合，呼び出される側の関数でパラメータ・レジスタをスタックへストアすることにより，値渡しした ST20 をメモリ上で参照することができます。

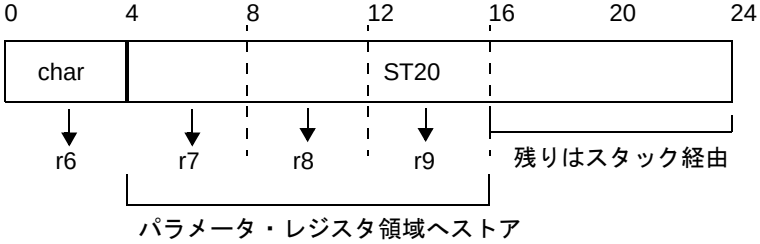




この領域が必要となる具体的なケースは，パラメータが以下の場合です。
これらいずれかに該当しなければ，パラメータ・レジスタをパラメータ・レジスタ領域へストアする必要はないため，パラメータ・レジスタ領域は不要（サイズ 0）となります。

(a) 構造体や共用体が，パラメータ・レジスタとスタックにまたがる場合

例 関数原型 : f(char, ST20)

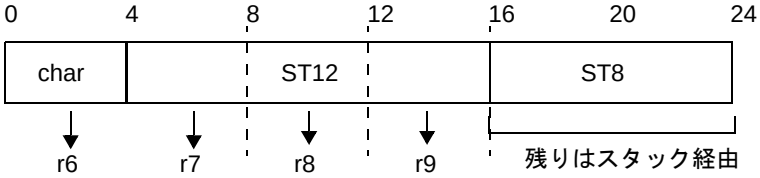


この場合には，r7 ～ r9 をパラメータ・レジスタ領域へストアします。

r6 は，ST20 と連続した位置のメモリ上に配置する必要がないのでストアしません。
したがって，パラメータ・レジスタ領域のサイズは 12 バイトとなります。

なお，構造体や共用体が，パラメータ・レジスタとスタックにまたがっていない場合は，パラメータ・レジスタ領域へのストアは不要になり，パラメータ・レジスタ領域はサイズ 0 バイトとなります。

例 関数原型 : f(char, ST12, ST8)

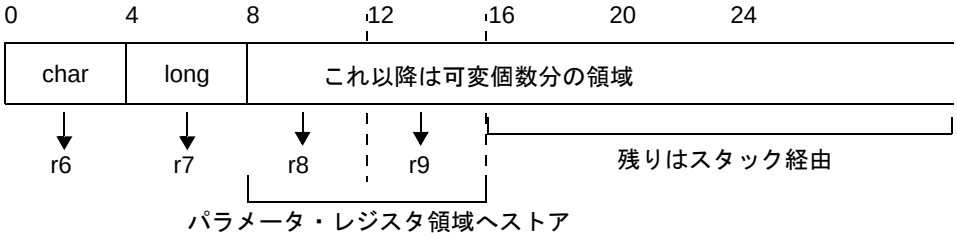


パラメータ・レジスタ領域へストアはありません。

この場合は，ST12 はパラメータ・レジスタ内にすべて収まり，ST8 はパラメータ・レジスタでは渡ってきません。
そのため，パラメータ・レジスタとスタックにはまたがらないので，パラメータ・レジスタ領域はサイズ 0 バイトとなります。
なお，パラメータ・レジスタで全体を受け取った構造体や共用体は，メモリ上に展開する際には，ローカル変数領域に展開されます。

(b) 可変個数の実引数を受け取る場合
可変個数の実引数を受け取る場合には，パラメータ・レジスタ領域へストアする必要があります。

例 関数原型 : f(char, long, ...)



この場合は，可変個数の実引数にあたるパラメータ・レジスタ（r8 と r9）がパラメータ・レジスタ領域へストアされます。
したがって，パラメータ・レジスタ領域のサイズは 8 バイトとなります。

9.2 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
CC-RH 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
CC-RH 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
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
CC-RH 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。

9.3 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
CC-RH 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
CC-RH 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
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
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
CC-RH 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。

9.4 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。

例 C 言語のプログラム

```
extern  char    c;
extern  int     i;

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

CC-RH アセンブラでは，次のように行います。

```
        .public _i
        .public _c
        .dseg   DATA
_i:
        .db4    0x0
_c:
        .db     0x0
```

9.5 汎用レジスタ

以下に，CC-RH における汎用レジスタの用い方を示します。

表 9.5 汎用レジスタの用い方

レジスタ	使用方法
r0	0 の値として演算時に使用 .data/.bss セクション参照のベース・レジスタ
r1	関数呼び出し前後で内容が保証されないレジスタ
r2	関数呼び出し前後で内容が保証されないレジスタ システム予約（OS）（オプションで切り替え）
r3（sp）	スタック・ポインタ
r4（gp）	PID 用グローバル・ポインタ 固定
r5（tp）	定数データ用グローバル・ポインタ 関数呼び出し前後で内容が保証されないレジスタ
r6 ～ r19	関数呼び出し前後で内容が保証されないレジスタ
r20 ～ r29	関数呼び出し前後で内容が保証されるレジスタ
r30（ep）	エレメント・ポインタ 固定，または関数呼び出し前後で内容が保証されるレジスタ（オプションで切り替え）
r31（lp）	リンク・ポインタ 関数呼び出し前後で内容が保証されるレジスタ

10. メッセージ

ここでは，CC-RH が出力するメッセージについて説明します。

10.1 概　　説

ここでは，CC-RH が出力する内部エラー・メッセージ，エラー・メッセージ，致命的エラー・メッセージ，インフォメーション・メッセージ，ワーニング・メッセージについて説明します。

10.2 メッセージ出力形式

ここでは，メッセージの出力形式について説明します。
メッセージの出力形式は，次のとおりです。

(1) ファイル名と行番号を含む場合

```
ファイル名 ( 行番号 ) : メッセージ種別 05 メッセージ番号 : メッセージ
```

(2) ファイル名と行番号を含まない場合

```
メッセージ種別 05 メッセージ番号 : メッセージ
```

備考　　下記内容が連続した文字列として出力されます。
メッセージ種別　　:　1 文字の英字
メッセージ　　:　5 桁の数値

10.3 メッセージ種別

ここでは，CC-RH が出力するメッセージ種別について説明します。
メッセージ種別（1 文字の英字）は，次のように分類されています。

表 10.1　　メッセージ種別

メッセージ種別	説明
C	内部エラー：処理を中止します。 オブジェクト・コードは生成しません。
E	エラー：一定数以上発生した場合，処理を中止します。 オブジェクト・コードは生成しません。
F	致命的エラー：処理を中止します。 オブジェクト・コードは生成しません。
M	インフォメーション：処理を続行します。 オブジェクト・コードを生成します。
W	ワーニング：処理を続行します。 オブジェクト・コードを生成します（ユーザが意図したものとは異なる可能性があります）。

10.4 メッセージ

ここでは，CC-RH が出力するメッセージについて説明します。

10.4.1 内部エラー

表 10.2 内部エラー

C05*nnnnn*	［メッセージ］	内部エラーが発生しました（*情報*）。
	［対処方法］	特約店，または当社までご連絡ください。
C0511200	［メッセージ］	内部エラーが発生しました (*error-information*)。
	［対処方法］	特約店，または当社までご連絡ください。
C0519996	［メッセージ］	Out of memory.
	［説明］	ccrh コマンドへの入力（ソース・ファイル名や指定オプション）の量が多すぎます。
	［対処方法］	ccrh コマンドへの入力を分割して，複数回に分けて起動してください。
C0519997	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0520000	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0529000	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530001	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530002	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530003	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530004	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530005	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0530006	［メッセージ］	Internal Error.
	［対処方法］	特約店，または当社までご連絡ください。
C0550802	［メッセージ］	Internal error(action type of icode strage).
	［対処方法］	特約店，または当社までご連絡ください。
C0550804	［メッセージ］	Internal error(section name ptr not found(*string*)).
	［対処方法］	特約店，または当社までご連絡ください。
C0550805	［メッセージ］	Internal error(section list ptr not found(*string*)).
	［対処方法］	特約店，または当社までご連絡ください。
C0550806	［メッセージ］	Internal error(current section ptr not found(*string*)).
	［対処方法］	特約店，または当社までご連絡ください。

C0550808	［メッセージ］	Internal error(*string*).
	［対処方法］	特約店，または当社までご連絡ください。
C0551800	［メッセージ］	Internal error.
	［対処方法］	特約店，または当社までご連絡ください。
C0564000	［メッセージ］	Internal error : (" *内部エラー番号* ") " *ファイル 行番号* " / " *コメント* "
	［説明］	最適化リンカの処理中に内部的な問題が発生しました。
	［対処方法］	メッセージ内の内部エラー番号，ファイル，行番号，コメントを添えて，特約店，または当社までご連絡ください。
C0564001	［メッセージ］	Internal error
	［対処方法］	特約店，または当社までご連絡ください。

10.4.2 エラー

表 10.3 エラー

E0511101	[メッセージ]	" *文字列* " オプションで指定された " *パス名* " はフォルダです。入力ファイルを指定してください。
E0511102	[メッセージ]	" *文字列* " オプションで指定されたファイル " *ファイル名* " が見つかりません。
E0511103	[メッセージ]	" *文字列* " オプションで指定された " *パス名* " はフォルダです。出力ファイルを指定してください。
E0511104	[メッセージ]	" *文字列* " オプションで指定された出力先フォルダ " *フォルダ名* " が見つかりません。
E0511107	[メッセージ]	" *文字列* " オプションで指定された " *パス名* " が見つかりません。
	[説明]	" *文字列* " オプションで指定された " *パス名* "（*ファイル名またはフォルダ名*）が見つかりません。
E0511108	[メッセージ]	" *文字列* " は認識されないオプションです。
E0511109	[メッセージ]	" *文字列* " オプションに引数は指定できません。
E0511110	[メッセージ]	" *文字列* " オプションに引数を指定してください。
	[説明]	" *文字列* " オプションは引数が必要です。引数を指定してください。
E0511113	[メッセージ]	" *文字列* " オプションに指定された引数が不正です。
E0511114	[メッセージ]	"-O *文字列* " オプションに指定された引数が不正です。
E0511115	[メッセージ]	"-O *文字列* " オプションの指定が不正です。
E0511116	[メッセージ]	"-O *文字列* " は認識されないオプションです。
E0511117	[メッセージ]	" *文字列* " オプションに指定されたパラメータが不正です。
E0511121	[メッセージ]	"-o" オプションと " *文字列* " オプションを同時に指定したとき，複数ソース・ファイルを入力できません。
E0511129	[メッセージ]	コマンド・ファイル " *ファイル名* " が複数回読まれています。
E0511130	[メッセージ]	コマンド・ファイル " *ファイル名* " が読み込めません。
E0511131	[メッセージ]	コマンド・ファイル " *ファイル名* " の構文が認識できません。
E0511132	[メッセージ]	テンポラリ・フォルダを作成できません。
E0511133	[メッセージ]	ソース・ファイルが複数の場合は "*option*" オプションにはフォルダを指定してください。
E0511134	[メッセージ]	指定された入力ファイル " *ファイル名* " が見つかりません。
E0511135	[メッセージ]	指定された入力ファイル " *パス名* " はフォルダです。
E0511145	[メッセージ]	" *文字列 1* " オプションで指定された " *文字列 2* " は使用できません。
E0511150	[メッセージ]	" *文字列 1* " オプションと " *文字列 2* " オプションが矛盾しています。
E0511152	[メッセージ]	" *文字列 1* " オプションには " *文字列 2* " オプションが必要です。
E0511154	[メッセージ]	*コンパイラ・パッケージ名*を構成するファイル " *ファイル名* " が見つかりません。再インストールしてください。
E0511178	[メッセージ]	*バージョン*に対応した professional 版のライセンスが確認できませんでした。" *文字列* " オプションを使用できません。professional 版の購入を検討ください。
E0511182	[メッセージ]	ファイルアクセスエラー。(*information*)

E0511200	［メッセージ］	内部エラーが発生しました (*error-information*)。
	［対処方法］	特約店，または当社までご連絡ください。
E0512001	［メッセージ］	テンポラリ・ファイル " *ファイル名* " の削除に失敗しました。
E0520006	［メッセージ］	ファイルの最後までコメントが閉じられていません。
E0520007	［メッセージ］	不明なトークンがあります。
E0520008	［メッセージ］	クォーテーションを閉じられていません。
E0520010	［メッセージ］	"#" はここには書けません。
	［説明］	"#" が正しくない位置に記述されています。
E0520011	［メッセージ］	不明な前処理指令があります。
E0520012	［メッセージ］	前に構文エラーがあるため，ここより文法の解析を再開します。
E0520013	［メッセージ］	ファイル名がありません。
E0520014	［メッセージ］	前処理指令の後に不正な文字があります。
E0520017	［メッセージ］	"]" がありません。
E0520018	［メッセージ］	")" がありません。
E0520019	［メッセージ］	数値の後に不正な文字があります。
E0520020	［メッセージ］	識別子 " *文字列* " は定義されていません。
E0520022	［メッセージ］	不正な 16 進数です。
E0520023	［メッセージ］	定数の値が大きすぎます。
E0520024	［メッセージ］	不正な 8 進数です。
	［説明］	不正な 8 進数です。8 進数に ‘8’，‘9’ は記述できません。
E0520025	［メッセージ］	引用文字列は少なくとも 1 文字を含まなければなりません。
E0520026	［メッセージ］	Too many characters in character constant.
	［説明］	文字定数中の文字が多すぎます。
E0520027	［メッセージ］	char 型の値が範囲を超えています。
E0520028	［メッセージ］	式は定数値を持つ必要があります。
E0520029	［メッセージ］	式がありません。
E0520030	［メッセージ］	浮動小数点数定数値が範囲を超えています。
E0520031	［メッセージ］	式は整数型を持つ必要があります。
E0520032	［メッセージ］	式は算術型を持つ必要があります。
E0520033	［メッセージ］	行番号がありません。
	［説明］	#line の後の行番号がありません。
E0520034	［メッセージ］	不正な行番号です。
	［説明］	#line の後の行番号が不正です。
E0520036	［メッセージ］	この前処理指令のための #if がありません。
E0520037	［メッセージ］	この前処理指令のための #endif がありません。

E0520038	[メッセージ]	この前処理指令は許可されていません -- #else はすでにあります。
	[説明]	#else が複数記述されているため，このディレクティブは不正です。
E0520039	[メッセージ]	0 で除算を行いました。
E0520040	[メッセージ]	識別子がありません。
E0520041	[メッセージ]	式は算術型かポインタ型を持つ必要があります。
E0520042	[メッセージ]	オペランドの型が適合しません (" *型1*" と " *型2*")。
E0520044	[メッセージ]	式はポインタ型を持つ必要があります。
E0520045	[メッセージ]	既定義名に対して #undef を使用できません。
E0520046	[メッセージ]	" *マクロ名* " を再定義することはできません。
	[説明]	" *マクロ名* " は既定義マクロです。再定義することはできません。
E0520047	[メッセージ]	マクロ " *マクロ名* " の適合しない再定義があります (宣言位置 行行番号)。
	[説明]	マクロ " *マクロ名* " の再定義が，行番号行での定義と適合しません。
E0520049	[メッセージ]	マクロの引数名が重複しています。
E0520050	[メッセージ]	マクロ定義の最初を "##" とすることはできません。
E0520051	[メッセージ]	マクロ定義の最後を "##" とすることはできません。
E0520052	[メッセージ]	マクロの引数名がありません。
E0520053	[メッセージ]	":" がありません。
E0520054	[メッセージ]	マクロに対する引数が足りません。
E0520055	[メッセージ]	マクロに対する引数が多すぎます。
E0520056	[メッセージ]	sizeof のオペランドに関数は書けません。
E0520057	[メッセージ]	この演算子は定数式では使用できません。
E0520058	[メッセージ]	この演算子はプリプロセッサ用の式には使用できません。
E0520059	[メッセージ]	定数式の中で関数を呼び出すことはできません。
E0520060	[メッセージ]	この演算子は整数型定数式には使用できません。
E0520061	[メッセージ]	整数演算の結果が範囲を超えました。
E0520062	[メッセージ]	シフト数が負数です。
E0520063	[メッセージ]	シフト数が多すぎます。
E0520064	[メッセージ]	宣言は何も宣言できません。
E0520065	[メッセージ]	";" がありません。
E0520066	[メッセージ]	enum の値が "int" の範囲を越えています。
E0520067	[メッセージ]	"}" がありません。
E0520069	[メッセージ]	整数変換で結果の値が丸められました。
E0520070	[メッセージ]	不完全型は許されていません。
E0520071	[メッセージ]	sizeof のオペランドにビット・フィールドは指定できません。
E0520075	[メッセージ]	"*" 演算子のオペランドはポインタ型である必要があります。
E0520077	[メッセージ]	宣言に記憶域クラスまたは型指定子がありません。

E0520078	[メッセージ]	引数宣言に初期化子は書けません。
E0520079	[メッセージ]	型指定子がありません。
E0520080	[メッセージ]	記憶域クラスはここでは指定できません。
E0520081	[メッセージ]	複数の記憶域クラスが指定されました。
	[説明]	複数の記憶域クラスが指定されました。記憶域クラスは 1 つしか指定できません。
E0520083	[メッセージ]	型修飾子が複数回指定されました。
	[説明]	型修飾子が複数回指定されました。型修飾子は 2 回以上指定できません。
E0520084	[メッセージ]	不正な型指定子の組み合わせです。
E0520085	[メッセージ]	引数に対する記憶域クラスが不正です。
E0520086	[メッセージ]	関数に対する記憶域クラスが不正です。
E0520087	[メッセージ]	型指定子はここでは使用できません。
E0520088	[メッセージ]	関数の配列は許されていません。
E0520089	[メッセージ]	void 型の配列は許されていません。
E0520090	[メッセージ]	関数を返す関数は許されていません。
E0520091	[メッセージ]	配列を返す関数は許されていません。
E0520092	[メッセージ]	引数の識別子リストは関数定義でのみ利用できます。
E0520093	[メッセージ]	関数型は typedef に指定できません。
E0520094	[メッセージ]	配列のサイズは正の整数でなければなりません。
E0520095	[メッセージ]	配列が大きすぎます。
E0520097	[メッセージ]	この型の値は関数返却値にできません。
E0520098	[メッセージ]	この型の配列は許されていません。
E0520099	[メッセージ]	ここでの宣言は引数宣言でなければなりません。
E0520100	[メッセージ]	引数名が重複しています。
E0520101	[メッセージ]	" *シンボル名* " はすでにこのスコープで宣言されています。
E0520102	[メッセージ]	列挙型の前方宣言は標準でありません。
E0520104	[メッセージ]	構造体または共用体が大きすぎます。
E0520105	[メッセージ]	ビット・フィールドのサイズが不正です。
E0520106	[メッセージ]	ビット・フィールドの型が不正です。
E0520107	[メッセージ]	サイズ 0 のビット・フィールドは名前を持てません。
E0520109	[メッセージ]	式は関数型または関数ポインタ型でなければなりません。
E0520110	[メッセージ]	タグ名または定義がありません。
E0520112	[メッセージ]	"while" がありません。
E0520114	[メッセージ]	タイプ " *シンボル* " は参照されていますが定義されていません。
E0520115	[メッセージ]	continue 文はループの中でのみ使用できます。
E0520116	[メッセージ]	break 文はループまたは switch の中でのみ使用できます。
E0520117	[メッセージ]	void でない 関数 " *名前* " は値を返す必要があります。

E0520118	[メッセージ]	void 関数は値を返せません。
E0520119	[メッセージ]	" *型名* " 型へのキャストは許されていません。
E0520120	[メッセージ]	返却値の型が関数の型と合っていません。
E0520121	[メッセージ]	case ラベルは switch の中でのみ使用できます。
E0520122	[メッセージ]	default ラベルは switch の中でのみ使用できます。
E0520124	[メッセージ]	default ラベルはすでにこの switch の中で使用されています。
E0520125	[メッセージ]	"(" がありません。
E0520127	[メッセージ]	文がありません。
E0520129	[メッセージ]	ブロック・スコープの関数は extern 記憶域クラスのみ指定できます。
E0520130	[メッセージ]	"{" がありません。
E0520132	[メッセージ]	式は構造体か共用体へのポインタでなければなりません。
E0520134	[メッセージ]	フィールド名がありません。
E0520136	[メッセージ]	*種別* " *シンボル名* " はフィールド " *フィールド名* " を持ちません。
E0520137	[メッセージ]	式は変更可能な左辺値である必要があります。
E0520138	[メッセージ]	レジスタ変数に対するアドレス演算子は許されていません。
E0520139	[メッセージ]	ビット・フィールドに対するアドレス演算子は許されていません。
E0520140	[メッセージ]	関数呼び出しに対する引数が多すぎます。
E0520141	[メッセージ]	名前なしでプロトタイプ宣言された引数は関数定義がある場合には許されていません。
E0520142	[メッセージ]	式はオブジェクト型へのポインタである必要があります。
E0520144	[メッセージ]	型 " *型名 1*" の値は型 " *型名 2*" の実体の初期化には使用できません。
E0520145	[メッセージ]	*種別* " *シンボル名* " は初期化できません。
E0520146	[メッセージ]	初期化子が多すぎます。
E0520147	[メッセージ]	宣言は " *宣言* "(宣言位置 行 *行番号*) と整合しません。
E0520148	[メッセージ]	*種別* " *シンボル名* " はすでに初期化されています。
E0520149	[メッセージ]	グローバル・スコープの宣言ではこの記憶域クラスを指定できません。
E0520151	[メッセージ]	typedef 名は引数として再宣言できません。
E0520154	[メッセージ]	式は構造体または共用体型である必要があります。
E0520158	[メッセージ]	式は左辺値か関数指示子である必要があります。
E0520159	[メッセージ]	宣言は以前の " *宣言* "(宣言位置 行 *行番号*) と整合しません。
E0520165	[メッセージ]	関数呼び出しに引数が足りません。
E0520166	[メッセージ]	不正な浮動小数点定数です。
E0520167	[メッセージ]	" *型名 1*" 型の引数は型 " *型名 2*" の引数と整合しません。
E0520168	[メッセージ]	関数型はここでは許されていません。
E0520169	[メッセージ]	宣言がありません。
E0520171	[メッセージ]	不正な型変換です。
E0520173	[メッセージ]	浮動小数点数は要求された整数型に入りません。

E0520175	[メッセージ]	添え字が範囲を越えました。
E0520179	[メッセージ]	"%" の右オペランドが 0 です。
E0520183	[メッセージ]	キャストの型は整数型である必要があります。
E0520184	[メッセージ]	キャストの型は算術型かポインタ型である必要があります。
E0520220	[メッセージ]	整数値が要求された浮動小数点型に入りません。
E0520221	[メッセージ]	浮動小数点数値が要求された浮動小数点型に入りません。
E0520222	[メッセージ]	浮動小数点演算の結果が範囲を超えました。
E0520228	[メッセージ]	カンマが仕様にあっていません。
	[説明]	最後のカンマは標準ではありません。
E0520230	[メッセージ]	ビット・フィールドに対し標準でない型です。
E0520235	[メッセージ]	変数 " *変数名* " が不完全型で宣言されました。
E0520238	[メッセージ]	引数の指定子が不正です。
E0520240	[メッセージ]	宣言の指示子が重複しています。
E0520247	[メッセージ]	*種別* " *シンボル名* " はすでに定義されています。
E0520253	[メッセージ]	"," がありません。
E0520254	[メッセージ]	型名は許されていません。
E0520256	[メッセージ]	型名 " *型名* " が不正に再宣言されています。
E0520260	[メッセージ]	明示的な型がありません。"int" として扱います。
E0520268	[メッセージ]	ブロック内で実行文の後に宣言を置けません。
E0520274	[メッセージ]	不適切に終了したマクロの呼び出しがあります。
E0520296	[メッセージ]	左辺値でない配列の不正な利用です。
E0520301	[メッセージ]	*typedef* 名はすでに同じ型で宣言されています。
E0520325	[メッセージ]	inline 指定子は関数宣言のみに利用できます。
E0520375	[メッセージ]	宣言は typedef 名を必要とします。
E0520393	[メッセージ]	不完全クラス型へのポインタは許されていません。
E0520404	[メッセージ]	関数 "main" は inline 宣言できません。
E0520409	[メッセージ]	*種別* " *シンボル名* " は不完全型 " *型名* " を返します。
E0520411	[メッセージ]	引数は許されていません。
E0520450	[メッセージ]	型 "long long" は標準ではありません。
E0520469	[メッセージ]	*文字列 1* のタグの種類は，*文字列 2* の宣言と一致しません。
E0520494	[メッセージ]	typedef を伴う void の引数リストの宣言は標準でありません。
E0520513	[メッセージ]	型 " *型名 1*" の値は型 " *型名 2*" の実体として代入できません。
E0520520	[メッセージ]	集成体は "{...}" により初期化してください。
E0520525	[メッセージ]	依存文に宣言は許されません。
	[説明]	if() の直後に "{" なしで宣言を書くことはできません。
E0520526	[メッセージ]	引数は void 型を持てません。

E0520618	[メッセージ]	構造体か共用体に名前のないメンバがあります。
E0520619	[メッセージ]	名前のないフィールドは標準ではありません。
E0520643	[メッセージ]	"restrict" は許されていません。
E0520644	[メッセージ]	関数へのポインタまたは参照は，"restrict" 修飾できません。
E0520654	[メッセージ]	宣言修飾子が前の宣言と合致しません。
E0520655	[メッセージ]	修飾子*名前*はこの宣言では許されていません。
E0520660	[メッセージ]	パッキング値が不正です。
E0520702	[メッセージ]	"=" がありません。
E0520731	[メッセージ]	不完全型の配列は標準ではありません。
E0520749	[メッセージ]	型指定子は許されていません。
E0520757	[メッセージ]	*名前*は型名ではありません。
E0520765	[メッセージ]	標準でないキャラクタがオブジェクト風なマクロ定義の最初に指定されました。
E0520816	[メッセージ]	関数定義において "void" の返却型を修飾することはできません。
E0520852	[メッセージ]	式は完全型のオブジェクト型へのポインタである必要があります。
E0520861	[メッセージ]	入力行に不正な文字があります。
E0520862	[メッセージ]	関数は不完全型 " *型名* " を返します。
E0520886	[メッセージ]	整数定数に不正な接尾子があります。
	[説明]	整数定数に不正な接尾子（サフィックス）があります。
E0520935	[メッセージ]	"typedef" はここでは指定できません。
E0520938	[メッセージ]	返却型 "int" が関数 "main" の宣言で省略されました。
E0520965	[メッセージ]	間違ったユニバーサル・キャラクタ名です。
E0520966	[メッセージ]	ユニバーサル・キャラクタ名が不正なキャラクタを指定されました。
E0520967	[メッセージ]	ユニバーサル・キャラクタ名は基本的なキャラクタ・セットの文字を指定できません。
E0520968	[メッセージ]	このユニバーサル・キャラクタは識別子として許されていません。
E0520969	[メッセージ]	識別子 __VA_ARGS__ は可変個引数マクロのリストのリプレースのみ使用できます。
E0520976	[メッセージ]	複合リテラルは整数定数式に使用できません。
E0520977	[メッセージ]	複合リテラル型*名前*は許されていません。
E0521029	[メッセージ]	サイズの不明な配列を持つ型は許されていません。
E0521030	[メッセージ]	静的変数は inline 関数で定義できません。
E0521031	[メッセージ]	内部リンケージを持つ実体は外部リンケージを持つ inline 関数で参照できません。
E0521036	[メッセージ]	予約語 " *シンボル* " は関数の中でのみ使用できます。
E0521037	[メッセージ]	このユニバーサル・キャラクタは識別子の先頭に使用できません。
E0521038	[メッセージ]	文字列リテラルがありません。
E0521039	[メッセージ]	認識されない STDC pragma です。
E0521040	[メッセージ]	"ON" か "OFF" または "DEFAULT" がありません。

E0521045	［メッセージ］	指示子の種類が不正です。
E0521049	［メッセージ］	可変長の配列に初期化子は指定できません。
E0521051	［メッセージ］	*名前*の型は直後の宣言により与えられなければなりません ("int" に仮定されます)。
E0521052	［メッセージ］	インライン関数*名前*には定義が必要です。
E0521072	［メッセージ］	宣言はラベルを持てません。
E0521144	［メッセージ］	記憶域クラスは auto または register である必要があります。
E0521158	［メッセージ］	返却型 void は修飾できません。
E0521260	［メッセージ］	不正なアライメント指定値です。
E0521261	［メッセージ］	整数リテラルがありません。
E0521381	［メッセージ］	キャリッジ・リターン文字がコメントおよび文字定数 / 文字列リテラルの外にありました。
	［説明］	キャリッジ・リターン文字 ('¥r') がコメントおよび文字定数／文字列リテラルの外にありました。
E0521578	［メッセージ］	case ラベル値はすでにこの switch の行*行番号*で現れています。
E0521584	［メッセージ］	文字列リテラルをかっこで囲むことは標準ではありません。
E0521649	［メッセージ］	*名前*とその置換テキストの間には空白が必要です。
E0523005	［メッセージ］	#pragma の構文が不正です。
	［説明］	#pragma の構文はフォーマットに合わせて書いてください。
E0523006	［メッセージ］	このシンボルは既に他の #pragma 指定がされています。
	［説明］	1 つのシンボルに対して，同時指定不可能な #pragma を 2 個以上指定しています。
E0523007	［メッセージ］	シンボル定義後の宣言にのみ #pragma 指定することはできません。
	［説明］	#pragma は対象のシンボル定義よりも先に宣言してください。
E0523008	［メッセージ］	不正な #pragma を指定しました。
	［説明］	このシンボルに対して，この #pragma を指定することはできません。
E0523026	［メッセージ］	PIC 機能使用時のアドレスの使用方法が不適切です。
E0523027	［メッセージ］	PID 機能使用時のアドレスの使用方法が不適切です。
E0523048	［メッセージ］	割り込み関数を不正に参照しています。
E0523057	［メッセージ］	セクションの属性指定文字が不正です。
	［説明］	セクションの属性指定文字として，使用不可能な文字列を使用しました。
E0523058	［メッセージ］	#pragma section の構文が正しくありません。
	［説明］	#pragma section の構文が文法に違反しています。
E0523065	［メッセージ］	ビット・フィールドの初期値にアドレス定数を記述できません。
E0523066	［メッセージ］	このセクションは現在のオプションでは使用できません。
E0523067	［メッセージ］	宣言子のネストが深すぎます。
E0523069	［メッセージ］	pm 番号を複数使うことはできません。
E0523070	［メッセージ］	"cmn" 指定変数は，r0 相対のみでアクセスできます。
E0523071	［メッセージ］	"cmn" 指定関数がアクセス可能な静的変数は r0 相対に限ります。

E0523072	［メッセージ］	"cmn" 指定関数が呼出し可能な "pmodule" 指定関数は，"cmn" 指定に限ります。
E0523073	［メッセージ］	*名前*で使用できない組み込み関数です。
E0523087	［メッセージ］	" *関数名* " を不正に参照しています。
E0523090	［メッセージ］	引数は __fp16 型を持てません。
E0523091	［メッセージ］	__fp16 を返す関数は許されていません。
E0523118	［メッセージ］	要素名が不正または指定されていません。
E0523119	［メッセージ］	ベクタ型要素のアドレスは取得できません。
E0523122	［メッセージ］	PIROD 機能使用時のアドレスの使用方法が不適切です。
E0523123	［メッセージ］	このセクションでは関数定義を行えません。
E0523124	［メッセージ］	このセクションでは変数定義を行えません。
E0523125	［メッセージ］	このセクションでは文字列リテラルを使用できません。
E0523126	［メッセージ］	このセクションの関数には "cmn" 指定できません。
E0523127	［メッセージ］	このセクションの文字列リテラルには "cmn" 指定できません。
E0550200	［メッセージ］	整列条件の指定に誤りがあります。
	［対処方法］	整列条件の指定を確認してください。
E0550201	［メッセージ］	扱うことのできない文字が現れました。
	［対処方法］	文字を確認してください。
E0550202	［メッセージ］	式の構成に誤りがあります。
	［対処方法］	式を確認してください。
E0550203	［メッセージ］	式の要素 *string* に誤りがあります。
	［対処方法］	式の要素を確認してください。
E0550207	［メッセージ］	異なる形式のラベル参照（#label，label，および $label）の間に演算が指定されています。
	［対処方法］	式を確認してください。
E0550208	［メッセージ］	異なるセクションに属するラベル間に演算が指定されています。
	［対処方法］	式を確認してください。
E0550209	［メッセージ］	ラベル同士の演算は同一ファイル内に定義してください。
	［対処方法］	式を確認してください。
E0550212	［メッセージ］	指定されたシンボルはすでに *label* として定義されています。
	［対処方法］	シンボル名を確認してください。
E0550213	［メッセージ］	ラベル *identifier* が複数回定義されています。
	［対処方法］	ラベル名を確認してください。
E0550214	［メッセージ］	*identifier* が複数回定義されています。
	［対処方法］	ラベル名を確認してください。
E0550220	［メッセージ］	名前に予約語 *identifier* が用いられています。
	［対処方法］	オペランドを確認してください。

E0550221	［メッセージ］	（ラベル - ラベル）の形式の式が指定されています。
	［対処方法］	式を確認してください。
E0550225	［メッセージ］	式の評価結果が負になりました。
	［対処方法］	式を確認してください。
E0550226	［メッセージ］	奇数のディスプレースメントが指定されています。
	［対処方法］	ディスプレースメントを確認してください。
E0550228	［メッセージ］	レジスタ以外のものが指定されています。
	［対処方法］	オペランドを確認してください。
E0550229	［メッセージ］	ベース・レジスタを指定する必要があります。
	［対処方法］	オペランドを確認してください。
E0550230	［メッセージ］	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。
	［対処方法］	ディスプレースメントを確認してください。
E0550231	［メッセージ］	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。
	［対処方法］	イミーディエトを確認してください。
E0550232	［メッセージ］	.local 疑似命令に指定されたパラメータが不正です。
	［対処方法］	パラメータを確認してください。
E0550234	［メッセージ］	.macro 疑似命令に指定されたパラメータが不正です。
	［対処方法］	パラメータを確認してください。
E0550235	［メッセージ］	.macro 疑似命令に定義されたマクロ名が不正です。
	［対処方法］	マクロ名を確認してください。
E0550236	［メッセージ］	マクロ呼び出しに指定された実引数が不正です。
	［対処方法］	パラメータを確認してください。
E0550237	［メッセージ］	.irp 疑似命令に指定された実引数が不正です。
	［対処方法］	引数を確認してください。
E0550238	［メッセージ］	.irp 疑似命令に指定されたパラメータが不正です。
	［対処方法］	パラメータを確認してください。
E0550239	［メッセージ］	RH850 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。
	［対処方法］	オペランドを確認してください。
E0550240	［メッセージ］	RH850 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。
	［対処方法］	オペランドを確認してください。
E0550242	［メッセージ］	ラベルはすでに定義されています。(*section*)
	［説明］	指定したラベルはすでに *section* セクションに定義されています。
	［対処方法］	ラベルを確認してください。
E0550244	［メッセージ］	.org 疑似命令において値（*value*）の指定に誤りがあります。
	［対処方法］	値を確認してください。

E0550245	［メッセージ］	予約語を用いることのできない場所において予約語 *identifier* が用いられています。
	［対処方法］	記述を確認してください。
E0550246	［メッセージ］	セクション中に記述することのできない命令が記述されています。
	［対処方法］	記述を確認してください。
E0550247	［メッセージ］	サイズの指定に誤りがあります。
	［対処方法］	指定を確認してください。
E0550248	［メッセージ］	シンボル *symbol* に対し '$'，または '#' が指定されています。
	［対処方法］	シンボルを確認してください。
E0550249	［メッセージ］	構成に誤りがあります。
	［対処方法］	記述を確認してください。
E0550250	［メッセージ］	*string* の構成に誤りがあります。
	［対処方法］	記述を確認してください。
E0550260	［メッセージ］	トークンの長さが限界を越えています。
	［説明］	トークンの長さが限界を越えています。限界値は 4,294,967,294 です。
	［対処方法］	トークンの長さを確認してください。
E0550261	［メッセージ］	指定された条件コードが不正です。
	［説明］	指定された条件コードが不正です。adf.sbf 命令の条件コードに 0xd は指定できません。
	［対処方法］	条件コードを確認してください。
E0550265	［メッセージ］	指定できないレジスタ (r0-r7, r16-r31) が指定されています。
	［対処方法］	汎用レジスタとして指定できるのは，r8-r15 のいずれかです。オペランドを確認してください。
E0550267	［メッセージ］	4 の倍数以外のディスプレースメントが指定されています。
	［対処方法］	ディスプレースメントを確認してください。
E0550269	［メッセージ］	RH850 " *コア名* " コア指定時には記述することのできない命令が記述されています。
	［説明］	指定されたコアでサポートされてない命令が記述されています。
	［対処方法］	命令のニモニックを確認してください。
E0550270	［メッセージ］	第 1，第 2 オペランドに同じレジスタを指定できません。
E0550271	［メッセージ］	" *文字列 1* " は先に指定された " *文字列 2* " と矛盾します。
	［説明］	" *文字列 1* " が先に指定された " *文字列 2* " と矛盾しています。ソースの記述を確認してください。 補足）"align=0" は，.section 疑似命令に align 指定がない場合を意味します。
E0550601	［メッセージ］	" *文字列* " オプションで指定された " *パス名* " はフォルダです。入力ファイルを指定してください。
E0550602	［メッセージ］	" *文字列* " オプションで指定されたファイル " *ファイル名* " が見つかりません。
	［対処方法］	ファイルが存在するか確認してください。
E0550603	［メッセージ］	" *文字列* " オプションで指定された " *パス名* " はフォルダです。出力ファイルを指定してください。

E0550604	[メッセージ]	" *文字列* " オプションで指定された出力先フォルダ " *フォルダ名* " が見つかりません。
E0550605	[メッセージ]	" *文字列 1*" オプションで指定された " *文字列 2*" はファイルです。フォルダを指定してください。
E0550606	[メッセージ]	" *文字列 1*" オプションで指定されたフォルダ " *文字列 2*" が見つかりません。
E0550607	[メッセージ]	" *文字列* " オプションで指定された " *パス名* " が見つかりません。
	[説明]	" *文字列* " オプションで指定された " *パス名* "（ファイル名，またはフォルダ名）が見つかりません。
E0550608	[メッセージ]	" *文字列* " は認識されないオプションです。
E0550609	[メッセージ]	" *文字列* " オプションに引数は指定できません。
E0550610	[メッセージ]	" *文字列* " オプションに引数を指定してください。
E0550611	[メッセージ]	" *文字列* " オプションに引数は指定できません。
E0550612	[メッセージ]	" *文字列* " オプションに引数を指定してください。
	[説明]	" *文字列* " オプションは引数が必要です。
	[対処方法]	引数を指定してください。
E0550613	[メッセージ]	" *文字列* " オプションに指定された引数が不正です。
E0550617	[メッセージ]	" *文字列* " オプションに指定された引数が不正です。
E0550629	[メッセージ]	コマンド・ファイル " *ファイル名* " が複数回読まれています。
E0550630	[メッセージ]	コマンド・ファイル " *ファイル名* " が読み込めません。
E0550631	[メッセージ]	コマンド・ファイル " *ファイル名* " の構文が認識できません。
E0550632	[メッセージ]	テンポラリ・フォルダを作成できません。
E0550633	[メッセージ]	ソース・ファイルが複数の場合は " *文字列* " オプションにはフォルダを指定してください。
E0550637	[メッセージ]	テンポラリ・フォルダ " *フォルダ名* " の削除に失敗しました。
E0550638	[メッセージ]	入力ファイル " *ファイル名* " のオープンに失敗しました。
E0550639	[メッセージ]	出力ファイル " *ファイル名* " のオープンに失敗しました。
E0550640	[メッセージ]	入力ファイル " *ファイル名* " のクローズに失敗しました。
E0550641	[メッセージ]	出力ファイル " *ファイル名* " の書き込みに失敗しました。
E0550645	[メッセージ]	" *文字列 1*" オプションで指定された " *文字列 2*" は使用できません。
E0550647	[メッセージ]	" *文字列* " オプションが複数指定されています。後の指定が有効になります。
E0550649	[メッセージ]	" *文字列 1*" オプションと " *文字列 2*" オプションが矛盾しています。" *文字列 2*" オプションを無視します。
E0550652	[メッセージ]	" *オプション 1*" オプションには " *オプション 2*" オプションが必要です。
	[説明]	*オプション 1* を使用するためには，*オプション 2* を同時に指定する必要があります。
E0550701	[メッセージ]	テンポラリ・ファイル " *ファイル名* " の削除に失敗しました。
E0551200	[メッセージ]	アセンブリ・ソースの記述に誤りがあります。
	[対処方法]	アセンブリ・ソースを確認してください。

E0551202	［メッセージ］	レジスタの記述に誤りがあります。
	［説明］	オペランドに指定できないレジスタの記述が含まれています。
	［対処方法］	オペランドに指定可能なレジスタを確認してください。
E0551203	［メッセージ］	リロケータブル項は記述できません。
	［説明］	リロケータブル項が許されていない箇所に記述されています。
	［対処方法］	該当箇所の記述形式を確認してください。
E0551204	［メッセージ］	オペランドの記述に誤りがあります。
	［説明］	オペランドに指定できない記述が含まれています。
	［対処方法］	オペランドに指定可能な形式を確認してください。
E0551205	［メッセージ］	文字列の記述に誤りがあります。
	［対処方法］	文字列の記述に誤りがないか確認してください。
E0551206	［メッセージ］	"$" は記述できません。
	［説明］	"$" が記述できない箇所に記述されています。
	［対処方法］	許されていない箇所に "$" を記述していないか確認してください。
E0551207	［メッセージ］	"*string*" は記述できません。
	［対処方法］	該当箇所の記述を確認ください。
E0551208	［メッセージ］	演算に誤りがあります ("op")。
	［説明］	"op" 演算の記述に誤りがあります。
	［対処方法］	"op" 演算の記述を確認してください。
E0551213	［メッセージ］	かっこの不整合，または演算子の対象となる式がありません。
	［説明］	右かっこがないか，または演算子の対象となる式がありません。
	［対処方法］	かっこの対応が正しくとれているか，また演算の対象となる式があるかを確認してください。
E0551214	［メッセージ］	"op" 演算子の記述に誤りがあります。
	［対処方法］	op 演算子の記述形式を確認してください。
E0551215	［メッセージ］	ラベルの記述に誤りがあります。
	［対処方法］	ラベルの記述を確認してください。
E0551218	［メッセージ］	(- ラベル) 形式の式が用いられています。
	［対処方法］	式を確認してください。
E0551219	［メッセージ］	ラベルの演算，または参照が不正です。
	［対処方法］	ラベルの演算，または参照を確認してください。
E0551220	［メッセージ］	未定義シンボルは記述できません。
	［説明］	未定義シンボルが記述できない箇所に記述されています。
	［対処方法］	シンボルの定義を確認してください。
E0551221	［メッセージ］	セクション名は記述できません。
	［説明］	セクション名が記述できない箇所に記述されています。
	［対処方法］	指定可能な記述を確認してください。

E0551222	［メッセージ］	文字の読み込みに失敗しました。
	［対処方法］	記述を確認してください。
E0551223	［メッセージ］	シングルクォーテーションの記述を確認してください。
	［説明］	シングルクォーテーション (') が閉じられていません。
	［対処方法］	シングルクォーテーションが閉じられているか確認してください。
E0551224	［メッセージ］	文字列の読み取りに失敗しました。
	［対処方法］	記述を確認してください。
E0551225	［メッセージ］	ダブルクォーテーションの記述を確認してください。
	［説明］	ダブルクォーテーション (") が閉じられていません。
	［対処方法］	ダブルクォーテーションが閉じられているか確認してください。
E0551226	［メッセージ］	式の途中に文字列の記述がありました。
E0551227	［メッセージ］	'?' は英数字として扱いません。
	［説明］	'?' は英数字として扱いません。シンボル名として使用することはできません。
E0551229	［メッセージ］	2 進数の表現に誤りがあります。
	［対処方法］	2 進数の表記が正しいか確認してください。
E0551230	［説明］	8 進数の表現に誤りがあります。
	［メッセージ］	8 進数の表記が正しいか確認してください。
E0551231	［対処方法］	10 進数の表現に誤りがあります。
	［メッセージ］	10 進数の表記が正しいか確認してください。
E0551232	［メッセージ］	16 進数の表現に誤りがあります。
	［対処方法］	16 進数の表記が正しいか確認してください。
E0551233	［メッセージ］	オペランド数が多すぎます。
	［対処方法］	正しい数のオペランドを指定してください。
E0551234	［メッセージ］	右ブラケットがありません。
E0551236	［メッセージ］	チルダの記述に誤りがあります。
E0551305	［メッセージ］	オペランドに指定した値が 8bit 幅を超えています。
E0551306	［メッセージ］	オペランドに指定した値が 16bit 幅を超えています。
E0551308	［メッセージ］	オペランドに指定した値が 32bit 幅を超えています。
E0551309	［メッセージ］	奇数値が指定されました。
E0551311	［メッセージ］	数値が 1-7 の範囲を超えています。
E0551313	［メッセージ］	"*reg*" は記述できません。
	［説明］	ここには *reg* レジスタは記述できません。
	［対処方法］	記述可能なオペランドを確認してください。
E0551401	［メッセージ］	" *文字列*" の記述に誤りがあります。
E0551402	［メッセージ］	命令の種類に誤りがあります。
E0551403	［メッセージ］	.DB8 疑似命令のオペランドには記述できません。

E0551406	［メッセージ］	" *文字列* " に "." で始まるシンボルを指定できません。
E0551407	［メッセージ］	"*name*" が自分自身を参照しています。
E0551501	［メッセージ］	"-output" オプションを指定したとき，複数ソース・ファイルを入力できません。
E0562000	［メッセージ］	Invalid option : " *オプション* "
	［説明］	" *オプション* " はサポートしていません。
E0562001	［メッセージ］	Option " *オプション* " cannot be specified on command line
	［説明］	" *オプション* " はコマンド・ライン上では指定できません。
	［対処方法］	サブコマンド・ファイル内で指定してください。
E0562002	［メッセージ］	Input option cannot be specified on command line
	［説明］	コマンド・ライン上で input オプションを指定しました。
	［対処方法］	コマンド・ライン上での入力ファイル指定は input オプションなしで指定してください。
E0562003	［メッセージ］	Subcommand option cannot be specified in subcommand file
	［説明］	サブコマンド・ファイル内に -subcommand オプションを指定しました。 -subcommand オプションはネストできません。
E0562004	［メッセージ］	Option " *オプション1* " cannot be combined with option " *オプション2* "
	［説明］	" *オプション1* " と " *オプション2* " は同時に指定できません。
E0562005	［メッセージ］	Option " *オプション* " cannot be specified while processing " *プロセス* "
	［説明］	" *プロセス* " 処理に対して " *オプション* " は指定できません。
E0562006	［メッセージ］	Option " *オプション1* " is ineffective without option " *オプション2* "
	［説明］	" *オプション1* " は " *オプション2* " が必要です。
E0562010	［メッセージ］	Option " *オプション* " requires parameter
	［説明］	" *オプション* " はパラメータ指定が必要です。
E0562011	［メッセージ］	Invalid parameter specified in option " *オプション* " : " *パラメータ* "
	［説明］	" *オプション* " で無効なパラメータを指定しました。
E0562012	［メッセージ］	Invalid number specified in option " *オプション* " : " *値* "
	［説明］	" *オプション* " 指定で無効な値を指定しました。
	［対処方法］	値の範囲を確認してください。
E0562013	［メッセージ］	Invalid address value specified in option " *オプション* " : " *アドレス* "
	［説明］	" *オプション* " で指定した " *アドレス* " は無効な値です。
	［対処方法］	0 ～ FFFFFFFF の間の 16 進数で指定してください。
E0562014	［メッセージ］	Illegal symbol/section name specified in " *オプション* " : " *名前* "
	［説明］	" *オプション* " で指定したセクションまたはシンボル名に不正文字が使用されています。
E0562016	［メッセージ］	Invalid alignment value specified in option " *オプション* " : " *整列条件数* "
	［説明］	" *オプション* " で指定した " *整列条件数* " は無効な値です。
	［対処方法］	1，2，4，8，16，または 32 を指定してください。

E0562020	［メッセージ］	Duplicate file specified in option " *オプション* " : " *ファイル* "
	［説明］	" *オプション* " 指定で同じファイルを二度指定しました。
E0562022	［メッセージ］	Address ranges overlap in option " *オプション* " : " *アドレス範囲* "
	［説明］	" *オプション* " で指定した " *アドレス範囲* " が重複しています。
E0562100	［メッセージ］	Invalid address specified in cpu option : " *アドレス* "
	［説明］	-cpu オプションで cpu では指定できないアドレスを指定しました。
E0562101	［メッセージ］	Invalid address specified in option " *オプション* " : " *アドレス* "
	［説明］	" *オプション* " で指定した " *アドレス* " は cpu で指定できるアドレス範囲，または -cpu オプションで指定した範囲を越えました。
E0562110	［メッセージ］	Section size of second parameter in rom option is not 0 : " *セクション* "
	［説明］	-rom オプションの第 2 パラメータにサイズが 0 でない " *セクション* " を指定しました。
E0562111	［メッセージ］	Absolute section cannot be specified in " *オプション* " option : " *セクション* "
	［説明］	" *オプション* " で絶対アドレス・セクションを指定しました。
E0562114	［メッセージ］	The generated duplicate section name "*section*" is confused
	［説明］	同名セクション *section* が複数回現れ，処理できませんでした。
E0562120	［メッセージ］	Library " *ファイル* " without module name specified as input file
	［説明］	入力ファイルとしてモジュール名なしのライブラリ・ファイルを指定しました。
E0562121	［メッセージ］	Input file is not library file : " *ファイル* (*モジュール*)"
	［説明］	入力ファイルで指定した " *ファイル* (*モジュール*)" はライブラリ・ファイルではありません。
E0562130	［メッセージ］	Cannot find file specified in option " *オプション* " : " *ファイル* "
	［説明］	" *オプション* " で指定したファイルが見つかりません。
E0562131	［メッセージ］	Cannot find module specified in option " *オプション* " : " *モジュール* "
	［説明］	" *オプション* " で指定したモジュールがありません。
E0562132	［メッセージ］	Cannot find " *名前* " specified in option " *オプション* "
	［説明］	" *オプション* " で指定したシンボルまたはセクションが存在しません。
E0562133	［メッセージ］	Cannot find defined symbol " *名前* " in option " *オプション* "
	［説明］	" *オプション* " で指定した外部定義シンボルが存在しません。
E0562140	［メッセージ］	Symbol/section " *名前* " redefined in option " *オプション* "
	［説明］	" *オプション* " で指定したシンボル，セクションはすでに定義されています。
E0562141	［メッセージ］	Module " *モジュール* " redefined in option " *オプション* "
	［説明］	" *オプション* " で指定したモジュールはすでに登録されています。
E0562200	［メッセージ］	Illegal object file : " *ファイル* "
	［説明］	ELF フォーマット以外を入力しました。
E0562201	［メッセージ］	Illegal library file : " *ファイル* "
	［説明］	" *ファイル* " はライブラリ・ファイルではありません。

E0562210	［メッセージ］	Invalid input file type specified for option " *オプション* " : " *ファイル* (*種別*)"
	［説明］	" *オプション* " 指定時に処理できない " *ファイル* (*種別*)" を入力しました。
E0562211	［メッセージ］	Invalid input file type specified while processing " *プロセス* " : " *ファイル* (*種別*)"
	［説明］	" *プロセス* " 処理に対して処理できない " *ファイル* (*種別*)" を入力しました。
E0562212	［メッセージ］	" *オプション* " cannot be specified for inter-module optimization information in " *ファイル* "
	［説明］	" *ファイル* " 内にリンク時最適化（モジュール間最適化）情報があるため，" *オプション* " オプションは使用できません。
	［対処方法］	コンパイル，アセンブル時に -goptimize オプションを使用しないでください。
E0562221	［メッセージ］	Section type mismatch : " *セクション* "
	［説明］	属性 (初期値有無) の異なる同名セクションを入力しました。
E0562224	［メッセージ］	Section type (relocation attribute) mismatch : " *セクション* "
	［説明］	再配置属性が異なる同名セクションを入力しました。
E0562300	［メッセージ］	Duplicate symbol " *シンボル* " in " *ファイル* "
	［説明］	" *シンボル* " は重複しています。
E0562301	［メッセージ］	Duplicate module " *モジュール* " in " *ファイル* "
	［説明］	" *モジュール* " は重複しています。
E0562310	［メッセージ］	Undefined external symbol " *シンボル* " referenced in " *ファイル* "
	［説明］	" *ファイル* " 内で未定義の " *シンボル* " を参照しています。
E0562311	［メッセージ］	Section " *セクション1* " cannot refer to overlaid section : " *セクション2- シンボル* "
	［説明］	同一アドレスを指定したオーバレイセクション間でシンボル参照がありました。
	［対処方法］	" *セクション1* " と " *セクション2* " を同じアドレスに割り付けないでください。
E0562320	［メッセージ］	Section address overflowed out of range : " *セクション* "
	［説明］	" *セクション* " のアドレスが使用可能なアドレス範囲を越えました。セクションの配置アドレスとサイズを確認してください。
E0562321	［メッセージ］	Section " *セクション1* " overlaps section " *セクション2* "
	［説明］	" *セクション1* " と " *セクション2* " のアドレスが重複しました。
	［対処方法］	start オプションのアドレス指定を変更してください。
E0562324	［メッセージ］	Section " *セクション* " in " *ファイル* " conflicts
	［説明］	" *セクション* " が存在するオブジェクト・ファイルを複数入力しました。
E0562326	［メッセージ］	Insufficient space to allocate prefetch section after section " *セクション* "
	［説明］	" *セクション* " の後ろにプリフェッチセクションを割り当てる領域が不足しています。
E0562330	［メッセージ］	Relocation size overflow : " *ファイル* "-" *セクション* "-" *オフセット* "
	［説明］	リロケーション演算結果がリロケーションサイズを越えました。分岐先が届かない，特定のアドレスに配置しなければならないシンボルを参照しているなどが考えられます。
	［対処方法］	アセンブル・リストで，" *セクション* " の " *オフセット* " 位置の参照シンボルが正しい位置に配置されているか確認してください。

E0562332	［メッセージ］	Relocation value is odd number : " *ファイル*"-" *セクション*"-" *オフセット*"
	［説明］	リロケーション演算結果が奇数になりました。
	［対処方法］	コンパイル，アセンブル・リストで，" *セクション*" の " *オフセット*" 位置の演算に問題がないか確認してください。
E0562340	［メッセージ］	Symbol name " *ファイル*"-" *セクション*"-" *シンボル*" is too long
	［説明］	" *セクション*" 内の " *シンボル*" の文字数がアセンブラの翻訳限界を越えました。
	［対処方法］	シンボル・アドレス・ファイルを出力する場合は，アセンブラの翻訳限界文字数以下になるようなシンボル名としてください。
E0562408	［メッセージ］	Register mode in " *ファイル*" conflicts with that in another file(" *モード*")
	［説明］	複数のファイルで異なるレジスタ・モードを指定されています。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562410	［メッセージ］	Address value specified by map file differs from one after linkage as to " *シンボル*"
	［説明］	" *シンボル*" のアドレス値がコンパイル時に使用した外部シンボル割り付け情報ファイル内のアドレスとリンク後のアドレスで異なっています。
	［対処方法］	以下を確認してください。 (1) コンパイル時の map オプション指定前後でプログラムを変更している場合は，プログラムの変更をやめてください。 (2) rlink の最適化によって，コンパイル時の map オプション指定前後のシンボル並び順が変わることがあります。コンパイル時 map オプションを無効にするか，rlink の最適化オプションを無効にしてください。
E0562411	［メッセージ］	Map file in " *ファイル*" conflicts with that in another file
	［説明］	入力ファイル間でコンパイル時に異なる外部シンボル割り付け情報ファイルを使用しています。
E0562412	［メッセージ］	Cannot open file : " *ファイル*"
	［説明］	" *ファイル*"(外部シンボル割り付け情報ファイル) がオープンできません。
	［対処方法］	ファイル名およびアクセス権が正しいか確認してください。
E0562413	［メッセージ］	Cannot close file : " *ファイル*"
	［説明］	" *ファイル*"(外部シンボル割り付け情報ファイル) がクローズできません。ディスク容量に空きがない可能性があります。
E0562414	［メッセージ］	Cannot read file : " *ファイル*"
	［説明］	" *ファイル*"(外部シンボル割り付け情報ファイル) が読みこめません。ディスク容量に空きがない可能性があります。
E0562415	［メッセージ］	Illegal map file : " *ファイル*"
	［説明］	" *ファイル*"(外部シンボル割り付け情報ファイル) のフォーマットが不正です。
	［対処方法］	ファイル名が正しいか確認してください。
E0562416	［メッセージ］	Order of functions specified by map file differs from one after linkage as to " *関数名*"
	［説明］	関数 " *関数名*" は，コンパイル時に使用した外部シンボル割り付け情報ファイル内の情報とリンク後の配置とで，ほかの関数との並び順が異なっています。関数内 static 変数のアドレスが，外部シンボル割り付け情報ファイルとリンク後の結果とで異なっている可能性があります。
E0562417	［メッセージ］	Map file is not the newest version : " *ファイル名*"
	［説明］	外部シンボル割り付け情報ファイルが最新バージョンではありません。

E0562420	［メッセージ］	" *ファイル 1*" overlap address " *ファイル 2*" : " *アドレス* "
	［説明］	" *ファイル 1*" と " *ファイル 2*" のアドレスが重複しています。
E0562430	［メッセージ］	Register ("*register1*") in "*file-name*" conflicts with that in another file("*register2*")
	［説明］	"*file-name*" 内で指定したレジスタ・モードの使用方法が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562431	［メッセージ］	GP register ("mode = *mode1*") in "*file-name*" conflicts with that in another file("mode = *mode2*")
	［説明］	"*file-name*" 内で指定した GP レジスタの使用方法が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562432	［メッセージ］	EP register ("mode = *mode1*") in "*file-name*" conflicts with that in another file("mode = *mode2*")
	［説明］	"*file-name*" 内で指定した EP レジスタの使用方法が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562433	［メッセージ］	TP register ("mode = *mode1*") in "*file-name*" conflicts with that in another file("mode = *mode2*")
	［説明］	"*file-name*" 内で指定した TP レジスタの使用方法が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562434	［メッセージ］	R2 register ("mode = *mode1*") in "*file-name*" conflicts with that in another file("mode = *mode2*")
	［説明］	"*file-name*" 内で指定した R2 レジスタの使用方法が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562435	［メッセージ］	Alignment of 8byte data (value="*alignment1*") in "*file-name*" conflicts with that in another file(value="*alignment2*")
	［説明］	"*file-name*" 内で指定した C 言語の 8 バイトの基本型に対する整列条件が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562436	［メッセージ］	Size of double/long double (value="*size1*") in "*file-name*" conflicts with that in another file(value="*size2*")
	［説明］	"*file-name*" 内で指定した C 言語の 8 バイトの基本型に対するサイズが，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562437	［メッセージ］	FPU Type (value="*FPU1*") in "*file-name*" conflicts with that in another file(value="*FPU2*")
	［説明］	"*file-name*" 内で指定した FPU 種別が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。

E0562438	［メッセージ］	SIMD Type (value="*SIMD1*") in "*file-name*" conflicts with that in another file(value="*SIMD2*")
	［説明］	"*file-name*" 内で指定した SIMD 種別が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562439	［メッセージ］	Number of additional global pointers (value="*number1*") in "*file-name*" conflicts with that in another file(value="*number2*")
	［説明］	"*file-name*" 内で指定した GP レジスタ数が，他ファイルと統一されていません。
	［対処方法］	コンパイル時のオプションを確認してください。
E0562450	［メッセージ］	Illegal ID(value="*number*") in section "*section-name*" in "*file-name*"
	［説明］	"*file-name*" 内の "*section-name*" で指定した "*number*" はサポートしていません。
	［対処方法］	コンパイラ，アセンブラのバージョンが正しいか確認してください。
E0562451	［メッセージ］	Illegal desc(*number*) in section "*section-name*" in "*file-name*"
	［説明］	"*file-name*" 内の "*section-name*" で指定した "*number*" はサポートしていません。
	［対処方法］	コンパイラ，アセンブラのバージョンが正しいか確認してください。
E0562600	［メッセージ］	Library " *ライブラリ* " requires " *ライセンス・エディション* "
	［説明］	指定した " *ライブラリ* " の利用には，" *ライセンス・エディション* " が必要です。
E0595001	［メッセージ］	checker : Missing input file
	［説明］	ファイル名が指定されていません。
E0595002	［メッセージ］	checker : Failed to open input file " *ファイル名* "
	［説明］	ファイルをオープンできません。
E0595003	［メッセージ］	checker : Incorrect usage
	［説明］	コマンド・ラインの指定方法に誤りがあります。
E0595004	［メッセージ］	checker : Incorrect file format
	［説明］	モトローラ・S タイプ以外のファイルが指定されています。
E0595005	［メッセージ］	checker : Failed to decode input file " *ファイル名* "
	［説明］	解析中にデコードに失敗しました。ファイルの形式が正しくないか，例外ベクタのベース・アドレス指定が正しくない可能性があります。
	［対処方法］	「11.4 例外ハンドラ先頭への SYNCP 命令配置確認ツール」を参照してください。
E0595010	［メッセージ］	NG : *name* : *address* : *cause*
	［説明］	例外要因 *name* の例外ハンドラから *address* までの間に SYNCP 命令が配置されていない可能性があります。
	［対処方法］	「11.4 例外ハンドラ先頭への SYNCP 命令配置確認ツール」を参照してください。

10.4.3 致命的エラー

表 10.4 致命的エラー

F0520003	［メッセージ］	#include ファイル " *ファイル名* " は自分自身をインクルードしています。
	［説明］	#include ファイル " *ファイル名* " は自分自身をインクルードしています。修正してください。
F0520004	［メッセージ］	メモリが足りません。
	［対処方法］	メモリが不足しています。ほかのアプリケーションを終了して，再度コンパイルし直してください。
F0520005	［メッセージ］	ソース・ファイル " *ファイル名* " を開くことができません。
F0520013	［メッセージ］	ファイル名がありません。
F0520035	［メッセージ］	#error 指令 : *文字列*
	［説明］	ソース・ファイル中に #error 指令がありました。
F0520143	［メッセージ］	プログラムはコンパイルするのに大きすぎるか複雑すぎます。
F0520163	［メッセージ］	テンポラリファイル *ファイル名* がオープンできません。
F0520164	［メッセージ］	テンポラリファイル用のフォルダ名が長すぎます *ファイル名*。
F0520182	［メッセージ］	ソース・ファイル *ファイル名* を開くことができません。サーチ・リストにフォルダがありません。
F0520189	［メッセージ］	ファイル " *ファイル名* " の書き込み中にエラーが生じました。
F0520563	［メッセージ］	不正なプリプロセッサ出力ファイルです。
F0520564	［メッセージ］	プリプロセッサ出力ファイルをオープンできません。
F0520571	［メッセージ］	不正なオプションです。: *オプション名*
F0520642	［メッセージ］	テンポラリファイル名を生成できません。
F0520920	［メッセージ］	出力ファイルがオープンできません : *ファイル名*。
F0523029	［メッセージ］	misra ルールファイルをオープンできません。
	［説明］	-Xmisra2004=" *ファイル名* " もしくは -Xmisra2012=" *ファイル名* " オプションで指定したファイルをオープンできません。
F0523030	［メッセージ］	ルールファイル内の記述が不正です。
	［説明］	-Xmisra2004=" *ファイル名* " もしくは -Xmisra2012=" *ファイル名* " オプションで指定したファイルの内容に，不正な記述があります。
F0523031	［メッセージ］	" *ルール番号* " はサポートしていません。
	［説明］	サポートしていないルール番号を指定しました。
F0523054	［メッセージ］	regID が範囲外の値です。
	［対処方法］	regID として使用可能な値を指定してください。
F0523055	［メッセージ］	selID が範囲外の値です。
	［対処方法］	selID として使用可能な値を指定してください。
F0523056	［メッセージ］	第一引数が範囲外の値です。
	［説明］	__set_il_rh(NUM, ADDR) の NUM として，使用不可能な値を指定しました。
F0523061	［メッセージ］	実引数は組み込み関数の仮引数と適合しません。

F0523062	［メッセージ］	返却値の型が組み込み関数の型と合っていません。
F0523073	［メッセージ］	*コア名*で使用できない組み込み関数です。
	［説明］	指定しているコアで使用できない組み込み関数を使用しています。
F0523089	［メッセージ］	ファイル " *ファイル名* " を読み込めません。
	［説明］	ファイルがみつからない，または読み込みに失敗しました。
F0530320	［メッセージ］	シンボル " *シンボル名* " が重複しています。
F0530321	［メッセージ］	セクション " *セクション名* " が存在するファイルを複数入力しました。
F0530800	［メッセージ］	" *シンボル名* " で示すシンボルの型がファイル間で異なっています。
F0530808	［メッセージ］	" *変数名* " で示す変数のアライメントがファイル間で異なっています。
F0530810	［メッセージ］	" *シンボル名* " で示すシンボルの #pragma 指定がファイル間で異なっています。
F0533015	［メッセージ］	シンボル数が限界値を越えました。
	［説明］	コンパイラが生成するシンボルの数が限界値を越えました。
F0533021	［メッセージ］	メモリが足りません。
	［対処方法］	ほかのアプリケーションを終了して，再度コンパイルし直してください。
F0533301	［メッセージ］	中間ファイルをクローズできません。
	［説明］	コンパイラが内部で生成する中間ファイルをクローズすることができません。
F0533302	［メッセージ］	中間ファイルの読み込み中にエラーが生じました。
F0533303	［メッセージ］	中間ファイルの書き込み中にエラーが生じました。
F0533306	［メッセージ］	コンパイル処理を中断しました。
	［説明］	コンパイル処理中に Cntl + C コマンドによる割り込みを検出しました。
F0533330	［メッセージ］	中間ファイルをオープンできません。
	［説明］	コンパイラが内部で生成する中間ファイルをオープンすることができません。
F0540027	［メッセージ］	ファイル ” *ファイル名* ” が読み込みできません。
F0540204	［メッセージ］	関数内で使用するスタックのサイズが大きすぎます。
	［説明］	関数内で使用するスタックのサイズが 2G バイトを越えています。
F0540300	［メッセージ］	中間ファイルをオープンできません。
	［説明］	コンパイラが内部で生成する中間ファイルをオープンすることができません。
F0540301	［メッセージ］	中間ファイルをクローズできません。
	［説明］	コンパイラが内部で生成する中間ファイルをクローズすることができません。
F0540302	［メッセージ］	中間ファイルの読み込み中にエラーが生じました。
F0540303	［メッセージ］	中間ファイルの書き込み中にエラーが生じました。
F0540400	［メッセージ］	#pragma " *識別子名* " に異なるパラメータが設定されています。
F0550503	［メッセージ］	ファイル *file* をオープンできません。
	［対処方法］	ファイルを確認してください。
F0550504	［メッセージ］	セクション定義疑似命令においてセクションの種類の指定に誤りがあります。
	［対処方法］	セクションの種類の指定を確認してください。

F0550505	［メッセージ］	メモリが足りません。
	［対処方法］	空きメモリを確認してください。
F0550506	［メッセージ］	内部データ領域（*string*）の確保に失敗しました。
	［対処方法］	空きメモリを確認してください。
F0550507	［メッセージ］	式の処理において作業領域が足りなくなりました。(*string*)
	［説明］	式の処理において作業領域が足りなくなりました。単純な式に変更してください。
	［対処方法］	式を確認してください。
F0550508	［メッセージ］	定義されていない識別子 *identifier* が参照されています。
	［対処方法］	識別子を確認してください。
F0550509	［メッセージ］	予期しない疑似命令 *string* が見つかりました。
	［対処方法］	疑似命令を確認してください。
F0550510	［メッセージ］	*string* 疑似命令に対応する疑似命令が存在しません。
	［対処方法］	疑似命令を確認してください。
F0550511	［メッセージ］	条件アセンブル疑似命令において対応する疑似命令 *string* が存在しません。
	［対処方法］	条件アセンブル疑似命令を確認してください。
F0550512	［メッセージ］	条件アセンブル疑似命令が 4294967294 回以上ネストして用いられています。
	［対処方法］	ネストを確認してください。
F0550513	［メッセージ］	*string* 疑似命令に対応する .endm 疑似命令が存在しません。
	［対処方法］	疑似命令を確認してください。
F0550514	［メッセージ］	実引数が 4294967294 個以上用いられています。
	［対処方法］	実引数を確認してください。
F0550516	［メッセージ］	.local 疑似命令により自動生成されたシンボルが限界数（4294967294）を越えました。
	［対処方法］	疑似命令を確認してください。
F0550531	［メッセージ］	1 ファイルに記述できるシンボル数を越えました。記述できるシンボル数の限界は，アセンブラが内部で登録するものを含め，4294967294 です。
F0550532	［メッセージ］	リンク可能なオブジェクト・ファイルを生成する段階で，ファイル・システムに依存するエラーが発生しました。
	［対処方法］	ファイル・システムを確認してください。
F0550534	［メッセージ］	1 ファイル中の命令の数が多すぎます。
	［説明］	1 ファイル中の命令の数が限界を越えています。限界値は 10,000,000 です。
	［対処方法］	命令の数を確認してください。
F0550537	［メッセージ］	セクション (*section*) のアドレスが使用可能なアドレス範囲を越えました。
	［説明］	絶対アドレス指定セクションの配置アドレスが 0xffffffff を越えています。
	［対処方法］	.org によるセクションの絶対アドレス指定は，セクションの終端命令の配置アドレスが 0xffffffff までになるように行ってください。

F0550538	［メッセージ］	セクション (*section1*) と (*section2*) のアドレスが重複しました。
	［説明］	絶対アドレス指定セクションの配置アドレス範囲が，ほかのセクションの配置アドレス範囲と重複しています。
	［対処方法］	.org で指定しているアドレスを確認してください。
F0550539	［メッセージ］	リロケーション・エントリに出力するシンボルが限界数 (16777215) を越えました。
	［説明］	16777216 個以上のシンボルが登録されており，かつ参照されています。
	［対処方法］	シンボル数を確認してください。
F0550540	［メッセージ］	ファイル *file* の読み込みができません。
	［説明］	ファイルが不正か，ファイルのサイズが読み込みできる限界を越えているかもしれません。
	［対処方法］	ファイルを確認してください。
F0551608	［メッセージ］	アドレスを指定してください。
F0551609	［メッセージ］	インクルード・ファイルの入れ子回数が上限を超えました。
	［説明］	インクルードの階層が深すぎるか，自分自身を再帰的にインクルードしている可能性があります。
	［対処方法］	インクルード・ファイルを見直してください。
F0551610	［メッセージ］	マクロ呼び出しの入れ子回数が上限を超えました。
	［説明］	マクロ呼び出しの階層が深すぎるか，自分自身を再帰的に呼び出している可能性があります。
	［対処方法］	マクロ定義を見直してください。
F0563000	［メッセージ］	No input file
	［説明］	入力ファイルがありません。
F0563001	［メッセージ］	No module in library
	［説明］	ライブラリ内のモジュール数が 0 になりました。
F0563002	［メッセージ］	Option " *オプション1*" is ineffective without option " *オプション2*"
	［説明］	" *オプション1*" は " *オプション2*" が必要です。
F0563003	［メッセージ］	Illegal file format " *ファイル* "
	［説明］	" *ファイル* " が利用できないファイル形式です。
F0563004	［メッセージ］	Invalid inter-module optimization information type in " *ファイル* "
	［説明］	" *ファイル* " 内にサポートしていないリンク時最適化（モジュール間最適化）情報タイプがありました。
	［対処方法］	コンパイラ，アセンブラのバージョンが正しいか確認してください。
F0563020	［メッセージ］	No cpu information in input files
	［説明］	CPU 種別を入力ファイルから識別できません。
	［対処方法］	バイナリ・ファイルは -binary オプションで指定し，共にリンクする .obj/.rel ファイルがあることを確認してください。

F0563100	［メッセージ］	Section address overflow out of range : " *セクション*"
	［説明］	" *セクション*" のアドレスが使用可能な上限の領域を越えました。
	［対処方法］	start オプションのアドレス指定を変更してください。 なお，アドレス空間の詳細についてはデバイスのユーザーズマニュアルを参照してください。
F0563102	［メッセージ］	Section contents overlap in absolute section " *セクション*" 【V1.05.00 以前】 Section contents overlap in absolute section " *セクション*" in " *ファイル*" 【V1.06.00 以降】
	［説明］	絶対アドレス・セクションのセクション内データ・アドレスが重複しています。
	［対処方法］	ソース・プログラムを修正してください。
F0563103	［メッセージ］	Section size overflow : "*section-name*"
	［説明］	セクション "*section-name*" が使用可能なサイズを超えました。
F0563110	［メッセージ］	Illegal cpu type " *マイコン種別*" in " *ファイル*"
	［説明］	異なるマイコン種別のファイルを入力しました。
F0563111	［メッセージ］	Illegal encode type " *エンディアン種別*" in " *ファイル*"
	［説明］	異なるエンディアン種別のファイルを入力しました。
F0563112	［メッセージ］	Invalid relocation type in " *ファイル*"
	［説明］	" *ファイル*" 内にサポートしていないリロケーション・タイプがありました。
	［対処方法］	コンパイラ，アセンブラのバージョンが正しいか確認してください。
F0563115	［メッセージ］	Cpu type in " *ファイル*" is not supported
	［説明］	" *ファイル*" の CPU 種別に対応していません。入力ファイルが正しいか確認してください。
F0563150	［メッセージ］	Multiple files cannot be specified while processing " *プロセス*"
	［説明］	" *プロセス*" 処理に対して，複数のファイルを指定できません。ファイルの指定を確認してください。
F0563200	［メッセージ］	Too many sections
	［説明］	セクション数が翻訳限界を越えました。複数ファイル出力を指定すると解決できる可能性があります。
F0563201	［メッセージ］	Too many symbols
	［説明］	シンボル数が翻訳限界を越えました。複数ファイル出力を指定すると解決できる可能性があります。
F0563202	［メッセージ］	Too many modules
	［説明］	モジュール数が翻訳限界を越えました。
	［対処方法］	ライブラリを分けて作成してください。
F0563203	［メッセージ］	Reserved module name "rlink_generates"
	［説明］	rlink_generates_** （** は，01 ～ 99 までの数値）は，最適化リンカで使用する予約名称です。.obj/.rel ファイル名，およびライブラリ内モジュール名として使用しています。
	［対処方法］	ファイル名，およびライブラリ内モジュール名で使用している場合は，変更してください。

F0563204	［メッセージ］	Reserved section name "$sss_fetch"
	［説明］	sss_fetch** （sss は任意の文字列，** は 01 ～ 99 までの数値）は，最適化リンカで使用する予約名称です。
	［対処方法］	シンボル名，またはセクション名を変更してください。
F0563300	［メッセージ］	Cannot open file : " *ファイル* "
	［説明］	" *ファイル* " をオープンできません。
	［対処方法］	ファイル名，およびアクセス権が正しいか，確認してください。
F0563301	［メッセージ］	Cannot close file : " *ファイル* "
	［説明］	" *ファイル* " をクローズできません。ディスク容量に空きがない可能性があります。
F0563302	［メッセージ］	Cannot write file : " *ファイル* "
	［説明］	" *ファイル* " に書き込めません。ディスク容量に空きがない可能性があります。
F0563303	［メッセージ］	Cannot read file : " *ファイル* "
	［説明］	" *ファイル* " を読めません。空ファイルを入力したか，ディスク容量に空きがない可能性があります。
F0563310	［メッセージ］	Cannot open temporary file
	［説明］	中間ファイルをオープンできません。
	［対処方法］	HLNK_TMP 指定が正しいか確認してください。またはディスク容量に空きがない可能性があります。
F0563314	［メッセージ］	Cannot delete temporary file
	［説明］	中間ファイルを削除できません。ディスク容量に空きがない可能性があります。
F0563320	［メッセージ］	Memory overflow
	［説明］	最適化リンカが内部で使用するメモリが不足しています。
	［対処方法］	メモリを増やしてください。
F0563410	［メッセージ］	Interrupt by user
	［説明］	標準入力端末から「(Ctrl)+C」キーによる割り込みを検出しました。
F0563430	［メッセージ］	The total section size exceeded the limit of the evaluation version of *バージョン*. Please consider purchasing the product.
	［説明］	無償評価版でリンク可能なサイズ制限を越えました。 リンク・サイズを 256K バイト以内に制限しています。製品版の購入をご検討ください。
F0563431	［メッセージ］	Incorrect device type, object file mismatch.
	［説明］	対応しない CPU 種別を入力しました。リンカ実行ファイルとオプションのファイルを確認してください。
F0563600	［メッセージ］	Option " *オプション* " requires parameter
	［説明］	" *オプション* " はパラメータ指定が必要です。
F0563601	［メッセージ］	Invalid parameter specified in option " *オプション* " : " *パラメータ* "
	［説明］	" *オプション* " で無効なパラメータを指定しました。
F0563602	［メッセージ］	"*character string*" option requires "*edition*".
	［説明］	" *文字列* " オプションの使用には，"*edition*" が必要です。

10.4.4 インフォメーション

表 10.5 インフォメーション

M0523028	［メッセージ］	Rule *ルール番号*: *内容*
	［説明］	MISRA-C:2004 の*ルール番号*と*内容*の該当箇所を検出しました。
M0523086	［メッセージ］	Rule *ルール番号*: *内容*
	［説明］	MISRA-C:2012 の*ルール番号*と*内容*の該当箇所を検出しました。
M0536001	［メッセージ］	制御レジスタを更新します。(*register-information*)
	［説明］	制御レジスタへの書き込みを検出しました。
M0560101	［メッセージ］	No stack information in " *ファイル* "
	［説明］	" *ファイル* " 内にスタック情報がありません。" *ファイル* " はアセンブラ出力ファイルの可能性があります。最適化リンカが出力するスタック情報ファイルに当該ファイルの内容は含まれません。
M0560004	［メッセージ］	" *ファイル* "-" *シンボル* " deleted by optimization
	［説明］	symbol_delete の最適化によって，" *ファイル* " 内の " *シンボル* " を削除しました。
M0560005	［メッセージ］	The offset value from the symbol location has been changed by optimization " *ファイル* "-" *セクション* "-" *シンボル* ±offset"
	［説明］	" *シンボル* ±offset" の範囲で最適化によるサイズ変更があったため offset 値を変更しました。問題ないか確認してください。offset 値の変更を抑止したい場合は，" *ファイル* " のアセンブル時に goptimize オプション指定を外してください。
M0560100	［メッセージ］	No inter-module optimization information in " *ファイル* "
	［説明］	" *ファイル* " 内にリンク時最適化（モジュール間最適化）情報がありません。" *ファイル* " をリンク時最適化の対象外にします。リンク時最適化の対象にする場合は，コンパイル，アセンブル時に goptimize オプションを指定してください。
M0560400	［メッセージ］	Unused symbol " *ファイル* "-" *シンボル* "
	［説明］	" *ファイル* " 内の " *シンボル* " は使用されていません。
M0560500	［メッセージ］	Generated CRC code at " *アドレス* "
	［説明］	" *アドレス* " に CRC コードを出力しました。
M0560512	［メッセージ］	Section " *セクション* " created by " *オプション* "
	［説明］	" *オプション* " によって，" *セクション* " を作成しました。
M0560700	［メッセージ］	Section address overflow out of range : " *セクション* "
	［説明］	" *セクション* " のアドレスが使用可能なアドレス範囲を超えました。

10.4.5 ワーニング

表 10.6 ワーニング

W0511105	[メッセージ]	" *文字列* " オプションで指定された " *パス名* " はファイルです。フォルダを指定してください。
W0511106	[メッセージ]	" *文字列* " オプションで指定されたフォルダ " *フォルダ名* " が見つかりません。
W0511123	[メッセージ]	" *文字列 1* " オプションが指定されたので " *文字列 2* " オプションは無視しました。
W0511143	[メッセージ]	FPU を持っていないデバイスが指定されたので，"-Xfloat" オプションを無視しました。
W0511146	[メッセージ]	" *文字列* " オプションで指定された " *シンボル名* " はマクロでは使用できません。
W0511147	[メッセージ]	" *文字列* " オプションが複数指定されています。後の指定が有効になります。
W0511149	[メッセージ]	" *文字列 1* " オプションと " *文字列 2* " オプションが矛盾しています。" *文字列 2* " オプションを無視します。
W0511151	[メッセージ]	" *文字列 1* " オプションが指定されていないので，" *文字列 2* " オプションを無視します。
W0511153	[メッセージ]	"-O *文字列* " が指定されたので最適化詳細オプションはクリアされました。最適化詳細オプションは "-O *文字列* " の後に指定してください。
W0511164	[メッセージ]	同じファイル名 " *ファイル名* " が複数指定されています。
	[説明]	コマンドラインに同じファイル名が複数指定されています。 CC-RH は，同じファイル名を複数扱うことができません。 最後に指定されたもののみが有効になります。
W0511179	[メッセージ]	この評価版は残り *数字* 日間有効です。
W0511180	[メッセージ]	*バージョン* の評価期間の有効期限が切れています。
W0511181	[メッセージ]	内部情報ファイルにエラーがあります。(*information*)
W0511183	[メッセージ]	ライセンスマネージャがインストールされていません。
	[説明]	ライセンス・マネージャがインストールされていません。対応するライセンス・マネージャをインストールしてください。
W0511184	[メッセージ]	"*option*" オプションが指定されたため，"-g" オプションが有効となります。
	[対処方法]	"-g" オプションを明示的に指定することにより，本メッセージを抑止することができます。
W0511185	[メッセージ]	professional 版の機能の試用期間は残り *数字* 日です。professional 版の購入を検討ください。
W0520009	[メッセージ]	コメントのネスティングは許されていません。
	[対処方法]	ネストしないようにしてください。
W0520011	[メッセージ]	不明な前処理指令があります。
W0520012	[メッセージ]	前に構文エラーがあるため，ここより文法の解析を再開します。
W0520021	[メッセージ]	型修飾子はこの宣言では無効です。
	[説明]	型修飾子はこの宣言では無効です。無視しました。
W0520026	[メッセージ]	文字定数中の文字が多すぎます。
	[説明]	文字定数中の文字が多すぎます。文字定数は複数の文字を含むことはできません。
W0520027	[メッセージ]	char 型の値が範囲を超えています。

W0520038	[メッセージ]	この前処理指令は許可されていません -- #else はすでにあります。
	[説明]	#else がすでにあるため，このディレクティブは不正です。
W0520039	[メッセージ]	0 で除算を行いました。
W0520042	[メッセージ]	オペランドの型が適合しません (" *型 1*" と " *型 2*")。
W0520055	[メッセージ]	マクロに対する引数が多すぎます。
W0520061	[メッセージ]	整数演算の結果が範囲を超えました。
W0520062	[メッセージ]	シフト数が負数です。
	[説明]	シフト数が負数です。未定義の動作となります。
W0520063	[メッセージ]	シフト数が多すぎます。
W0520064	[メッセージ]	この宣言は何も宣言できません。
W0520068	[メッセージ]	整数変換で結果の符号が反転しました。
W0520069	[メッセージ]	整数変換で結果の値が丸められました。
W0520070	[メッセージ]	不完全型は許されていません。
W0520076	[メッセージ]	マクロに対する引数がありません。
W0520077	[メッセージ]	宣言に記憶域クラスまたは型指定子がありません。
W0520082	[メッセージ]	記憶域クラスが最初にありません。
	[説明]	記憶域クラスが最初にありません。記憶域クラスは宣言の最初に指定してください。
W0520083	[メッセージ]	型修飾子が複数回指定されました。
W0520099	[メッセージ]	ここでの宣言は引数宣言でなければなりません。
W0520108	[メッセージ]	1 ビットの符号付きビット・フィールドです。
W0520111	[メッセージ]	文は実行されません。
W0520117	[メッセージ]	void でない関数 " *関数名* " は値を返す必要があります。
W0520127	[メッセージ]	文がありません。
W0520128	[メッセージ]	ループはその前のコードから到達しません。
W0520138	[メッセージ]	レジスタ変数に対するアドレス演算子は許されていません。
W0520140	[メッセージ]	関数呼び出しに対する引数が多すぎます。
W0520147	[メッセージ]	宣言は " *宣言* "(宣言位置 行 *行番号*) と整合しません。
W0520152	[メッセージ]	0 でない値がポインタに変換されました。
W0520159	[メッセージ]	宣言は以前の *名前* と整合しません。
W0520161	[メッセージ]	認識されない #pragma です。
W0520165	[メッセージ]	関数呼び出しに引数が足りません。
W0520167	[メッセージ]	" *型名 1*" 型の引数は型 " *型名 2*" の引数と整合しません。
W0520170	[メッセージ]	ポインタがオブジェクトから外れた位置を指しました。
W0520172	[メッセージ]	外部または内部リンケージが以前の宣言と整合しません。
W0520173	[メッセージ]	浮動小数点数は要求された整数型に入りません。

W0520174	[メッセージ]	式は作用しません。
	[説明]	式は作用しません。無効です。
W0520175	[メッセージ]	添字が範囲を越えました。
W0520177	[メッセージ]	*種別* " *シンボル名* " は宣言されましたが参照されていません。
W0520179	[メッセージ]	"%" の右オペランドが 0 です。
W0520180	[メッセージ]	実引数が仮引数と整合しません。
W0520186	[メッセージ]	符号なし整数と 0 の比較は無意味です。
W0520187	[メッセージ]	"==" と思われる "=" の使用があります。
W0520188	[メッセージ]	列挙型に別の型が混在しています。
W0520191	[メッセージ]	型修飾子はキャスト型に意味を持ちません。
W0520192	[メッセージ]	認識されないエスケープ・シーケンスがあります。
W0520220	[メッセージ]	整数値が要求された浮動小数点型に入りません。
W0520221	[メッセージ]	浮動小数点値が要求された浮動小数点型に入りません。
W0520222	[メッセージ]	浮動小数点演算の結果が範囲を越えました。
W0520223	[メッセージ]	関数*名前*は暗黙に宣言されました。
W0520229	[メッセージ]	ビット・フィールドは列挙型のすべての値を保持できません。
W0520231	[メッセージ]	宣言は関数の外で見えません。
W0520236	[メッセージ]	制御式が定数です。
W0520240	[メッセージ]	宣言の指示子が重複しています。
W0520257	[メッセージ]	const 変数 " *名前* " は初期化子が必要です。
W0520260	[メッセージ]	明示的な型がありません。"int" として扱います。
W0520301	[メッセージ]	typedef 名はすでに同じ型で宣言されています。
W0520375	[メッセージ]	宣言は typedef 名を必要とします。
W0520494	[メッセージ]	typedef を伴う void の引数リストの宣言は標準でありません。
W0520513	[メッセージ]	型*名前 1* の値は型*名前 2* の実体として代入できません。
W0520520	[メッセージ]	集成体は "{...}" により初期化してください。
W0520546	[メッセージ]	初期化されないパスがあります。 : *種別* " *シンボル名* "(宣言位置 行*行番号*)
W0520549	[メッセージ]	*種別* " *シンボル名* " は値が設定される前に使用されました。
W0520550	[メッセージ]	*種別* " *シンボル名* " は設定されていますが利用されていません。
W0520609	[メッセージ]	この種類の pragma はここでは使用できません。
W0520618	[メッセージ]	構造体か共用体に名前のないメンバがあります。
W0520660	[メッセージ]	パッキング値が不正です。
W0520676	[メッセージ]	*種別* " *シンボル名* "(宣言位置 行*行番号*) の宣言のスコープ外で使用されました。
W0520767	[メッセージ]	ポインタが幅の小さな整数に変換されました。
W0520815	[メッセージ]	返却型に対する型修飾子は意味がありません。
W0520819	[メッセージ]	"..." は許されていません。

W0520867	［メッセージ］	"size_t" の宣言は期待された型*名前*と一致しません。
W0520870	［メッセージ］	不正な多バイト文字列です。
W0520940	［メッセージ］	void でない関数 " *名前* " に return 文がありません。
W0520951	［メッセージ］	"main" 関数の返却型は "int" である必要があります。
W0520966	［メッセージ］	ユニバーサル・キャラクタ名が不正なキャラクタを指定されました。
W0520967	［メッセージ］	ユニバーサル・キャラクタ名は基本的なキャラクタ・セットの文字を指定できません。
W0520968	［メッセージ］	このユニバーサル・キャラクタは識別子として許されていません。
W0520993	［メッセージ］	" *型名 1* " と " *型名 2* " のポインタ型の減算は標準ではありません。
W0521000	［メッセージ］	記憶域クラスはここでは指定できません。
W0521037	［メッセージ］	このユニバーサル・キャラクタは識別子の先頭に使用できません。
W0521039	［メッセージ］	認識されない STDC pragma です。
W0521040	［メッセージ］	"ON" か "OFF" または "DEFAULT" がありません。
W0521046	［メッセージ］	浮動小数点の値が正しく記述されていません。
W0521051	［メッセージ］	*名前*の型は直後の宣言により与えられなければなりません ("int" に仮定されます)。
W0521053	［メッセージ］	整数がそれより幅の小さなポインタに変換されました。
W0521056	［メッセージ］	ローカル変数へのポインタが返却されました。
W0521057	［メッセージ］	ローカルなテンポラリへのポインタがリターンされました。
W0521072	［メッセージ］	宣言はラベルを持てません。
W0521105	［メッセージ］	#warning 指令 : *文字列*。
	［説明］	ソース・ファイル中に #warning 指令がありました。
W0521222	［メッセージ］	不正なエラー番号です。
W0521223	［メッセージ］	不正なエラータグです。
W0521224	［メッセージ］	エラー番号かエラータグがありません。
W0521273	［メッセージ］	alignment-of オペレータが不完全型に指定されました。
W0521297	［メッセージ］	定数が long long 型には大きすぎます。unsigned long long 型にしてください (標準ではありません)。
W0521422	［メッセージ］	多バイト文字リテラルです。潜在的な移植性の問題があります。
W0521644	［メッセージ］	宣言がファイル末尾で完了していません。
	［説明］	宣言終了のセミコロンがないままファイル末尾に達しました。
W0521649	［メッセージ］	" *マクロ名* " とその置換テキストの間には空白が必要です。
	［対処方法］	マクロ名とその置換テキストの間に空白を入れて区切ってください。
W0523038	［メッセージ］	ひとつの構造体 / 共用体 / クラスの中に，異なる pack 値を持つものが混在しています。
W0523042	［メッセージ］	SuperH コンパイラとの互換性に影響ある " *機能項目* "(オプションや #pragma など) が使用されています。
	［対処方法］	SuperH コンパイラとの互換性に影響する可能性があります。仕様相違の詳細をご確認ください。

W0523061	［メッセージ］	実引数は組み込み関数の仮引数と適合しません。
W0523062	［メッセージ］	返却値の型が組み込み関数の型と合っていません。
W0523068	［メッセージ］	"*cpu*" でアトミック転送関数が使用されています。
W0523116	［メッセージ］	" *文字列* " と他の設定が矛盾しています。
W0530809	［メッセージ］	" *変数名* " で示す変数の const 修飾がファイル間で異なっています。
W0530811	［メッセージ］	" *シンボル名* " で示すシンボルの型がファイル間で異なっています。
W0550001	［メッセージ］	マクロ呼び出し時に指定された実引数が多すぎます。
	［対処方法］	実引数を確認してください。
W0550005	［メッセージ］	*option* オプションに指定されたシンボル *symbol* が不正です。
	［説明］	*option* オプションに指定されたシンボル *symbol* が不正です。オプション指定は無視されます。
	［対処方法］	オプション指定のシンボルを確認してください。
W0550010	［メッセージ］	ディスプレースメントの値が指定可能な値の範囲を越えています。
	［説明］	ディスプレースメントの値が指定可能な値の範囲を越えています。下位の有効な桁数分だけが指定されたものとみなし，アセンブルを続行します。
	［対処方法］	ディスプレースメントの値を確認してください。
W0550011	［メッセージ］	イミーディエトの値が指定可能な値の範囲を越えています。
	［説明］	イミーディエトの値が指定可能な値の範囲を越えています。 下位の有効な桁数分だけが指定されたものとみなし，アセンブルを続行します。
	［対処方法］	イミーディエトの値を確認してください。
W0550012	［メッセージ］	オペランドに指定した値が指定可能な値の範囲を越えています。
	［説明］	オペランドに指定した値が指定可能な値の範囲を越えています。 下位の有効な桁数分だけが指定されたものとみなし，アセンブルを続行します。
	［対処方法］	オペランドの値を確認してください。
W0550013	［メッセージ］	*register* がレジスタとしてオペランドに指定されています。
	［対処方法］	レジスタの指定を確認してください。
W0550014	［メッセージ］	prepare/dispose 命令のレジスタ・リストに指定した値が不正です。
	［説明］	prepare/dispose 命令のレジスタ・リストに指定した値が不正です。 下位の有効な桁数分だけが指定されたものとみなし，アセンブルを続行します。
	［対処方法］	レジスタ・リストの値を確認してください。
W0550015	［メッセージ］	prepare/dispose 命令のレジスタ・リストに指定したレジスタが不正です。
	［説明］	prepare/dispose 命令のレジスタ・リストに指定したレジスタが不正です。 不正なレジスタを無視して，アセンブルを続行します。
	［対処方法］	レジスタ・リストのレジスタを確認してください。
W0550017	［メッセージ］	sld/sst 命令のベース・レジスタに ep 以外を指定しています。
	［対処方法］	ベース・レジスタの指定を確認してください。
W0550018	［メッセージ］	*inst* 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。
	［対処方法］	システム・レジスタの番号を確認してください。

W0550019	［メッセージ］	オペランドに指定した値は*文字列*の倍数である必要があります。
	［説明］	オペランドに指定した値は*文字列*の倍数である必要があります。 端数を切捨てて，アセンブルを続行します。
	［対処方法］	オペランドの値を確認してください。
W0550021	［メッセージ］	*string* が，以前に指定したレジスタ数と異なる数で指定されています。
	［説明］	*string* が，以前に指定したレジスタ数と異なる数で指定されています。すでに指定されている数を使用します。この指定は無視されます。
	［対処方法］	レジスタ数を確認してください。
W0550026	［メッセージ］	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。
	［説明］	奇数番号の付いたレジスタ（r1，r3，…，r31）が指定されています。 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。 偶数番号の付いたレジスタ（r0，r2，r4，…，r30）を指定したとして，アセンブルを続行します。
	［対処方法］	レジスタの指定を確認してください。
W0550028	［メッセージ］	-Xreg_mode オプションによる指定と $REG_MODE 制御命令による指定が異なっています。
	［説明］	-Xreg_mode オプションによる指定と $REG_MODE 制御命令による指定が異なっています。-Xreg_mode オプションを優先し，$REG_MODE 制御命令によるレジスタ・モード指定は無視されます。
	［対処方法］	オプションの指定を確認してください。
W0550031	［メッセージ］	定義されていない識別子 *identifier* が参照されています。
	［対処方法］	識別子を確認してください。
W0561000	［メッセージ］	Option " *オプション* " ignored
	［説明］	" *オプション* " は無効です。" *オプション* " を無視します。
W0561001	［メッセージ］	Option " *オプション1* " is ineffective without option " *オプション2* "
	［説明］	" *オプション1* " は " *オプション2* " が必要です。" *オプション1* " を無視します。
W0561002	［メッセージ］	Option " *オプション1* " cannot be combined with option " *オプション2* "
	［説明］	" *オプション1* " と " *オプション2* " は同時に指定できません。" *オプション1* " を無視します。
W0561003	［メッセージ］	Divided output file cannot be combined with option " *オプション* "
	［説明］	" *オプション* " 指定時，出力ファイルの分割指定はできません。オプションの指定を無視します。先頭入力ファイル名を出力ファイル名として使用します。
W0561004	［メッセージ］	Fatal level message cannot be changed to other level : "*option*"
	［説明］	致命的エラー・メッセージはレベル変更できません。"*option*" の指定を無視します。 change_message オプションで変更できるメッセージは，Information/Warning/Error レベルです。
W0561005	［メッセージ］	Subcommand file terminated with end option instead of exit option
	［説明］	end オプションの後に処理指定がありません。exit オプションを仮定して処理します。
W0561006	［メッセージ］	Options following exit option ignored
	［説明］	exit オプションの後のオプションを無視しました。

W0561007	［メッセージ］	Duplicate option : "*オプション*"
	［説明］	"*オプション*" が重複しています。最後に指定したオプションを有効にします。
W0561008	［メッセージ］	Option "*オプション*" is effective only in cpu type "*マイコン種別*"
	［説明］	"*オプション*" は "*マイコン種別*" 以外では無効です。"*オプション*" を無視します。
W0561010	［メッセージ］	Duplicate file specified in option "*オプション*" : "*ファイル名*"
	［説明］	"*オプション*" で同じファイルを二度指定しました。二度目の指定を無視します。
W0561011	［メッセージ］	Duplicate module specified in option "*オプション*" : "*モジュール*"
	［説明］	"*オプション*" で同じモジュールを二度指定しました。二度目の指定を無視します。
W0561012	［メッセージ］	Duplicate symbol/section specified in option "*オプション*" : "*名前*"
	［説明］	"*オプション*" で同じシンボル名またはセクション名を二度指定しました。二度目の指定を無視します。
W0561013	［メッセージ］	Duplicate number specified in option "*オプション*" : "*番号*"
	［説明］	"*オプション*" で同じエラー番号を指定しました。最後に指定した方を有効にします。
W0561014	［メッセージ］	License manager is not installed
	［説明］	ライセンス・マネージャがインストールされていません。対応するライセンス・マネージャをインストールしてください。
W0561016	［メッセージ］	The evaluation version of *バージョン* is valid for the remaining *日数* days. After that, link size limit (256 Kbyte) will be applied. Please consider purchasing the product.
	［説明］	サイズ制限なし無償評価版として動作できる評価期間の残り日数を示しています。評価期間の後はサイズ制限あり無償評価版として動作します。製品版の購入をご検討ください。
W0561017	［メッセージ］	Paid license of "*バージョン*" is not found, and the evaluation period has expired. Please consider purchasing the product.
	［説明］	有償ライセンスが確認できません。また無償評価期間も終了しています。製品版の購入をご検討ください。
W0561100	［メッセージ］	Cannot find "*名前*" specified in option "*オプション*"
	［説明］	"*オプション*" で指定したシンボル名，またはセクション名が見つかりません。"*名前*" の指定を無視します。
W0561101	［メッセージ］	"*名前*" in option "*オプション*" conflicts between symbol and section
	［説明］	"*オプション*" で指定した "*名前*" がセクション名とシンボル名の両方に存在します。シンボル名を変更の対象にします。
W0561102	［メッセージ］	Symbol "*シンボル*" redefined in option "*オプション*"
	［説明］	"*オプション*" で指定したシンボルはすでに定義されています。そのまま処理を続けます。
W0561103	［メッセージ］	Invalid address value specified in option "*オプション*" : "*アドレス*"
	［説明］	"*オプション*" で指定した "*アドレス*" は無効な値です。"*アドレス*" の指定を無視します。
W0561104	［メッセージ］	Invalid section specified in option "*オプション*" : "*セクション*"
	［説明］	"*オプション*" で無効なセクションを指定しています。
	［対処方法］	以下を確認してください。 -output オプションは，初期値のないセクションを指定できません。

W0561120	［メッセージ］	Section address is not assigned to " *セクション* "
	［説明］	" *セクション* " のアドレス指定がありません。" *セクション* " を最後尾に配置します。
	［対処方法］	rlink オプション -start を使用して，セクションのアドレスを設定してください。
W0561121	［メッセージ］	Address cannot be assigned to absolute section " *セクション* " in start option
	［説明］	" *セクション* " は絶対アドレス・セクションです。絶対アドレス・セクションに対するアドレス指定を無視します。
W0561122	［メッセージ］	Section address in start option is incompatible with alignment : " *セクション* "
	［説明］	start オプションで指定した " *セクション* " のアドレスは整列条件数と矛盾しています。整列条件数に合わせてセクションアドレスを補正します。
W0561130	［メッセージ］	Section attribute mismatch in rom option : " *セクション1* "," *セクション2* "
	［説明］	rom オプションで指定した " *セクション1* " と " *セクション2* " の属性，整列条件数が異なります。" *セクション2* " の整列条件数はどちらか大きい方を有効とします。
W0561140	［メッセージ］	Load address overflowed out of record-type in option " *オプション* "
	［説明］	アドレス値よりも小さい record 形式を指定しました。指定した record 形式を越える範囲は，別の record 形式で出力します。
W0561141	［メッセージ］	Cannot fill unused area from " *アドレス* " with the specified value
	［説明］	空きエリアのサイズが space オプションで指定された値の倍数となっていないため，" *アドレス* " 以降に指定データを出力できませんでした。
W0561142	［メッセージ］	Cannot find symbol which is a pair of " *シンボル* "
	［説明］	空き領域の範囲を表す " *シンボル* " と対になるシンボルが見つかりませんでした。
W0561150	［メッセージ］	Sections in " *オプション* " option have no symbol
	［説明］	" *オプション* " で指定したセクションは外部定義シンボルがありません。
W0561160	［メッセージ］	Undefined external symbol " *シンボル* "
	［説明］	未定義の " *シンボル* " を参照しています。
W0561181	［メッセージ］	Fail to write " *出力コード種別* "
	［説明］	出力ファイルへの，" *出力コード種別* " の書き込みを失敗しました。 出力ファイルに，" *出力コード種別* " の書き込み先アドレスが含まれていない可能性があります。 出力コード種別 : CRC コード書き込み失敗時 : "CRC Code"
W0561191	［メッセージ］	Area of "FIX" is within the range of the area specified by "*cpu*=< *メモリ属性* >" :"<*start*>-<*end*>"
	［説明］	cpu オプションで，メモリ属性 FIX と FIX 以外の <*start*>-<*end*> 範囲が重複していたため，FIX を有効にしました。
W0561193	［メッセージ］	Section " *セクション名* " specified in option " *オプション* " is ignored
	［説明］	-cpu=stride の機能で分割したセクションの，後半部への " *オプション* " 指定は無効となります。
	［対処方法］	後半部のセクションは " *オプション* " で指定しないでください。
W0561200	［メッセージ］	Backed up file " *ファイル1* " into " *ファイル2* "
	［説明］	入力ファイル " *ファイル1* " は書き換えられました。書き換える前の " *ファイル1* " の内容は " *ファイル2* " にバックアップされています。

W0561210	［メッセージ］	Section "*section-name*" ID(value="*number*") in "*file-name*" is reserved
	［説明］	入力ファイル内のセクション情報で予約とした番号があります。"*number*" 指定を無視します。コンパイラ，アセンブラのバージョンが正しいか確認してください。
W0561300	［メッセージ］	Option " *オプション* " is ineffective without debug information
	［説明］	入力ファイル内にデバッグ情報がありません。" *オプション* " 指定を無視します。
	［対処方法］	コンパイル，アセンブル時に該当するオプションを指定しているか確認してください。
W0561301	［メッセージ］	No inter-module optimization information in input files
	［説明］	入力ファイル内にリンク時最適化（モジュール間最適化）情報がありません。optimize オプションを無視します。
	［対処方法］	コンパイル，アセンブル時に goptimize オプションを指定してください。
W0561302	［メッセージ］	No stack information in input files
	［説明］	入力ファイル内にスタック情報がありません。stack オプションを無視します。入力ファイルがアセンブラ出力ファイルの場合は，stack オプションは無効です。
W0561305	［メッセージ］	Entry address in " *ファイル* " conflicts : " *アドレス* "
	［説明］	異なるエントリーアドレスのファイルが複数入力されています。
W0561310	［メッセージ］	" *セクション* " in " *ファイル* " is not supported in this tool
	［説明］	" *ファイル* " 内に非サポートセクションがありました。" *セクション* " を無視します。
W0561311	［メッセージ］	Invalid debug information format in " *ファイル* "
	［説明］	" *ファイル* " 内のデバッグ情報は dwarf2 ではありません。debug 情報を削除します。
W0561320	［メッセージ］	Duplicate symbol " *シンボル* " in " *ファイル* "
	［説明］	" *シンボル* " は重複しています。先に入力したファイル内シンボルを優先します。
W0561322	［メッセージ］	Section alignment mismatch : " *セクション* "
	［説明］	整列条件数の異なる同名セクションを入力しました。整列条件数は最大の指定を有効にします。
W0561323	［メッセージ］	Section attribute mismatch : " *セクション* "
	［説明］	属性の異なる同名セクションを入力しました。絶対セクションと相対セクションの場合は，絶対セクションとして扱います。read/write 属性が異なる場合は，どちらも許可します。
W0561324	［メッセージ］	Symbol size mismatch : " *シンボル* " in " *ファイル* "
	［説明］	サイズの異なるコモン・シンボルまたは定義シンボルが入力されました。定義シンボルを優先します。コモン・シンボル同士の場合は，先に入力したファイル内シンボルを優先します。
W0561326	［メッセージ］	Reserved symbol " *シンボル* "is defined in " *ファイル* "
	［説明］	予約された名称のシンボル " *シンボル* " が " *ファイル* " 内で定義されています。
W0561327	［メッセージ］	Section alignment in option "aligned_section" is small : " *セクション* "
	［説明］	aligned_section オプション指定時の整列条件数 16 の方が，" *セクション* " の整列条件数より小さいため，指定セクションに対するオプション指定を無視します。

W0561331	［メッセージ］	Section alignment is not adjusted : " *セクション* "
	［説明］	整列条件数の異なる同名セクションを入力しました。整列条件数は最大の指定を有効にします。入力時の整列条件を満たしていない可能性があります。
W0561402	［メッセージ］	Parentheses specified in option "start" with optimization
	［説明］	start オプションで括弧 "()" を記述した場合，最適化機能は使用できません。 最適化機能を無効にします。
W0561410	［メッセージ］	Cannot optimize " *ファイル* "-" *セクション* " due to multi label relocation operation
	［説明］	複数ラベルのリロケーション演算を持つセクションは最適化できません。" *ファイル* " 内の " *セクション* " を最適化対象外にします。
W0561510	［メッセージ］	Input file was compiled with option "smap" and option "map" is specified at linkage
	［説明］	smap を指定してコンパイルしたファイルがあります。
	［対処方法］	smap を指定したファイルは，2 回目のビルドで map オプションを指定してコンパイルしないでください。
W0595020	［メッセージ］	Warning : *name* : *address* : *cause*
	［説明］	例外要因 *name* の例外ハンドラに SYNCP 命令が配置されていない可能性があります。
	［対処方法］	「11.4 例外ハンドラ先頭への SYNCP 命令配置確認ツール」を参照してください。

11. 注意事項

この章では，CC-RH を用いる際に注意すべき点について説明します。

11.1 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。
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
- 命令の実行順序が変わる
- メモリのアクセス回数・アクセス幅が変わる

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;
void func(void){
    if(a <= 0){
        b++;
    } else {
        b+=2;
    }
    b++;
}
```

```
_func:
        MOVHI   HIGHW1(#_a), R0, R6
        LD.W    LOWW(#_a)[R6], R6
        CMP     0x00000000, R6
        MOVHI   HIGHW1(#_b), R0, R6
        LD.W    LOWW(#_b)[R6], R6
        BGT     .BB1_2          ; bb3
.BB1_1:     ; bb1
        ADD     0x00000001, R6
        BR      .BB1_3          ; bb9
.BB1_2:     ; bb3
        ADD     0x00000002, R6
.BB1_3:     ; bb9
        ADD     0x00000001, R6
        MOVHI   HIGHW1(#_b), R0, R7
        ST.W    R6, LOWW(#_b)[R7]
        JMP     [R31]
```

【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;
void func(void){
    if(a <= 0){
        b++;
    } else {
        b+=2;
    }
    b++;
}
```

```
_func:
        MOVHI   HIGHW1(#_a), R0, R6
        LD.W    LOWW(#_a)[R6], R6
        CMP     0x00000000, R6
        BGT     .BB1_2          ; bb3
.BB1_1:    ; bb1
        MOVHI   HIGHW1(#_b), R0, R6
        LD.W    LOWW(#_b)[R6], R6
        ADD     0x00000001, R6
        BR      .BB1_3          ; bb9
.BB1_2:    ; bb3
        MOVHI   HIGHW1(#_b), R0, R6
        LD.W    LOWW(#_b)[R6], R6
        ADD     0x00000002, R6
.BB1_3:    ; bb9
        MOVHI   HIGHW1(#_b), R0, R7
        ST.W    R6, LOWW(#_b)[R7]
        LD.W    LOWW(#_b)[R7], R6
        ADD     0x00000001, R6
        ST.W    R6, LOWW(#_b)[R7]
        JMP     [R31]
```

11.2 アセンブラにおける -Xcpu オプション指定

-Xcpu オプションに指定する引数によって，アセンブルできる命令が異なります。使用できる命令については，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。使用できない命令を記述した場合はアセンブルエラーとし，次のメッセージを出力します。

E0550269 : RH850（コア名）指定時には記述することのできない命令が記述されています。

11.3 ビット操作命令の出力を制御する方法 【V1.05.00 以降】

組み込み関数を使用せずにビット操作命令を出力したい場合は，次の条件をすべて満たしてください。

(a) 定数値を代入する

(b) 代入先を 1 バイト型で 1 ビット幅のビットフィールドにする

(c) 代入先を volatile 修飾する

ビット操作命令を出力したくない場合は，上記の (c) を満たしたうえで，(a) の代入値を定数値以外にするか，または (b) の型を 1 バイト型以外の型にしてください。

上記のいずれにも該当しない場合，ビット操作命令を出力するかどうかは，最適化レベルやソース・プログラムの記述内容によってコンパイラが自動的に判別します。

注意 1 バイト型は，（char/unsigned char/signed char/_Bool）を指します。

例

```
volatile struct {
  unsigned char bit0:1;
  unsigned int  bit1:1;
} data;

void func(void) {
  data.bit0 = 1;  /* ビット操作命令を出力する */
  data.bit1 = 1;  /* ビット操作命令を出力しない */
}
```

11.4 例外ハンドラ先頭への SYNCP 命令配置確認ツール

例外ハンドラ先頭への SYNCP 命令配置確認ツール（syncp_checker）は，G3M コア用のプロジェクトにおいて，ビルドでモトローラ・S タイプ・ファイルを出力する際に，例外ハンドラ先頭に SYNCP 命令が配置されていることを確認するツールです。CS+ V4.00.00 以降をインストールした場合に利用可能となります。

例外ハンドラ先頭への SYNCP 命令配置については，「RH850G3M ユーザーズマニュアル ソフトウェア編」（Rev1.10 以降）を参照してください。

本ツールでは，指定した例外ベクタのベース・アドレス（デフォルトは 0 番地）からモトローラ・S タイプファイルを解析して，次のいずれかを表示します。

OK	SYNCP 命令が配置されている，または SYNCP 命令は不要である
E0595010:NG	SYNCP 命令が必要であるにもかかわらず配置されていない可能性がある
W0595020:Warning	本ツールでは OK か NG か判定不能である

NG，または Warning が表示された場合は，例外ハンドラの先頭コードを確認してください。

NG，または Warning が表示された場合でも，出力ファイルは生成されます。

統合開発環境 CS+ では，CC-RH（ビルド・ツール）のプロパティパネルの［ヘキサ出力オプション］タブで，本ツールを制御できます。［その他］カテゴリの［例外ハンドラ先頭への SYNCP 命令配置確認を行う］プロパティで［いいえ］を選択することで本ツールの起動を抑止できます。例外ベクタ・テーブルを含まないプロジェクトなどではこの方法で抑止してください。

本ツールのオプションは次のとおりです。

-b=*address*	解析を開始するための例外ベクタのベース・アドレスを指定します。デフォルト値は 0 です。
-n=*num*	ユーザ割り込み（EIINT）に対応する例外ベクタ・テーブルのエントリ数を指定します。デフォルト値は 16 です。

11.5 コンパイラ・パッケージのバージョンについて

最適化リンカを使用する際は，入力するすべてのオブジェクト・ファイル，リロケータブル・ファイル，ライブラリ・ファイルを生成したコンパイラ・パッケージと同じか，より新しいコンパイラ・パッケージに付属しているものを使用してください。

標準ライブラリを使用する際は，使用する最適化リンカと同じコンパイラ・パッケージに付属しているものを使用してください。

A. クイック・ガイド

この章では，CC-RH 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。

A.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。

A.1.1 短い命令長でアクセスできる領域へ配置する

CC-RH では，変数のアクセスは通常 movhi 命令（4 バイト）と ld/st 命令（4 バイト）の 2 命令（計 8 バイト）を使用しますが，#pragma section 指令を活用することで ld/st 命令（4 バイト，または 6 バイト），または sld/sst 命令（2 バイト）の 1 命令でアクセスするコードを生成します。これにより，コード・サイズを削減することができます。以下に詳細を示します。

A.1.1.1 GP 相対アクセス

GP（グローバル・ポインタ）と，ld/st 命令によってアクセスできるセクションに変数を配置することにより変数のアクセスを 1 命令で行うコードを生成します。
変数の定義／ 参照時は，#pragma section 指令を使用し，属性指定文字に gp_disp16/ gp_disp23 のいずれかを指定します。

注意 gp_disp32 には，コード・サイズを削減する効果はありません。

```
#pragma section 属性指定文字
変数宣言／定義
#pragma section default
```

例 1. GP 相対で 4 バイトのロード／ストア命令によりアクセスする場合

```
#pragma section gp_disp16
int a = 1;              /*.sdata セクションに配置する */
int b;                  /*.sbss セクションに配置する */
#pragma section default
```

例 2. GP 相対で 6 バイトのロード／ストア命令によりアクセスする場合

```
#pragma section gp_disp23
int a = 1;             /*.sdata23 セクションに配置する */
int b;                 /*.sbss23 セクションに配置する */
#pragma section default
```

A.1.1.2 EP 相対アクセス

EP（エレメント・ポインタ）と，sld/sst 命令，または ld/st 命令によってアクセスできるセクションに変数を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。以下のいずれかの方法により，EP 相対でアクセスするセクションに変数を配置させることができます。

(1) -Omap/-Osmap オプション指定
外部変数アクセス最適化を行います。最適化リンケージエディタが生成する外部シンボル割り付け情報を元に，アクセス頻度の高い外部変数に対して EP 相対アクセスするコードを出力します。

(2) #pragma section 指令
変数の定義／ 参照時は，#pragma section 指令を使用し，属性指定文字に ep_disp4/ep_disp5/ep_disp7/ ep_disp8/ep_disp16/ep_disp23/ep_auto のいずれかを指定します。

注意 ep_disp32 には，コード・サイズを削減する効果はありません。

```
#pragma section 属性指定文字
変数宣言／定義
#pragma section default
```

例 1. EP 相対で 2 バイトのロード／ストア命令によりアクセスする場合

```
#pragma section ep_disp4
int a = 1;              /*.tdata4 セクションに配置する */
int b;                  /*.tbss4 セクションに配置する */
#pragma section default
```

属性指定文字として，ep_disp5/ep_disp7/ep_disp8 を指定しても同様のアクセスとなります。

例 2. EP 相対で 4 バイトのロード／ストア命令によりアクセスする場合

```
#pragma section ep_disp16
int a = 1;              /*.edata セクションに配置する */
int b;                  /*.ebss セクションに配置する */
#pragma section default
```

例 3. EP 相対で 6 バイトのロード／ストア命令によりアクセスする場合

```
#pragma section ep_disp23
int a = 1;              /*.edata23 セクションに配置する */
int b;                  /*.ebss23 セクションに配置する */
#pragma section default
```

A.1.2 配置領域を変更する

変数のデフォルトの配置先セクションは，次のとおりになります。

- 初期値なし変数：.bss セクション
- 初期値あり変数：.data セクション
- const 変数：.const セクション

A.1.2.1 #pragma section 指令を使用して配置領域を変更する

配置する領域（セクション）を変更するには，#pragma section 指令で属性指定文字を指定します。

例 #pragma section 指令記述

```
#pragma section gp_disp16 "mysdata"
int a = 1;              /*mysdata.sdata セクションに配置する */
int b;                  /*mysdata.sbss セクションに配置する */
#pragma section default
```

#pragma section 指令の詳細な使用方法は，「4.2.6.1 関数とデータのセクション割り当て」を参照してください。

なお，#pragma section 指令を使った変数を別のソース・ファイルの関数から参照する場合には，参照する側のファイルで，該当する変数を同じ #pragma section 指令付きで extern 宣言する必要があります。
定義と宣言で指定が異なり，指定したセクション属性ではアクセスできない場合に，以下のエラーを出力します。

```
E0562330 : Relocation size overflow : " ファイル"-" セクション"-" オフセット"
```

例 1. 変数を定義しているファイル

```
#pragma section zconst
const unsigned char table_data[9] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
                                                    /*.zconst セクションに配置する */
#pragma section default
```

例 2. 変数を参照するファイル

```
#pragma section zconst
extern  const unsigned char  table_data[];      /*.zconst セクションに配置する */
#pragma section default
```

なお，SH ファミリ用コンパイラの C ソース移植性を考慮して，以下のような表記も記述可能です。

例 #pragma section 指令記述

```
#pragma section mydata
int a = 1;              /*mydata.data セクションに配置する */
int b;                  /*mydata.bss セクションに配置する */
#pragma section default
```

A.1.2.2 -Xsection オプションを使用して配置領域を変更する

-Xsection オプションによって，デフォルトの配置先セクションを変更できます。アクセス効率の良いセクションに配置させることでコード・サイズを削減することができます。

(1) -Xsection オプションを使用して，デフォルトのセクション種別を指定します。

例 .sdata/.sbss セクションに配置させる場合

```
>ccrh main.c -Xsection=data=gp_disp16
```

ただし，(1) で指定したセクションに変数が収まりきらない場合は，リンク時に以下のエラーが出力されます。この場合は，C ソース・ファイル上で変数のセクションを変更してください。

```
E0562330 : Relocation size overflow : " ファイル"-" セクション"-" オフセット "
```

例 変数を .sdata23/.sbss23 セクションに変更する場合

```
int a = 1;              /*(1) で指定したセクションに配置する */
int b;                  /*(1) で指定したセクションに配置する */
#pragma section gp_disp23
int c = 1;              /*.sdata23 セクションに配置する */
int d;                  /*.sbss23 セクションに配置する */
#pragma section default
int e = 1;              /* (1) で指定したセクションに配置する */
int f;                  /* (1) で指定したセクションに配置する */
```

A.1.2.3 -Xpreinclude オプションを使用して配置領域を変更する

-Xpreinclude オプションによって，C ソース・ファイルを変更せずに，ファイル内で宣言・定義したすべての変数を任意のセクションに配置させることが可能です。アクセス効率の良いセクションに配置させることでコード・サイズを削減することができます。

(1) #pragma section 指令を記述したヘッダ・ファイル（.h）を準備します。

例 .sdata/.sbss セクションに配置させる場合 [section.h]

```
#pragma section gp_disp16
```

(2) -Xpreinclude オプションによって，（1）で作成したヘッダ・ファイルをコンパイル単位の先頭にインクルードします。

例 セクション指定したヘッダ・ファイル名が section.h の場合

```
>ccrh main.c -Xpreinclude=section.h
```

main.c の先頭に，section.h がインクルードされているものとしてコンパイルします。

ただし，（1）で指定したセクションに変数が収まりきらない場合は，リンク時に以下のエラーが出力されます。この場合は，C ソース・ファイル上で変数のセクションを変更してください。

```
E0562330 : Relocation size overflow : " ファイル"-" セクション"-" オフセット"
```

例 変数を .sdata23/.sbss23 セクションに変更する場合

```
int a = 1;              /*(1) で指定したセクションに配置する */
int b;                  /*(1) で指定したセクションに配置する */
#pragma section gp_disp23
int c = 1;              /*.sdata23 セクションに配置する */
int d;                  /*.sbss23 セクションに配置する */
#pragma section default
int e = 1;              /* デフォルトの .data セクションに配置する */
int f;                  /* デフォルトの .bss セクションに配置する */
```

A.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になりません。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，volatile 指定された変数に値を代入するときは必ずメモリへ値を書き込みます。また，volatile 指定された変数のアクセス順序や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。

例 1. volatile 指定しなかった場合のソースと出力コードイメージの例
“ 変数 a”，“ 変数 b” 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，最適化される場合があります。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;
void func(void){
    if(a <= 0){
        b++;
    } else {
        b+=2;
    }
    b++;
}
```

```
_func:
        movhi   highw1(#_a), r0, r6
        ld.w    loww(#_a)[r6], r6
        cmp     0x00000000, r6
        movhi   highw1(#_b), r0, r6
        ld.w    loww(#_b)[r6], r6
        bgt     .bb1_2          ; bb3
.bb1_1:     ; bb1
        add     0x00000001, r6
        br      .bb1_3          ; bb9
.bb1_2:     ; bb3
        add     0x00000002, r6
.bb1_3:     ; bb9
        add     0x00000001, r6
        movhi   highw1(#_b), r0, r7
        st.w    r6, loww(#_b)[r7]
        jmp     [r31]
```

例 2. 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，値を代入するときは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。
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;
void func(void){
    if(a <= 0){
        b++;
    } else {
        b+=2;
    }
    b++;
}
```

```
_func:
        movhi   highw1(#_a), r0, r6
        ld.w    loww(#_a)[r6], r6
        cmp     0x00000000, r6
        bgt     .bb1_2          ; bb3
.bb1_1:    ; bb1
        movhi   highw1(#_b), r0, r6
        ld.w    loww(#_b)[r6], r6
        add     0x00000001, r6
        br      .bb1_3          ; bb9
.bb1_2:    ; bb3
        movhi   highw1(#_b), r0, r6
        ld.w    loww(#_b)[r6], r6
        add     0x00000002, r6
.bb1_3:    ; bb9
        movhi   highw1(#_b), r0, r7
        st.w    r6, loww(#_b)[r7]
        ld.w    loww(#_b)[r7], r6
        add     0x00000001, r6
        st.w    r6, loww(#_b)[r7]
        jmp     [r31]
```

A.1.4 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 .zconst セクションに割り当てるときは #pragma section zconst 指定をしてください。
.const セクションを .zconst23 セクションに割り当てるときは #pragma section zconst23 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（.data など）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（.const/.zconst/.zconst23）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（.const/.zconst/.zconst23）に配置されます。

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

A.2 関　　数

この節では，関数について説明します。

A.2.1 配置領域を変更する

プログラム領域のセクション名を変更する場合は，以下のように #pragma section 指令を使用して関数を指定します。

```
#pragma section text  [" セクション名 "]
```

#pragma section 指令で，任意の text 属性のセクションを作成する場合，実際に生成されるセクション名は “ 指定した文字列＋ .text” となります。
セクションの開始アドレスは，以下のように -start オプションで指定します。

```
-start=sec.text/1000
```

アドレスは 16 進数で指定してください。アドレスの指定がない場合は，0 番地から割り付けます。

A.2.2 離れた関数をコールする

C コンパイラは，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。
上記エラーを解決する方法として，まずは -Xcall_jump=32 を指定していただくことで，jarl32，および jr32 命令を生成することができます。
-Xcall_jump=22 オプションを指定されている場合は，C コンパイラの -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。

例　　-Xfar_jump オプションで指定するファイル

```
_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}
```

A.2.3 アセンブラ命令の埋め込み

CC-RH 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。これは，関数そのものをアセンブラ命令とみなして，呼び出し箇所にインライン展開します。

- #pragma 指令

```
#pragma inline_asm func
static int func(int a, int b) {
        /* アセンブラ命令 */
}
```

詳細は「アセンブラ命令の記述」を参照してください。

A.2.4 RAM で特定のルーチンを実行する

プログラム中の特定のルーチンを RAM 上で実行したい場合，最適化リンカの -rom オプションを使用して，次の手順で実現できます。

(1) 対象ルーチンの ROM 上の配置位置と，RAM 上の実行位置を決めて，それぞれにセクションを割り当てる。

(2) 次のような転送用ルーチンを用意し，プログラムに組み込む。
(2-1) 対象ルーチンを，ROM 上のセクションから RAM 上のセクションへコピーする。
(2-2) コピー後，RAM 上の対象ルーチンを呼び出す。

(3) プログラムのビルド時に，次のように指定して最適化リンカを実行する。
(3-1) 対象ルーチンの ROM 上の配置先のセクション名を，-rom オプションの ROM 側に指定する。
(3-2) 対象ルーチンを実行する RAM 上のセクション名を，-rom オプションの RAM 側に指定する。

A.3 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。

A.3.1 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      サイズ
```

他ファイルからも参照可能にするには，そのラベルを .public 疑似命令で宣言する必要があります。

```
.public  ラベル名
```

例　　初期値なし変数定義

```
        .dseg   sbss
        .public _val0           ;_val0 を他ファイルから参照可能にします
        .public _val1           ;_val1 を他ファイルから参照可能にします
        .public _val2           ;_val2 を他ファイルから参照可能にします
        .align  4               ;_val0  を 4 バイト整列します
_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
```

A.3.2 初期値あり変数を定義する

初期値あり変数領域を確保するには，初期値ありセクションの中で，.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
```

- 4 バイトの場合

```
[ ラベル :]  .db4     値
```

他ファイルからも参照可能にするには，そのラベルを .public 疑似命令で宣言する必要があります。

```
.public  ラベル名
```

例　　初期値あり変数定義

```
        .dseg   sdata
        .public _val0           ;_val0 を他ファイルから参照可能にします
        .public _val1           ;_val1 を他ファイルから参照可能にします
        .public _val2           ;_val2 を他ファイルから参照可能にします
        .align  4               ;_val0 を 4 バイト整列します
_val0:
        .db4    100             ;_val0 は 4 バイト分領域を確保し，100 を格納します
_val1:
        .db2    10              ;_val0 は 2 バイト分領域を確保し，10 を格納します
_val2:
        .db     1               ;_val0 は 1 バイト分領域を確保し，1 を格納します
```

A.3.3 const 定数を定義する

const 定数を定義するには，.const/.zconst/.zconst23 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
```

- 4 バイトの場合

```
[ ラベル :]  .db4     値
```

例　　const 定数定義

```
        .cseg   const
        .public _p              ;_p を他ファイルから参照可能にします
        .align  4               ;_p を 4 バイト整列します
_p:
        .db2    10              ;_p は 2 バイト分領域を確保し，10 を格納します
```

改訂記録

Rev.	発行日	改定内容	
		ページ	ポイント
1.00	2015.09.14	-	初版発行
1.01	2016.07.01	16	排他制御チェック設定ファイルを追加しました。
		23, 他	コンパイル・オプション -Xcheck_exclusion_control を追加しました。
		44	［詳細説明］を変更しました。
		55, 56	次の MISRA-C:2012 ルールを追加しました。 2.6 2.7 9.2 9.3 12.1 12.3 12.4 14.4 15.1 15.2 15.3 15.4 15.5 15.6 15.7 16.1 16.2 16.3 16.4 16.5 16.6 16.7 17.1 17.7 18.4 18.5 19.2 20.1 20.2 20.3 20.4 20.5 20.6 20.7 20.8 20.9 20.10 20.11 20.12 20.13 20.14
		167	［詳細説明］を変更しました。
		664, 822	動的メモリ管理関数を追加しました。
		824, 825	セキュリティ機能の説明を追加しました。
		828, 他	スタートアップの説明を変更しました。
		855	26 レジスタ・モードを削除しました。
		858, 他	不要なメッセージを削除しました。
		865	E0521158 を追加しました。
		879	F0523089 を追加しました。
		885	W0511143 を追加しました。
		886	W0520171 を追加しました。
1.02	2016.12.01	12	「ライセンスについて」の説明を変更しました。
		12	「standard 版と professional 版について」を追加しました。
		12	「無償評価版について」を追加しました。
		19	サブコマンド・ファイル指定の動作を変更しました。
		21 他	コンパイル・オプション -g_line を追加しました。
		22, 他	コンパイル・オプション -Xuse_fp16 を追加しました。
		36	省略時解釈を変更しました。
		55, 57, 59, 247	__fp16 型を追加しました。
		56, 57	次の MISRA-C:2012 ルールを追加しました。 2.2 3.2 5.1 5.6 5.7 5.8 5.9 8.3 8.9 9.1 12.2 21.1 21.2 21.3 21.4 21.5 21.6 21.7 21.8 21.9 21.10

Rev.	発行日	改定内容	
		ページ	ポイント
1.02	2016.12.01	66	［詳細説明］を変更しました。
		78	［詳細説明］を変更しました。
		91	［詳細説明］を変更しました。
		95	［詳細説明］を変更しました。
		110	-D，-U，-I オプションの分類を変更しました。
		160, 173	セクション配置できなかった場合の動作を変更しました。
		233, 他	処理系依存の次の項目を変更しました。 4.1.3 (1), (4), (6), (7), (9), (12), (14), (16), (27), (30), (36), (37), (38)
		236	half.h ファイルの説明を追加しました。
		244	10 進定数が表現できる型の説明を変更しました。
		254	以下の予約語を追加しました。 __fp16，__set_il_rh，__ldsr_rh，__stsr_rh
		263	説明を変更しました。
		272	備考の内容を変更しました。
		275	「組み込み関数」の説明を変更しました。
		276	__stcw の返却型を変更しました。
		279	以下の説明を変更しました。 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サイズ
		284	［注意事項］を変更しました。
		286	不要な記述を削除しました。
		288	説明を変更しました。
		289, 290	「半精度浮動小数点数」の説明を追加しました。
		357	［注意事項］を変更しました。
		666	「その他の命令」の説明を変更しました。
		695	［詳細説明］を変更しました。
		795- 798	［戻り値］を変更しました。
		831	説明を変更しました。
		840	説明を変更しました。
		885, 902	E0595001，E0595002，E0595003，E0595004，E0595005，E0595010， W0595020 を追加しました。
		889	F0563020 を追加しました。
		890	F0563115 を追加しました。
		892	M0560700 を追加しました。
		893	W0511179 の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.02	2016.12.01	893	W0511180，W0511183 を追加しました。
		896	W0523068 を追加しました。
		898	W0561014 を追加しました。
		904	「アセンブラにおける -Xcpu オプション指定」の説明を変更しました。
		904	「ビット操作命令の出力を制御する方法」を追加しました。
		905	「例外ハンドラ先頭への SYNCP 命令配置確認ツール」を追加しました。
		912	説明を変更しました。
1.03	2017.06.01	12	ライセンスの判定処理を変更しました。
		57, 58	次の MISRA-C:2012 ルールを追加しました。 12.5 13.2 13.5 17.5 17.8 21.13 21.15 21.16
		61	-Xuse_fp16 オプションの詳細説明を変更しました。
		23, 75, 100	-insert_dbtag_with_label オプションを追加しました。
		23, 75, 101, 258, 293, 294	-store_reg オプション，#pragma register_group 指令を追加しました。
		107, 142	-Xno_warning メッセージで制御できるメッセージ番号の範囲を記載しました。
		109	-Xasm_option オプションの使用例を変更しました。
		155, 156, 178, 179, 212, 214	-SHow オプションに relocation_attribute 指定を追加しました。
		161	-END_RECORD オプションを追加しました。
		170	-PADDING オプションの使用例を変更しました。
		171	-OVERRUN_FETCH オプションの詳細説明を変更しました。
		242	sizeof 演算子の型を追記しました。
		243, 257, 258, 264, 266, 268, 269, 276, 279, 288-290	#pragma 指令の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.03	2017.06.01	248	-Xansi オプション指定時に使用できない型の説明に，__fp16 型を追加しました。
		277, 278	組み込み関数 __mul32()，__mul32u() の説明を変更しました。
		364	.db 疑似命令の説明を変更しました。
		667, 668	cmovf.d，cmovf.s 命令使用時に出力するエラー番号の記載を修正しました。
		673-675	特殊シンボルを追加しました。
		678	half.h の記載を追加しました。
		678	リエントラント性の説明を変更しました。
		840-846	標準ライブラリ関数が使用するデータ用セクション，リエントラント性の一覧を追加しました。
		852	スタートアップ・ルーチン内の FPU の初期設定時の，FPU 有無の動的判定処理を追加しました。
		867	ROM，RAM セクション配置の注意事項を追加しました。
		878-916	次のメッセージの説明を追加しました。 C0511200，C0519996，C0519997，C0530001，C0530002，C0530003，C0530004，C0530005，C0530006，C0550802，C0550804，C0550805，C0550806，C0550808，C0551800，C0564001，E0511200，E0523069，E0523070，E0523071，E0523072，E0550270，E0550605，E0550606，E0550633，E0550637，E0550638，E0550639，E0550640，E0550641，E0550647，E0550649，E0551401，E0551402，E0551403，E0551406，E0551501，E0562430，E0562431，E0562432，E0562433，E0562434，E0562435，E0562436，E0562437，E0562438，E0562439，E0562450，E0562451，F0563103，M0536001，W0511184，W0511185，W0523120，W0550010，W0561015，W0561016，W0561017，W0561210
		905	次のメッセージを変更しました。 F0563102
		888-912	次のメッセージの説明を削除しました。 E0550204，E0550205，E0550206，E0550263，E0550264，E0550266，E0550268，E0550642，E0551228，E0551235，E0551301，E0551307，E0551312，E0551316，F0551602，W0550605，W0550606，W0550645，W0550647，W0550649
		888-913	次のメッセージの説明を変更しました。 E0550212，E0550236，E0550237，E0550239，E0550240，F0550511，F0550512，F0550537，F0550539，W0550013，W0550019，W0561004
		928, 929	記述例のコメント形式を変更しました。
1.04	2017.12.01	10, 12, 264, 310, 691	C99 規格に対応しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.04	2017.12.01	22, 49, 50, 56, 226, 264, 265, 267, 271	コンパイル・オプション -lang を追加しました。
		22, 49, 51, 62, 265, 267, 271	コンパイル・オプション -strict_std を追加しました。
		23, 76, 83	コンパイル・オプション -r4 を追加しました。
		23, 76, 104-106	コンパイル・オプション -control_flow_integrity を追加しました。
		23, 76, 107	コンパイル・オプション -pic を追加しました。
		23, 76, 108	コンパイル・オプション -pirod を追加しました。
		23, 76, 109	コンパイル・オプション -pid を追加しました。
		24, 114, 116	コンパイル・オプション -change_message を追加しました。
		58, 59	次の MISRA-C:2012 ルールを追加しました。 8.14 9.4 9.5 13.1 17.6 18.7 21.11 21.12
		66-68	コンパイル・オプション -O の［詳細説明］を変更しました。
		67, 68	コンパイル・オプション -Oinline_init を追加しました。
		81	コンパイル・オプション -Xreg_mode の［詳細説明］を変更しました。
		84	コンパイル・オプション -Xep の［詳細説明］を変更しました。
		87	コンパイル・オプション -Xfar_jump の［詳細説明］を変更しました。
		98	コンパイル・オプション -Xstack_protector の［詳細説明］を変更しました。
		100	コンパイル・オプション -Xsection に以下の引数を追加しました。 pcconst16，pcconst23

Rev.	発行日	改定内容	
		ページ	ポイント
1.04	2017.12.01	115	コンパイル・オプション -Xno_warning の指定可能範囲を変更しました。
		122, 143, 147	アセンブル・オプション -pic を追加しました。
		122, 143, 148	アセンブル・オプション -pirod を追加しました。
		122, 143, 149	アセンブル・オプション -pid を追加しました。
		144	アセンブル・オプション -Xreg_mode の［詳細説明］を変更しました。
		146	アセンブル・オプション -Xep の［詳細説明］を変更しました。
		155	アセンブル・オプション -Xno_warning の指定形式の説明を変更しました。
		159, 168, 170, 179, 184	リンク・オプション -FIX_RECORD_LENGTH_AND_ALIGN を追加しました。
		159, 168-170, 191	リンク・オプション -CFI を追加しました。
		159, 168-170, 192	リンク・オプション -CFI_ADD_Func を追加しました。
		159, 168-170, 193	リンク・オプション -CFI_IGNORE_Module を追加しました。
		169,	-nocompress オプションを追加しました。
		169, 170	-SHow オプションに all，total_size 指定を追加しました。
		170	-crc オプションを追加しました。
		183	リンク・オプション -BYte_count の使用可能条件を変更しました。
		196, 197, 230, 236	リンク・オプション -SHow の引数に cfi を追加しました。
		227	「#pragma 指定との関係」の説明を変更しました。
		256	「翻訳限界」の説明を変更しました。
		261-263	「あらかじめ定義されるマクロ名」の説明を変更しました。
		274	「コンパイラ生成シンボル」の説明を変更しました。
		275-279	「関数とデータのセクション割り当て」の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.04	2017.12.01	280, 283, 290, 291, 305	#pragma 指令の同時指定の説明を変更しました。
		283	「インライン展開を抑止したい関数」の説明を変更しました。
		303	「コア番号指定の形式」の説明を変更しました。
		305	「スタック破壊検出機能」の説明を変更しました。
		309, 310	「制御レジスタへの書き込みの検出，同期化処理挿入」の説明を変更しました。
		360, 361	.cseg 疑似命令に再配置属性 pctext，pcconst16，pcconst23，pcconst32 を追加しました。
		362-364	.dseg 疑似命令に再配置属性 sdata32，sbss32，edata32，ebss32 を追加しました。
		366	.org 疑似命令の［注意事項］を変更しました。
		374	.stack 疑似命令の説明を変更しました。
		422	「予約語」の説明を変更しました。
		423	「あらかじめ定義されたマクロ名」の説明を変更しました。
		556	jarl 命令の説明を変更しました。
		688	「特殊シンボル」の説明を変更しました。
		689, 847, 860	間接関数呼び出しチェック関数を追加しました。
		691	以下のヘッダ・ファイルを追加しました。 iso646.h，stdbool.h，stdint.h
		771, 776, 779, 783, 784	llabs，lldiv，atoll，strtoll，strtoull の［注意事項］を変更しました。
		833, 834	atan2f，atan2 の［戻り値］の説明を変更しました。
		850	__heap_chk_fail 関数の［使用例］の説明を変更しました。
		865	「スタック領域」の説明を変更しました。
		875, 876	「シンボル」の説明を変更しました。
		878-888	「PIC/PID 機能」の説明を追加しました。
		889	「関数呼び出し前後で内容が保証されないレジスタ（Caller-Save レジスタ）」の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.04	2017.12.01	905, 907, 911, 923-925, 933, 935	次のメッセージを追加しました。 E0520411，E0523087，E0550652，F0563003，F0563150，F0563431，F0563600，F0563601，F0563602，W0561142，W0561331
		915-917, 923, 933	次のメッセージを変更しました。 E0562311，E0562340，E0562417，F0563004，W0561130
		928	W0520062 の説明を変更しました。
		900-935	次のメッセージを削除しました。 E0511120，E0562017，E0562021，E0562112，E0562113，E0562142，E0562203，E0562220，E0562223，E0562323，E0562331，E0562402，E0562404，E0562405，E0562500，F0563115，F0563120，F0563311，F0563312，F0563313，F0563400，F0563420，M0560102，M0560103，M0560300，M0560510，M0560511，W0561015，W0561110，W0561180，W0561182，W0561183，W0561190，W0561192，W0561194，W0561321，W0561325，W0561430，W0561500，W0561501，W0561502
		939	「GP 相対アクセス」「EP 相対アクセス」の説明を変更しました。
		940, 941	「配置領域を変更する」の説明を変更しました。
1.05	2018.06.01	12	「ライセンスについて」の説明を変更しました。
		17	「指定形式」の説明を変更しました。
		22, 65, 76	コンパイル・オプション -library を追加しました。
		23, 77, 87	コンパイル・オプション -Xfxu を追加しました。
		23, 77, 94	コンパイル・オプション -use_recipf を追加しました。
		23, 77, 95	コンパイル・オプション -relaxed_math を追加しました。
		23, 77, 106	コンパイル・オプション -Xresbank_mode を追加しました。
		37	コンパイル・オプション -Xcommon の説明を変更しました。
		38	コンパイル・オプション -Xcpu の説明を変更しました。
		50	コンパイル・オプション -lang の説明を変更しました。
		89, 90	コンパイル・オプション -Xfar_jump の説明を変更しました。
		120	コンパイル・オプション -Xno_warning の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.05	2018.06.01	121	コンパイル・オプション -change_message の説明を変更しました。
		139	アセンブル・オプション -Xcommon の説明を変更しました。
		140	アセンブル・オプション -Xcpu の説明を変更しました。
		160	アセンブル・オプション -Xno_warning の説明を変更しました。
		167	リンク・オプション -Input の説明を変更しました。
		170	リンク・オプション -Binary の説明を変更しました。
		171	リンク・オプション -DEFine の説明を変更しました。
		174	リンク・オプション -FOrm の説明を変更しました。
		179	リンク・オプション -RECord の説明を変更しました。
		180	リンク・オプション -END_RECORD の説明を変更しました。
		182	リンク・オプション -OUtput の説明を変更しました。
		186	リンク・オプション -NOMessage の説明を変更しました。
		164, 173, 194, 195	リンク・オプション -CRc の説明を変更しました。
		197	リンク・オプション -CFI_ADD_Func の説明を変更しました。
		198	リンク・オプション -CFI_IGNORE_Module の説明を変更しました。
		204	リンク・オプション -STARt の説明を変更しました。
		209	リンク・オプション -CPu の説明を変更しました。
		218	リンク・オプション -REName の説明を変更しました。
		219	リンク・オプション -DELete の説明を変更しました。
		220	リンク・オプション -REPlace の説明を変更しました。
		223	リンク・オプション -CHange_message の説明を変更しました。
		262, 263	「整数型の限界値（stdint.h ファイル）」を追加しました。
		290	「マスカブル割り込みの受け付けの許可／禁止」の説明を変更しました。
		290-292, 297, 298	「割り込み／例外ハンドラの記述方法」の説明を変更しました。
		300-302	「組み込み関数」の説明を変更しました。
		302, 303	以下の組み込み関数を追加しました。 __dbcp(), __dbpush(), __dbtag(), __clipb(), __clipbu(), __cliph(), __cliphu(), __ldlbu(), __ldlhu(), __stcb(), __stch()
		312	「コア番号指定の形式」の説明を変更しました。
		688	cmpf.s の説明を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.05	2018.06.01	804-829	「数学関数」の説明を変更しました。
		835	_REL_tracestring を削除しました。
		804, 827-829, 848	以下の関数を追加しました。 fmax，fmaxf，fmin，fminf，copysign，copysignf
		890-921	次のメッセージを追加しました。 E0511133，E0511182，E0520069，E0520117，E0520175，E0520296， E0520393，E0520404，E0520469，E0520643，E0520644，E0520654， E0520655，E0520702，E0520749，E0520757，E0520765，E0520938， E0520965，E0520966，E0520967，E0520968，E0520969，E0520976， E0520977，E0521029，E0521030，E0521031，E0521037，E0521038， E0521039，E0521040，E0521045，E0521049，E0521051，E0521052， E0521144，E0521260，E0521261，E0521649，E0523026，E0523027， E0523048，E0523065，E0523067，E0523073，E0523090，E0523091， E0523118，E0523119，E0523122，E0523123，E0523124，E0523125， E0523126，E0523127，F0520163，F0520164，F0520182，F0520571， F0520642，F0520920，F0523073，W0511181，W0520055，W0520083， W0520140，W0520159，W0520220，W0520221，W0520222，W0520223， W0520240，W0520257，W0520513，W0520609，W0520660，W0520767， W0520819，W0520867，W0520940，W0520951，W0520966，W0520967， W0520968，W0521037，W0521039，W0521040，W0521046，W0521051， W0521057，W0521072，W0521222，W0521223，W0521224，W0521273， W0521297，W0523038，W0523116
		890-921	次のメッセージを削除しました。 E0511118，E0511136，E0511142，E0511148，E0511161，E0520001， E0520002，E0520005，E0520096，E0520123，E0520126，E0520157， E0520170，E0520255，E0520257，E0520259，E0520518，E0520544， E0520545，E0520606，E0520661，E0520668，E0520767，E0520940， E0520989，E0520992，E0520993，E0521066，E0521075，E0521076， E0521254，E0521255，E0521282，E0521420，E0523042，E0523059， F0512003，F0520016，F0520219，F0520583，F0520584，F0523071， W0520001，W0520014，W0520144，W0520171，W0520181，W0520185， W0520224，W0520225，W0520226，W0520514，W0520902，W0521396， W0523060，W0523120
		929	「コンパイラ・パッケージのバージョンについて」を追加しました。
1.06	2018.12.01	13	「最適化リンカ（rlink）」の説明を変更しました。
		22, 66, 78, 129, 143, 144, 168, 176, 178, 179, 206-211, 214, 226, 234	リンク時最適化のために，次のオプションを追加しました。 コンパイル・オプション -goptimize アセンブル・オプション -goptimize リンク・オプション -optimize/-nooptimize, -section_forbid, -absolute_forbid, -symbol_forbid

Rev.	発行日	改定内容	
		ページ	ポイント
1.06	2018.12.01	22, 49, 62	コンパイル・オプション -misra_intermodule を追加しました。
		58	次の MISRA-C:2012 ルールを追加しました。 8.5 8.6
		169, 221, 228	リンク・オプション -LIB_REName を追加しました。
		178, 179	リンク・オプション -FOrm の詳細説明を変更しました。
		227	リンク・オプション -REName の詳細説明を変更しました。
		236	リンク・オプション -Total_size の備考を変更しました。
		268-285	「基本言語仕様」の構成を見直しました。
		292-294	「拡張言語仕様」の構成を見直しました。
		331-333	「制御レジスタへの書き込みの検出，同期化処理挿入」の説明を変更しました。
		455-492	「命令の説明」を削除し，「アセンブリ言語の拡張」を追加しました。
		603-631	次のライブラリ関数を追加しました。 acosl, asinl, atanl, atan2l, cosl, sinl, tanl, coshl, sinhl, tanhl, expl, frexpl, ldexpl, logl, log10l, modfl, fabsl, powl, sqrtl, ceill, floorl, round, roundf, roundl, lround, lroundf, lroundl, llround, llroundf, llroundl, trunc, truncf, truncl, fmodl, copysignl, fmaxl, fminl
		621, 628	pow 関数群，fmod 関数群の詳細説明を変更しました。
		658	「RAM セクションの初期化」の説明を変更しました。
		668, 669	「ROM 化」を削除し，「ROM イメージの作成」を追加しました。
		707, 714, 726	次のメッセージの説明を変更しました。 E0562212, E0562320, F0563004, W0561301
		710, 717, 719	次のメッセージを追加しました。 E0562600, M0560004, M0560005, M0560100, W0520070
		724	W0561101 のメッセージを変更しました。
		737	「RAM で実行する」を削除し，「RAM で特定のルーチンを実行する」を追加しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.07	2019.11.01	10	「著作権について」の説明を変更しました。
		17	「操作例」の説明を変更しました。
		57, 58	次の MISRA-C:2012 ルールを追加しました。 8.13 14.2 14.3
		87	コンパイル・オプション -Xfloat の「省略時解釈」と［詳細説明］を変更しました。
		22, 78, 94	コンパイル・オプション -relaxed_math の記述場所を変更しました。
		94	コンパイル・オプション -relaxed_math の説明を全体的に変更しました。
		22, 78, 97	コンパイル・オプション -approximate を追加しました。
		122	コンパイル・オプション -Xno_warning の［詳細説明］を変更しました。
		167, 170, 177, 180, 227	リンク・オプション -ALLOW_DUPLICATE_MODULE_NAME を追加しました。
		193	リンク・オプション -BYte_count の［備考］を変更しました。
		229	リンク・オプション -LIB_RENname の［指定形式］と［使用例］を変更しました。
		287	「C99 の処理系定義」の次の項目を変更しました。 (109)
		293, 294	次の表のヘッダーを変更しました。 表 4.2, 表 4.3, 表 4.4
		302	表 4.8 のマクロ名 __MULTI_LEVEL__ の説明を変更しました。
		310, 311	#pragma section 指令の効果についての説明を変更しました。
		350	「C ソースの修正」の説明を変更しました。
		472, 他	「アセンブリ言語の拡張」の全体にわたる表にヘッダー行を追記しました。
		518, 519	表 7.1 の「提供ライブラリ」列を変更しました。
		520	表 7.2 から以下の行を削除しました。 inttypes.h
		669, 他	「データ用セクションの使用，リエントラント性一覧」の表を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.08	2020.11.01	表紙	対象の CPU コアを追記しました。
		15	表 2.1 にツール使用情報ファイルを追加しました。
		22, 78, 107	コンパイル・オプション -stuff を追加しました。
		46	コンパイル・オプション -Xpreinclude の［詳細説明］を変更しました。
		68	最適化項目 align を追加しました。
		98	コンパイル・オプション -Xunordered_cmpf の［詳細説明］を変更しました。
		170, 223, 239	リンク・オプション -VERBOSE を追加しました。
		303	「予約語」の説明を変更しました。
		310, 311	#pragma section 指令の書式説明を変更しました。
		311, 312	表 4.16 を表 4.16 ～表 4.18 に分割しました。
		330- 332	次の表の関数宣言を ANSI-C 形式に変更しました 表 4.20，表 4.21
		407	.section 疑似命令の［指定形式］,［用途］を変更しました。
		726	次のメッセージを追加しました。 E0550271

Rev.	発行日	改定内容	
		ページ	ポイント
1.09	2021.11.01	10	「概説」の説明を変更しました。
		16	「コマンド・ラインでの操作方法」の「指定形式」の説明を変更しました。
		22, 78, 80	コンパイル・オプション -misalign を追加しました。
		45	コンパイル・オプション -I の［詳細説明］を変更しました。
		67, 68	次の最適化項目に関する説明を変更しました。 tail_call，align
		148	アセンブル・オプション -D の［詳細説明］を変更しました。
		151	アセンブル・オプション -I の［詳細説明］を変更しました。
		307	「#pragma 指令」の説明を変更しました。
		322	「割り込み仕様」の表に注記を追加しました。
		324- 326, 328	例外ハンドラの復帰処理内の ldsr 命令の順序を変更しました。
		356, 400	識別子に $ を追加しました。
		414, 420	.dbl_size 疑似命令を追加しました。
		431, 434	.extern 疑似命令の説明を変更しました。
		522	「特殊シンボル」の説明を変更しました。
		714	「他言語で定義された変数の参照」において，参照例の誤記を訂正しました。
		718, 743, 745, 751	次のメッセージを変更しました。 E0511178，F0563430，W0511180，W0511185，W0561016，W0561017

Rev.	発行日	改定内容	
		ページ	ポイント
1.10	2022.12.01	66	コンパイル・オプション -O の「省略時解釈」を変更しました。
		101	コンパイル・オプション -Xpatch の［詳細説明］を変更しました。
		148	アセンブル・オプション -D の［詳細説明］を変更しました。
		170, 180, 201	リンク・オプション -RESERVE_PREFETCH_AREA を追加しました。
		202, 203	リンク・オプション -CRc の［詳細説明］,［使用例］を変更しました。
		525, 526	表 7.2 を追加しました。
		736, 744	次のメッセージを追加しました。 E0562326，F0563115
		740, 753	次のメッセージを変更しました。 F0520571，W0561017
1.11	2023.12.01	16	「コマンド・ラインでの操作方法」の説明を変更しました
		20, 39	コンパイル・オプション -P の説明，［詳細説明］,［使用例］を変更しました。
		61	コンパイル・オプション -misra_intermodule の［備考］を変更しました。
		101	コンパイル・オプション -Xpatch の［詳細説明］を変更しました。
		170, 211, 216	リンク・オプション -ALLOW_OPTIMIZE_ENTRY_BLOCK を追加しました。
		188	リンク・オプション -ROm の［詳細説明］を変更しました。
		201	リンク・オプション -RESERVE_PREFETCH_AREA の［備考］の誤記を修正しました。
		203	リンク・オプション -CRc の［詳細説明］を変更しました。
		526	表 7.2 を変更しました。
		719, 741	次のメッセージを変更しました。 C0519996，F0520003
		719, 736	次のメッセージを追加しました。 C0520000，C0529000，E0562114
1.12	2024.12.01	16	「コマンド・ラインでの操作方法」の「指定形式」の説明を変更しました。
		44	コンパイル・オプション -D の［詳細説明］を変更しました。
		58	コンパイル・オプション -Xmisra2012 の［詳細説明］を変更しました。
		67	コンパイル・オプション -O の［詳細説明］を変更しました。
		89	コンパイル・オプション -Xfloat の［詳細説明］を変更しました。
		90	コンパイル・オプション -Xfxu の［詳細説明］を変更しました。
		204	リンク・オプション -CRc の［詳細説明］を変更しました。

Rev.	発行日	改定内容	
		ページ	ポイント
1.12	2024.12.01	256	シンボル種別の説明を変更しました。
		288	「C99 の処理系定義」(58) の説明を変更しました。
		404	表 5.6 を変更しました。
		411	.section 疑似命令の［用途］を変更しました。
		416	.equ 疑似命令の［指定形式］,［機能］,［詳細説明］を変更しました。
		434, 438	.weak 疑似命令を追加しました。
		737	次のメッセージを追加しました。 E0551407

CC-RH ユーザーズマニュアル

発行年月日　2015年　9月 14日　Rev.1.00
　　2024年 12月　1日　Rev.1.12
発行　　ルネサス エレクトロニクス株式会社
　　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

CC-RH

RENESAS

ルネサス エレクトロニクス株式会社

R20UT3516JJ0112