User's Manual

RENESAS

SuperH™ RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタ

コンパイラパッケージ V.9.04　　ユーザーズマニュアル

ルネサスマイクロコンピュータ開発環境システム

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

ルネサス エレクトロニクス
www.renesas.com

Rev.1.02　2022.03

ご注意書き

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

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

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

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

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

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

はじめに

本マニュアルは、「SuperH RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタ」の使用方法を述べたものです。

本製品はC言語、C++言語、DSP-C言語*1およびアセンブリ言語で記述したソースプログラムを、SuperH RISC engine用オブジェクトプログラムおよびロードモジュールに変換するソフトウェアシステムです。

ご使用になる前に、本マニュアルを良く読んで理解してください。

■表記上の注意事項

本マニュアルの説明の中で用いられる記号は、次の意味を示しています。

〈　〉	この記号で囲まれた内容を指定することを示します。
［　］	省略してもよい項目を示します。
．．．	直前の項目を1回以上指定することを示します。
△	1個以上の空白を示します。
｜	|で区切られた項目を選択できることを示します。

本マニュアルはIBM PC*2互換機およびその互換機上で動作するMicrosoft® Windows® 2000、Windows® XP、Windows® Vista、またはWindows® 7*3に対応するように書かれています。

【注】*1　DSP-C言語は、オランダのACE社(Associated Compiler Experts)がDSPコンパイラ実装時に必要となる言語拡張を検討し、1998年にISO標準委員会に提案した仕様です。

*2　IBM PCは、米国 International Business Machines Corporationの登録商標です。

*3　Microsoft®, Windows®は、米国Microsoft Corporationの米国及びその他の国における登録商標または商標です。

※　その他、本マニュアルの文中に使われている会社名および製品名、システム名などは各社の登録商標または商標です。

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。

目次

1. 概要 1
1.1 プログラムの開発手順 1
1.2 コンパイラの概要 2
1.3 アセンブラの概要 2
1.4 最適化リンケージエディタの概要 3
1.5 プレリンカの概要 3
1.6 標準ライブラリ構築ツールの概要 3
1.7 CallWalkerの概要 3
2. C/C++コンパイラ操作方法 5
2.1 オプション指定規則 5
2.2 オプション解説 5
2.2.1 ソースオプション 5
2.2.2 オブジェクトオプション 9
2.2.3 リストオプション 22
2.2.4 最適化オプション 25
2.2.5 その他オプション 44
2.2.6 マイコンオプション 56
2.2.7 残りのオプション 63
3. アセンブラ操作方法 67
3.1 オプション指定規則 67
3.2 オプション解説 67
3.2.1 ソースオプション 67
3.2.2 オブジェクトオプション 71
3.2.3 リストオプション 75
3.2.4 その他オプション 81

3.2.5 マイコンオプション 87
3.2.6 残りのオプション 91

4. 最適化リンケージエディタ操作方法 97

4.1 オプション指定規則 97
4.1.1 コマンドラインの形式 97
4.1.2 サブコマンドファイルの形式 97
4.2 オプション解説 98
4.2.1 入力オプション 98
4.2.2 出力オプション 102
4.2.3 リストオプション 121
4.2.4 最適化オプション 124
4.2.5 セクションオプション 130
4.2.6 ベリファイオプション 133
4.2.7 その他オプション 137
4.2.8 サブコマンドファイルオプション 145
4.2.9 マイコンオプション 146
4.2.10 残りのオプション 147

5. 標準ライブラリ構築ツール操作方法 149

5.1 オプション指定規則 149
5.1.1 コマンドラインの形式 149
5.2 オプション解説 149
5.2.1 追加オプション 149
5.2.2 指定不可オプション 152
5.2.3 オプション指定時の注意事項 153

6. CallWalker 操作方法 155

6.1 CallWalkerの起動 155

7. 環境変数 157

7.1 環境変数一覧 157

7.2 プリデファインドマクロ 159

8. ファイル仕様 161

8.1 ファイル名の付け方 161
8.2 コンパイルリストの参照方法 162
8.2.1 コンパイルリストの構成 162
8.2.2 ソースリスト情報 162
8.2.3 オブジェクト情報 165
8.2.4 統計情報 166
8.2.5 コマンド指定情報 167
8.3 アセンブルリストの参照方法 168
8.3.1 アセンブルリストの構成 168
8.3.2 ソースリスト情報 169
8.3.3 クロスリファレンスリスト 170
8.3.4 セクション情報リスト 171
8.4 リンケージリストの参照方法 172
8.4.1 リンケージリストの構成 172
8.4.2 オプション情報 173
8.4.3 エラー情報 173
8.4.4 リンケージマップ情報 174
8.4.5 シンボル情報 175
8.4.6 シンボル削除最適化情報 176
8.4.7 クロスリファレンス情報 177
8.4.8 合計セクションサイズ 178
8.4.9 ベクタ情報 178
8.4.10 CRC 情報 179
8.5 ライブラリリストの参照方法 180
8.5.1 ライブラリリストの構成 180
8.5.2 オプション情報 180
8.5.3 エラー情報 181
8.5.4 ライブラリ情報 181
8.5.5 ライブラリ内モジュール、セクション、シンボル情報 182

9. プログラミング 183
9.1 プログラムの構造 183
9.1.1 セクション 183
9.1.2 C/C++プログラムのセクション 183
9.1.3 アセンブリプログラムのセクション 186
9.1.4 セクションの結合 187
9.2 初期設定プログラムの作成 190
9.2.1 メモリ領域の割り付け 191
9.2.2 実行環境の設定 198
9.3 C/C++プログラムとアセンブリプログラムとの結合 229
9.3.1 外部名の相互参照方法 229
9.3.2 関数呼び出し規約 231
9.3.3 引数割り付けの具体例 239
9.3.4 レジスタとスタック領域の使用法 241
9.4 プログラム作成上の注意事項 242
9.4.1 コーディング上の注意事項 242
9.4.2 C プログラムを C++コンパイラでコンパイルするときの注意事項 245
9.4.3 プログラム開発上の注意事項 246

10. C/C++言語仕様 247
10.1 言語仕様 247
10.1.1 コンパイラの仕様 247
10.1.2 データの内部表現 252
10.1.3 浮動小数点型の仕様 262
10.1.4 演算子の評価順序 268
10.2 DSP-C仕様 269
10.2.1 固定小数点型 269
10.2.2 型修飾子 270
10.2.3 定数 271
10.2.4 型変換 272
10.2.5 算術変換 273

10.2.6 ポインタ変換 274
10.2.7 演算子 274
10.2.8 ライブラリ 275
10.3 拡張機能 277
10.3.1 #pragma 277
10.3.2 セクションアドレス演算子 305
10.3.3 組み込み関数 306
10.4 C/C++ライブラリ 367
10.4.1 標準 C ライブラリ 367
10.4.2 EC++クラスライブラリ 496
10.4.3 リエントラントライブラリ 566
10.4.4 未サポートライブラリ 568
10.4.5 DSP ライブラリ 569

11. アセンブラ言語仕様 619

11.1 プログラムの要素 619
11.1.1 ソースステートメント 619
11.1.2 キーワード 621
11.1.3 シンボル 622
11.1.4 定数 624
11.1.5 ロケーションカウンタ 632
11.1.6 式 633
11.1.7 文字列 639
11.1.8 ローカルラベル 639
11.2 実行命令 641
11.2.1 実行命令の概要 641
11.2.2 実行命令に関する注意事項 645
11.3 DSP命令 666
11.3.1 プログラムの要素 666
11.3.2 DSP 命令 669
11.4 アセンブラ制御命令 676
11.5 ファイルインクルード機能 721

11.6 条件つきアセンブリ機能 723
11.6.1 条件つきアセンブリ機能の概要 723
11.6.2 条件つきアセンブリ機能に関する制御文 727
11.7 マクロ機能 739
11.7.1 マクロ機能の概要 739
11.7.2 マクロ機能に関する制御文 741
11.7.3 マクロ本体 744
11.7.4 マクロコール 747
11.7.5 文字列操作関数 749
11.8 リテラルプール自動生成機能 752
11.8.1 リテラルプール自動生成機能の概要 752
11.8.2 リテラルプール自動生成機能に関する拡張命令 752
11.8.3 リテラルプール自動生成機能のサイズモード 753
11.8.4 リテラルプールの出力 754
11.8.5 リテラルの共有 756
11.8.6 リテラルプール出力の抑止 757
11.8.7 リテラルプール自動生成に関する注意事項 758
11.9 リピートループ命令自動生成機能 760
11.9.1 リピートループ命令自動生成機能の概要 760
11.9.2 リピートループ命令自動生成機能に関する拡張命令 761
11.9.3 REPEAT の記述方法 761
11.9.4 コーディング例 762
11.9.5 REPEAT 拡張命令に関する注意事項 764
11.10 拡張リピートループ命令自動生成機能 766
11.10.1 拡張リピートループ命令自動生成機能の概要 766
11.10.2 拡張リピートループ命令自動生成機能に関する拡張命令 766
11.10.3 EREPEAT の記述方法 767
11.10.4 コーディング例 767
11.10.5 拡張 REPEAT 命令(EREPEAT 命令)に関する注意事項 769

12. コンパイラのエラーメッセージ 771

12.1 エラー形式とエラーレベル 771

12.2 メッセージ一覧 771
12.3 C標準ライブラリ関数のエラーメッセージ 841

13. アセンブラのエラーメッセージ 845

13.1 エラー形式とエラーレベル 845
13.2 メッセージ一覧 845

14. 最適化リンケージエディタのエラーメッセージ 865

14.1 エラー形式とエラーレベル 865
14.2 エラーの返値 865
14.3 メッセージ一覧 866

15. 翻訳限界 887

15.1 コンパイラの翻訳限界 887
15.2 アセンブラの翻訳限界 888

16. バージョンアップにおける注意事項 889

16.1 バージョンアップ時の注意事項 889
16.1.1 プログラムの動作保証 889
16.1.2 旧バージョンとの互換性 890
16.1.3 旧バージョンのオブジェクトとの互換性 891
16.1.4 コマンドラインインタフェース 892
16.1.5 提供内容 894
16.1.6 リストファイル仕様 894
16.2 追加・改善内容 895
16.2.1 共通の追加・改善(パッケージ Ver.6) 895
16.2.2 コンパイラの追加・改善機能 895
16.2.3 アセンブラの追加・改善機能 903
16.2.4 最適化リンケージエディタの追加・改善機能 903

17. 付録 909

17.1 モトローラS形式、インテルHEX形式ファイル 909
17.1.1 モトローラ S 形式ファイル 909

17.1.2 インテル HEX 形式ファイル 911
17.2 ASCIIコード一覧表.....913

1. 概要

1.1 プログラムの開発手順

プログラムの開発手順を**図 1.1**に示します。網掛け部分は、**本コンパイラパッケージ**で提供するソフトウェアを示します。

本マニュアルでは、C/C++コンパイラ、アセンブラ、最適化リンケージエディタ、標準ライブラリ構築ツール、CallWalker について説明します。

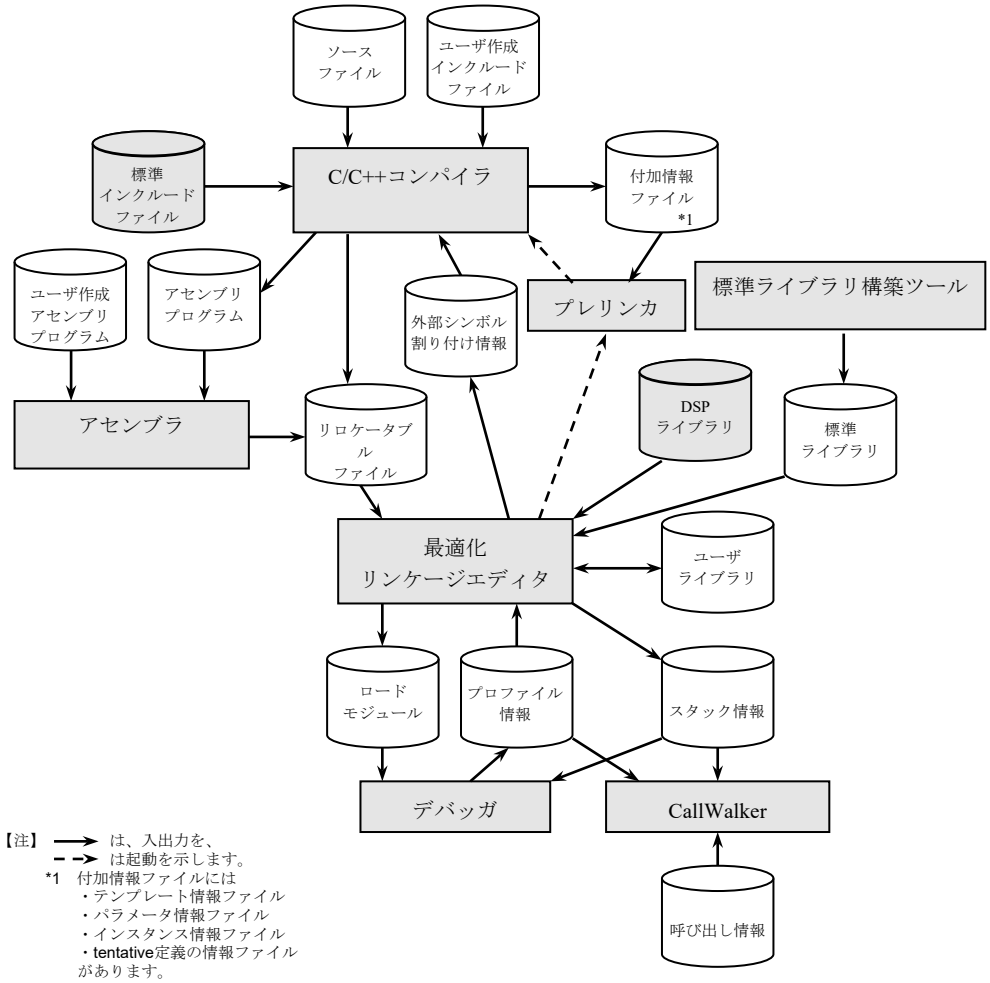


図 1.1 プログラムの開発手順

以下、C/C++コンパイラ、アセンブラ、最適化リンケージエディタ、プレリンカ、標準ライブラリ構築ツール、CallWalker の概要を述べます。

1.2 コンパイラの概要

「C/C++コンパイラ」は、C 言語および C++言語で記述したソースプログラムを入力し、リロケータブルオブジェクトプログラムまたはアセンブリソースプログラムを出力します。

本コンパイラには次の特長があります。

(1) 機器組み込み用として ROM 化可能なオブジェクトプログラムを生成します。
(2) オブジェクトプログラムの実行速度向上や ROM/RAM サイズ縮小のための最適化機能をサポートしています。
(3) プログラム記述言語として、C 言語、C++言語をサポートしています。
(4) C/C++言語でサポートしていない割り込み関数やシステム命令記述など、組み込み用プログラム作成に必要な機能を、拡張機能としてサポートしています。
(5) デバッガによる C/C++ソースレベルデバッグを行うためのデバッグ情報出力を指定できます。
(6) アセンブリソースプログラムまたはリロケータブルオブジェクトプログラムを選択して出力することができます。
(7) 最適化リンケージエディタによるリンク時最適化を行うためのモジュール間最適化情報出力を指定できます。

1.3 アセンブラの概要

「アセンブラ」は、アセンブリ言語で記述したソースプログラムを入力し、リロケータブルオブジェクトプログラムを出力します。

本アセンブラには次の特長があります。

(1) 次に示すプリプロセッサ機能により、効率よくソースプログラムを記述できます。
- ファイルインクルード機能
- 条件付アセンブリ機能
- マクロ機能

(2) 実行命令、アセンブラ制御命令のニーモニック(名称)は、IEEE-694 仕様で規定された命名規則に準拠し、統一された体系となっています。

1.4 最適化リンケージエディタの概要

「最適化リンケージエディタ」は、コンパイラおよびアセンブラが出力した複数のオブジェクトプログラムを入力し、ロードモジュールまたはライブラリファイルを出力します。

本最適化リンケージエディタには次の特長があります。

(1) コンパイラでは最適化できないメモリ配置や関数の呼び出し関係に依存した最適化を、オブジェクトプログラムをまたがって実行します。

(2) 以下の5種類のロードモジュールを選択出力できます。
- リロケータブル ELF 形式
- アブソリュート ELF 形式
- モトローラ S 形式
- インテル(拡張)HEX 形式
- バイナリ形式

(3) ライブラリファイルを作成・編集できます。

(4) シンボル参照回数リストを出力できます。

(5) ライブラリ、ロードモジュールファイルのデバッグ情報を削除できます。

(6) CallWalkerで使用するスタック情報ファイルの出力を指定できます。

1.5 プレリンカの概要

「プレリンカ」は、最適化リンケージエディタから呼ばれ、C++プログラムの template, typeid, dynamic_cast 機能を使用している場合に、コンパイラを起動して必要なオブジェクトファイルを生成します。

通常はプレリンカを意識する必要はありませんが、C++プログラムの template, typeid, dynamic_cast 機能を使用していない場合、最適化リンケージエディタの noprelink オプションを指定してプレリンカの起動を抑止することにより、リンク速度を向上できます。

1.6 標準ライブラリ構築ツールの概要

「標準ライブラリ構築ツール」は、コンパイラが提供する標準ライブラリファイルをユーザ指定オプションで構築するソフトウェアシステムです。

本コンパイラが提供する標準ライブラリ関数には、C ライブラリ関数群、組み込み向け C++クラスライブラリ関数群、ランタイムライブラリ群(プログラムを実行する上で必要な算術演算)があります。ソースプログラム上でライブラリ関数を使用していない場合でも、ランタイムライブラリが必要な場合がありますので注意してください。

1.7 CallWalker の概要

「CallWalker」は、最適化リンケージエディタが出力したスタック情報ファイルを入力し、C/C++プログラムのスタック使用量を算出します。

2. C/C++コンパイラ操作方法

2.1 オプション指定規則

コンパイラを起動するコマンドラインの形式は以下の通りです。

```
shc[△<オプション>...][△<ファイル名>[△<オプション>...]...]
  <オプション>：-<オプション>[=<サブオプション>][,...]
```

2.2 オプション解説

コマンドライン形式の場合は、英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。
また、統合開発環境の対応するダイアログメニューを、タブ名<カテゴリ名>[項目] ...で示します。
オプションの順序は、統合開発環境のタブに対応しています。
なお、最適化に関わるオプションは、条件により、適用されない場合もあります。当該最適化が適用されたかどうかは出力コードで確認してください。

2.2.1 ソースオプション

表 2.1 ソースカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	インクルードファイルフォルダ	Include = <パス名>[,...]	コンパイラ<ソース> [オプション項目 :] [インクルードファイルディレクトリ]	インクルードファイルパス名を指定
2	デフォルトインクルードファイル	PREInclude = <ファイル名>[,...]	コンパイラ<ソース> [オプション項目 :] [デフォルトインクルードファイル]	指定したファイルをコンパイル単位の先頭にインクルード
3	マクロ名の定義	DEFine = <sub>[,...] <sub> : <マクロ名> [= <文字列>]	コンパイラ<ソース> [オプション項目 :] [マクロ定義]	<文字列>を<マクロ名>として定義
4	インフォメーションメッセージ	MEssage NOMEssage [= <エラー番号> [- <エラー番号>][,...]]	コンパイラ<ソース> [オプション項目 :] [インフォメーションメッセージ] [インフォメーションレベルメッセージの表示]	出力あり 出力なし
5	ファイル間インライン展開フォルダ指定	FILE_INLINE_PATH= <パス名>[,...]	コンパイラ<ソース> [オプション項目 :] [ファイル間インライン展開ディレクトリ]	ファイル間インライン展開対象のファイルの取り込み先パス名指定
6	メッセージレベル	CHAnge_message =<sub>[,...] <sub>:<level> [=<n>[-m],...] <level>:{Information | Warning | Error }	コンパイラ<ソース> [メッセージレベル :]	メッセージレベルの変更

インクルードファイルフォルダ

Include

コンパイラ<ソース>[オプション項目 :][インクルードファイルディレクトリ]

書　式　　Include = <パス名>[,...]

説　明　　インクルードファイルの存在するパス名を指定します。
パス名が複数ある場合にはカンマ(,)で区切って指定することができます。
システムインクルードファイルの検索は、include オプション指定フォルダ、環境変数 SHC_INC 指定フォルダ、環境変数 SHC_LIB 指定フォルダの順序で行います。
ユーザインクルードファイルの検索は、カレントフォルダ、include オプション指定フォルダ、環境変数 SHC_INC 指定フォルダ、環境変数 SHC_LIB 指定フォルダの順序で行います。

列　　shc -include=c:¥usr¥inc,c:¥usr¥shc test.c
フォルダ c:¥usr¥inc と c:¥usr¥shc をインクルードファイルパスとして検索します。

デフォルトインクルードファイル

PREInclude

コンパイラ<ソース>[オプション項目 :][デフォルトインクルードファイル]

書　式　　PREInclude = <ファイル名>[,...]

説　明　　指定したファイルの内容をコンパイル単位の先頭に取り込みます。ファイル名が複数ある場合にはカンマ(,)で区切って指定することができます。

列　　shc -preinclude=a.h test.c

```
<test.c>の内容
    int a;
    main() { … }

コンパイル時解釈
    #include "a.h"
    int a;
    main() { … }
```

マクロ名定義

DEFine

コンパイラ<ソース>[オプション項目 :][マクロ定義]

書　式　　DEFine = <sub>[,...]
　　<sub>:<マクロ名> [= <文字列>]

説　明　　ソースファイル内で記述する#define と同等の効果を得ます。
<マクロ名>=<文字列>と記述することで<文字列>をマクロ名として定義できます。
サブオプションに<マクロ名>を単独で指定した場合は、そのマクロ名が定義されたものと仮定します。<文字列>には、名前または整定数を記述することができます。

インフォメーションメッセージ

MEssage
NOMEssage

コンパイラ<ソース>[オプション項目 :][インフォメーションメッセージ]
[インフォメーションレベルメッセージの表示]

書　式　　MEssage
NOMEssage [= <エラー番号> [- <エラー番号>][,...]]

説　明　　インフォメーションレベルメッセージの出力有無を指定します。
message オプションを指定した場合、インフォメーションレベルメッセージを出力します。
nomessage オプションを指定した場合、インフォメーションレベルメッセージの出力を抑止します。また、サブオプションでエラー番号を指定すると、指定したインフォメーションレベルメッセージの出力だけを抑止します。<エラー番号>-<エラー番号>のようにハイフン(-)で抑止するエラー番号の範囲を指定することもできます。
本オプションの省略時解釈は、nomessage です。

例　　shc -message test.c
インフォメーションレベルメッセージを出力します。

ファイル間インライン展開フォルダ指定

FILE_INLINE_PATH

コンパイラ<ソース>[オプション項目 :][ファイル間インライン展開ディレクトリ]

書　式　　FILE_INLINE_PATH = <パス名>[,...]

説　明　　ファイル間インライン展開対象となるファイルの存在するパス名を指定します。
パス名が複数ある場合にはカンマ(,)で区切って指定することができます。
ファイル間インライン展開対象ファイルの検索は、file_inline_path オプション指定フォルダ、カレントフォルダの順序で行います。

例　　shc -file_inline_path=c:¥usr¥file -file_inline=test2.c test.c
フォルダ c:¥usr¥file をファイル間インライン展開指定フォルダとし、file_inline オプションで指定された test2.c を検索します。

メッセージレベル

CHAnge_message

コンパイラ<ソース>[メッセージレベル :]

書　式　　CHAnge_message = <sub>[,…]
<sub>　　: <エラーレベル>[=<エラー番号>[- <エラー番号>][,…]]
<エラーレベル>　　: { Information | Warning | Error }

説　明　　インフォメーション、ウォーニングのメッセージレベルを変更します。

例　　change_message=information=エラー番号
ウォーニングレベルの指定エラー番号のみインフォメーションレベルに変更します。

change_message=warning=エラー番号
インフォメーションレベルの指定エラー番号のみウォーニングレベルに変更します。

change_message=error=エラー番号
インフォメーション、ウォーニングレベルの指定エラー番号のみエラーレベルに変更します。

change_message=information
全てのウォーニングメッセージをインフォメーションレベルに変更します。

change_message=warninge
全てのインフォメーションメッセージをウォーニングレベルに変更します。

change_message=error
全てのインフォメーション、ウォーニングメッセージをエラーレベルに変更します。

備　考　　インフォメーションレベルに変更したメッセージについては、nomessage オプション指定により出力を抑止できます。

2.2.2 オブジェクトオプション

表 2.2 オブジェクトカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	プリプロセッサ展開	PREProcessor [= <ファイル名>]	コンパイラ <オブジェクト> [出力ファイル形式 :] [プリプロセッサ展開プログラム]	プリプロセッサ展開後のソースプログラムを出力
		NOLINe	[プリプロセッサ展開プログラム(#line 出力抑止)]	プリプロセッサ展開時に#line の出力を抑止
2	オブジェクト形式	Code =	コンパイラ <オブジェクト> [出力ファイル形式 :]	
		{ Machinecode	[機械語プログラム]	機械語プログラムを出力
		Asmcode }	[アセンブリプログラム]	アセンブリプログラムを出力
3	デバッグ情報	DEBug	コンパイラ <オブジェクト> [デバッグ情報出力]	出力あり
		NODEBug		出力なし
4	セクション名	SEction = <sub>[,…] <sub>:{	コンパイラ <オブジェクト> [詳細…]	
		Program=<セクション名>	[セクション :]	プログラム領域のセクション名
		Const = <セクション名>	[プログラム領域 (P)]	定数領域のセクション名
		Data = <セクション名>	[定数領域 (C)]	初期化データ領域のセクション名
		Bss = <セクション名> }	[初期化データ領域 (D)] [未初期化データ領域 (B)]	未初期化データ領域のセクション名
5	文字列出力領域	STring = { Const	コンパイラ <オブジェクト> [詳細…]	定数領域へ出力
		Data }	[文字列データ格納 :]	初期化データ領域へ出力
6	オブジェクトファイル名指定	OBjectfile = <ファイル名>	コンパイラ <オブジェクト> [オブジェクト出力ディレクトリ :]	指定したファイル名のオブジェクトファイルを出力
7	テンプレートインスタンス生成機能	Template = { None	コンパイラ <オブジェクト> [詳細…]	インスタンスを生成しない
		Static		参照されたものだけ内部リンケージとして生成
		Used	[テンプレート生成 :]	参照されたものだけ外部リンケージとして生成
		ALl		宣言、参照されたものを生成
		AUto }		リンク時に生成
8	ABS16/20/28/32 宣言	<ABS>=<sub>[,…] <ABS>: { ABs16 | ABS20 | ABS28 | ABS32} <sub>: { Program | Const | Data | Bss | Run | All }	コンパイラ <オブジェクト> [詳細 2…] [アドレス宣言 :]	指定したセクションに属するラベルアドレスもしくはランタイムライブラリを配置するメモリ空間を指定

9	除算方式選択 (SH-1 以外)	DIvision = Cpu = { Inline | Runtime }	コンパイラ <オブジェクト> [詳細…] [除算方式選択 :]	マイコンの除算命令を使用 乗算に変換しインライン展開 ランタイムライブラリ呼び出し
10	浮動小数点 レジスタ退避・ 回復抑止 (SH-2E, SH2A-FPU, SH-4,SH-4A)	IFUnc	コンパイラ <オブジェクト> [詳細…] [浮動小数点レジスタ退避・回復抑止]	浮動小数点レジスタの退避・回復を抑止
11	ラベルの 16/32 バイト整合	ALIGN16 ALIGN32 NOALign	コンパイラ <オブジェクト> [詳細…] [分岐先アドレスの境界調整 :]	プログラムセクション内ラベルで、無条件分岐命令の後で最初に出現するラベルをすべて 16 バイト整合する プログラムセクション内ラベルで、無条件分岐命令の後で最初に出現するラベルをすべて 32 バイト整合する ラベルを 16/32 バイト整合しない
12	TBR 相対関数 呼び出し指定 (SH-2A, SH2A-FPU)	TBR [=<セクション名>]	コンパイラ <オブジェクト> [詳細 2…] [TBR 指定 :]	関数呼び出しを TBR 相対で行う
13	未初期化変数 の出力順	BSs_order = { DEClaration | DEFinition }	コンパイラ <オブジェクト> [詳細 2…] [未初期化変数の出力順 :]	宣言順に出力 定義順に出力
14	変数の配置 指定	STUff [= {Bss | Data | Const} [,…]] NOSTuff	コンパイラ <オブジェクト> [詳細 2…] [変数の配置指定 :]	変数サイズに応じたセクションに配置する 変数サイズに関わらず同じセクションに配置する
15	$G0,$G1 セクションの変数の配置指定	STUFF_GBR	コンパイラ <オブジェクト> [詳細 2…] [$G0/$G1 セクション内変数の配置 :]	変数サイズに応じたセクションに配置($G0,$G1 セクション)
16	分岐先アドレスの境界調整	ALIGN4 ={ ALL | LOOP | INMOSTLOOP }	コンパイラ <オブジェクト> [詳細…] [分岐先アドレスの境界調整 :]	分岐先アドレスの境界調整 全ての分岐先 全てのループ先頭アドレス 全ての最内側ループ先頭アドレス
17	const volatile 属性変数の 割り付け	CONST_VOLATILE = { DATA | CONST }	コンパイラ <オブジェクト> [詳細…] [const volatile 修飾変数:]	const volatile 割り付け先選択 初期化データ領域 定数領域

プリプロセッサ展開

PREProcessor
NOLINe

コンパイラ<オブジェクト>[出力ファイル形式 :][プリプロセッサ展開プログラム]
[プリプロセッサ展開プログラム(#line 出力抑止)]

書　式　　PREProcessor [= <ファイル名>]
NOLINe

説　明　　プリプロセッサ展開後のソースプログラムを出力します。
<ファイル名>を指定しない場合は、ソースファイル名と同じファイル名で拡張子が「p」(入力ソースファイルがCプログラムの場合)、または「pp」(入力ソースプログラムがC++プログラムの場合)のファイルが作成されます。
本オプションを指定した場合は、オブジェクトプログラムを出力しません。
noline を指定した場合、プリプロセッサ展開時に#line の出力を抑止します。

備　考　　本オプションを指定した場合、以下のオプション以外は無効になります。
define、include、comment、euc、sjis、latin1、subcommand、preinclude、message、lang、logo、cpu、change_message

オブジェクト形式

Code

コンパイラ<オブジェクト>[出力ファイル形式 :][機械語プログラム][アセンブリプログラム]

書　式　　Code = { Machinecode | Asmcode }

説　明　　オブジェクトプログラムの出力形式を指定します。
code=machinecode を指定した場合、リロケータブルオブジェクト(機械語)プログラムを出力します。
code=asmcode を指定した場合、アセンブリプログラムを出力します。
本オプションの省略時解釈は、code=machinecode です。

備　考　　code=asmcode を指定した場合、show=object、goptimize オプションは無効になります。

デバッグ情報

DEBug
NODEBug

コンパイラ<オブジェクト>[デバッグ情報出力]

書　式　　DEBug
NODEBug

説　明　　debug オプションを指定した場合、C ソースレベルデバッグに必要なデバッグ情報をオブジェクトファイルに出力します。debug オプションは、最適化オプションを指定した場合も有効となります。
nodebug オプションを指定した場合、デバッグ情報をオブジェクトファイルに出力しません。
本オプションの省略時解釈は、nodebug です。

セクション名

SEction

コンパイラ<オブジェクト>[詳細…][セクション :][プログラム領域 (P)][定数領域 (C)]
[初期化データ領域 (D)][未初期化データ領域 (B)]

書　式

```
SEction  =  <sub>[,...]
         <sub>: { Program = <セクション名> |
                  Const   = <セクション名> |
                  Data    = <セクション名> |
                  Bss     =  <セクション名> }
```

説　明　　オブジェクトプログラム中のセクション名を指定します。
section=program=<セクション名>は、プログラム領域のセクション名を指定します。
section=const=<セクション名>は、定数領域のセクション名を指定します。
section=data=<セクション名>は、初期化データ領域のセクション名を指定します。
section=bss=<セクション名>は、未初期化データ領域のセクション名を指定します。
<セクション名>は、英字、数字、下線(_)または、$の列で、先頭が数字以外のものです。セクション名は、8192 文字目まで有効です。
本オプションの省略時解釈は、section=program=P,const=C,data=D,bss=B です。

備　考　　プログラムとセクション名の対応についての詳細は、「9.1 プログラムの構造」を参照してください。
領域が異なるセクションに同じセクション名を指定できません。

文字列出力領域

STring

コンパイラ<オブジェクト>[詳細…][文字列データ格納 :]

書　式　　STring = { Const | Data }

説　明　　文字列の出力先を指定します。
string=const を指定した場合、定数領域に出力します。
string=data を指定した場合、初期化データ領域に出力します。
初期化データ領域へ出力した文字列はプログラム実行時に変更することができますが、ROM 上と RAM 上に二重に領域を確保し、プログラム実行開始時に ROM から RAM へ転送する必要があります。初期化データ領域の初期設定、メモリ割り付けの方法については、**「9.2.1 メモリ領域の割り付け」**を参照してください。
本オプションの省略時解釈は、string=const です。

オブジェクトファイル名指定

OBjectfile

コンパイラ<オブジェクト>[オブジェクト出力ディレクトリ :]

書　式　　OBjectfile = <ファイル名>

説　明　　出力するオブジェクトファイル名を指定します。
本オプションを指定しない場合には、ソースファイルと同じファイル名で拡張子が「obj」(出力ファイルがリロケータブルオブジェクトプログラムの場合)、または「src」(出力ファイルがアセンブリソースプログラムの場合)のオブジェクトファイルを出力します。
ファイル拡張子が「obj」か「src」かは、code オプションで決まります。

テンプレートインスタンス生成機能

Template

コンパイラ<オブジェクト>[詳細…][テンプレート生成 :]

書　式

```
Template ={ None     |
            Static   |
            Used     |
            ALl      |
            AUto     }
```

説　明

テンプレートのインスタンス生成方法を指定します。
`template=none` を指定した場合、インスタンスの生成を行いません。
`template=static` を指定した場合、コンパイル単位内で参照されたテンプレートのみインスタンスを作成します。ただし、生成される関数は内部リンケージを持ちます。
`template=used` を指定した場合、コンパイル単位内で参照されたテンプレートのみインスタンスを作成します。ただし、生成される関数は外部リンケージを持ちます。
`template=all` を指定した場合、コンパイル単位内で宣言または参照されている全てのテンプレートのインスタンスを作成します。
`templete=auto` を指定した場合、リンク時に必要なインスタンスの生成を行います。

備　考

`code=asmcode` を指定した場合は、常に `template=static` になります。

ABS16/20/28/32 宣言

ABs16
ABS20
ABS28
ABS32

コンパイラ<オブジェクト>[詳細…][アドレス宣言 :]

書　式

```
ABs16 = { Program | Const | Data | Bss | Run | All }[,...]
ABS20 = { Program | Const | Data | Bss | Run | All }[,...]
ABS28 = { Program | Const | Data | Bss | Run | All }[,...]
ABS32 = { Program | Const | Data | Bss | Run | All }[,...]
```

説　明

サブオプションで指定したセクションに属するラベルアドレスもしくはランタイムライブラリを配置するメモリ空間を指定します。
本オプションの省略時解釈は、abs32=all です。

表 2.3　アドレス範囲

オプション名	アドレス範囲	
	下位	上位
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

【注】* 0x07FFFF7Fとなることに注意。

表 2.4　サブオプションの説明

サブオプション名	意味
program	プログラム領域を対象にする
const	定数領域を対象にする
data	初期化データ領域を対象にする
bss	未初期化データ領域を対象にする
run	ランタイムライブラリを対象にする
all	すべての領域を対象にする

列

```
-abs20=program -abs28=const,data
                → -abs20=program -abs28=const,data -abs32=bss,run と同じ
-abs20=data -abs16=data
                → ウォーニングを出力し、-abs16=data が有効
```

備　考

本オプションと#pragma abs16|abs20|abs28|abs32 が同時に指定された場合、#pragma 指定が有効になります。
本オプションと#pragma gbr_base|gbr_base1 が同時に指定された場合、#pragma gbr_base|gbr_base1 指定された変数に対しては、本オプション指定は適用されません。
abs20|abs28 オプションは、cpu=sh2a|sh2afpu 以外を指定した場合は無効になります。

除算方式選択

DIvision

コンパイラ<オブジェクト>[詳細…][除算方式選択 :]

書　式

```
DIvision = { Cpu [= { Inline | Runtime }]|
             Peripheral                  |
             Nomask                      }
```

説　明

プログラム中の整数型除算、剰余算の方式を選択します。
division=cpu=inline を指定した場合、定数除算は乗算に変換してインライン展開し、変数除算は、マイコンが SH-2A または SH2A-FPU の場合 DIVS または DIVU 命令を、それ以外のマイコンの場合 DIV1 命令によるランタイムライブラリをそれぞれ選択します。本オプションは、cpu=sh1 を指定した場合は無効になります。
division=cpu=runtime を指定した場合、シフト演算で行えない除算について、マイコンが SH-2A または SH2A-FPU の場合 DIVS または DIVU 命令を、それ以外のマイコンの場合 DIV1 命令によるランタイムライブラリをそれぞれ選択します。本オプションは、cpu=sh1 を指定した場合は無効になります。
division=cpu のみ指定した場合は、size オプション指定時は division=cpu=runtime、speed、nospeed オプション指定時は division=cpu=inline とみなします。
division=peripheral を指定した場合、除算器を用いたランタイムライブラリを選択(割り込みマスクに 15 を設定)します。マイコンが SH-2(SH7604)の時のみ実行可能です。
division=nomask を指定した場合、除算器を用いたランタイムライブラリを選択(割り込みマスクは変更なし)します。マイコンが SH-2(SH7604)の時のみ実行可能です。
peripheral、nomask 指定時は以下の点に注意してください。

(1) ゼロ除算のチェックおよび errno の設定は行いません。
(2) nomask 指定時には、除算器動作中に割り込みがかかり、割り込み処理ルーチンで除算器を用いた場合、動作は保証しません。
(3) オーバフロー割り込みはサポートしていません。
(4) ゼロ除算、オーバフローなどの演算結果は除算器の仕様に従います。cpu サブオプション指定時と異なる場合があります。

浮動小数点レジスタ退避･回復抑止

IFUnc

コンパイラ<オブジェクト>[詳細…][浮動小数点レジスタ退避・回復抑止]

書　式

```
IFUnc
```

説　明

浮動小数点レジスタの退避・回復を抑止します。

備　考

#pragma ifunc で関数単位での指定もできます。
ifunc オプションを指定した場合、浮動小数点命令を生成するソースプログラムをコンパイルするとエラーになります。
本オプションは cpu=sh2e|sh2afpu|sh4|sh4a を指定した場合のみ有効です。

ラベルの 16/32 バイト整合

ALIGN16
ALIGN32
NOALign

コンパイラ<オブジェクト>[詳細…][ラベルの 16/32 バイト整合 :]

書　式　　ALIGN16
ALIGN32
NOALign

説　明　　align16 オプションを指定した場合、プログラムセクション内のラベルで無条件分岐命令の後で最初に出現するラベルを、すべて 16 バイト整合します。
align32 オプションを指定した場合、プログラムセクション内のラベルで無条件分岐命令の後で最初に出現するラベルを、すべて 32 バイト整合します。
noalign オプションを指定した場合、無条件分岐命令の後で最初に出現するラベルを 16/32 バイト整合しません。
本オプションの省略時解釈は、noalign です。

備　考　　align16 オプションと align32 オプションを同時に指定することはできません。
noalign16 オプションを指定した場合、noalign オプションが指定されたものとみなします。

TBR 相対関数呼び出し指定

TBR

コンパイラ<オブジェクト>[詳細…][TBR 指定 :]

書　式　　TBR [= <セクション名>]

説　明　　関数呼び出しを TBR 相対で行います。
<セクション名>を指定した場合は、関数定義に対し関数アドレステーブルを$TBR<セクション名>セクションに出力します。
<セクション名>を指定しない場合は、関数定義に対し関数アドレステーブルを$TBR セクションに出力します。
詳細については「10.3.1(2) #pragma tbr」を参照してください。

備　考　　本オプションは、cpu=sh2a|sh2afpu を指定した場合のみ有効です。
本オプションと#pragma tbr が同時に指定された場合、#pragma tbr 指定が有効になります。本オプションと pic=1 が同時に指定された場合、本オプションは無効になります。
関数アドレステーブルを作成する関数が 255 個を超えた場合、エラーになります。

未初期化変数の出力順

BSs_order

コンパイラ<オブジェクト>[詳細…][未初期化変数の出力順 :]

書　式　　BSs_order = { declaration | definition }

説　明　　bss_order=declaration を指定した場合、未初期化変数の割り付けを宣言順に行います。
bss_order=definition を指定した場合、未初期化変数の割り付けを定義順に行います。
本オプション省略時解釈は、bss_order=declaration です。

例

```
extern int a1;
extern int a2;
int a3;
extern int a4;
int a5;
int a2;
int a1;
int a4;
```

＜bss_order=declaration 指定時＞

```
.SECTION B,DATA,ALIGN=4
_a1:
.RES.L 1
_a2:
.RES.L 1
_a3:
.RES.L 1
_a4:
.RES.L 1
_a5:
.RES.L 1
```

＜bss_order=definition 指定時＞

```
.SECTION B,DATA,ALIGN=4
_a3:
.RES.L 1
_a5:
.RES.L 1
_a2:
.RES.L 1
_a1:
.RES.L 1
_a4:
.RES.L 1
```

備　考　　stuff オプションを指定した場合は、本オプション指定に関わらず常に未初期化変数の割り付けを定義順に行います。

変数の配置指定

STUff
NOSTuff

コンパイラ<オブジェクト>[詳細 2…][変数の配置指定 :]

書　式

```
STUff [= <セクション種別>[,...]]
NOSTuff
<セクション種別>:{ Bss | Data | Const }
```

説　明

stuff オプションを指定した場合、指定した<セクション種別>に属する変数をデータサイズに応じてアライメント数が 4 のセクション、2 のセクション、1 のセクションに配置します(表 2.5)。
<セクション種別>を省略した場合は、すべての変数が対象になります。
C、D、B は section オプションまたは#pragma section で指定したセクション名になります。各セクション内のデータは bss_order オプション指定に関わらず常に定義順に出力されます。

表 2.5　変数のサイズとセクション名の関係

	セクション種別	変数のサイズ (byte) 4n	4n-2	2n-1
const 型変数	const	C$4	C$2	C$1
初期値あり変数	data	D$4	D$2	D$1
初期値なし変数	bss	B$4	B$2	B$1

nostuff オプションを指定した場合、すべての変数をアライメント数が 4 のセクションに配置します。各セクション内の出力は C、D セクションは定義順、B セクションは bss_order オプションに従います。
本オプション省略時解釈は、nostuff です。

例

```
int a;
char b=0;
const short c=0;
struct {
   char x;
   char y;
} ST;
```

＜stuff オプション指定時＞

```
.SECTION C$2,DATA,ALIGN=2
_c:
.DATA.W H'0000
.SECTION D$1,DATA,ALIGN=1
_b:
.DATA.B H'00
.SECTION B$4,DATA,ALIGN=4
_a:
.RES.L  1
.SECTION B$2,DATA,ALIGN=2
_ST:
.RES.B  2
```

＜nostuff オプション指定時＞

```
.SECTION C,DATA,ALIGN=4
_c:
.DATA.W H'0000
.SECTION D,DATA,ALIGN=4
_b:
.DATA.B H'00
.SECTION B,DATA,ALIGN=4
_a:
.RES.L  1
_ST:
.RES.B  2
```

備　考　　#pragma gbr_base|gbr_base1 または#pragma global_register を指定した変数は、本オプションの対象外になります。

$G0,$G1 セクションの変数の配置指定

STUFF_GBR

コンパイラ<オブジェクト>[詳細 2…][$G0/$G1 セクション内変数の配置：]

書　式　　STUFF_GBR

説　明　　#pragma gbr_base|gbr_base1 指定変数を、変数のサイズに従って表 2.6 に示したセクションに配置します。これにより境界調整による空き領域を削減することができます。

表 2.6　変数のサイズとセクション名の関係

	変数のサイズ(バイト)		
	4n	4n-2	2n-1
#pragma gbr_base 指定変数	$G0$4	$G0$2	$G0$1
#pragma gbr_base1 指定変数	$G1$4	$G1$2	$G1$1

備　考　　本オプションは gbr=user を指定した場合のみ有効です。
$G0、$G1 で始まるセクションは、表 2.7 に従って配置してください。

表 2.7　セクションの配置

セクション名	配置方法
$G0	先頭アドレスが 4 の倍数になるように配置する
$G0$1,$G0$2,$G0$4	$G0 の先頭アドレスから 128 バイト以内に収まるように配置する
$G1	$G0 の先頭アドレスから 128 バイト先に配置する
$G1$1	$G0 の先頭アドレスから 256 バイト以内に収まるように配置する
$G1$2	$G0 の先頭アドレスから 512 バイト以内に収まるように配置する
$G1$4	$G0 の先頭アドレスから 1024 バイト以内に収まるように配置する

分岐先アドレスの境界調整

ALIGN4

コンパイラ<オブジェクト>[詳細…][分岐先アドレスの境界調整：]

書　式　　ALIGN4 = { ALL | LOOP | INMOSTLOOP }

説　明　　align4=all を指定した場合、すべての分岐先アドレスを 4 バイト整合します。
align4=loop を指定した場合、すべてのループの先頭アドレスを 4 バイト整合します。
align4=inmostloop を指定した場合､最内側ループの先頭アドレスを 4 バイト整合します。

備　考　　本オプションは align16 オプション、align32 オプションと同時に指定することはできません。
本オプションを指定した場合、関数の先頭アドレスは常に 4 バイト整合されます。
分岐先アドレスを 4 バイト整合した関数は、リンク時最適化の対象外になります。

const volatile 属性変数の割り付け

CONST_VOLATILE

コンパイラ<オブジェクト>[詳細…][const volatile 修飾変数:]

書　式　　-CONST_VOLATILE={ DATA | CONST}

説　明　　const 修飾かつ volatile 修飾された初期値あり変数の割付先を選択します。
const_volatile=const を指定した場合、定数領域に割り付けます。
const_volatile=data を指定した場合、初期化データ領域に割り付けます。
本オプション省略時解釈は、const_volatile=data です。

例

(1) const volatile int c=3; の、変数 c の割付先
- const_volatile=data 指定時、初期化データ領域(D セクション)
- const_volatile=const 指定時、定数領域(C セクション)
- const_volatile=const -stuff 指定時、定数領域(C$4 セクション)
- const_volatile=const -section=const=N 指定時、定数領域(N セクション)

(2) __X const volatile __fixed x=0.5r; の、変数 x の割付先
- const_volatile=data 指定時、初期化データ領域($XD セクション)
- const_volatile=const 指定時、定数領域($XC セクション)

2.2.3 リストオプション

表 2.8 リストカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	リスト ファイル	Listfile [= <ファイル名>] NOListfile	コンパイラ<リスト> [コンパイルリスト出力]	出力あり 出力なし
2	リスト内容と 形式	SHow = <sub>[,…] <sub>:{ SOurce | NOSOurce | Object | NOObject | STatistics | NOSTatistics | Include | NOInclude | Expansion|NOExpansion| Width = <数値> | Length = <数値> | Tab = {4 | 8} }	コンパイラ<リスト> [リスト内容 :]	 ソースリストの有無 オブジェクトリストの有無 統計情報の有無 インクルード展開後リストの有無 マクロ展開後リストの有無 １行の最大文字数: 0, 80～132 ページ内の最大行数: 0, 40～255 タブ使用時のカラム数: 4 か 8

リストファイル

Listfile
NOListfile

コンパイラ<リスト>[コンパイルリスト出力]

書　式　　Listfile [= <ファイル名>]
NOListfile

説　明　　リストファイルの出力有無を指定します。
listfile オプションを指定した場合、リストファイルを出力します。<ファイル名>を指定することもできます。
nolistfile オプションを指定した場合、リストファイルは出力しません。
<ファイル名>は、「8.1 ファイル名の付け方」に従って指定できます。
listfile オプションで<リストファイル名>を指定しない場合には、ソースファイルと同じファイル名で、拡張子が「lst」(入力ソースが C プログラムの場合)、または「lpp」(入力ソースが C++プログラムの場合)のリストファイルが作成されます。
本オプションの省略時解釈は、nolistfile です。

リスト内容と形式

SHow

コンパイラ<リスト>[リスト内容 :]

書　式

```
SHow = <sub>[,...]
    <sub>:{ SOurce      |NOSOurce      |
            Object      |NOObject      |
            STatistics  |NOSTatistics  |
            Include     |NOInclude     |
            Expansion   |NOExpansion   |
            Width = <数値>             |
            Length = <数値>            |
            Tab = { 4 | 8 }            }
```

説　明

コンパイラが出力するリストの内容とその形式、および出力の解除を指定します。
本項で記した各リストの具体例については「8.2 コンパイルリストの参照方法」を参照してください。
本オプションの省略時解釈は、show=nosource,object,statistics,noinclude,noexpansion,width=0,length=0,tab=8 です。

備　考

サブオプションの一覧を表 2.9 に示します。

表 2.9 show オプションのサブオプション一覧

	サブオプション名	意味
1	source	ソースプログラムのリストを出力します。
2	nosource	ソースプログラムのリストを出力しません。
3	object	オブジェクトプログラムのリストを出力します。
4	noobject	オブジェクトプログラムのリストを出力しません。
5	statistics	統計情報のリストを出力します。
6	nostatistics	統計情報のリストを出力しません。
7	include	インクルードファイル展開した後のソースプログラムリストを出力します。nosource サブオプションが同時に指定された場合には、include サブオプションは無効となり、ソースプログラムリストは出力されません。
8	noinclude	インクルードファイル展開する前のソースプログラムリストを出力します。nosource サブオプションが同時に指定された場合には、noinclude サブオプションは無効となり、ソースプログラムリストは出力されません。
9	expansion	マクロ展開した後のソースプログラムリストを出力します。nosource サブオプションが同時に指定された場合には、expansion サブオプションは無効となり、ソースプログラムリストは出力されません。
10	noexpansion	マクロを展開する前のソースプログラムリストを出力します。nosource サブオプションが同時に指定された場合には、noexpansion サブオプションは無効となり、ソースプログラムリストは出力されません。
11	width = <数値>	<数値>で指定する文字数をリストの 1 行の最大文字数とします。<数値>は 10 進数で指定し、0 または 80 から 132 の間の数値を指定することができます。<数値>が 0 の場合、リストの 1 行の最大文字数は規定されません。
12	length = <数値>	<数値>で指定する行数を、リストの 1 ページの最大行数とします。<数値>は 10 進数で指定し、0 または 40 から 255 の間の数値を指定することができます。<数値>が 0 の場合、リストの 1 ページの最大行数は規定されません。
13	tab = { 4 | 8 }	リスト表示時のタブのサイズを指定します。

2.2.4　最適化オプション

表 2.10　最適化カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	最適化レベル	OPtimize = { 0| 1 | Debug_only }	コンパイラ<最適化> [最適化]	最適化なし 最適化あり デバッグ情報に影響のないコード生成
2	実行速度･サイズ最適化	SPeed SIze NOSPeed	コンパイラ<最適化> [最適化方法 :] [スピード優先] [サイズ優先] [サイズ&スピード]	最適化項目の選択
3	モジュール間最適化	Goptimize	コンパイラ<最適化> [モジュール間最適化]	モジュール間最適化用付加情報出力
4	外部変数アクセス最適化	MAP = <ファイル名>	コンパイラ<最適化> [外部変数アクセス最適化 :] [モジュール間]	外部変数アクセス最適化を行う
5	外部変数アクセス最適化	SMap	コンパイラ<最適化> [外部変数アクセス最適化 :] [モジュール内]	コンパイル対象ファイル内で定義された外部変数に対し､外部変数アクセス最適化を行う
6	GBR 相対アクセスコード自動生成	GBr = { Auto | User }	コンパイラ<最適化> [GBR 相対アクセスコード自動生成 :]	GBR 相対アクセスコード自動生成を行う GBR 相対アクセスコード自動生成を行わない
7	switch 文展開方式	CAse = { Ifthen | Table }	コンパイラ<最適化> [switch 文展開 :]	if_then 方式で展開 テーブルジャンプ方式で展開
8	シフト演算展開	SHIft = { Inline | Runtime }	コンパイラ<最適化> [シフト演算展開 :]	インライン展開する 展開命令数の多いシフト演算をランタイムライブラリ呼び出しにする
9	転送コード展開	BLOckcopy = { Inline | Runtime }	コンパイラ<最適化> [転送コード展開 :]	インライン展開する 転送サイズが大きい場合はランタイムライブラリ呼び出しにする
10	非アライメントデータ転送	Unaligned = { Inline | Runtime }	コンパイラ<最適化> [非境界調整データ転送 :]	インライン展開する ランタイムライブラリ呼び出しにする
11	自動インライン展開	INLine[= <数値>] NOINLine	コンパイラ<最適化> [詳細…] [インライン展開] [自動インライン展開]	自動インライン展開する 自動インライン展開しない
12	ファイル間インライン展開	FILe_inline= <ファイル名>[,…]	コンパイラ<最適化> [詳細…] [インライン展開] [インライン展開ファイル]	ファイル間インライン展開対象ファイルの指定
13	外部変数の volatile 化	GLOBAL_Volatile={ 0 | 1 }	コンパイラ<最適化> [詳細…] [外部変数] [外部変数 volatile 化]	外部変数について volatile 宣言されたものとして扱わない(volatile 宣言されたものは除く) すべての外部変数について volatile 宣言されたものとして扱う

14	外部変数最適化範囲指定	OPT_Range={All | NOLoop | NOBlock }	コンパイラ<最適化> [詳細…] [外部変数] [外部変数の最適化範囲 :]	関数内の全範囲で外部変数を最適化対象とする ループ制御変数やループ内の外部変数のループ外への移動を抑止 ループや、分岐をまたいだ外部変数に対する最適化をすべて抑止
15	空ループ削除	DEL_vacant_loop={ 0 | 1 }	コンパイラ<最適化> [詳細…] [その他] [空ループ削除]	空ループ削除を抑止 空ループ削除を行う
16	ループ展開最適化	LOop NOLOop	コンパイラ<最適化> [詳細…] [その他] [ループ展開 :]	ループ展開の最適化を行う ループ展開の最適化を行わない
17	ループ最大展開数の指定	MAX_unroll=<数値> <数値>:1-32	コンパイラ<最適化> [詳細…] [その他] [ループ最大展開数 :]	ループ展開時の最大展開数を指定 default:1(speed,loop 指定時 2)
18	無限ループ前の式削除	INFinite_loop={ 0 | 1 }	コンパイラ<最適化> [詳細…] [外部変数] [無限ループ前の外部変数への代入式削除]	無限ループ前の外部変数代入削除を抑止 無限ループ前の外部変数代入削除を行う
19	外部変数のレジスタ割り付け	GLOBAL_Alloc={ 0 | 1 }	コンパイラ<最適化> [詳細…] [外部変数] [外部変数のレジスタ割付 :]	外部変数のレジスタ割り付けを抑止 外部変数のレジスタ割り付けを行う
20	構造体/共用体メンバのレジスタ割り付け	STRUCT_Alloc={ 0 | 1 }	コンパイラ<最適化> [詳細…] [その他] [構造体/共用体メンバのレジスタ割付 :]	構造体/共用体メンバのレジスタ割り付けを抑止 構造体/共用体メンバのレジスタ割り付けを行う
21	const 定数伝播	CONST_Var_propagate={ 0 | 1 }	コンパイラ<最適化> [詳細…] [外部変数] [外部定数の定数伝播 :]	const 修飾型外部変数の定数伝播を抑止 const 修飾型外部変数の定数伝播を行う
22	定数ロードの命令展開	CONST_Load={ Inline | Literal }	コンパイラ<最適化> [詳細…] [その他] [定数ロードの命令展開 :]	定数ロードを命令展開 定数ロードをリテラルアクセス defalut:size 指定時 2-3 命令まで展開 それ以外リテラルアクセス
23	命令並べ替え	SChedule={ 0 | 1 }	コンパイラ<最適化> [詳細…] [外部変数] [命令並べ替え :]	命令並べ替えを行わない 命令並べ替えを行う

24	ソフトウェアパイプライニング (SH-2A, SH2A-FPU, SH-4,SH-4A, SH4AL-DSP)	SOftpipe	コンパイラ<最適化> [詳細…] [その他] [ソフトウェアパイプライニング :]	ソフトウェアパイプライニングを有効にする
25	最適化範囲分割	SCOpe NOSCope	コンパイラ<最適化> [詳細…] [その他] [最適化範囲の分割をしない :]	最適化範囲を分割する 最適化範囲を分割しない
26	GBR 相対論理演算生成	LOGIc_gbr	コンパイラ<最適化> [GBR 相対アクセスコード自動生成 :]	外部変数に対する論理演算を GBR 相対で行うコードを生成
27	C++インライン関数の展開抑止	CPP_NOINLINE	コンパイラ<最適化> [詳細…] [インライン展開] [C++インライン関数の展開抑止 :]	C++インライン関数の展開を抑止する
28	ポインタ指示先の型を考慮した最適化	ALIAS = {ANSI | NOANSI }	コンパイラ<最適化> [詳細…] [その他] [ポインタ指示先の型を考慮した最適化 :]	ポインタ指示先の型を考慮した最適化を実施する ポインタ指示先の型を考慮した最適化を実施しない

最適化レベル

OPtimize

コンパイラ<最適化>[最適化]

書　式　　`OPtimize = { 0 | 1 | Debug_only }`

説　明　　オブジェクトプログラムの最適化レベルを指定します。
`optimize=debug_only`を指定した場合、最適化を実施しません。これにより、デバッグ情報を高い精度で出力でき、ソースレベルデバッグがしやすくなります。
`optimize=0` を指定した場合、自動変数のレジスタ割付、関数出口ブロックの統合、統合可能な複数命令の統合など、一部最適化を実施します。これにより、`optimize=debug_only`指定時よりもコードサイズを削減できます。
`optimize=1` を指定した場合、最適化を実施します。
本オプションの省略時解釈は、`optimize=1` です。

実行速度・サイズ最適化

SPeed
SIze
NOSPeed

コンパイラ<最適化>[最適化方法 :][スピード優先][サイズ優先][サイズ&スピード]

書　式　　SPeed
SIze
NOSPeed

説　明　　speed、size、nospeed オプションにおける主な最適化項目を表 2.11 に示します。
これらの最適化項目は、オプションで制御可能です。
本オプションの省略時解釈は、nospeed です。

表 2.11　最適化項目一覧

	自動インライン展開	ループ展開最適化	シフトコード展開	転送コード展開	整数定数除算展開	非アライメントデータ転送
speed	inline=20	loop	inline	inline	inline	inline
size	noinline	noloop	runtime	runtime	runtime	runtime
nospeed	noinline	noloop	inline	inline	inline	inline

モジュール間最適化

Goptimize

コンパイラ<最適化>[モジュール間最適化]

書　式　　Goptimize

説　明　　モジュール間最適化用付加情報を出力します。
本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。

外部変数アクセス最適化

MAP

コンパイラ<最適化>[外部変数アクセス最適化 :][モジュール間]

書　式　　MAP = <ファイル名>

説　明　　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ベースアドレスを設定し、外部変数もしく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。gbr=auto を指定した場合、条件によりベースアドレスを GBR レジスタに設定し、外部変数もしくは静的変数のアクセスを GBR 相対で行うコードを生成します。
本オプションを使用する際は、プログラムを本オプションを指定しないで一度コンパイルし、リンク時に map=<ファイル名>を指定して外部シンボル割り付け情報ファイルを作成し、再度 map=<ファイル名>を指定してコンパイルしてください。

例　　プログラム(test.c)

```
int A,B,C;
void main()
{
   A = 0;
   B = 0;
   C = 0;
}
```

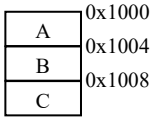
(1) コマンド： shc test.c

＜出力コード＞

```
_main:
       MOV.L      L11,R6    ;_A
       MOV        #0,R2
       MOV.L      R2,@R6
       MOV.L      L11+4,R6  ;_B
       MOV.L      R2,@R6
       MOV.L      L11+8,R6  ;_C
       RTS
       MOV.L      R2,@R6
L11:
       .DATA.L    _A
       .DATA.L    _B
       .DATA.L    _C
```

(2) コマンド： shc test.c
optlnk -map=test.bls -start=P/400,B/1000 test.obj
shc -map=test.bls test.c

リンク後のデータ配置



```
<出力コード>
_main:
        MOV.W       L11,R1    ;_A       ※A のアドレスをベースアドレスに設定
        MOV         #0,R0
        MOV.L       R0,@R1
        MOV.L       R0,@(4,R1)
        RTS
        MOV.L       R0,@(8,R1)
L11:
        .DATA.W     _A                  ※A のアドレスは 2 バイト
```

備　考　　外部変数もしくは静的変数の定義順を変更した場合は、外部シンボルアドレス情報ファイルを生成し直す必要があります。
本オプション以外で 1 回目のコンパイル時に指定したオプションと異なるオプションを指定した場合と、関数内の処理を追加した場合は、動作は保証しません。これらの場合は必ず外部シンボルアドレス情報ファイルを生成しなおしてください。

外部変数アクセス最適化

SMap

コンパイラ<最適化>[外部変数アクセス最適化 :][モジュール内]

書　式　　SMap

説　明　　コンパイル対象ファイル内で定義された外部変数もしく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。
gbr=auto を指定した場合、条件によりベースアドレスを GBR レジスタに設定し、外部変数もしくは静的変数のアクセスを GBR 相対で行うコードを生成します。

例

```
int A,B,C;
void main()
{
   A = 0;
   B = 0:
   C = 0;
}

MOV.L       L11,R6     ; _A
MOV         #0,R2      ; H'00000000
MOV.L       R2,@R6
MOV.L       R2,@(4,R6)
RTS
MOV.L       R2,@(8,R6)
```

備　考　　map=<ファイル名>と同時に指定した場合は、map オプションが有効になります。

GBR 相対アクセスコード自動生成

GBr

コンパイラ<最適化>[GBR 相対アクセスコード自動生成 :]

書　式　　GBr = { Auto | User }

説　明　　gbr=auto を指定した場合、条件によりコンパイラが自動的に GBR 相対論理演算コードを生成します。gbr=auto かつ map=<ファイル名>を指定した場合は、条件により GBR にベースアドレスを設定し、外部変数アクセスを GBR 相対で行うコードを生成します。
gbr=user を指定した場合、GBR の設定、参照、GBR 相対アクセスは#pragma gbr_base、#pragma gbr_base1、または GBR 関連組み込み関数でユーザが指定します。
本オプション省略時解釈は、gbr=auto です。

例　　プログラム

```
char A,B,C;
void main()
{
   A |= 1;
   B &= 1;
   C ^= 1;
}
```

＜出力コード(gbr=auto)＞

```
_main:
       STC         GBR,@-R15     ; GBR を退避
       MOV         #0,R0
       LDC         R0,GBR        ; GBR に 0 を設定
       MOV.L       L11+2,R0      ; R0 <- A のアドレス
       OR.B        #1,@(R0,GBR)  ; A |= 1
       MOV.L       L11+6,R0      ; R0 <- B のアドレス
       AND.B       #1,@(R0,GBR)  ; B &= 1
       MOV.L       L11+10,R0     ; R0 <- C のアドレス
       XOR.B       #1,@(R0,GBR)  ; C ^= 1
       RTS
       LDC         @R15+,GBR     ; GBR の回復
L11:
       .RES.W      1
       .DATA.L     _A
       .DATA.L     _B
       .DATA.L     _C
```

備　考　　#pragma gbr_base、#pragma gbr_base1 を使用したプログラムを gbr=auto を指定してコンパイルするとウォーニングメッセージを出力し、#pragma 指定を無視します。また、gbr 関連組み込み関数を使用したプログラムを gbr=auto を指定してコンパイルするとエラーになります。
gbr=auto を指定した場合、関数の呼び出し前後で GBR レジスタを保証します。

switch 文展開方式

CAse

コンパイラ<最適化>[switch 文展開 :]

書　式　　CAse = { Ifthen | Table }

説　明　　switch 文のコード展開方式を指定します。
case=ifthen を指定した場合、switch 文を if_then 方式で展開します。if_then 方式は、switch 文の評価式の値と case ラベルの値を比較し、一致すれば case ラベルの文へ飛ぶ処理を case ラベルの回数繰り返す展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例してオブジェクトコードのサイズが増大します。
case=table を指定した場合、switch 文をテーブル方式で展開します。テーブル方式は、case ラベルの飛び先をジャンプテーブルに確保し、1 回のジャンプテーブルの参照で switch 文の評価式と一致する case ラベルの文へ飛ぶ展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例してリテラルプールに確保されるジャンプテーブルのサイズが増えますが、実行速度は常に一定です。
本オプション省略時は、いずれかの展開方式をコンパイラが自動的に選択します。

シフト演算展開

SHIft

コンパイラ<最適化>[シフト演算展開 :]

書　式　　SHIft = { Inline | Runtime }

説　明　　シフト数が 0 以上かつ(被シフト数のビット幅-1)以下の定数であるシフト演算の方式を選択します。
shift=inline を指定した場合、すべてのシフト演算を命令展開します。
shift=runtime を指定した場合、展開する命令が多い場合はランタイムライブラリ呼び出しになります。
size オプションを指定した場合のデフォルトは shift=runtime です。また speed、nospeed オプションを指定した場合のデフォルトは shift=inline です。

転送コード展開

BLOckcopy

コンパイラ<最適化>[転送コード展開 :]

書　式　　BLOckcopy = { Inline | Runtime }

説　明　　blockcopy=inline を指定した場合、すべてのメモリ間転送コードを命令展開します。
blockcopy=runtime を指定した場合、転送サイズが大きい場合はランタイムライブラリ呼び出しになります。
size オプションを指定した場合のデフォルトは、blockcopy=runtime です。また speed、nospeed オプションを指定した場合のデフォルトは、blockcopy=inline です。

非アライメントデータ転送

Unaligned

コンパイラ<最適化>[非境界調整データ転送 :]

書　式　　Unaligned = { Inline | Runtime }

説　明　　unaligned=inline を指定した場合、非アライメントデータ転送を命令展開します。
unaligned=runtime を指定した場合、非アライメントデータ転送のサイズが大きい場合は、ランタイムライブラリ呼び出しになります。
size オプションを指定した場合のデフォルトは、unaligned=runtime です。また speed、nospeed オプションを指定した場合のデフォルトは、unaligned=inline です。

備　考　　本オプションはアライメント数が 1 の構造体の転送が対象になります。

自動インライン展開

INLine
NOINLine

コンパイラ<最適化>[詳細…][インライン展開][自動インライン展開]

書　式　　INLine [= <数値>]
NOINLine

説　明　　関数の自動インライン展開を行うかどうかを指定します。
inline オプションを指定した場合、自動インライン展開を行います。
inline=<数値>で、プログラムサイズが何%増加するまでインライン展開を行うかを指定できます。例えば inline=50 を指定した場合、プログラムサイズが 50%増加するまで(1.5 倍になるまで)インライン展開を行います。
noinline オプションを指定した場合、自動インライン展開を行いません。
speed オプションを指定した場合のデフォルトは、inline=20 です。nospeed、size,optimize=0 または optimize=debug_only オプションを指定した場合のデフォルトは、noinline です。

ファイル間インライン展開

FILe_inline

コンパイラ<最適化>[詳細…][インライン展開][インライン展開ファイル]

書　式　　FILe_inline=<ファイル名>[,…]

説　明　　ファイル名で指定されたファイルについて、ファイル間にまたがった関数インライン展開を行います。

例

```
<a.c>
func(){
    g();
}
<b.c>
g(){
    h();
}
```

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

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

備　考　　file_inline オプションと noinline オプションを同時に指定した場合、#pragma inline 指定された関数についてのみインライン展開します。
file_inline オプションで指定された複数のファイルで同じ名前の extern 関数が定義されていた場合、動作は保証しません(任意に選んだ 1 つの関数定義を用いてインライン展開します)。
<ファイル名>で指定するファイル名の拡張子を省略することはできません。
コンパイル対象のファイルを file_inline オプションで指定することはできません。
<ファイル名>にワイルドカード(*,?)を指定することはできません。

外部変数の volatile 化

GLOBAL_Volatile

コンパイラ<最適化>[詳細…][外部変数][外部変数 volatile 化]

書　式　　GLOBAL_Volatile = { 0 | 1 }

説　明　　global_volatile=0 を指定した場合、volatile 宣言のない外部変数に対して最適化を行います。したがって、外部変数のアクセス回数、アクセス順序が C/C++プログラムで記述した場合と異なることがあります。
global_volatile=1 を指定した場合、すべての外部変数を volatile 宣言したものとして扱います。したがって、外部変数のアクセス回数、アクセス順序は C/C++プログラムで記述した通りになります。
本オプション省略時解釈は、global_volatile=0 です。

外部変数最適化範囲指定

OPT_Range

コンパイラ<最適化>[詳細…][外部変数][外部変数の最適化範囲 :]

書　式　　OPT_Range = { All | NOLoop | NOBlock }

説　明　　opt_range=all を指定した場合、関数内の全範囲を対象に外部変数に対する最適化を行います。
opt_range=noloop を指定した場合、ループ内にある外部変数やループ判定式で使用されている外部変数を最適化の対象外にします。
opt_range=noblock を指定した場合、分岐をまたいだ外部変数の最適化(ループを含む)をすべて抑止します。
本オプション省略時解釈は、optimize=0 または optimize=debug_only を指定した場合は opt_range=noblock、それ以外の場合は opt_range=all です。

例　　(1) 分岐をまたいだ最適化例(opt_range=all/noloop 指定時に行う)

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a)
        B = 1;
    C = A;
}
```

＜最適化後のソースイメージ＞

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a)
        B = 1;
    C = 1;      /* A の参照を削除し、A=1 を伝播する */
}
```

(2) ループにおける最適化例(opt_range=all 指定時に行う)

```
int A,B,C[100];
void f() {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

＜最適化後のソースイメージ＞

```
int A,B,C[100];
void f() {
    int i;
    int temp_A, temp_B;
    temp_A = A; /* ループ判定式の A の参照をループ外に移動*/
    temp_B = B; /* ループ内の B の参照をループ外に移動     */
    for (i=0;i<temp_A;i++) {      /* A のループ内での参照を削除 */
        C[i] = temp_B;            /* B のループ内での参照を削除 */
    }
}
```

備　考　　opt_range=noloop を指定した場合、常に max_unroll=1 がデフォルトになります。
opt_range=noblock を指定した場合、常に max_unroll=1、const_var_propagate=0、global_alloc=0 がデフォルトになります。

空ループ削除

DEL_vacant_loop

コンパイラ<最適化>[詳細…][その他][空ループ削除]

書　式　　DEL_vacant_loop = { 0 | 1 }

説　明　　del_vacant_loop=0 を指定した場合、ループ内処理がない場合でもループを削除しません。
del_vacant_loop=1 を指定した場合、ループ内処理がないループは削除します。
本オプション省略時解釈は、del_vacant_loop=0 です。

備　考　　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：空ループを削除する
V7.0.06 以降 ：空ループを削除しない

ループ展開最適化

LOop
NOLOop

コンパイラ<最適化>[詳細…][その他][ループ展開 :]

書　式　　LOop
NOLOop

説　明　　ループ展開の最適化を行うかどうかを指定します。
loop オプションを指定した場合、繰り返し文(for,while,do-while)を展開します。
noloop オプションを指定した場合、繰り返し文を展開しません。
本オプションの省略時解釈は、optimize=1 かつ speed を指定した場合は loop、それ以外の場合は noloop です。

ループ最大展開数の指定

MAX_unroll

コンパイラ<最適化>[詳細…][その他][ループ最大展開数 :]

書　式　　MAX_unroll = <数値>

説　明　　ループ展開時の最大展開数を指定します。<数値>には 1 から 32 までの整数を指定することができます。それ以外の値を指定した場合はエラーになります。
本オプション省略時解釈は、speed または loop オプションを指定した場合は max_unroll=2、それ以外の場合は max_unroll=1 です。

備　考　　opt_range=noloop/noblock を指定した場合、常に max_unroll=1 がデフォルトになります。

無限ループ前の式削除

INFinite_loop

コンパイラ<最適化>[詳細…][外部変数][無限ループ前の外部変数への代入式削除]

書　式　　INFinite_loop = { 0 | 1 }

説　明　　infinite_loop=0 を指定した場合、無限ループ直前の外部変数への代入式を削除しません。
infinite_loop=1 を指定した場合、無限ループ直前にあり無限ループ内で参照されない外部変数への代入式を削除します。
本オプション省略時解釈は、infinite_loop =0 です。

例

```
int A;
void f() {
    A = 1;        /* 外部変数 A への代入式 */
    while(1) {}   /* A は参照されない     */
}

＜infinite_loop=1 指定時のイメージ＞
void f() {
                  /* 外部変数 A への代入式を削除 */
    while(1) {}
}
```

備　考　　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。

V7.0.04 まで ：無限ループ直前のループ内で参照されない外部変数への代入式を削除する
V7.0.06 以降 ：無限ループ直前の外部変数への代入式を削除しない

外部変数のレジスタ割り付け

GLOBAL_Alloc

コンパイラ<最適化>[詳細…][外部変数][外部変数のレジスタ割付 :]

書　式　　GLOBAL_Alloc = { 0 | 1 }

説　明　　global_alloc=0 を指定した場合、外部変数のレジスタ割り付けを抑止します。
global_alloc=1 を指定した場合、外部変数のレジスタ割り付けを行います。

備　考　　本オプションの省略時解釈は、opt_range=noblock または optimize=debug_only を指定した場合、global_alloc=0 です。optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。

V7.0.04 まで ：外部変数のレジスタ割り付けを行う
V7.0.06 以降 ：外部変数のレジスタ割り付けを抑止する

それ以外の場合は、global_alloc=1 です。

構造体/共用体メンバのレジスタ割り付け

STRUCT_Alloc

コンパイラ<最適化>[詳細…][その他][構造体/共用体メンバのレジスタ割付 :]

書　式　　STRUCT_Alloc = { 0 | 1 }

説　明　　struct_alloc=0 を指定した場合、構造体/共用体メンバのレジスタ割り付けを抑止します。
struct_alloc=1 を指定した場合、構造体/共用体メンバのレジスタ割り付けを行います。

備　考　　opt_range=noblock もしくは global_alloc=0 を指定し、かつ struct_alloc=1 を指定した場合、ローカル構造体/共用体メンバのみレジスタ割り付けを行います。
本オプションの省略時解釈は、optimize=debug_only を指定した場合、struct_alloc=0 です。optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：構造体/共用体メンバのレジスタ割り付けを行う
V7.0.06 以降 ：構造体/共用体メンバのレジスタ割り付けを抑止する
それ以外の場合は、struct_alloc=1 です。

const 定数伝播

CONST_Var_propagate

コンパイラ<最適化>[詳細…][外部変数][外部定数の定数伝播 :]

書　式　　CONST_Var_propagate = { 0 | 1 }

説　明　　const_var_propagate=0 を指定した場合、const 修飾型外部変数の定数伝播を抑止します。
const_var_propagate=1 を指定した場合、const 修飾型外部変数についても定数伝播を行います。

例

```
const int X = 1;
int A;
void f() {
    A = X;
}

＜const_var_propagate=1 指定時のソースイメージ＞
void f() {
    A = 1;      /* X=1 を伝播 */
}
```

備　考　　C++プログラムの const 修飾型変数については本オプションで制御することはできません(常に定数伝播されます)。
本オプションの省略時解釈は、optimize=0、optimize=debug_only、または opt_range=noblock を指定した場合は const_var_propagate=0、それ以外の場合は const_var_propagate=1 です。

定数ロードの命令展開

CONST_Load

コンパイラ<最適化>[詳細…][その他][定数ロードの命令展開 :]

書　式　　CONST_Load = { Inline | Literal }

説　明　　const_load=inline を指定した場合、符号あり 2 バイト以内の定数ロードを命令展開します。
const_load=literal を指定した場合、2 バイト以上の定数ロードをリテラルアクセスします。
本オプションの省略時解釈は、以下のようになります。

オプション指定	省略時解釈
-optimize=1 かつ-speed	const_load=inline
-optimize=1 かつ-size -optimize=1 かつ-nospeed	2 バイト定数の場合は 2 命令で、4 バイト定数の場合は 3 命令で展開できる場合は const_load=inline それ以外は const_load=literal
-optimize=0 または -optimize=debug_only	const_load=literal

例

```
int f() {
    return (257);
}
```

(1) const_load=inline または speed 指定時

```
    MOV     #1,R0   ; R0 <- 1
    SHLL8   R0      ; R0 <- 256 (1<<8)
    RTS
    ADD     #1,R0   ; R0 <- 257 (256+1)
```

(2) const_load=literal または size/nospeed 指定時

```
    MOV.W   L11,R0
    RTS
    NOP
L11:
    .DATA.W H'0101
```

命令並べ替え

SChedule

コンパイラ<最適化>[詳細…][外部変数][命令並べ替え :]

書　式　　SChedule = { 0 | 1 }

説　明　　schedule=0 を指定した場合、命令並べ替えを行いません。したがって C/C++プログラムで記述した順番で処理を行います。
schedule=1 を指定した場合、パイプライン処理、スーパスカラ (SH-2A、SH2A-FPU、SH-4、SH-4A、SH4AL-DSP) を考慮した命令並べ替えを行います。
本オプション省略時解釈は、optimize=0 または optimize=debug_only を指定した場合は schedule=0、それ以外の場合は schedule=1 です。

ソフトウェアパイプライニング

SOftpipe

コンパイラ<最適化>[詳細…][その他][ソフトウェアパイプライニング :]

書　式　　SOftpipe

説　明　　ソフトウェアパイプライニングを有効にします。

備　考　　本オプションは、cpu=sh2a|sh2afpu|sh4|sh4a|sh4aldsp かつ optimize=1 指定時のみ有効です。

最適化範囲分割

SCOpe
NOSCope

コンパイラ<最適化>[詳細…][その他][最適化範囲の分割をしない :]

書　式　　SCOpe
NOSCope

説　明　　scope オプションはサイズの大きい関数について、最適化範囲を複数に分割してコンパイルします。
noscope オプションは最適化範囲を分割せずにコンパイルします。最適化範囲が広がることによりコンパイル速度は遅くなりますが、一般的にはオブジェクト性能が向上します。ただし、レジスタ数が不足するとオブジェクト性能が低下する場合があります。
本オプションは、プログラムによってオブジェクト性能に影響しますので、性能チューニング時に試してください。

GBR 相対論理演算生成

LOGIc_gbr

コンパイラ<最適化>[GBR 相対アクセスコード自動生成 :]

書　式　　LOGIc_gbr

説　明　　外部変数に対する論理演算を GBR 相対で行うコードを生成します。

備　考　　gbr=auto を指定した場合は、本オプションは無効になります。
本オプション使用時は、組み込み関数 set_gbr() で GBR に$G0 セクションの先頭アドレスを設定してください。

CPP_NOINLINE

CPP_NOINLINE

コンパイラ<最適化>[詳細…][インライン展開][C++インライン関数の展開抑止 :]

書　式　　CPP_NOINLINE

説　明　　C++ソースコンパイル時に、inline 指定子付きの関数およびクラスや構造体の中で定義したメンバ関数のインライン展開を抑止し、ファイル内に静的関数として定義した関数を呼び出すコードを生成します。

備　考　　本オプションは、C++コンパイル時のみ有効です。
inline オプションまたは speed オプションを指定した場合、もしくは#pragma inline を使用した場合、本オプションでインライン展開を抑止した関数がインライン展開されることがあります。

ポインタ指示先の型を考慮した最適化

ALIAS

コンパイラ<最適化>[詳細…][その他][ポインタ指示先の型を考慮した最適化 :]

書　式　　-ALIAS = {ANSI | NOANSI }

説　明　　alias=ansi を指定した場合、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。
一般には、alias=noansi を指定した場合よりもオブジェクト性能が向上しますが、実行結果が旧バージョンのコンパイラと異なる場合があります。
alias=noansi を指定した場合、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。
本オプションの省略時解釈は、alias=noansi です。

例

```
long x,n;
void func(short * ps)
{
   n = 1;
    *ps = 2;
    x = n;
 }

[alias=noansi 指定時]
;; *ps = 2; によって、n の値が書き換わる可能性があるとみなし
;; (A)で n の値を再ロードします。
      MOV        #1,R2      ; H'00000001
      MOV.L      L11+2,R6   ; _n
      MOV.L      R2,@R6     ; n
      MOV        #2,R2      ; H'00000002
      MOV.W      R2,@R4     ; *(ps)
      MOV.L      @R6,R2     ; n          (A) n を再ロードする
      MOV.L      L11+6,R6   ; _x
      RTS
      MOV.L      R2,@R6     ; x

[alias=ansi 指定時]
;; *ps と n は型が異なるため、*ps = 2; によって n の値は書き換わらないと
;; みなし、(B) で、n = 1 を再利用します。そのため、
;; *ps = 2; によって n の値が書き換わった場合、結果が変わります。
      MOV        #1,R2      ; H'00000001
      MOV.L      L11+2,R6   ; _n
      MOV.L      R2,@R6     ; n
      MOV        #2,R2      ; H'00000002
      MOV.W      R2,@R4     ; *(ps)
      MOV        #1,R2      ; H'00000001 (B) n = 1 を再利用する
      MOV.L      L11+6,R6   ; _x
      RTS
      MOV.L      R2,@R6     ; x
```

備　考　　本オプションは、optimize=1 を指定した場合のみ有効です。

2.2.5 その他オプション

表 2.12 その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	組み込み向け C++言語	ECpp	コンパイラ<その他> [その他のオプション :] [EC++言語に基づいたチェック]	EC++言語仕様に基づいてシンタックスチェック
2	DSP-C 言語 (SH2-DSP, SH3-DSP, SH4AL-DSP)	DSpc	コンパイラ<その他> [その他のオプション :] [DSP-C 言語に基づいたチェック]	DSP-C 言語仕様に基づいてシンタックスチェック
3	コメントのネスト	COMment = { Nest | NONest }	コンパイラ<その他> [その他のオプション :] [コメント(/* */)のネストを許す]	コメント(/* */)のネストを許す コメント(/* */)のネストを許さない
4	MAC レジスタ保証	Macsave = { 0 | 1 }	コンパイラ<その他> [その他のオプション :] [MAC レジスタを保証する]	関数呼び出し前後で MAC レジスタを保証しない 関数呼び出し前後で MAC レジスタを保証する
5	SSR/SPC レジスタ退避･回復 (SH-3～SH-4)	SAve_cont_reg={ 0 | 1 }	コンパイラ<その他> [その他のオプション :] [SSR/SPC 退避･回復]	SSR/SPC レジスタの退避･回復を行わない SSR/SPC レジスタの退避･回復を行う
6	リターン値の拡張	RTnext NORTnext	コンパイラ<その他> [その他のオプション :] [返却値の拡張を行う]	返却値の符号/ゼロ拡張する 返却値の符号/ゼロ拡張しない
7	浮動小数点定数除算の乗算化	APproxdiv	コンパイラ<その他> [その他のオプション :] [浮動小数点定数除算の乗算化]	浮動小数点定数除算の乗算化を行う
8	SH7055 不具合回避 (SH-2E)	PAtch=7055	コンパイラ<その他> [その他のオプション :] [SH7055 不具合回避]	SH7055 で不正動作となる命令並びを生成しない
9	FPSCR レジスタの切り替え (SH2A-FPU, SH-4,SH-4A)	FPScr = { Safe | Aggressive }	コンパイラ<その他> [その他のオプション :] [FPSCR レジスタの切り替え]	関数呼び出し前後で FPU の精度モードを単精度に保証する 関数呼び出し前後で FPU の精度モードを保証しない
10	ループ判定式の最適化抑止	Volatile_loop	コンパイラ<その他> [その他のオプション :] [ループ判定式の最適化抑止]	ループ判定式の最適化を抑止する
11	列挙型サイズ	AUto_enum	コンパイラ<その他> [その他のオプション :] [列挙型サイズの自動選択]	列挙型サイズの自動選択
12	register 記憶クラス変数優先割り付け	ENAble_register	コンパイラ<その他> [その他のオプション :] [register 指定変数の優先レジスタ割り付け]	register 記憶クラスを指定した変数を優先的にレジスタ割り付け
13	ANSI 準拠	STRIct_ansi	コンパイラ<その他> [その他のオプション :] [ANSI 準拠対応拡張]	以下を ANSI 準拠で行う ・unsigned int と long 型演算 ・浮動小数点演算の結合則

14	浮動小数点除算変換 (SH-2E, SH2A-FPU, SH-4,SH-4A)	FDIv	コンパイラ<その他> **[**その他のオプション :**]** **[**整数除算の浮動小数点除算置き換え**]**	整数除算を浮動小数点除算に変換する
15	浮動小数点→固定小数点変換 (SH2-DSP, SH3-DSP, SH4AL-DSP)	FIXED_Const	コンパイラ<その他> **[**その他のオプション :**]** **[**浮動小数点→固定小数点変換**]**	浮動小数点定数を固定小数点定数として扱う
16	1.0r→_ _fixed 型最大値変換 (SH2-DSP, SH3-DSP, SH4AL-DSP)	FIXED_Max	コンパイラ<その他> **[**その他のオプション :**]** **[**1.0r→_ _fixed 型最大値変換**]**	1.0r(R)を(long)_ _fixed 型最大値として扱う
17	_ _fixed 乗算結果の型変換省略 (SH2-DSP, SH3-DSP, SH4AL-DSP)	FIXED_Noround	コンパイラ<その他> **[**その他のオプション :**]** **[**_ _fixed 型乗算結果の型変換省略**]**	_ _fixed 型乗算での演算結果に対して型変換を省略
18	DSP リピートループ (SH3-DSP, SH4AL-DSP)	REPeat	コンパイラ<その他> **[**その他のオプション :**]** **[**DSP 拡張リピートループ**]**	DSP 拡張リピートループを使用
19	浮動小数点数-整数変換時の範囲チェック省略 (SH-2E, SH2A-FPU, SH-4,SH-4A)	SIMple_float_conv	コンパイラ<その他> **[**その他のオプション :**]** **[**浮動小数点数-整数変換時の範囲チェック省略**]**	符号なし整数型と浮動小数点型の間の型変換に対して、変換対象の値の範囲のチェックを省略したコードを生成
20	DIVS, DIVU 命令生成抑止	NOUSE_DIV_INST	コンパイラ<その他> **[**その他のオプション :**]** **[**DIVS, DIVU 命令生成抑止**]**	除算、剰余算を DIVS, DIVU 命令で生成しない
21	浮動小数点演算式の演算順序変更	FLOAT_ORDER	コンパイラ<その他> **[**その他のオプション :**]** **[**浮動小数点演算式の演算順序変更を積極的に行なう**]**	浮動小数点演算式の演算順序変更を積極的に行なう

組み込み向け C++言語

ECpp

コンパイラ<その他>[その他のオプション :][EC++言語に基づいたチェック]

書　式　　ECpp

説　明　　Embedded C++言語仕様に基づいて、C++プログラムのシンタックスチェックを行います。
Embedded C++言語仕様では、catch、const_cast、dynamic_cast、explicit、mutable、namespace、reinterpret_cast、static_cast、template、throw、try、typeid、typename、using をサポートしていません。これらのキーワードを記述した場合、エラーメッセージを出力します。

備　考　　Embedded C++言語仕様では、多重継承、仮想基底クラスをサポートしていません。
多重継承、仮想基底クラスを記述した場合は、エラーメッセージ
"C5882 (E) Embedded C++ does not support multiple or virtual inheritance"
を出力します。
本オプションは、exception オプションと同時に指定することはできません。

DSP-C 言語

DSpc

コンパイラ<その他>[その他のオプション :][DSP-C 言語に基づいたチェック]

書　式　　DSpc

説　明　　DSP-C 言語仕様に基づいて、DSP-C プログラムのシンタックスチェックを行います。
DSP-C 言語仕様についての詳細は、「10.2　DSP-C 仕様」を参照してください。

備　考　　本オプションは、cpu=sh2dsp|sh3dsp|sh4aldsp 以外のときは指定することはできません。
本オプションは、C++プログラムに指定することはできません。

コメントのネスト

COMment

コンパイラ<その他>[その他のオプション :][コメント(/* */)のネストを許す]

書　式　　COMment = { Nest | NONest }

説　明　　comment=nest を指定した場合、ネストしたコメントの記述を可能にします。
comment=nonest を指定した場合、コメントのネストを記述するとエラーになります。
本オプションの省略時解釈は、comment=nonest です。

例

```
/* This is an example of /* nested */ comment */
                                      ↑
                                     [1]
```

comment=nest を指定した場合は全てコメントと解釈しますが、comment=nonest を指定した場合は[1]でコメントが終わっていると解釈します。

MAC レジスタ保証

Macsave

コンパイラ<その他>[その他のオプション :][MAC レジスタを保証する]

書　式　　Macsave = { 0 | 1 }

説　明　　MACH、MACL レジスタを関数の呼び出し前後で保証するかどうかを指定します。
macsave=0 を指定した場合、関数の呼び出し前後で MACH、MACL レジスタを保証しません。
macsave=1 を指定した場合、関数の呼び出し前後で MACH、MACL レジスタを保証します。
macsave=1 でコンパイルした関数から macsave=0 でコンパイルした関数を呼び出すことはできません。逆に macsave=0 でコンパイルした関数から macsave=1 でコンパイルした関数を呼び出すことは可能です。
本オプションの省略時解釈は、macsave=1 です。

SSR/SPC レジスタ退避・回復

SAve_cont_reg

コンパイラ<その他>[その他のオプション :][SSR/SPC 退避･回復]

書　式　　SAve_cont_reg = { 0 | 1 }

説　明　　SSR/SPC レジスタの退避・回復を行うかどうかを指定します。
save_cont_reg=0 を指定した場合、SSR/SPC レジスタの退避・回復を行いません。
save_cont_reg=1 を指定した場合、SSR/SPC レジスタの退避・回復を行います。
本オプションは、cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp を指定した時に、#pragma interrupt 指定した関数に対してのみ有効です。
本オプションの省略時解釈は、save_cont_reg=1 です。

リターン値拡張

RTnext
NORTnext

コンパイラ<その他>[その他のオプション :][返却値の拡張を行う]

書　式　　RTnext
NORTnext

説　明　　関数原型がある関数について､返却値の型が char, signed char, unsigned char, short, signed short, または unsigned short 型のいずれかである場合に、返却値レジスタ R0 の符号拡張あるいはゼロ拡張を行うかどうか指定します。
rtnext オプションを指定した場合、関数返却値の符号/ゼロ拡張を行います。
nortnext オプションを指定した場合、関数返却値の符号/ゼロ拡張を行いません。
本オプションの省略時解釈は、nortnext です。

浮動小数点定数除算の乗算化

APproxdiv

コンパイラ<その他>[その他のオプション :][浮動小数点定数除算の乗算化]

書　式　　APproxdiv

説　明　　浮動小数点定数除算を、定数の逆数の乗算に変換します。

備　考　　本オプションを指定した場合、浮動小数点定数除算の実行速度は向上しますが、演算の精度が変わる場合がありますので注意が必要です。

SH7055 不具合回避

PAtch

コンパイラ<その他>[その他のオプション :][SH7055 不具合回避]

書　式　　PAtch = 7055

説　明　　SH7055 で不正動作となる命令並びをコンパイラで出力しません。

備　考　　本オプションは、cpu=sh2e を指定した場合のみ有効です。

FPSCR レジスタ精度モード切り替え

FPScr

コンパイラ<その他>[その他のオプション :][FPSCR レジスタの切り替え]

書　式　　FPScr = { Safe | Aggressive }

説　明　　FPSCR レジスタの精度モードを関数呼び出し前後で保証するかどうか指定します。
SH2A-FPU、SH-4、SH-4A では float 演算、double 演算を実行するときに FPSCR レジスタの精度モードを単精度、倍精度に設定します。
fpscr=safe を指定した場合、関数呼び出しから戻ったときの FPSCR レジスタの精度モードは常に単精度モードになります。
fpscr=aggressive を指定した場合、関数呼び出しから戻ったときの FPSCR レジスタの精度モードの値は保証しません。
本オプションは、cpu=sh2afpu|sh4|sh4a で fpu=single、fpu=double のどちらの指定もないときに有効です。
本オプションの省略時解釈は、fpscr=aggressive です。

ループ判定式最適化抑止

Volatile_loop

コンパイラ<その他>[その他のオプション :][ループ判定式の最適化抑止]

書　式　　Volatile_loop

説　明　　ループ判定式に外部変数を含む場合、最適化対象外にします。
ただし、型変換を伴う場合、外部変数を 2 つ以上含む場合または複合演算の場合は、最適化抑止対象にならない場合があります。

備　考　　本オプションを指定しなかった場合、ループ判定式がループ内で不変の時に削除される場合があります。

列挙型サイズ

AUto_enum

コンパイラ<その他>[その他のオプション :][列挙型サイズの自動選択]

書　式　　AUto_enum

説　明　　enum 宣言した列挙型のデータを、列挙値が収まる最小型として処理します。
本オプションの省略時解釈は、列挙型サイズを int 型として処理します。
列挙型のとりうる値と型の関係を**表 2.13** に示します。

表 2.13　列挙型のとりうる値と型の関係

列挙子		型
最小値	最大値	
-128	127	signed char
0	255	unsigned char
-32768	32767	signed short
0	65535	unsigned short
上記以外		int

***register** 記憶クラス変数優先割り付け*

ENAble_register

コンパイラ<その他>[その他のオプション :][register 指定変数の優先レジスタ割り付け]

書　式　　ENAble_register

説　明　　register 記憶クラスを指定した変数を優先的にレジスタに割り付けます。

備　考　　レジスタに割り付かなかった場合は、インフォメーションメッセージ
C0102 (I) Register is not allocated to "変数名" in "関数名"
を出力します。ただし、引数がレジスタに割り付かなかった場合は、本メッセージは出力しません。

ANSI 準拠

STRIct_ansi

コンパイラ<その他>[その他のオプション :][ANSI 準拠対応拡張]

書　式　　STRIct_ansi

説　明　　以下処理を ANSI 準拠で行います。

(1) unsigned int と long 型演算

例：

```
long sl;
unsigned int ui;
sl /= ui;        /* strict_ansi 未指定時は long、
                    指定時は unsigned int で演算します */
```

(2) 浮動小数点演算の結合則

備　考　　本オプションを指定した場合、演算結果が旧バージョンのコンパイラと異なる場合があります。

浮動小数点除算変換

FDIv

コンパイラ<その他>[その他のオプション :][整数除算の浮動小数点除算置き換え]

書　式　　FDIv

説　明　　整数除算を浮動小数点除算に変換します。これにより、除算の実行速度を向上させることができます。

備　考　　本オプションは、Cpu=Sh2e|Sh2afpu|Sh4|Sh4a を指定した場合のみ有効です。
本オプション指定は、ifunc オプション指定時および#pragma ifunc を指定した関数に対しては無効になります。
cpu=sh2afpu|sh4|sh4a かつ fpu=double を指定した場合は除数、被除数が共に 4byte 以内のときに、それ以外の場合は除数、被除数が共に 2byte 以内のときに変換を行います。

浮動小数点→固定小数点変換

FIXED_Const

コンパイラ<その他>[その他のオプション :][浮動小数点→固定小数点変換]

書　式　　FIXED_Const

説　明　　本オプションを指定した場合、浮動小数点型定数を固定小数点型定数としてオブジェクトを生成します。

備　考　　本オプションは cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ有効です。
明示的に浮動小数点型定数の表現形式で表現されている場合は、本オプション指定しても浮動小数点型定数としてオブジェクトを生成します。

1.0r→ _ _fixed 型最大値変換

FIXED_Max

コンパイラ<その他>[その他のオプション :][1.0r→_ _fixed 型最大値変換]

書　式　　FIXED_Max

説　明　　本オプションを指定した場合、1.0r と記述すると_ _fixed 型の最大値としてオブジェクトを生成し、1.0R と記述すると long__fixed 型の最大値としてオブジェクトを生成します。
最大値の詳細は「10.4.1(8)　fixed.h」を参照してください。

備　考　　本オプションは cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ有効です。

_ _fixed 乗算結果の型変換省略

FIXED_Noround

コンパイラ<その他>[その他のオプション :][_ _fixed 型乗算結果の型変換省略]

書　式　　FIXED_Noround

説　明　　本オプションを指定した場合、_ _fixed 型乗算で long _ _fixed 型となった演算結果に対して _ _fixed 型への型変換を省略します。

備　考　　本オプションを指定すると、演算結果の精度が変わる可能性があります。
本オプションは cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ有効です。

DSP 拡張リピートループ

REPeat

コンパイラ<その他>[その他のオプション :][DSP 拡張リピートループ]

書　式　　REPeat

説　明　　repeat オプションを指定した場合、ループを DSP 拡張リピートループを使用したコードに展開する場合があります。

備　考　　拡張リピートループは LDRC 命令をサポートしたマイコンのみ使用することができます。
本オプションは cpu=sh3dsp|sh4aldsp を指定した場合のみ有効です。

浮動小数点数-整数変換時の範囲チェック省略

SIMple_float_conv

コンパイラ<その他>[その他のオプション :][浮動小数点数-整数変換時の範囲チェック省略]

書　式　SIMple_float_conv

説　明　符号なし整数型と浮動小数点型の間の型変換に対して、変換対象の値の範囲のチェックを省略したコードを生成します。

例 1

```
unsigned long func(float f)
{
        return ((unsigned int)f);
}
```

[オプション未使用時]

```
        MOV         #79,R2     ; 0x0000004F
        SHLL8       R2
        SHLL16      R2         ; 0x4F000000
        LDS         R2,FPUL
        FSTS        FPUL,FR8
        FCMP/GT     FR4,FR8
        BT          L12
        FADD        FR8,FR8  ; f ≧ 0x4F000000 の場合、
        FSUB        FR8,FR4  ; 変換前の値を(f - 0x4F800000)とする
L12:
        FTRC        FR4,FPUL ; float から signed long への変換
        STS         FPUL,R0
```

[オプション使用時]

```
        FTRC        FR4,FPUL ; float 型 から signed long 型への変換
        STS         FPUL,R0
```

例 2

```
float func2(unsigned long u)
{
        return ((float)u);
}
```

[オプション未使用時]

```
        LDS         R4,FPUL
        CMP/PZ      R4
        BT/S        L12
        FLOAT       FPUL,FR0   ; signed long から float への変換
        MOVA        L13+2,R
        FMOV.S      @R0,FR9    ; u ≧ 0x80000000u の場合、
        FADD        FR9,FR0    ; 変換後の値に 0x4F800000 を加算する。
L12:
        RTS
        NOP
L13:
        RES.W      1
        DATA.L     H'4F800000
```

[オプション使用時]

```
        LDS         R4,FPUL
        RTS
        FLOAT       FPUL,FR0   ; signed long から float への変換
```

備　考　　本オプションは、cpu=sh2e|sh2afpu|sh4|sh4a を指定した場合のみ有効です。
型変換前の値が 0 以上 2147483647 以下の整数値もしくは 0.0 以上 2147483647.0 以下の浮動小数点数値でない場合は動作を保証しません。これらの範囲外の値を使用する場合は本オプションを使用しないでください。

DIVU, DIVS 命令生成抑止

NOUSE_DIV_INST

コンパイラ<その他>[その他のオプション :][DIVS, DIVU 命令生成抑止]

書　式　　-NOUSE_DIV_INST

説　明　　プログラム中の全ての整数除算、および剰余算を、DIVS 命令、および DIVU 命令を使わないコードに展開します。
本オプションは、cpu=sh2a | sh2afpu を指定した場合のみ有効です。

浮動小数点演算式の演算順序変更

FLOAT_ORDER

コンパイラ<その他>[その他のオプション :][浮動小数点演算式の演算順序変更を積極的に行なう]

書　式　　-FLOAT_ORDER

説　明　　浮動小数点演算式の演算順序変更の最適化を積極的に行ないます。一般には、float_order を指定しない場合よりも
オブジェクト性能が向上しますが、演算の精度が旧バージョンのコンパイラと異なる場合があります。

例

```
/* -float_order 指定ありの場合、(b + c) * 100.0f で行う*/
float a,b,c;
f()
{
        a = b * 100.0f + c * 100.0f;
}
```

備　考　　本オプションは、optimize=1 を指定した場合のみ有効です。

2.2.6 マイコンオプション

表 2.14 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	マイコン/ 動作モード	CPu = { SH1 | SH2 | SH2E | SH2A | SH2AFPU | SH2DSP | SH3 | SH3DSP | SH4 | SH4A | SH4ALDSP }	CPU [CPU 種別 :]	SH-1 のオブジェクトを生成 SH-2 のオブジェクトを生成 SH-2E のオブジェクトを生成 SH-2A のオブジェクトを生成 SH2A-FPU のオブジェクトを生成 SH2-DSP のオブジェクトを生成 SH-3 のオブジェクトを生成 SH3-DSP のオブジェクトを生成 SH-4 のオブジェクトを生成 SH-4A のオブジェクトを生成 SH4AL-DSP のオブジェクトを生成
2	メモリのバイト並び順 (SH-2, SH-3～SH-4)	ENdian = { Big | Little }	CPU [Endian 選択 :]	Big Endian Little Endian
3	浮動小数点演算モード (SH2A-FPU, SH-4,SH-4A)	FPu = { Single | Double }	CPU [浮動小数点演算モード :]	倍精度浮動小数点演算を単精度で演算 単精度浮動小数点演算を倍精度で演算
4	丸め方式 (SH2A-FPU, SH-4,SH-4A)	Round = { Zero | Nearest }	CPU [丸め方式 :]	Round to Zero で丸める Round to Nearest で丸める
5	非正規化数の扱い (SH-4,SH-4A)	DENormalize = { OFF | ON }	CPU [非正規化数を非正規化数として扱う]	非正規化数を 0 として扱う 非正規化数を非正規化数として扱う
6	プログラムセクションポジションインディペンデント (SH-1 以外)	Pic = { 0 | 1 }	CPU [ポジションインディペンデントコード生成]	プログラムセクションのポジションインディペンデントコードを生成しない プログラムセクションのポジションインディペンデントコードを生成する
7	double→float 変換 (SH2A-FPU, SH-4,SH-4A 以外)	DOuble = Float	CPU [double→float 変換]	double 型の変数を float 型として扱う
8	ビットフィールド並び順指定	BIt_order={ Left | Right }	CPU [ビットフィールドメンバを下位 bit から格納]	ビットフィールドメンバを上位 bit から格納 ビットフィールドメンバを下位 bit から格納

9	構造体、共用体、クラスメンバのアライメント数	PACK={ 1 | 4 }	CPU [構造体メンバの境界調整数を 1 とする]	アライメント数を 1 とする データのアライメントに従う
10	例外処理機能	EXception NOEXception	CPU [C++の try、throw、catch を有効にする]	例外処理機能を有効にする 例外処理機能を無効にする
11	実行時型情報	RTTI = { ON | OFF }	CPU [C++の dynamic_cast、typeid を有効にする]	dynamic_cast、typeid を有効にする dynamic_cast、typeid を無効にする
12*	除算方式選択 (SH-2 のみ)	DIvision = { Cpu | Peripheral | Nomask }	CPU [除算方式選択 :]	マイコンの除算命令を使用 除算器を使用 (割り込みマスクあり) 除算器を使用 (割り込みマスクなし)

【注】* オプションの詳細は「2.2.2 オブジェクトオプション」を参照してください。

マイコン種別

CPu

CPU[CPU 種別 :]

書　式

```
CPu = { SH1      |
        SH2      |
        SH2E     |
        SH2A     |
        SH2AFPU  |
        SH2DSP   |
        SH3      |
        SH3DSP   |
        SH4      |
        SH4A     |
        SH4ALDSP }
```

説　明　　作成するオブジェクトプログラムのマイコン種別を指定します。サブオプションの一覧を表 2.15 に示します。
本オプションの省略時解釈は、cpu=sh1 です。

表 2.15　cpu オプションのサブオプション一覧

	サブオプション名	意　味
1	sh1	SH-1 のオブジェクトを作成します。
2	sh2	SH-2 のオブジェクトを作成します。
3	sh2e	SH-2E のオブジェクトを作成します。
4	sh2a	SH-2A のオブジェクトを作成します。
5	sh2afpu	SH2A-FPU のオブジェクトを作成します。
6	sh2dsp	SH2-DSP のオブジェクトを作成します。
7	sh3	SH-3 のオブジェクトを作成します。
8	sh3dsp	SH3-DSP のオブジェクトを作成します。
9	sh4	SH-4 のオブジェクトを作成します。
10	sh4a	SH-4A のオブジェクトを作成します。
11	sh4aldsp	SH4AL-DSP のオブジェクトを作成します。

メモリのバイト並び順

ENdian

CPU[Endian 選択 :]

書　式　　ENdian = { Big | Little }

説　明　　endian=big を指定した場合、データのバイト並びが Big Endian になります。
endian=little を指定した場合、データのバイト並びが Little Endian になります。
Little Endian のオブジェクトプログラムは、SH-1、SH-2E、SH-2A、SH2A-FPU、SH2-DSP では実行できません。
本オプションの省略時解釈は、endian=big です。

浮動小数点演算モード

FPu

CPU[浮動小数点演算モード :]

書　式　　FPu = { Single | Double }

説　明　　fpu=single を指定した場合、倍精度浮動小数点演算を単精度で演算します。
fpu=double を指定した場合、単精度浮動小数点演算を倍精度で演算します。
プログラム中に浮動小数点演算がない場合には、fpu=single を指定してください。
本オプションは、cpu=sh2afpu|sh4|sh4a を指定した場合のみ有効です。

注　意　　fpu オプション未指定時、あるいは fpu=single 指定時に、割り込み関数内で単精度浮動小数点演算を行う時は精度モードの設定が必要になる場合があります。詳細は「9.4.1(6) マイコン種別が SH2A-FPU|SH4|SH4A の場合の割り込み関数について」を参照してください。

丸め方式

Round

CPU[丸め方式 :]

書　式　　Round = { Zero | Nearest }

説　明　　浮動小数点定数をオブジェクトコードに変換する際の丸め方式を指定します。
round=zero を指定した場合、Round to Zero で丸めます。
round=nearest を指定した場合、Round to Nearest で丸めます。
本オプションは、cpu=sh2afpu|sh4|sh4a を指定した場合のみ有効です。
本オプションの省略時解釈は、round=zero です。

非正規化数の扱い

DENormalize

CPU[非正規化数を非正規化数として扱う]

書　式　　DENormalize = { OFF | ON }

説　明　　浮動小数点定数に非正規化数を記述した場合の扱いを指定します。
denormalize=off を指定した場合、非正規化数を 0 として扱います。
denormalize=on を指定した場合、非正規化数を非正規化数として扱います。
本オプションは、cpu=sh4|sh4a を指定した場合のみ有効です。
本オプションの省略時解釈は、denormalize=off です。

ポジションインディペンデントコード

Pic

CPU[ポジションインディペンデントコード生成]

書　式　　Pic = { 0 | 1 }

説　明　　pic=1 を指定した場合、リンク後のプログラムセクションを任意のアドレスに配置して実行できます。データセクションはリンク時に決定したアドレス以外には配置できません。ポジションインディペンデントコードとして実行する場合は、関数のアドレスを初期値として指定することはできません。C++コンパイルでは、仮想関数、関数メンバへのポインタも関数のアドレスを初期値として必要とするため、仮想関数やメンバ関数へのポインタを含んだ C++プログラムは、ポジションインディペンデントコードとして実行できません。

例 1

```
extern int f();
int (*fp)() = f;    ←指定不可
```

例 2

```
struct A{virtual void f();}; ←指定不可
void (A::*ap)() = &A::f; ←指定不可
```

cpu=sh1 を指定した場合は、pic=1 指定を無視します。
本オプションの省略時解釈は、pic=0 です。

double→float 変換

DOuble = Float

CPU[double→float 変換]

書　式　　DOuble = Float

説　明　　double(倍精度浮動小数点)型の数値を float(単精度浮動小数点)型としてオブジェクトを生成します。

備　考　　cpu=sh2afpu|sh4|sh4a を指定した場合、本オプションを指定すると本オプションは無効となり、fpu=single が指定されたとみなします。

ビットフィールド並び順指定

BIt_order

CPU[ビットフィールドメンバを下位 bit から格納]

書　式　　BIt_order = { Left | Right }

説　明　　ビットフィールドのメンバの並び順を指定します。
bit_order=left を指定した場合は上位ビットからメンバを割り付けます。
bit_order=right を指定した場合は下位ビットからメンバを割り付けます。
本オプション省略時解釈は、bit_order=left です。

備　考　　ビットフィールドのメンバの割り付けについては、「10.1.2 データの内部表現」、および「10.3.1 #pragma」の#pragma bit_order を参照してください。

構造体、共用体、クラスメンバのアライメント数

PACK

CPU[構造体メンバの境界調整数を 1 とする]

書　式　　PACK = { 1 | 4 }

説　明　　構造体、共用体、クラスメンバのアライメント数を指定します。
構造体メンバのアライメント数は、#pragma pack 拡張子でも指定できます。オプションと#pragma の両方が指定された場合には、拡張子の指定を優先します。
構造体、共用体、クラスのアライメント数は、メンバの最大のアライメント数と同じになります。
詳細は「10.1.2(2) 構造体/共用体(C 言語)、クラス型(C++言語)」を参照してください。
本オプションの省略時解釈は、pack=4 です。

備　考　　ルネサス統合開発環境で作成された iodefine.h を使用する場合、#pragma もしくはオプションでアライメント数を 1 にすると、I/O レジスタ用構造体のメンバが示すアドレスが正しいアドレスを指しません。iodefine.h の先頭に、#pragma pack 4 を、iodefine.h の最後に、#pragma unpack を記述してください。
pack オプション指定時の構造体メンバのアライメント数を表 2.16 に示します。

表 2.16　pack オプション指定時の構造体、共用体、クラスメンバのアライメント数

メンバの型	pack=1	pack=4	指定なし
(unsigned)char	1	1	1
(unsigned)short、__fixed	1	2	2
(unsigned)int、(unsigned)long、(unsigned)long long、long __fixed、__accum、long __accum、浮動小数点型、ポインタ型	1	4	4
アライメント数が 1 の構造体、共用体、クラス	1	1	1
アライメント数が 2 の構造体、共用体、クラス	1	2	2
アライメント数が 4 の構造体、共用体、クラス	1	4	4

例外処理機能

EXception
NOEXception

CPU[C++の try、throw、catch を有効にする]

書　式　　EXception
　　NOEXception

説　明　　exception オプションを指定した場合、C++例外処理機能(try,catch,throw)を有効にします。
noexception オプションを指定した場合、C++例外処理機能(try,catch,throw)を無効にします。
exception オプションを指定した場合、コード性能が低下する可能性があります。
本オプション省略時解釈は、noexception です。

備　考　　ファイル間で例外処理機能を有効にするには以下を行ってください。
(1) rtti=on を指定する。
(2) 最適化リンケージエディタで noprelink オプションを指定しない。
exception オプションと ecpp オプションを同時に指定することはできません。
exception オプションを指定して作成したオブジェクトファイルをライブラリに登録したり、最適化リンケージエディタでリロケータブル形式で出力しないでください。シンボルの二重定義エラーや未定義エラーになることがあります。

実行時型情報

RTTI

CPU[C++の dynamic_cast、typeid を有効にする]

書　式　　RTTI = { ON | OFF }

説　明　　実行時型情報の有効/無効を指定します。
rtti=on を指定した場合、dynamic_cast、typeid を有効にします。
rtti=off を指定した場合、dynamic_cast、typeid を無効にします。
本オプション省略時解釈は、rtti=off です。

備　考　　本オプションを指定して作成したオブジェクトファイルをライブラリに登録したり、最適化リンケージエディタでリロケータブル形式で出力しないでください。シンボルの二重定義エラーや未定義エラーになることがあります。

2.2.7 残りのオプション

表 2.17 残りのオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	C/C++言語の選択	LAng = { C | CPp }	- (拡張子で判断)	C プログラムとしてコンパイル C++プログラムとしてコンパイル
2	コピーライト出力抑止	LOGO NOLOGO	- (常に nologo が有効)	コピーライトを出力 コピーライトの出力を抑止
3	文字列内の文字コード	Euc SJis LATin1	コンパイラ<その他> [ユーザ指定オプション :]	euc コードを選択 sjis コードを選択 latin1 コードを選択
4	オブジェクト内漢字コード指定	OUtcode = { EUc | SJis }	コンパイラ<その他> [ユーザ指定オプション :]	euc コード sjis コード
5	サブコマンドファイルの指定	SUbcommand = <ファイル名>	-	<ファイル名>で指定したファイルからコマンドオプションを取りこむ

C/C++言語選択

LAng

なし（常に拡張子で判断）

書　式　　LAng = { C | CPp }

説　明　　ソースプログラムの言語を指定します。
lang=c を指定した場合、C プログラムとしてコンパイルします。
lang=cpp を指定した場合、C++プログラムとしてコンパイルします。
本オプションを省略した場合は、ソースプログラムの拡張子によって判断します。拡張子が c のときには C プログラムとしてコンパイルします。また、拡張子が cpp、cc、cp のときには C++プログラムとしてコンパイルします。ソースプログラムの拡張子を指定しなかった場合は、C プログラムとしてコンパイルします。

例

```
shc test.c              C プログラムとしてコンパイルします。
shc test.cpp            C++プログラムとしてコンパイルします。
shc -lang=cpp test.c    C++プログラムとしてコンパイルします。
shc test                test.c を仮定し、C プログラムとしてコンパイルします。
```

備　考　　lang=c を指定した場合、ecpp オプションは無効になります。

コピーライト出力制御

LOGO
NOLOGO

なし(常に nologo が有効)

書　式　　LOGO
NOLOGO

説　明　　コピーライトの出力を抑止します。
logo オプションを指定した場合、コピーライト表示を出力します。
nologo オプションを指定した場合、コピーライト表示の出力を抑止します。
本オプション省略時解釈は、logo です。

文字列内文字コード

Euc
SJis
LATin1

コンパイラ<その他>[ユーザ指定オプション :]

書　式　　Euc
SJis
LATin1

説　明　　文字列、文字定数およびコメント内に日本語または ISO-Latin1 コードを記述できます。
ホストマシンと文字列内コードとの関係を表 2.18 に示します。

表 2.18　ホストマシンと文字列内コード

ホストマシン	オプション指定			
	euc	sjis	latin1	指定なし
PC	euc	sjis	latin1	sjis
SPARC	euc	sjis	latin1	euc
HP9000/700	euc	sjis	latin1	sjis

備　考　　latin1 オプションを指定した場合、outcode オプションは無効になります。

オブジェクト内漢字コード指定

OUtcode

コンパイラ<その他>[ユーザ指定オプション :]

書　式　　OUtcode = { EUc | SJis }

説　明　　文字列、文字定数内に日本語を記述したときに、オブジェクトプログラムに出力する漢字コードを指定します。
outcode=euc は、漢字コードを euc コードで出力します。
outcode=sjis は、漢字コードを sjis コードで出力します。
ソースプログラム上の漢字コードは、euc または sjis オプションで指定できます。

サブコマンドファイル指定

SUbcommand

なし

書　式　　SUbcommand ＝ <ファイル名>

説　明　　subcommand オプションは、コンパイラ起動時のコンパイラオプションをサブコマンドファイルで指定します。サブコマンドファイル中の書式は、コマンドラインの書式と同一です。

例

```
opt.sub              : -listfile -show=object -debug
コマンドライン指定   : shc -cpu=sh4 -subcommand=opt.sub test.c
コンパイラ解釈       : shc -cpu=sh4 -listfile -show=object -debug test.c
```

3. アセンブラ操作方法

3.1 オプション指定規則

アセンブラを起動するコマンドラインの形式は以下のとおりです。

asmsh [△<オプション> …] [△<ファイル名> [, …*]] [△<オプション> …]
<オプション>：-<オプション> [=<サブオプション> [, …]]

【注】* 複数のソースファイル名を指定すると、それらのファイルを指定順に連結したものがアセンブル処理の単位になります。この場合、.END は最後のファイルにだけ記述してください。

3.2 オプション解説

コマンドライン形式の英大文字は短縮形を、下線は省略時解釈を示します。
また、統合開発環境の対応するダイアログメニューをタブ名<カテゴリ名> [項目] …で示します。オプションの順序は統合開発環境のタブとその中のカテゴリに対応しています。

3.2.1 ソースオプション

表 3.1 ソースカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	インクルードファイルフォルダ	Include = <パス名>[, …]	アセンブラ<ソース> [オプション項目:] [インクルードファイルディレクトリ]	インクルードファイルパス名を指定します。
2	置換シンボルの定義	DEFine = <sub>[, …] <sub>： <置換シンボル> = "<文字列>"	アセンブラ<ソース> [オプション項目:] [シンボル定義]	文字列の置き換えを定義します。
3	整数型プリプロセッサ変数の定義	ASsignA = <sub>[, …] <sub>： <変数名> = <整数定数>	アセンブラ<ソース> [オプション項目:] [プリプロセッサ変数定義]	整数型のプリプロセッサ変数を定義します。
4	文字型プリプロセッサ変数の定義	ASsignC = <sub>[, …] <sub>： <変数名> = "<文字列>"	アセンブラ<ソース> [オプション項目:] [プリプロセッサ変数定義]	文字型のプリプロセッサ変数を定義します。

インクルードファイルフォルダ

Include

アセンブラ<ソース>[オプション項目 :][インクルードファイルディレクトリ]

書　式　　Include = <パス名>[,…]

説　明　　インクルードするファイルのフォルダ名を指定します。
フォルダ名はホストマシンの標準的な指定方法に従います。
フォルダ名の指定数はコマンドラインで 1 行入力可能な限り有効です。
検索は、まずカレントフォルダ、続いて include オプションで指定したフォルダを指定した順序に従って行います。

例　　asmsh aaa.src -include=C:¥common,C:¥local
aaa.src 内で.INCLUDE "file.h"を指定した場合、file.h をカレントフォルダ、C:¥common、C:¥local の順に検索します。

備　考　　アセンブラ制御文との関係

オプション	制御文	結　果
include	(指定に関わらず)	.INCLUDE で指定したフォルダ
		include オプションで指定したフォルダ*
(指定なし)	.INCLUDE <ファイル名>	.INCLUDE で指定したフォルダ

【注】* .INCLUDE で指定したフォルダ文字列の前に include オプションで指定したディレクトリ文字列を付加したフォルダ名を使用します。

置換シンボル定義

DEFine

アセンブラ<ソース>[オプション項目 :][シンボル定義]

書　式　　DEFine = <sub>[,…]
<sub>：<置換シンボル> = "<文字列>"

説　明　　プリプロセッサで置換シンボルを対応する文字列に置き換えます。
define オプションと assignc オプションの機能の違いは、.DEFINE と.ASSIGNC の機能の違いに対応します。

備　考　　アセンブラ制御文との関係

オプション	制御文	結　果
define	.DEFINE*	define オプションで指定した文字列
	(指定なし)	define オプションで指定した文字列
(指定なし)	.DEFINE	.DEFINE で指定した文字列

【注】* define オプションで置換シンボルに文字列を設定した場合、当該置換シンボルへの.DEFINE による定義がすべて無効になります。また、".AENDI", ".AENDR",".AENDW", ".AIFDEF" , ".END", ".ENDM"の 6 制御命令は置換されません。

整数型プリプロセッサ変数定義

ASsignA

アセンブラ<ソース>[オプション項目 :][プリプロセッサ変数定義]

書　式

```
ASsignA = <sub>[,…]
      <sub>:<プリプロセッサ変数名> = <整数定数>
```

説　明

プリプロセッサ変数に整数定数を設定します。
プリプロセッサ変数名の書き方はシンボル名の書き方と同じです。
整数定数は基数(B'、Q'、D'、H')と数値を組み合わせて指定します。基数を省略し、数値のみを指定した場合は10進数として扱います。
整数定数に指定できる値の範囲は-2,147,483,648～4,294,967,295です。ただし、負の値を設定する場合は10進以外の基数で指定してください。

アセンブラ制御文との関係

オプション	制御文	結　果
assigna	.ASSIGNA*	assigna オプションで指定した値
	(指定なし)	assigna オプションで指定した値
(指定なし)	.ASSIGNA	.ASSIGNA で指定した値

【注】* assigna オプションでプリプロセッサ変数に値を設定した場合、当該プリプロセッサへの.ASSIGNA による定義が無効になります。

例

```
asmsh  aaa.src -assigna=_$=H'FF
```

プリプロセッサ変数_$に値H'FFを設定します。ソースプログラム内のプリプロセッサ変数_$のすべての参照箇所¥&_$をH'FFに設定します。

文字型プリプロセッサ変数定義

ASsignC

アセンブラ<ソース>[オプション項目 :][プリプロセッサ変数定義]

書　式

```
ASsignC = <sub>[,…]
      <sub>:<プリプロセッサ変数名> = "<文字列>"
```

説　明

プリプロセッサ変数に文字列を設定します。
プリプロセッサ変数名の書き方はシンボル名の書き方と同じです。
文字列は文字をダブルクォーテーション(")で囲んで指定します。
文字列には 255 文字まで指定できます。

アセンブラ制御文との関係

オプション	制御文	結　果
assignc	.ASSIGNC*	assignc オプションで指定した文字列
	(指定なし)	assignc オプションで指定した文字列
(指定なし)	.ASSIGNC	.ASSIGNC で指定した文字列

【注】* assignc オプションでプリプロセッサ変数に文字列を設定した場合、当該プリプロセッサ変数への.ASSIGNC による定義がすべて無効になります。

例

```
asmsh  aaa.src -assignc=_$="ON!OFF"
```

プリプロセッサ変数_$に文字列 ON!OFF を設定します。ソースプログラム内のプリプロセッサ変数_$のすべての参照箇所¥&_$を文字列 ON!OFF に設定します。

3.2.2 オブジェクトオプション

表 3.2 オブジェクトカテゴリオプション一覧

項目	コマンドライン形式	ダイアログメニュー	指定内容
1 デバッグ情報の出力制御	Debug NODebug	アセンブラ <オブジェクト> [デバッグ情報出力:]	デバッグ情報の出力を制御します。
2 プリプロセッサの展開結果出力	EXPand [= <出力ファイル名>]	アセンブラ <オブジェクト> [プリプロセッサ展開結果出力]	プリプロセッサの展開結果を出力します。
3 リテラルプール出力ポイントの指定	LITERAL = <point> [, …] <point> : { Pool | Branch | Jump | Return }	アセンブラ <オブジェクト> [リテラルテーブル出力位置指定:]	リテラルプールを出力する場所を指定します。
4 オブジェクトモジュールの出力制御	Object [= <出力ファイル名>] NOObject	アセンブラ <オブジェクト> [オブジェクト出力ディレクトリ:]	オブジェクトモジュールの出力を制御します。
5 未解決シンボルのサイズ指定 (SH-2A, SH2A-FPU)	DIspsize={4|12}	アセンブラ ＜オブジェクト＞ [未解決シンボルサイズ指定]	未解決シンボルのサイズを指定します。

デバッグ情報

Debug
NODebug

アセンブラ<オブジェクト>[デバッグ情報出力:]

書　式　　Debug
NODebug

説　明　　debug オプションを指定した場合、デバッグ情報を出力します。
nodebug オプションを指定した場合、デバッグ情報を出力しません。
debug、nodebug オプションの指定は、オブジェクトモジュールを出力する場合に限り有効です。
本オプションの省略時解釈は、nodebug です。

備　考　　デバッグ情報はデバッガでプログラムをデバッグするのに必要です。ソースプログラムの行に関する情報やシンボルに関する情報 (シンボルデバッグ情報) などを含みます。

アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果 (オブジェクトモジュール出力時)
debug	(指定に関わらず)	デバッグ情報を出力する。
nodebug	(指定に関わらず)	デバッグ情報を出力しない。
(指定なし)	.OUTPUT DBG	デバッグ情報を出力する。
	.OUTPUT NODBG	デバッグ情報を出力しない。
	(指定なし)	デバッグ情報を出力しない。

プリプロセッサ展開結果出力

EXPand

アセンブラ<オブジェクト>[プリプロセッサ展開結果出力]

書　式　　EXPand [= <出力ファイル名>]

説　明　　マクロ展開、条件つきアセンブル、ファイルのインクルードを行った後のアセンブリソースファイルを出力します。
本オプションを指定するとオブジェクトの生成は行いません。
出力ファイル名の指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル拡張子は exp になります。
・主ファイル名、ファイル拡張子とも指定を省略した場合
　主ファイル名は入力ソースファイル(1 つめに指定したもの)と同じになります。
　また、ファイル拡張子は exp になります。

備　考　　入力ソースファイルと出力ファイルに、同じファイル名を指定しないでください。

リテラルプール出力ポイント指定

LITERAL

アセンブラ<オブジェクト>[リテラルテーブル出力位置指定:]

書　式　　LITERAL = <point>[,…]
　　<point>:{ Pool | Branch | Jump | Return }

説　明　　リテラルプール自動生成機能によって生成されるリテラルプールの出力位置を指定します。
・pool　　‥‥ .POOL の位置に出力します
・branch ‥‥ BRA/BRAF 命令の後に出力します
・jump　　‥‥ JMP 命令の後に出力します
・return ‥‥ RTS/RTE 命令の後に出力します
本オプションの省略時解釈は、literal=pool,branch,jump,return です。

オブジェクトモジュール出力制御

Object
NOObject

アセンブラ<オブジェクト>[オブジェクト出力ディレクトリ :]

書　式

Object [= <出力ファイル名>]
NOObject

説　明

object オプションを指定した場合、オブジェクトファイルを出力します。
noobject オプションを指定した場合、オブジェクトファイルを出力しません。
出力ファイル名の指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル拡張子は obj になります。
・主ファイル名、ファイル拡張子とも指定を省略した場合
　主ファイル名は入力ソースファイル (1 つめに指定したもの) と同じになります。
　また、ファイル拡張子は obj になります。
本オプションの省略時解釈は、object です。

備　考

アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果
object	(指定に関わらず)	オブジェクトファイルを出力する。
noobject	(指定に関わらず)	オブジェクトファイルを出力しない。
(指定なし)	.OUTPUT OBJ	オブジェクトファイルを出力する。
	.OUTPUT NOOBJ	オブジェクトファイルを出力しない。
	(指定なし)	オブジェクトファイルを出力する。

入力ソースファイルと出力オブジェクトファイルに、同じファイル名を指定しないでください。同じファイル名を指定した場合、入力ソースファイルが上書きされます。

未解決シンボルサイズ指定

DIspsize

アセンブラ<オブジェクト>[未解決シンボルサイズ指定 :]

書　式　　DIspsize = {4|12}

説　明　　外部参照シンボル及び、値が解決しないシンボルのサイズを指定します。
指定対象は、アドレッシングモードにディスプレースメントサイズとして 4、及び 12 を持つ命令です。
ディスプレースメントサイズが 4 のみの命令は対象外となります。

本オプション省略時解釈は、dispsize=12 です。

備　考　　本オプションは、マイコン種別が SH-2A、SH2A-FPU の場合のみ有効です。
本オプションを指定しても、確保サイズ(:12)による指定を優先します。

3.2.3　リストオプション

表 3.3　リストカテゴリオプション一覧

項目	コマンドライン形式	ダイアログメニュー	指定内容
1 アセンブルリストの出力制御	LISt [= <出力ファイル名>] NOLISt	アセンブラ<リスト> [アセンブルリスト出力]	アセンブルリストの出力を制御します。
2 ソースプログラムリストの出力制御*	SOurce NOSOurce	アセンブラ<リスト> [アセンブルリスト出力] [ソースプログラム:]	ソースプログラムリストの出力を制御します。
3 ソースプログラムリストの部分出力の制御およびタブサイズの設定*	SHow [= <出力種別>[, …]] NOSHow [= <出力種別>[,…]] <出力種別> : { CONditionals | Definitions | CAlls | Expansions | CODe | TAB={ 4 | 8 } }	アセンブラ<リスト> [ソースプログラムリスト部分出力:] [条件つき不成立:] [定義:] [コール:] [展開:] [オブジェクトコード:] [タブサイズ:]	ソースプログラムリストの部分出力の制御およびタブサイズの設定を行います。
4 クロスリファレンスリストの出力制御*	CRoss_reference NOCRoss_reference	アセンブラ<リスト> [アセンブルリスト出力] [クロスリファレンス:]	クロスリファレンスリストの出力を制御します。
5 セクション情報リストの出力制御*	SEction NOSEction	アセンブラ<リスト> [アセンブルリスト出力] [セクション:]	セクション情報リストの出力を制御します。

【注】*　これらのオプションは、list オプションを指定した場合のみ有効となります。

アセンブルリスト出力制御

LISt
NOLISt

アセンブラ<リスト>[アセンブルリスト出力]

書　式　　LISt [= <出力ファイル名>]
NOLISt

説　明　　list オプションを指定した場合、アセンブルリストを出力します。
nolist オプションを指定した場合、アセンブルリストを出力しません。
出力ファイル名の指定を省略すると次のようになります。
・ファイル拡張子の指定を省略した場合
　ファイル拡張子は lis になります。
・主ファイル名、ファイル拡張子とも指定を省略した場合
　主ファイル名は入力ソースファイル(1 つめに指定したもの)と同じになります。
　また、ファイル拡張子は lis になります。
本オプションの省略時解釈は、nolist です。

備　考　　アセンブラ制御命令との関係(アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果
list	(指定に関わらず)	アセンブルリストを出力する。
nolist	(指定に関わらず)	アセンブルリストを出力しない。
(指定なし)	.PRINT LIST	アセンブルリストを出力する。
	.PRINT NOLIST	アセンブルリストを出力しない。
	(指定なし)	アセンブルリストを出力しない。

入力ソースファイルと出力リストファイルに、同じファイル名を指定しないでください。
同じファイル名を指定した場合、入力ソースファイルが上書きされます。

ソースプログラムリスト出力制御

SOurce
NOSOurce

アセンブラ<リスト>[アセンブルリスト出力][ソースプログラム:]

書　式　　SOurce
NOSOurce

説　明　　source オプションを指定した場合、アセンブルリストにソースプログラムリストを付加します。
nosource オプションを指定した場合、アセンブルリストにソースプログラムリストを付加しません。
source、nosource オプションの指定は、アセンブルリストを出力する場合のみ有効です。
本オプションの省略時解釈は、source です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果（アセンブルリスト出力時）
source	(指定に関わらず)	ソースプログラムリストを出力する。
nosource	(指定に関わらず)	ソースプログラムリストを出力しない。
(指定なし)	.PRINT SRC	ソースプログラムリストを出力する。
	.PRINT NOSRC	ソースプログラムリストを出力しない。
	(指定なし)	ソースプログラムリストを出力する。

ソースプログラムリスト部分出力制御およびタブサイズの設定

SHow
NOSHow

アセンブラ<リスト>[ソースプログラムリスト部分出力:]
[条件つき不成立:][定義:][コール:][展開:][オブジェクトコード:][タブサイズ:]

書　式

```
SHow [= <出力種別>[,…]]
NOSHow [= <出力種別>[,...]]
<出力種別>:{   CONditionals    | Definitions      |
               CAlls           | Expansions       |
               CODe            | TAB = { 4 | 8 }  }
```

説　明

show オプションを指定した場合、ソースプログラムリストのプリプロセッサ機能のソースステートメント、オブジェクトコード表示行を指定したタブサイズで出力します。show=<出力種別>を指定した場合は、指定した項目のみ出力します。タブサイズを指定しない場合は、デフォルトサイズで出力します。
noshow オプションを指定した場合、ソースプログラムリストのプリプロセッサ機能のソースステートメント、オブジェクトコード表示行を出力しません。noshow=<出力種別>を指定した場合は、指定した項目のみ出力しません。
show、noshow オプションによる指定はアセンブルリストを出力する場合のみ有効です。
出力種別の内容は次のとおりです。
本オプションの省略時解釈は、show です。

出力種別	意　味	内　容
conditionals	条件つき不成立	.AIF, .AIFDEF の不成立部分
definitions	定義	マクロ定義部分 .AREPEAT, .AWHILE 定義部分 .INCLUDE .ASSIGNA, .ASSIGNC
calls	コール	マクロコール文 .AIF, .AIFDEF, .AENDI
expansions	展開	マクロ展開部分 .AREPEAT, .AWHILE 展開部分
code	オブジェクトコード表示行	制御命令のオブジェクトコード表示が、ソースステートメントの行数を超える部分
tab={4|8}	タブサイズ	リスト表示時のタブのサイズ

備　考

出力種別を 2 つ以上指定する時はカッコ () で囲んで指定してください。

アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果
show[=<出力種別>]	(指定に関わらず)	出　力
noshow[=<出力種別>]	(指定に関わらず)	出力抑止
(指定なし)	.LIST <出力種別>(出力)	出　力
	.LIST <出力種別>(出力抑止)	出力抑止
	(指定なし)	出　力

クロスリファレンスリスト出力制御

CRoss_reference
NOCRoss_reference

アセンブラ<リスト>[アセンブルリスト出力][クロスリファレンス:]

書　式　　CRoss_reference
NOCRoss_reference

説　明　　cross_reference オプションを指定した場合、アセンブルリストにクロスリファレンスリストを付加します。
nocross_reference オプションを指定した場合、アセンブルリストにクロスリファレンスリストを付加しません。
cross_reference、nocross_reference オプションの指定は、アセンブルリストを出力する場合のみ有効です。
本オプションの省略時解釈は、cross_reference です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果 (アセンブルリスト出力時)
cross_reference	(指定に関わらず)	クロスリファレンスリストを出力する。
nocross_reference	(指定に関わらず)	クロスリファレンスリストを出力しない。
(指定なし)	.PRINT　CREF	クロスリファレンスリストを出力する。
	.PRINT　NOCREF	クロスリファレンスリストを出力しない。
	(指定なし)	クロスリファレンスリストを出力する。

セクション情報リスト出力制御

SEction
NOSEction

アセンブラ<リスト>[アセンブルリスト出力][セクション:]

書　式　　SEction
NOSEction

説　明　　section オプションを指定した場合、アセンブルリストにセクション情報リストを付加します。
nosection オプションを指定した場合、アセンブルリストにセクション情報リストを付加しません。
section、nosection オプションの指定はアセンブルリストを出力する場合のみ有効です。
本オプションの省略時解釈は、section です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果 (アセンブルリスト出力時)
section	(指定に関わらず)	セクション情報リストを出力する。
nosection	(指定に関わらず)	セクション情報リストを出力しない。
(指定なし)	.PRINT SCT	セクション情報リストを出力する。
	.PRINT NOSCT	セクション情報リストを出力しない。
	(指定なし)	セクション情報リストを出力する。

3.2.4 その他オプション

表 3.4 その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	リテラルプール自動生成機能のサイズモードを指定	AUTO_literal	アセンブラ<その他> [その他のオプション:] [リテラルプール自動生成機能をサイズ選択モードに指定]	リテラルプール自動生成機能をサイズ選択モードに設定します。
2	未参照外部参照シンボル情報の出力抑止	Exclude NOExclude	アセンブラ<その他> [その他のオプション:] [未定義外部参照シンボル情報の出力抑止]	未参照外部参照シンボルのシンボル情報の出力、出力抑止を指定します。
3	特権モード命令チェックを指定	CHKMd	アセンブラ<その他> [その他のオプション:] [特権モード命令チェックを指定]	特権モード命令のチェックを指定します。
4	LDTLB 命令チェックを指定	CHKTlb	アセンブラ<その他> [その他のオプション:] [LDTLB 命令チェックを指定]	LDTLB 命令のチェックを指定します。
5	キャッシュ関連命令チェックを指定	CHKCache	アセンブラ<その他> [その他のオプション:] [キャッシュ関連命令チェックを指定]	キャッシュ関連命令のチェックを指定します。
6	DSP 関連命令チェックを指定	CHKDsp	アセンブラ<その他> [その他のオプション:] [DSP 関連命令チェックを指定]	DSP 関連命令のチェックを指定します。
7	FPU 関連命令チェックを指定	CHKFpu	アセンブラ<その他> [その他のオプション:] [FPU 関連命令チェックを指定]	FPU 関連命令のチェックを指定します。
8	.FDATA の 8byte ALIGN チェックを指定	CHKAlign8	アセンブラ<その他> [その他のオプション:] [.FDATA の 8byte ALIGN チェックを指定]	.FDATA の 8byte ALIGN チェックを指定します。

リテラルプール自動生成機能サイズモード設定

AUTO_literal

アセンブラ<その他>[リテラルプール自動生成機能をサイズ選択モードに指定]

書　式　　AUTO_literal

説　明　　リテラルプール自動生成機能のサイズモードを指定します。
本オプションを指定した場合、リテラルプール自動生成機能はサイズ選択モードとなり、オペレーションサイズ指定のないデータ転送命令(MOV #imm,Rn)はアセンブラが imm の値の範囲を判定し、必要ならばリテラルプールを自動生成します。
本オプションを指定しない場合、リテラルプール自動生成機能はサイズ指定モードとなり、オペレーションサイズ指定がないデータ転送命令は 1 バイトのデータ転送命令としてアセンブルします。
サイズ選択モードでは、オペレーションサイズ指定のないデータ転送命令は符号付きの範囲で判定するため、H'00000080～H'000000FF(128～255)はワードサイズとして扱います。

imm の値の範囲	選択されるサイズまたはエラー	
	サイズ選択モード	サイズ指定モード
H’80000000 ～ H’FFFF7FFF (-2,147,483,648～-32,769)	ロングワード	ウォーニング 835
H’FFFF8000 ～ H’FFFFFF7F (-32,768～-129)	ワード	ウォーニング 835
H’FFFFFF80 ～ H’0000007F (-128～127)	バイト	バイト
H’00000080 ～ H’000000FF (128～255)	ワード	バイト
H’00000100 ～ H’00007FFF (256～32,767)	ワード	ウォーニング 835
H’00008000 ～ H’7FFFFFFF (32,768～2,147,483,647)	ロングワード	ウォーニング 835

【注】( )内は 10 進表示

未参照シンボルの情報の出力抑止

Exclude
NOExclude

アセンブラ<その他>[未定義外部参照シンボル情報の出力抑止]

書　式　　Exclude
NOExclude

説　明　　exclude オプションは、未参照外部参照シンボルのシンボル情報を出力しません。
noexclude オプションは、未参照外部参照シンボルのシンボル情報を出力します。
未参照外部参照シンボルのシンボル情報を出力抑止することにより、オブジェクトモジュールのサイズを小さくできます。

例　　asmsh aaa.src -exclude
　未参照外部参照シンボルのシンボル情報を出力しません。

asmsh aaa.src -noexclude
　未参照外部参照シンボルのシンボル情報を出力します。

特権モード命令のチェック

CHKMd

アセンブラ<その他>[特権モード命令チェックを指定]

書　式　　CHKMd

説　明　　マイコン種別が SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP で本オプションを指定した場合、各マイコンのユーザモード命令のみ有効とし、特権モード命令を記述するとウォーニング 704 を通知します。

備　考　　マイコン種別が SH3-DSP または SH4AL-DSP で CHKDSP オプションの指定がない場合は、以下の特権モード命令をユーザモード扱いにします。

```
LDC         Rm,SR
LDC.L       @Rm+,SR
STC         SR,Rm
STC.L       SR,@-Rn
```

LDTLB 命令のチェック

CHKTlb

アセンブラ<その他>[LDTLB 命令チェックを指定]

書　式　　CHKTlb

説　明　　マイコン種別が SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP で本オプションを指定した場合、LDTLB 命令を記述するとウォーニング 705 を通知します。

キャッシュ関連命令のチェック

CHKCache

アセンブラ<その他>[キャッシュ関連命令チェックを指定]

書　式　　CHKCache

説　明　　マイコン種別が SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP で本オプションを指定した場合、キャッシュ関連命令を記述するとウォーニング 706 を通知します。

DSP 関連命令のチェック

CHKDsp

アセンブラ<その他>[DSP 関連命令チェックを指定]

書　式　　CHKDsp

説　明　　マイコン種別が SH3-DSP または SH4AL-DSP で本オプションを指定した場合、DSP 関連命令を記述するとウォーニング 707 を通知します。

FPU 関連命令のチェック

CHKFpu

アセンブラ<その他>[FPU 関連命令チェックを指定]

書　式　　CHKFpu

説　明　　マイコン種別が SH2A-FPU、SH-4 または SH-4A で本オプションを指定した場合、FPU 関連命令を記述するとウォーニング 708 を通知します。

.FDATA の 8byte ALIGN のチェック

CHKAlign8

アセンブラ<その他>[.FDATA の 8byte ALIGN チェックを指定]

書　式　　CHKAlign8

説　明　　マイコン種別が SH-4A または SH4AL-DSP で本オプションを指定した場合、.FDATA.D の 8byte ALIGN チェックを行います。
.FDATA.D で指定する倍精度浮動小数点定数データが 8byte ALIGN 境界に存在しない場合にウォーニング 816 を通知します。

3.2.5 マイコンオプション

表 3.5 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	マイコン種別の指定	CPU = <マイコン種別>	CPU [CPU 種別:]	マイコン種別を指定します。
2	メモリのバイト並び順の指定	ENdian = { Big | Little }	CPU [Endian 選択:]	メモリのバイト並び順を指定します。
3	浮動小数点定数の丸め方式を指定	Round = { Nearest | Zero }	CPU [丸め方式:]	浮動小数点定数の丸め方式を指定します。
4	非正規化数となる浮動小数点定数の取扱いを指定	DENormalize = { ON | OFF }	CPU [非正規化数を非正規化数として扱う:]	浮動小数点定数が非正規化数となったときの取扱いを指定します。

マイコン種別

CPU

CPU[CPU 種別:]

書　式　　CPU = <マイコン種別>

説　明　　アセンブルするソースプログラムの対象とするマイコン種別を指定します。
マイコン種別の内容は次のとおりです。

マイコン種別	対象マイコン
SH1	SH-1 用
SH2	SH-2 用
SH2E	SH-2E 用
SHDSP	SH2-DSP 用
SH2A	SH-2A 用
SH2AFPU	SH2A-FPU 用
SH3	SH-3 用
SH3DSP	SH3-DSP 用
SH4	SH-4 用
SH4A	SH-4A 用
SH4ALDSP	SH4AL-DSP 用

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	環境変数	結　果
cpu=<マイコン種別>	(指定に関わらず)	(指定に関わらず)	cpu オプションで指定したマイコン種別
(指定なし)	.CPU <マイコン種別>	(指定に関わらず)	.CPU で指定したマイコン種別
	(指定なし)	SHCPU=<マイコン種別>	環境変数のマイコン種別
		(指定なし)	SH-1 用

メモリのバイト並び順

Endian

CPU[Endian 選択:]

書　式　　Endian = { Big | Little }

説　明　　ターゲットマイコンのバイトの並び方が、Big Endian か Little Endian かを指定します。
本オプションの省略時解釈は、endian=big です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果 (アセンブルリスト出力時)
endian=big	(指定に関わらず)	Big Endian でアセンブルする。
endian=little	(指定に関わらず)	Little Endian でアセンブルする。
(指定なし)	.ENDIAN BIG	Big Endian でアセンブルする。
	.ENDIAN LITTLE	Little Endian でアセンブルする。
	(指定なし)	Big Endian でアセンブルする。

丸め方式

Round

CPU[丸め方式:]

書　式　　Round = { Nearest | Zero }

説　明　　浮動小数点データ制御命令に記述した定数をオブジェクトコードに変換する際の丸め方式を指定します。
Round=nearest を指定した場合、round to NEAREST even で丸めます。
Round=zero を指定した場合、round to ZERO で丸めます。
本オプションを省略した場合、マイコン種別により丸め方式は次のようになります。

マイコン種別	丸め方式
SH1	round to NEAREST even
SH2	round to NEAREST even
SH2E	round to ZERO
SH2A	round to NEAREST even
SH2AFPU	round to ZERO
SHDSP	round to NEAREST even
SH3	round to NEAREST even
SH4	round to ZERO
SH3DSP	round to NEAREST even
SH4A	round to ZERO
SH4ALDSP	round to NEAREST even

備　考　　マイコン種別が SH2E で、丸め方式に round to NEAREST even を選択した場合、ソースプログラムで最初に出現した浮動小数点データ制御命令に対してウォーニング 818 とし、round to NEAREST even でオブジェクトコードを出力します。

非正規化数の扱い

DENormalize

CPU[非正規化数を非正規化数として扱う:]

書　式　　DENormalize = { ON | OFF }

説　明　　浮動小数点データ制御命令に非正規化数を記述したとき有効な値とするか、無効な値とするかを指定します。非正規化数を有効な値(ON)とした場合と、無効な値(OFF)とした場合ではオブジェクトコードが異なります。
denormalize=on を指定した場合、非正規化数を有効な値とします。
denormalize=off を指定した場合、非正規化数を無効な値とします。
非正規化数を有効とした場合は、ウォーニング 842 とし、オブジェクトコードを出力します。
非正規化数を無効とした場合は、ウォーニング 841 とし、0 のオブジェクトコードを出力します。
本オプションを省略した場合、マイコン種別により非正規化数の扱いは次のようになります。

マイコン種別	有効／無効
SH1	有効な値(ON)とする。
SH2	有効な値(ON)とする。
SH2E	無効な値(OFF)とする。
SH2A	有効な値(ON)とする。
SH2AFPU	無効な値(OFF)とする。
SHDSP	有効な値(ON)とする。
SH3	有効な値(ON)とする。
SH3DSP	有効な値(ON)とする。
SH4	無効な値(OFF)とする。
SH4A	無効な値(OFF)とする。
SH4ALDSP	有効な値(ON)とする。

備　考　　マイコン種別が SH2E または SH2AFPU で、非正規化数の扱いを有効な値とした場合、ソースプログラムで最初に出現した浮動小数点データ制御命令に対してウォーニング 818 とし、非正規化数の扱いを有効な値としてオブジェクトコードを出力します。

3.2.6 残りのオプション

表 3.6 残りのオプション

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	異常終了とするエラーレベルの変更	ABort = { Warning | Error }	アセンブラ<その他> [ユーザ指定オプション :]	アセンブラが異常終了するエラーのレベルを変更します。
2	ISO-Latin1 コード文字	LATIN1	アセンブラ<その他> [ユーザ指定オプション :]	ソースファイル内にLatin1 コード文字を使えるようにします。
3	漢字コードをシフト JIS に指定	SJIS	アセンブラ<その他> [ユーザ指定オプション :]	ソースファイル内の漢字コードをシフト JIS コードとして扱います。
4	漢字コードを EUC に指定	EUC	アセンブラ<その他> [ユーザ指定オプション :]	ソースファイル内の漢字コードを EUC コードとして扱います。
5	出力漢字コードの指定	OUtcode = { SJIS | EUC }	アセンブラ<その他> [ユーザ指定オプション :]	オブジェクトファイルに出力する漢字コードを指定します。
6	アセンブルリストの行数指定	LINes = <行数>	アセンブラ<その他> [ユーザ指定オプション :]	アセンブルリストの行数を設定します。
7	アセンブルリストの桁数指定	COlumns = <桁数>	アセンブラ<その他> [ユーザ指定オプション :]	アセンブルリストの桁数を設定します。
8	コピーライト出力抑止	LOGO NOLOGO	- (常に nologo が有効)	コピーライトを出力します。 コピーライトの出力を抑止します。
9	サブコマンドファイルの指定	SUBcommand = <ファイル名>	-	コマンドラインをファイルから入力します。

エラーレベル変更

ABort

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　ABort = { Warning | Error }

説　明　　異常終了するエラーレベルを変更します。
abort=warning を指定した場合、ウォーニングが発生すると処理を中断します。
abort=error を指定した場合、エラーが発生すると処理を中断します。
OS へのリターン値が 1 以上の場合、オブジェクトモジュールの出力を抑止します。
abort オプションによる指定はオブジェクトモジュールを出力する場合に限り有効です。
OS へのリターン値は次のとおりです。
本オプションの省略時解釈は、abort=error です。

発生個数			オプション指定時のリターン値	
ウォーニング	エラー	致命的エラー	abort=warning	abort=error
0	0	0	0	0
1 以上	0	0	2	0
―	1 以上	0	2	2
―	―	1 以上	4	4

LATIN コード文字

LATIN1

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　LATIN1

説　明　　文字列およびコメント内に LATIN コード文字を記述することができます。
本オプションは、sjis、euc、または outcode オプションと一緒に指定しないでください。

シフト JIS

SJIS

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　SJIS

説　明　　文字列およびコメント内の日本語をシフト JIS コードとして解釈します。
本オプション、euc オプションとも省略した場合は、文字列、コメント内の日本語はホストマシンに依存する日本語コードとして解釈します。
本オプションは、latin1、または euc オプションと一緒に指定しないでください。

EUC

EUC

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　EUC

説　明　　文字列およびコメント内の日本語を EUC コードとして解釈します。
本オプション、sjis オプションとも省略した場合は、文字列、コメント内の日本語はホストマシンに依存する日本語コードとして解釈します。
本オプションは、latin1、または sjis オプションと一緒に指定しないでください。

出力漢字コード

OUtcode

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　OUtcode = { SJIS | EUC }

説　明　　outcode=sjis を指定した場合、ソースファイル内の日本語記述をシフト JIS コードに変換し、オブジェクトファイルに出力します。
outcode=euc を指定した場合、ソースファイル内の日本語記述を EUC コードに変換し、オブジェクトファイルに出力します。
本オプションとソースファイル内の漢字コードの指定(sjis、euc オプション)によるオブジェクトファイルへ出力する漢字コードは次のとおりです。

outcode オプション	ソースファイル内の漢字コード		
	sjis	euc	指定なし
sjis	シフト JIS コード	シフト JIS コード	シフト JIS コード
euc	EUC コード	EUC コード	EUC コード
指定なし	シフト JIS コード	EUC コード	デフォルト漢字コード

本オプションを省略した場合の漢字コードは次のとおりです。

ホストマシン	漢字コード
SPARC ステーション	EUC コード
HP9000/700 シリーズ	シフト JIS コード
PC	シフト JIS コード

アセンブルリスト行数設定

LINes

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　LINes = <行数>

説　明　　アセンブルリストの 1 ページあたりの行数を設定します。
行数として有効な値は 20～255 です。
本オプションはアセンブルリストを出力する場合のみ有効です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果
lines=<行数>	(指定に関わらず)	1 ページあたり、lines オプションで指定した行数になる。
(指定なし)	.FORM LIN=<行数>	1 ページあたり、.FORM で指定した行数になる。
	(指定なし)	1 ページあたり、60 行になる。

アセンブルリスト桁数設定

COlumns

アセンブラ<その他>[ユーザ指定オプション :]

書　式　　COlumns = <桁数>

説　明　　アセンブルリストの 1 行あたりの桁数を設定します。
桁数として有効な値は 79～255 です。
本オプションはアセンブルリストを出力する場合のみ有効です。

備　考　　アセンブラ制御命令との関係 (アセンブラはオプションによる指定を優先します)

オプション	制御命令	結　果
columns=<桁数>	(指定に関わらず)	1 行あたり、columns オプションで指定した桁数になる。
(指定なし)	.FORM COL=<桁数>	1 行あたり、.FORM で指定した桁数になる。
	(指定なし)	1 行あたり、132 桁になる。

コピーライト出力制御

LOGO
NOLOGO

なし(常に nologo が有効)

書　式　　LOGO
NOLOGO

説　明　　コピーライトの出力を制御します。
logo オプションを指定した場合、コピーライト表示が出力されます。
nologo オプションを指定した場合、コピーライトの表示の出力が抑止されます。
本オプション省略時解釈は、logo です。

サブコマンドファイル指定

SUBcommand

なし

書　式　　SUBcommand = <ファイル名>

説　明　　コマンドラインをファイルから入力します。
通常のコマンドライン指定と同じ順番で、入力ファイル名とオプションを指定してください。
1 行に 1 つの入力ファイル名またはオプションを指定してください。
subcommand オプションは、サブコマンドファイル内に指定しないでください。

例　　asmsh aaa.src -subcommand=aaa.sub
サブコマンドファイルの内容をコマンドラインに展開し、アセンブルします。
aaa.sub の内容：

```
bbb.src
-list
-noobj
```

展開結果：

```
asmsh aaa.src,bbb.src -list -noobj
```

備　考　　サブコマンドファイル全体のサイズの限界値は 65,535 バイトです。

4.　最適化リンケージエディタ操作方法

4.1　オプション指定規則

4.1.1　コマンドラインの形式

コマンドラインの形式は以下のとおりです。

```
optlnk [ {△<ファイル名> | △<オプション列>}...]
  <オプション列>：-<オプション>[= <サブオプション>[,...]]
```

4.1.2　サブコマンドファイルの形式

サブコマンドファイルの形式は以下のとおりです。

```
<オプション> {= | △}[<サブオプション>[,...]] [△&] [;<コメント>]
  &：継続行指定
```

サブコマンドファイル形式の詳細は、「**4.2.8 サブコマンドファイルオプション**」を参照してください。

4.2 オプション解説

オプション、サブオプションの英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。

また、統合開発環境の対応するダイアログメニューを、タブ名<カテゴリ名>[項目]…で示します。オプションの順序は、統合開発環境のタブとその中のカテゴリに対応しています。

ファイル名、パス名には、括弧記号"("および")"を含まないようにしてください。

4.2.1 入力オプション

表 4.1 入力カテゴリオプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	入力 ファイル	Input = <sub> [{,|△}...] <sub>: <ファイル名> [(<モジュール名>[,...])]	リンカ<入力> [オプション項目 :] [リロケータブルファイル/オブジェクトファイル]	入力ファイルを指定 (コマンドラインでは input なしで指定します)
2	ライブラリ ファイル	LIBrary = <ファイル名>[,…]	リンカ<入力> [オプション項目 :] [ライブラリファイル]	入力ライブラリファイルを指定
3	バイナリ ファイル	Binary = <sub>[,...] <sub> : <ファイル名> (<セクション名> [:<アライメント数>] [/<セクション属性>] [,<シンボル名>])	リンカ<入力> [オプション項目 :] [バイナリファイル]	入力バイナリファイルを指定
4	シンボル 定義	DEFine = <sub>[,...] <sub>: <シンボル名> = {<シンボル名> | <数値> }	リンカ<入力> [オプション項目 :] [シンボル定義]	未定義シンボルの強制定義 シンボル名と同値として定義 数値で定義
5	実行開始 アドレス	ENTry = { <シンボル名> | <アドレス> }	リンカ<入力> [エントリポイント :]	エントリシンボルを指定 エントリアドレスを指定
6	プレリンカ	NOPRElink	リンカ<入力> [プレリンカ制御 :]	プレリンカの起動を抑止

入力ファイル

Input

リンカ<入力>[オプション項目 :][リロケータブルファイル/オブジェクトファイル]

書　式　　Input = <サブオプション>[{,|△}…]
<サブオプション> : <ファイル名>[(<モジュール名>[,…])]

説　明　　入力ファイルを指定します。複数ある場合にはカンマ(,)または空白文字で区切って指定します。
ワイルドカード(*,?)も指定できます。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。数字と英文字は数字が先、英大文字と英小文字は英大文字が先になります。
入力ファイルとして指定できるのは、コンパイラ、アセンブラ出力オブジェクトファイル、最適化リンケージエディタ出力のリロケータブルファイルおよびアブソリュートファイルです。またライブラリ名(<モジュール名>)の形式で、ライブラリ内モジュールを入力ファイルとして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場合、モジュール名がない時は「obj」、モジュール名がある時は「lib」を仮定します。

例　　input=a.obj lib1(e)　　; a.obj と lib1.lib 内のモジュール e を入力します
input=c*.obj　　; c で始まる拡張子 obj のファイルを全て入力します

備　考　　form=object および extract 指定時、本オプションは無効です。
コマンドライン上で入力ファイルを指定する場合は、input 無しで指定します。

ライブラリファイル

LIBrary

リンカ<入力>[オプション項目 :][ライブラリファイル]

書　式　　LIBrary = <ファイル名>[,…]

説　明　　ライブラリファイルを指定します。複数ある場合にはカンマ(,)で区切って指定します。
ワイルドカード(*,?)も指定できます。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。数字と英文字は数字が先、英大文字と英小文字は英大文字が先になります。
入力ファイル名に拡張子の指定がない場合は、「lib」を仮定します。
form=library オプションまたは extract オプション指定時は、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ケージ処理後に、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、ライブラリオプション指定ユーザライブラリファイル(指定順)、ライブラリオプション指定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_LIBRARY1,2,3)の順序で行います。

例　　library=a.lib,b　　; a.lib と b.lib を入力します。
library=c*.lib　　; c で始まる拡張子 lib のファイルを全て入力します。

バイナリファイル

Binary

リンカ<入力>[オプション項目 :][バイナリファイル]

書　式

```
Binary = <サブオプション>[,…]
<サブオプション> : <ファイル名>(<セクション名>
                    [:<アライメント数>][/<セクション属性>][,<シンボル名>])
<セクション属性> : CODE | DATA
<アライメント数> : 1 | 2 | 4 | 8 | 16 | 32 (デフォルトは1)
```

説　明

入力バイナリファイルを指定します。複数ある場合にはカンマ(,)で区切って指定します。
ファイル名に拡張子の指定がない場合は、「bin」を仮定します。
入力したバイナリデータは、指定したセクション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。セクションは省略できません。
またシンボルを指定することにより、定義シンボルとしてリンクすることもできます。C/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。
セクション属性は、CODE または DATA を指定できます。
セクション属性の指定が無い場合、デフォルトとして書き込み、読み取り、実行全ての属性が有効になります。
アライメント数に指定可能な値は 2 の累乗です。それ以外の値を指定することはできません。
アライメント数の指定がない場合、デフォルト値として 1 が有効になります。

例

```
input=a.obj
start=P,D*/200
binary=b.bin(D1bin),c.bin(D2bin:4,_datab)
form=absolute
```

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。

備　考

form={object | library}または strip 指定時、本オプションは無効です。
また入力オブジェクトファイル指定がない場合、本オプションは指定できません。

シンボル定義

DEFine

リンカ<入力>[オプション項目 :][シンボル定義]

書　式

```
DEFine = <サブオプション>[,…]
<サブオプション> : <シンボル名> = {<シンボル名> | <数値>}
```

説　明

未定義シンボルを外部定義シンボルまたは数値で強制定義します。
数値は 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して解釈します。先頭に 0 を付加した場合は常に数値と解釈します。
シンボル名が C/C++変数名の場合、プログラム中での定義名先頭に_を付加します。C++関数名の場合は(main 関数は除く)引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。ただし引数が void の場合は、"関数名()"で指定します。

例　define=_sym1=data　;_sym1 を外部定義シンボル data と同値として定義します。
define=_sym2=4000　;_sym2 を 0x4000 として定義します。

備　考　form={object | relocate | library}指定時、本オプションは無効です。

実行開始アドレス

ENTry

リンカ<入力>[エントリポイント :]

書　式　ENTry = {<シンボル名> | <アドレス>}

説　明　実行開始アドレスを外部定義シンボルまたはアドレスで指定します。
アドレスは 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判断します。先頭に 0 を付加した場合は常にアドレスと解釈します。
シンボル名は、C 関数名の場合プログラム中での定義名先頭に_を付加します。C++関数名の場合は(main 関数は除く)引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。ただし引数が void の場合は、"関数名()"で指定します。
コンパイル、アセンブル時に entry シンボルを指定している場合、本オプション指定を優先します。

例　entry=_main　;C/C++の main 関数を実行開始アドレスとして設定します。
entry="init()"　;C++の init 関数を実行開始アドレスとして設定します。
entry=100　;0x100 を実行開始アドレスとして設定します。

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
未参照シンボル削除最適化(optimize=symbol_delete)指定時には、実行開始アドレスは必ず必要です。指定がない場合は、未参照シンボル削除最適化指定は無効です。本オプションでアドレスを指定している場合は、未参照シンボル削除最適化を無効にします。

プレリンカ

NOPRElink

リンカ<入力>[プレリンカ制御 :]

書　式　NOPRElink

説　明　プレリンカの起動を抑止します。
プレリンカは、C++テンプレートインスタンスの自動生成機能および実行時型検査機能をサポートします。C++テンプレート機能および実行時型検査機能を使用していない場合は、noprelink オプションを指定してください。リンク時間が短くなります。

備　考　extract または strip 指定時、本オプションは無効です。
C++テンプレート機能および実行時型検査機能を使用し、form=lib または、form=rel を指定する場合には、noprelink を指定しないでください。

4.2.2 出力オプション

表 4.2 出力カテゴリオプション一覧

	項目	オプション	ダイアログメニュー	指定内容
1	出力形式	FOrm = { Absolute | Relocate | Object | Library [= {S | U}] | Hexadecimal | Stype | Binary }	リンカ<出力> [出力形式 :]	アブソリュート形式 リロケータブル形式 オブジェクト形式 ライブラリ形式 インテル HEX 形式 モトローラ S 形式 バイナリ形式
2	デバッグ情報	DEBug SDebug NODEBug	リンカ<出力> [デバッグ情報 :]	出力あり(出力ファイル内) デバッグ情報ファイル出力 出力なし
3	レコードサイズ統一	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }	リンカ<出力> [レコードサイズ統一 :]	インテル HEX レコード インテル拡張 HEX レコード インテル 32bitHEX レコード S1 レコード S2 レコード S3 レコード
4	ROM 化支援	ROm = <sub>[,...] <sub> : <ROM セクション名> =<RAM セクション名>	リンカ<出力> [オプション項目 :] [ROM から RAM へマップするセクション]	RAM に領域を確保し、シンボルを RAM 上のアドレスでリロケーション解決
5	出力ファイル	OUtput = <sub>[,…] <sub> : <ファイル名> [=<出力範囲>] <出力範囲>: { <先頭アドレス> - <終了アドレス> | <セクション名>[:...] }	リンカ<出力> [オプション項目 :] [出力ファイル/インフォメーション抑止] [出力ファイルの分割]	出力ファイルを指定 (範囲指定、分割出力可能)
6	外部シンボル割り付け情報ファイル	MAp [= <ファイル名>]	リンカ<出力> [外部シンボル割り付け情報ファイル出力]	外部シンボル割り付け情報ファイル出力を指定(SuperH ファミリ, RX ファミリ向け)
7	空きエリア出力指定	SPace [= {<数値> | Random}]	リンカ<出力> [オプション項目 :] [空きエリア出力指定] [空きエリア出力]	空きエリアへの出力値の指定
8	インフォメーションメッセージ	Message NOMessage [= <sub>[,...]] <sub> : <エラー番号> [- <エラー番号>]	リンカ<出力> [オプション項目 :] [出力ファイル/インフォメーション抑止] [インフォメーションレベルメッセージ抑止]	出力あり 出力なし (エラー番号、範囲指定可能)

	項目	オプション	ダイアログメニュー	指定内容
9	参照されない定義シンボルの通知	MSg_unused	リンカ<出力> **[オプション項目 :]** **[メッセージ出力指定]** **[参照されない定義シンボルの通知]**	1 回も参照されない定義シンボルをメッセージ出力により通知
10	セクション内データの詰め込み配置	DAta_stuff	リンカ<出力> **[オプション項目 :]** **[セクション内データの詰め込み配置]**	コンパイル単位間の空き領域を詰めてデータを配置(SuperH ファミリ, H8,H8S,H8SX ファミリ向け)
11	データレコードのバイト数指定	BYte_count=<数値>	リンカ<出力> **[データレコード長 :]**	データレコードの最大バイト数を指定
12	CRC 演算	CRc = <サブオプション> <サブオプション> : <出力位置>=<計算範囲>[/<多項式>][:<エンディアン>] <出力位置> : <アドレス> <計算範囲> : <先頭アドレス>-<終了アドレス>[,...] <多項式> : { CCITT | 16 } <エンディアン> : {BIG|LITTLE}	リンカ<出力> **[オプション項目 :]** **[CRC コード]**	リンク時に計算範囲の CRC(Cyclic Redundancy Check)演算を行い、計算結果を出力位置に埋め込む
13	セクション終端にパディング	PADDING	リンカ<出力> **[パディング]**	アライメントにあわせてセクション終端にパディングを出力
14	特定ベクタ番号のアドレス設定	VECTN=<サブオプション>[,...] <サブオプション>: <ベクタ番号>={<シンボル>|<アドレス>}	リンカ<出力> **[オプション項目 :]** **[ベクタ]** **[特定ベクタ :]**	可変ベクタの特定ベクタ番号へのアドレスを設定(RX ファミリ、M16C ファミリ向け)
15	可変ベクタの空き領域のアドレス設定	VECT={<シンボル>|<アドレス>}	リンカ<出力> **[オプション項目 :]** **[ベクタ]** **[空きベクタ :]**	可変ベクタの空き領域へのアドレスを設定(RX ファミリ、M16C ファミリ向け)
16	utl30 情報出力	UTL	リンカ<出力> [UTL 情報]	UTL30 向け情報を出力(M16C ファミリ向け)
17	ジャンプテーブル出力	JUMP_ENTRIES_FOR_PIC=<セクション名>[,...]	リンカ<出力> [ジャンプテーブル出力]	ジャンプテーブルを出力(RX ファミリの PIC 機能向け)

出力形式

FOrm

リンカ<出力>[出力形式 :]

書　式　　FOrm = {Absolute | Relocate | Object | Library[={S|U}]
　　| Hexadecimal | Stype | Binary}

説　明　　出力形式を指定します。
本オプションの省略時解釈は、form=absolute です。サブオプションの一覧を表 4.3 に示します。

表 4.3　form オプションのサブオプション一覧

	サブオプション名	内　容
1	absolute	アブソリュートファイルを出力します。
2	relocate	リロケータブルファイルを出力します。
3	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。
4	library	ライブラリファイルを出力します。 library=s 指定時、出力ライブラリファイルをシステムライブラリとします。 library=u 指定時、出力ライブラリファイルをユーザライブラリとします。 省略時解釈は、library=u です。
5	hexadecimal	インテル HEX 形式ファイルを出力します。インテル HEX フォーマットは「16.1.2 インテル HEX 形式ファイル」を参照してください。
6	stype	モトローラ S 形式ファイルを出力します。モトローラ S フォーマットは「16.1.1 モトローラ S 形式ファイル」を参照してください。
7	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、他オプションとの関係を表 4.4 に示します。

表 4.4　出力形式と入力ファイル、他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*1
1	Absolute	strip あり	アブソリュートファイル	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, sdebug, cpu, ps_check, start, rom, entry, output, map, hide, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, section_forbid, absolute_forbid, profile, cachesize, sbr, compress, rename, delete, define, fsymbol, stack, noprelink, memory, msg_unused, data_stuff*5, show=symbol, reference, xreference, jump_entries_for_pic, aligned_section
2	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, debug/nodebug, output, hide, rename, delete, noprelink, msg_unused, data_stuff*5, show=symbol, xreference
3	Object	extract あり	ライブラリファイル	library, output
4	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, ps_check, start, rom, entry, output, map, space, optimize/nooptimize, samesize, symbol_forbid, samecode_forbid, variable_forbid, function_forbid, section_forbid, absolute_forbid, profile, cachesize, sbr, rename, delete, define, fsymbol, stack, noprelink, record, s9*2, byte_count*3, memory, msg_unused, data_stuff*5, show=symbol, reference, xreference, jump_entries_for_pic, aligned_section
			アブソリュートファイル	input, output, record, s9*2, byte_count*3, show=symbol, reference, xreference
5	Library	strip あり	ライブラリファイル	library, output, memory*4, show=symbol, section
		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, noprelink, memory*4, show=symbol, section

【注】　*1　message/nomessage, change_message, logo/nologo, form, list, subcommand は常に指定できます。
　　*2　s9 は出力形式が form=stype のときだけ指定できます。

*3 byte_count は出力形式が form= hexadecimal のときだけ指定できます。

*4 hide 指定する場合は使用できません。

*5 data_stuff は出力形式が form=relocate のときは指定できません。

デバッグ情報

DEBug
SDebug
NODEBug

リンカ<出力>[デバッグ情報 :]

書　式　DEBug
SDebug
NODEBug

説　明　debug 情報の出力有無を指定します。
debug オプションは、出力ファイル中にデバッグ情報を出力します。
sdebug オプションは、<出力ファイル名>.dbg ファイルにデバッグ情報を出力します。
nodebug オプションは、デバッグ情報を出力しません。
form=relocate 指定時に sdebug オプションを指定したときは、debug オプションと解釈します。
output オプションで複数ファイル出力を指定時に debug オプションを指定したときは、sdebug オプションと解釈して、<先頭出力ファイル名>.dbg に出力します。
本オプション省略時解釈は、debug です。

備　考　form={object | library | hexadecimal | stype | binary}、strip、または、extract 指定時、本オプションは無効です。

レコードサイズ統一

REcord

リンカ<出力>[レコードサイズ統一 :]

書　式　REcord = {H16 | H20 | H32 | S1 | S2 | S3}

説　明　アドレス範囲に関係なく、一定のデータレコードで出力します。
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、アドレスに合わせてデータレコードを選択します。
本オプション省略時は、それぞれのアドレスに合わせて混在したデータレコードを出力します。

備　考　form=hexadecimal または stype 指定がないとき、本オプションは無効です。

ROM 化支援

ROm

リンカ<出力>[オプション項目 :][ROM から RAM へマップするセクション]

書　式　ROm = <サブオプション>[,…]
<サブオプション> : <ROM セクション名>=<RAM セクション名>

説　明　初期化データ領域の 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になるようリロケーションします。
ROM セクション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。

例　rom=D=R
start=D/100,R/8000
D セクションと同サイズの R セクションを確保し、D セクション内定義シンボルを R セクション上のアドレスでリロケーションします。

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。

出力ファイル

OUtput

リンカ<出力>[オプション項目 :][出力ファイル/インフォメーション抑止][出力ファイルの分割]

書　式　OUtput ＝ <サブオプション>[,…]
<サブオプション> : <ファイル名>[=<出力範囲>]
<出力範囲>: {<先頭アドレス>-<終了アドレス> | <セクション名>[:…]}

説　明　出力ファイル名を指定します。form={absolute | hexadecimal | stype | binary}のときは、複数ファイルを指定できます。アドレスは 16 進数で指定します。先頭が A～F の場合は先にセクションを検索し、該当するセクションがなければアドレスと判断します。先頭に 0 を付加した場合は常にアドレスと解釈します。
本オプションの省略時解釈は、<先頭入力ファイル名>.<デフォルト拡張子>です。
デフォルト拡張子は、次のようになります。
form=absolute：「abs」、form=relocate　：「rel」、form=object：「obj」
form=library ：「lib」、form=hexadecimal：「hex」、form=stype ：「mot」
form=binaruy ：「bin」

例　output=file1.abs=0-ffff,file2.abs=10000-1ffff
0～0xffff 間を file1.abs に、0x10000～0x1ffff 間を file2.abs に出力します。

output=file1.abs=sec1:sec2,file2.abs=sec3
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。

備　考　マイコン種別が RX ファミリでビッグエンディアンのときに、セクション単位で出力する場合は、セクションサイズを 4 の倍数にしてください。

外部シンボル割り付け情報ファイル出力

MAp

リンカ<出力>[外部シンボル割り付け情報ファイル出力]

書　式　MAp [= <ファイル名>]

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を出力します。
<ファイル名>を指定しなかった場合は、output オプションで指定したファイル名、もしくは先頭入力ファイル名で、拡張子が bls のファイルを出力します。
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変わっている場合はエラーを出力します。

備　考　form={absolute | hexadecimal | stype | binary}を指定した場合のみ、本オプションは有効です。
マイコン種別が SuperH ファミリおよび RX ファミリで有効です。

空きエリア出力指定

SPace

リンカ<出力>[オプション項目 :][空きエリア出力指定][空きエリア出力]

書　式　SPace [＝ {<数値> | Random}]

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で充填します。
充填するデータとしては、乱数、もしくは 16 進数の数値を指定することができます。
空きエリアを埋める方法は、output オプション指定時の出力範囲指定方法によって下記のように異なります。

・出力範囲:セクション指定
　指定されたセクション間に空きが存在した場合に指定データを出力
・出力範囲:アドレス範囲指定
　指定された範囲内に空きが存在した場合に指定データを出力

出力データサイズは、1,2,4 バイト単位で有効となります。出力データサイズは space オプションに指定する 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、上位桁を 0 拡張し 4 バイトのデータとして扱われます。また、奇数桁データを指定した場合も、上位桁に 0 拡張して偶数桁入力として扱われます。
空きエリアのサイズが出力データサイズの倍数でない場合、出力できるだけ出力し、メッセージによる警告を行います。

備　考　本オプションにてサブオプションの指定がされなかった場合は、空きエリアへの出力は行いません。
本オプションは form={ binary |stype | hexadecimal}オプションを指定した場合にのみ有効となります。
output オプションによる出力範囲指定がされなかった場合は、本オプション指定は無効となります。

インフォメーションメッセージ

Message
NOMessage

リンカ<出力>[オプション項目 :][出力ファイル/インフォメーション抑止][インフォメーションレベルメッセージ抑止]

書　式　　Message
NOMessage [= <サブオプション>[,…]]
<サブオプション> : <エラー番号>[-<エラー番号>]

説　明　　インフォメーションレベルメッセージの出力有無を指定します。
message オプション指定時は、インフォメーションレベルメッセージを出力します。
nomessage オプション指定時は、インフォメーションレベルメッセージの出力を抑止します。またエラー番号を指定すると、指定したエラー番号のメッセージ出力を抑止できます。ハイフン(-)を使用して抑止するエラー番号の範囲を指定することもできます。エラー番号としてウォーニング、エラーレベルメッセージ番号を指定した場合、change_message でインフォメーションレベルに変更したと仮定し、メッセージ出力を抑止します。
本オプションの省略時解釈は nomessage です。

例　　nomessage=4,200-203,1300
L0004 および L0200～L0203 および L1300 のメッセージ出力を抑止します。

参照されない定義シンボルの通知

MSg_unused

リンカ<出力>[オプション項目 :][メッセージ出力指定][参照されない定義シンボルの通知]

書　式　MSg_unused

説　明　本オプションを指定した場合、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知らせます。

例　optlnk -msg_unused a.obj

備　考　入力ファイルが absolute 形式の場合、本オプション指定は無効です。
メッセージ出力させるためには、同時に message オプションの指定が必要です。
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、関数定義に static 宣言することで、メッセージ出力を抑えることができます。
以下のいずれかに該当する場合、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。

- アセンブル時に goptimize オプションが指定されておらず、同一ファイル内、かつ同一セクションへの分岐がある場合 (マイコンが H8,H8S,H8SX ファミリの場合のみ)
- 同一ファイル内の定数シンボルへの参照
- 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
- コンパイル時に外部変数アクセス最適化が有効な場合 (マイコンが SuperH ファミリの場合のみ)
- ソースファイル上で#pragma tbr を記述した際にオフセット値を直接指定している場合 (マイコンが SH-2A/SH2A-FPU の場合のみ)
- リンク時の最適化によって、定数やリテラルの統合が生じる場合

セクション内データの詰め込み配置

DAta_stuff

リンカ<出力>[オプション項目 :][セクション内データの詰め込み配置]

書　式　　DAta_stuff

説　明　　リンク時に、セクション内のデータを詰め込んで配置します。本オプション機能の対象となるセクションは、定数領域、初期化データ領域、未初期化データ領域です。
本オプションを指定した場合、コンパイル単位のセクションのアライメントにより生じる空き領域を詰めてリンクを行います。
ただし、データの配置順は変更しません。
本オプションを指定しない場合、コンパイル単位のセクションのアライメントに従いリンクを行います。本オプションの指定により、アライメントで生じる冗長な空き領域を詰めることができ、データセクション全体のサイズ低減が期待できます。

例

```
<tp1.c>            <tp2.c>
----------------  --------------------
long a;            char  d;
char b,c;          long  e;
                   char  f;

<コンパイル後のデータセクションサイズ (SuperH ファミリ用コンパイラの出力例)>
  tp1.obj : 4+1+1 = 6 バイト
  tp2.obj : 1+3[*]+4+1 = 9 バイト

<tp1.obj と tp2.obj、リンク後のデータセクションサイズ>
1) data_stuff 指定なし
  オブジェクトファイルを各セクションのアライメントに従ってリンクします(従来処理)。
  6 バイト[tp1] + 2 バイト[*] + 9 バイト[tp2] = 17 バイト

2) data_stuff 指定あり
  セクション内のデータを詰めて配置させ、アライメントによる 冗長な空き領域を埋めて
  リンクします。
  (4+1+1)バイト + 1 バイト + 1 バイト[*] + 4 バイト + 1 バイト = 13 バイト
```

【注 1】 * : アライメントのために生じる空き領域
【注 2】 コンパイル後のデータセクションサイズは、コンパイル時のオプション指定などによって変化するので、上記例のようにならない場合があります。

備　考　　SuperH ファミリ用コンパイラの smap オプションを指定したオブジェクトファイルをリンクする際に本オプションを指定した場合、動作は保証しません。
アセンブラ出力のオブジェクトファイルに対しては、本オプション機能は適用されません。
下記のいずれかの条件の場合、本オプション指定は無効です。
- form=library,object,relocate 指定時
- アブソリュートファイル入力時
- memory=low 指定時
- nooptimize 指定がない場合

本オプションを指定して生成したリロケータブルファイルに対してはリンク時の最適化が適用されません。
マイコン種別が RX ファミリ、M16C シリーズ、R8C ファミリの場合は、本機能を使用できません。

データレコードのバイト数指定

BYte_count

リンカ<出力>[データレコード長 :]

書　式　BYte_count=<数値>

説　明　Intel-Hex 形式ファイルを生成する際に、データレコードのバイト数最大値を指定するためのオプションです。バイト数としては、1byte の 16 進数(01～FF)を指定することができます。本オプションを記述しない場合、バイト数最大値は FF として Intel-Hex ファイルを生成します。

例　byte_count=10

備　考　生成するファイル形式が Intel-Hex 形式(form=hex)ではない場合、本オプションは無効です。

CRC 演算

CRc

リンカ<出力>[オプション項目 :][CRC コード]

書　式

```
CRc = <サブオプション>
  <サブオプション>: <出力位置>=<計算範囲>[/<多項式>][:<エンディアン>]
    <出力位置> :  <アドレス>
    <計算範囲> :  <先頭アドレス>-<終了アドレス>[,...]
    <多項式>   :  { CCITT | 16 }
    <エンディアン>: {BIG|LITTLE}
```

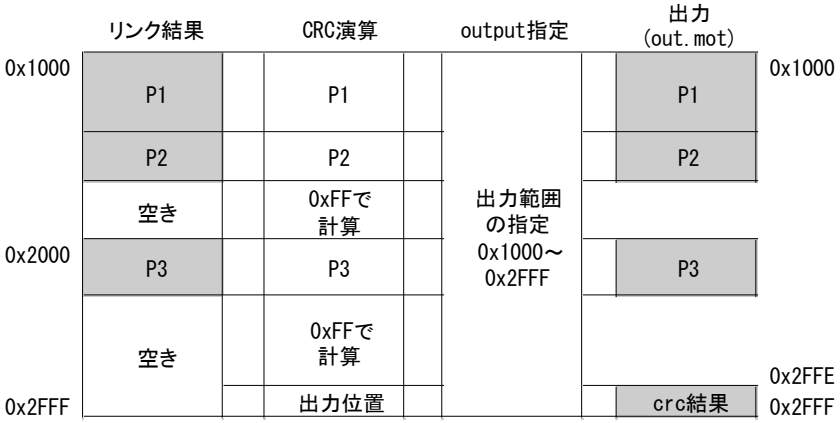
説　明　計算範囲で指定された内容を下位アドレスから上位アドレスの順で CRC(Cyclic Redundancy Check)演算を行い、計算結果を出力位置のアドレスに出力します。
エンディアンは、RX ファミリの場合に指定可能なオプションです。エンディアンを指定した場合は、エンディアンにしたがって、計算結果を出力位置のアドレスに出力します。指定しない場合は、アブソリュートファイルのエンディアンで計算結果を出力位置のアドレスに出力します。
多項式は CRC-CCITT または CRC-16 を選択できます。(デフォルトは CRC-CCITT)

多項式

CRC-CCITT
X^16+X^12+X^5+1
ビット表現(10001000000100001)

CRC-16
X^16+X^15+X^2+1
ビット表現(11000000000000101)

例 1　　optlnk *.obj -form=stype -start=P1,P2/1000,P3/2000
　　-crc=2FFE=1000-2FFD -output=out.mot=1000-2FFF



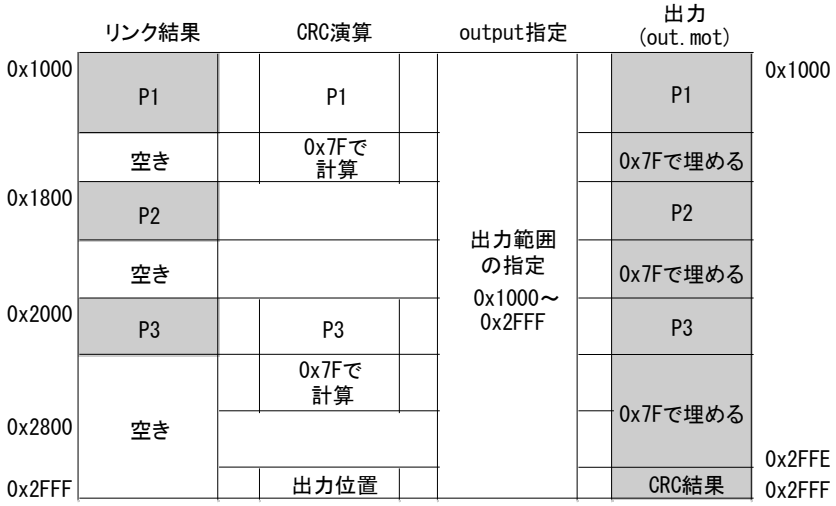
crc オプション：-crc=2FFE=1000-2FFD
　　0x1000～0x2FFD の領域に対して CRC 演算を行い、その結果を 0x2FFE 番地に出力します。計算範囲にある空き領域は space オプションが指定されていない場合、space=0xFF が指定されていると仮定して、CRC 演算を行います。

output オプション：-output=out.mot=1000-2FFF
　　space オプションが指定されていないため、空きの領域は「out.mot」ファイルに出力されません。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。

【注】 1.　CRC 出力位置は、計算範囲に含むことは出来ません。

2.　CRC 出力位置は output オプションの出力範囲に含まれている必要があります。

例 2
```
optlnk *.obj -form=stype -start=P1/1000,P2/1800,P3/2000
        -space=7F -crc=2FFE=1000-17FF,2000-27FF
        -output=out.mot=1000-2FFF
```



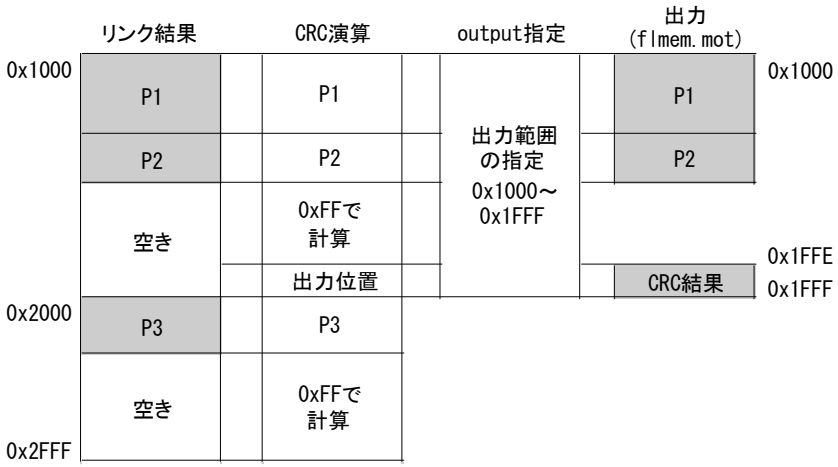
crc オプション：-crc=2FFE=1000-2FFD,2000-27FF
0x1000～0x17FF と 0x2000～0x27FF の２つの領域に対して CRC 演算を行い、その結果を 0x2FFE 番地に出力します。
CRC 演算は計算対象として、連続していない複数の計算範囲を指定できます。

space オプション：-space=7F
指定された計算範囲の空き領域は space オプションの値(0x7F)で計算されます。

output オプション：-output=out.mot=1000-2FFF
space オプションが指定されているため、空き領域は「out.mot」ファイルに出力されます。空き領域は 0x7F で充填されます。

【注】 1. CRC 演算の計算順は計算範囲の指定順ではありません。下位アドレスから上位アドレスの順に計算されます。
2. crc オプションと space オプションを同時に指定する場合、space オプションに random または２バイト以上の値を指定することは出来ません。１バイトのデータを指定してください。

例 3
```
optlnk *.obj -form=stype -start=P1,P2/1000,P3/2000
        -crc=1FFE=1000-1FFD,2000-2FFF
        -output=flmem.mot=1000-1FFF
```



crc オプション：-crc=1FFE=1000-1FFD,2000-2FFF

0x1000～0x1FFD と 0x2000～0x2FFF の領域に対して CRC 演算を行い、その結果を 0x1FFE 番地に出力します。
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、space=0xFF が指定されていると仮定して、CRC 演算を行います。

output オプション：-output=flmem.mot=1000-1FFF

space オプションが指定されていないため、空きの領域は「flmem.mot」ファイルに出力されません。
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。

備　考　　複数のアブソリュートファイル入力時は、本オプションは無効です。
出力形式が form={hexadecimal | stype}の場合に有効です。
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になります。

サンプルコード

crc オプションで計算された CRC 演算結果を比較するためのサンプルコードです。
サンプルコードのプログラムは、optlnk の CRC 演算結果と一致します。

多項式 CRC-CCITT の場合

```
typedef  unsigned  char    uint8_t;
typedef  unsigned  short   uint16_t;
typedef  unsigned  long    uint32_t;

uint16_t CRC_CCITT(uint8_t *pData, uint32_t iSize)
{
  uint32_t ui32_i;
  uint8_t       *pui8_Data;
  uint16_t ui16_CRC = 0xFFFFu;

  pui8_Data = (uint8_t *)pData;

  for(ui32_i = 0; ui32_i < iSize; ui32_i++)
  {
    ui16_CRC  = (uint16_t)((ui16_CRC >> 8u) |
                  ((uint16_t)((uint32_t)ui16_CRC << 8u)));
    ui16_CRC ^= pui8_Data[ui32_i];
    ui16_CRC ^= (uint16_t)((ui16_CRC & 0xFFu) >> 4u);
    ui16_CRC ^= (uint16_t) ((ui16_CRC << 8u) << 4u);
    ui16_CRC ^= (uint16_t)(((ui16_CRC & 0xFFu) << 4u) << 1u);
  }
 ui16_CRC = (uint16_t)( 0x0000FFFFul &
               ((uint32_t)~(uint32_t)ui16_CRC) );
  return ui16_CRC;
}
```

多項式 CRC-16 の場合

```
#define POLYNOMIAL  0xa001      // 生成多項式 CRC-16

typedef  unsigned  char    uint8_t;
typedef  unsigned  short   uint16_t;
typedef  unsigned  long    uint32_t;

uint16_t CRC16(uint8_t *pData, uint32_t iSize)
{
  uint16_t crcdData = (uint16_t)0;
  uint32_t data = 0;
  uint32_t i,cycLoop;

  for(i=0;i<iSize;i++){
    data = (uint32_t)pData[i];
    crcdData = crcdData ^ data;
    for (cycLoop = 0; cycLoop < 8; cycLoop++) {
        if (crcdData & 1) {
            crcdData = (crcdData >> 1) ^ POLYNOMIAL;
        } else {
            crcdData = crcdData >> 1;
        }
    }
  }
  return crcdData;
}
```

セクション終端にパディング埋め込み

PADDING

リンカ<出力>[パディング :]

書　式　　PADDING

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、セクション終端にデータを埋め込みます。

例　　-start=P,C/0 -padding
P セクションのアライメント:4 バイト
P セクションのサイズ:0x06 バイト
C セクションのアライメント:1 バイト
C セクションのサイズ:0x03 バイト
の場合、
P セクションに 2 バイトのパディングデータを埋め込んで、サイズを 0x08 バイトにしてリンクする。

-start=P/0,C/7 -padding
P セクションのアライメント:4 バイト
P セクションのサイズ:0x06 バイト
C セクションのアライメント:1 バイト
C セクションのサイズ:0x03 バイト
の場合、
P セクションに 2 バイトのパディングデータを埋め込んで、サイズを 0x08 バイトにしてリンクすると、C セクションと重複してしまうため、L2321 エラーを出力する。

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、パディングを行いませんので、絶対アドレスセクションはユーザにてサイズを調整してください。
マイコン種別が SuperH ファミリおよび RX ファミリのときに有効です。

特定ベクタ番号のアドレス設定

VECTN

リンカ<出力>[オプション項目 :] [特定ベクタ]

書　式　　VECTN = <サブオプション>[,...]
<サブオプション> : <ベクタ番号> = {<シンボル> | <アドレス>}

説　明　　可変ベクタテーブルセクションの特定ベクタ番号に対して、オプションで指定されたアドレスを設定します。
本オプションを使用した場合、ソース上に割り込み関数記述がなくても、可変ベクタテーブルセクションを作成し、テーブルへアドレスを設定します。

<ベクタ番号>は、10 進数で 0～255 の範囲で指定してください。
<シンボル>は、対象関数の外部名で指定してください。
<アドレス>は、指定アドレスを 16 進数で指定してください。

例　　-vectn=30=_f1,31=0000F100 ;ベクタ番号 30 番に_f1 のアドレスを、
;ベクタ番号 31 番に 0x0f100 を設定します

備　考　　マイコン種別が RX ファミリ、M16C シリーズ、R8C ファミリの場合に有効です。
ユーザが可変ベクタテーブルセクションをソースプログラムで作成している場合、可変ベクタテーブルの自動生成は行なわないため、本オプションは無効になります。

空きベクタ領域のアドレス設定

VECT

リンカ<出力> [オプション項目:] [空きベクタ]

書　式　　VECT={<シンボル>|<アドレス>}

説　明　　可変ベクタテーブルセクションで、アドレス未設定のベクタ番号に対してオプション指定のアドレスを設定します。
本オプションを使用した場合、ソース上の割り込み関数記述がなくても、可変ベクタテーブルセクションをリンカが作成し、テーブルへアドレスを設定します。
<シンボル>は、対象関数の外部名を記述してください。
<アドレス>は、設定するアドレスを 16 進数表記で記述してください。

備　考　　マイコン種別が RX ファミリ、M16C シリーズ、R8C ファミリの場合に有効です。
ユーザが可変ベクタテーブルセクションをソースプログラムで作成している場合、可変ベクタテーブルの自動生成は行なわないため、本オプションは無効になります。
{<シンボル>|<アドレス>}の記述で、先頭を 0 と記述したものは全てアドレスとして判断します。

Ut130 向け情報ファイル出力

UTL

リンカ<その他>[その他のオプション]utl ファイル出力

書　式　　UTL

説　明　　コンパイラパッケージ付属のツール(utl30)に入力するための外部ファイル(utl ファイル)を生成します。
生成するファイルの名称は「<出力ファイル名>.utl」となります。

例　　tp.obj
utl
output=test.abs
tp.obj 内のインスペクタ情報を test.utl に出力します。

備　考　　本オプションは、M16C マイコン向けのコンパイラを使用した場合のみ有効です。
本オプションは、abs ファイル入力時の処理には使用できません。
form={object | library} 指定時、本オプションは無効です。

ジャンプテーブル出力

JUMP_ENTRIES_FOR_PIC

リンカ<出力>[ジャンプテーブル:]

書　式　JUMP_ENTRIES_FOR_PIC = <セクション名>[,...]

説　明　指定セクション内の外部定義シンボルへ分岐するジャンプテーブルのアセンブラソースを出力します。
RX ファミリ用コンパイラの PIC 機能向けに用意されたオプションです。
ファイル名は、<出力ファイル>.jmp です。

例　jump_entries_for_pic=sct2,sct3
output=test.abs
セクション sct2,sct3 の外部定義シンボルへ分岐するジャンプテーブルを test.jmp に出力します。

[test.jmp の出力例]

```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 2009.07.19
          .glb _func01
          .glb _func02
          .SECTION    P,CODE
_func01:
          MOV.L       #1000H,R14
          JMP         R14
_func02:
          MOV.L       #2000H,R14
          JMP         R14
          .END
```

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
マイコン種別が RX 系以外の場合は、本オプションは無効です。
生成するジャンプテーブルは、P セクションへ出力します。
セクション名に指定できるセクション種別は、プログラムセクションのみです。

4.2.3　リストオプション

表 4.5　リストカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	リスト ファイル	LISt [= <ファイル名>]	リンカ <リスト> [リンケージリスト出力]	リストファイル出力を指定
2	リスト 内容	SHow [= <sub>[,...]] <sub> : {SYmbol| Reference| SEction| Xreference| Total_size| VECTOR| ALL }	リンカ <リスト> [リスト内容 :]	 シンボル情報 参照回数 セクション情報 クロスリファレンス情報 合計セクションサイズ ベクタ情報出力 全情報出力

リストファイル

LISt

リンカ <リスト>[リンケージリスト出力]

書　式　　LISt [= <ファイル名>]

説　明　　リストファイル出力およびリストファイル名を指定します。
リストファイル名を指定しない場合には、出力 (または先頭出力) ファイルと同じファイル名で、拡張子が form=library または extract 指定時「lbp」、それ以外のとき「map」のリストファイルが作成されます。

リスト内容

SHow

リンカ <リスト>[リスト内容 :]

書　式　　SHow[= <sub>[,…]]
　　<sub> : { SYmbol | Reference | SEction | Xreference | Total_size
　　|VECTOR | ALL }

説　明　　リストの出力内容を指定します。
サブオプションの一覧を表 4.6 に示します。
各リストの具体例については「7.3　リンケージリスト」、「7.4　ライブラリリスト」を参照してください。

表 4.6　show オプションのサブオプション一覧

	出力形式	サブオプション名	意味
1	form=library または extract 指定時	symbol	モジュール内シンボル名一覧を出力します。（extract 指定時）
		reference	指定できません。
		section	モジュール内セクション一覧を出力します。（extract 指定時）
		xreference	指定できません。
		total_size	指定できません。
		vector	指定できません。
		all	指定できません(extract 指定時)。 モジュール内シンボル名、セクション一覧を出力します(form=library 指定時)。
2	form=library 以外 かつ extract 指定なし時	symbol	シンボルアドレス、サイズ、種別、最適化内容を出力します。
		reference	シンボルの参照回数を出力します。
		section	指定できません。
		xreference	クロスリファレンス情報を出力します。
		total_size	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表示します。
		vector	ベクタ情報を出力します。
		all	show=symbol,xreferrence,total_size 指定時と同内容を出力します。(form=rel) show=symbol,total_sizel 指定時と同内容を出力します。(form=rel,data_stuff) show=symbol,reference,xreference ,total_size 指定時と同内容を出力します。(form=abs) show=symbol,reference,xreference,total_size 指定時と同内容を出力します。(form=hex/stype/bin) form=obj のときは指定できません。

備　考　　オプション form とオプション show および show=all で有効/無効になる組み合わせは以下のようになります。

		Symbol	Reference	Section	Xreference	Vector	Total_size
form=abs	show のみ	有効	有効	無効	無効	無効	無効
	show=all	有効	有効	無効	有効	有効	有効
form=lib	show のみ	有効	無効	有効	無効	無効	無効
	show=all	有効	無効	有効	無効	無効	無効
form=rel	show のみ	有効	無効	無効	無効	無効	無効
	show=all	有効	無効	無効	有効*1	無効	有効
form=obj	show のみ	有効	有効	無効	無効	無効	無効
	show=all	無効	無効	無効	無効	無効	無効
form=hex/bin/sty	show のみ	有効	有効	無効	無効	無効	無効
	show=all	有効	有効	無効	有効	有効*1	有効*1

*1　入力ファイルが absolute 形式の場合は無効です。

クロスリファレンス情報の出力に関しては、下記制限があります。

- 出力ファイルが relocatable 形式で、かつ data_stuff オプションを使用している場合、クロスリファレンス情報は出力できません。
- 入力ファイルが absolute 形式の場合、参照側アドレスの情報は出力されません。
- アセンブル時に goptimize オプションが指定されていない場合、同一ファイル内への分岐に関する情報は出力されません。(マイコンが H8,H8S,H8SX ファミリの場合のみ)
- 同一ファイル内の、定数シンボルへの参照に関する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。
- 外部変数アクセス最適化が有効な場合、ベースとなるシンボルを除いて、変数の参照情報は出力されません。(マイコンが SuperH ファミリおよび RX ファミリの場合のみ)
- ソースファイル上で#pragma tbr を記述した際にオフセット値を直接指定している場合、当該関数についての情報は出力されません。(マイコンが SH-2A/SH2A-FPU の場合のみ)
- リンク時の最適化を指定した場合、定数やリテラルの統合が生じると、その定数やリテラルに関する参照情報は出力されません。
- show=total_size で表示する情報は、別オプション total_size での表示内容と同じです。
- show=vector は、マイコン種別が RX ファミリ、M16C シリーズ、R8C ファミリのとき使用できます。
- show=reference が有効な場合に、#pragma address で指定された変数の参照回数が 0 として出力されます。(マイコンが SuperH ファミリおよび RX ファミリの場合のみ)

4.2.4 最適化オプション

表 4.7 最適化カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	最適化	OPtimize [= <sub>[,...]]	リンカ<最適化>	最適化あり
		<sub> : { STring_unify	[最適化方法 :]	定数/文字列の統合
		| SYmbol_delete	[最適化設定]	未参照シンボルの削除
		| Register	[設定 :]	レジスタ退避/回復の最適化
		| SAMe_code		共通コードの統合
		| Branch		分岐命令の最適化
		| SPeed		実行速度優先の最適化
		| SAFe }		安全な最適化
		NOOPtimize		最適化なし
2	共通コードサイズ	SAMESize = <サイズ>	リンカ<最適化>	共通コード統合の対象となる最小サイズの指定
		(省略時：sames=1e)	[統合サイズ :]	
3	プロファイル情報	PROfile = <ファイル名>	リンカ<最適化>	プロファイル情報ファイルの指定
			[プロファイル情報 :]	(動的最適化を行います)
4	キャッシュサイズ	CAchesize = <sub>	リンカ<最適化>	
		<sub>: Size = <サイズ>|	[キャッシュサイズ :]	キャッシュサイズの指定
		Align = <ラインサイズ>		キャッシュラインサイズの指定
		(省略時：ca=s=8,a=20)		(SuperH ファミリ向け)
5	最適化部分抑止	SYmbol_forbid =	リンカ<最適化>	未参照シンボル削除抑止シンボル
		<シンボル名>[,...]	[最適化方法 :]	
		SAMECode_forbid =	[最適化部分抑止]	共通コード統合抑止シンボル
		<関数名>[,...]		
		Variable_forbid =		短絶対アドレッシングモード活用抑止シンボル
		<シンボル名>[,...]		
		FUnction_forbid =		間接アドレッシングモード活用抑止シンボル
		<関数名>[,...]		
		SEction_forbid = <sub>[,...]		最適化抑止セクション
		<sub> : [<ファイル名>|		
		<モジュール名>]		
		(<セクション名>[,...])		
		Absolute_forbid =		最適化抑止アドレス範囲
		<アドレス> [+ <サイズ>] [,...]		

最適化

OPtimize
NOOPtimize

リンカ<最適化>[最適化方法 :][最適化設定][設定 :]

書　式　　OPtimize[= <サブオプション>[,…]]
NOOPtimize
<サブオプション> : {STring_unify | SYmbol_delete | Register | SAMe_code | Branch | SPeed | SAFe}

説　明　　モジュール間最適化実行有無を指定します。
optimize オプション指定時は、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最適化を行います。
nooptimize オプション指定時は、モジュールの最適化を行いません。
本オプションの省略時解釈は、optimize です。サブオプションの一覧を表 4.8 に示します。

表 4.8　optimize オプションのサブオプション一覧

サブオプション	意　味
パラメータなし	V.9.04 Release 01 以前 optimize=string_unify, symbol_delete, register, same_code, branch と同じです。 V.9.04 Release 02 以降 optimize=string_unify, symbol_delete, same_code, branch と同じです。
string_unify	const 属性を持つ定数に対し、同一値定数を統合します。const 属性を持つ定数には次のものが含まれます。 ・C/C++プログラム中の const 修飾型変数 ・文字列データの初期値/リテラル定数
symbol_delete	1 度も参照のない変数/関数を削除します。必ずコンパイル 時に#pragma entry を指定するか、optlnk で entry オプションを指定してください。
register	関数の呼び出し関係を解析し、レジスタの再割付および冗長なレジスタ退避/回復コードを削除します。必ずコンパイル 時に#pragma entry を指定するか、optlnk で entry オプションを指定してください。
same_code	複数の同一命令列をサブルーチン化します。
branch	プログラムの配置情報に基づいて、分岐命令サイズを最適化します。他の最適化項目を実行すると、指定の有無に関わらず必ず実行します。
speed	オブジェクトスピード低下を招く可能性のある最適化以外を実行します。 V.9.04 Release 01 以前 optimize=string_unify, symbol_delete, register, branch と同じです。 V.9.04 Release 02 以降 optimize=string_unify, symbol_delete, branch と同じです。
safe	変数や関数の属性によって制限される可能性のある最適化以外を実行します。 V.9.04 Release 01 以前 optimize=string_unify, register, branch と同じです。 V.9.04 Release 02 以降 optimize=string_unify, branch と同じです。

備　考　form=｛object|relocate|library｝または strip 指定時、本オプションは無効です。
コンパイル時に外部変数アクセス最適化を指定した場合、定数/リテラル統合最適化(optimize=string_unify)は無効になります。
マイコン種別が SH-2A/SH2A-FPU の場合、optimize=register の機能によってコードサイズが増加する場合があります。
プログラム内に#pragma entry で実行開始関数を指定、または実行開始アドレス(entry)を指定していない場合、optimize=symbol_delete は無効になります。

共通コードサイズ

SAMESize

リンカ<最適化>[統合サイズ :]

書　式　SAMESize = <サイズ>

説　明　共通コード統合最適化(optimize=same_code)で、最適化対象となる最小コードサイズを指定します。8～7FFF までの 16 進数で指定してください。
本オプションの省略時解釈は、samesize=1E です。

備　考　optimize=same_code の指定がないとき、本オプションは無効です。

プロファイル情報

PROfile

リンカ<最適化>[プロファイル情報 :]

書　式　　PROfile = <ファイル名>

説　明　　プロファイル情報ファイルを指定します。
プロファイル情報ファイルとして指定できるのは、ルネサス統合開発環境 Ver. 2.0 以降が出力するプロファイル情報ファイルだけです。
プロファイル情報ファイルを指定すると、モジュール間最適化で動的情報に基づいた最適化を実行できます。
プロファイル情報入力により影響がある最適化を表 4.9 に示します。

表 4.9　プロファイル情報と最適化の関係

サブオプション	意　味	最適化対象プログラム*1			
		SHC	SHA	H8C	H8A
variable_access	動的アクセス回数の多い変数を優先的に割り当てます。	×	×	○	○
function_call	動的アクセス回数の多い関数の最適化優先順位を下げます。	×	×	○	○
branch	動的に呼び出し回数の多い関数を呼び出し元の関数の近くに配置します。 SuperH ファミリ用プログラムの場合は、cachesize オプションで指定するキャッシュサイズを意識した配置最適化を行います。	○	△ *2	○	△

【注】　*1　SHC: SuperH ファミリ用 C/C++プログラム、SHA: SuperH ファミリ用アセンブリプログラム、
H8C: H8,H8S,H8SX ファミリ用 C/C++プログラム、H8A: H8,H8S,H8SX ファミリ用アセンブリプログラム

*2　関数単位の移動は行いませんが、入力ファイル単位の移動は実行します。

備　考　　optimize 指定がないとき、本オプションは無効です。

キャッシュサイズ

CAchesize

リンカ<最適化>[キャッシュサイズ :]

書　式　　CAchesize = <sub>
<sub>:Size = <サイズ> | Align = <ラインサイズ>

説　明　　キャッシュサイズおよびキャッシュラインサイズを指定します。
profile オプション指定時、分岐命令最適化(optimize=branch)で使用します。
サイズはキロバイト単位、ラインサイズはバイト単位の 16 進数で指定してください。
本オプションの省略時解釈は、cachesize=size=8,align=20 です。

備　考　　profile 指定がないとき、本オプションは無効です。

最適化部分抑止

SYmbol_forbid
SAMECode_forbid
Variable_forbid
FUnction_forbid
SEction_forbid
Absolute_forbid

リンカ<最適化>[最適化方法 :][最適化部分抑止]

書　式　　SYmbol_forbid = <シンボル名>[,…]
SAMECode_forbid = <関数名>[,…]
Variable_forbid = <シンボル名>[,…]
FUnction_forbid = <関数名>[,…]
SEction_forbid = <sub>[,…]
<sub>: [<ファイル名>|<モジュール名>](<セクション名>[,…])
Absolute_forbid = <アドレス>[+ <サイズ>][,…]

説　明　　特定のシンボル、セクション、アドレス範囲の最適化を抑止します。アドレス、サイズは 16 進数で指定してください。C/C++変数名、C 関数名はプログラム中での定義名先頭に_を付加します。C++関数の場合は、引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。但し引数が void の場合は、"関数名()"で指定します。各オプションの意味を**表 4.10** に示します。

表 4.10　最適化部分抑止オプション一覧

オプション	パラメータ	意味
symbol_forbid	関数名|変数名	未参照シンボル削除最適化を抑止します。
samecode_forbid	関数名	共通コード統合最適化を抑止します。
variable_forbid	変数名	短絶対アドレッシングモード活用最適化を抑止します。
function_forbid	関数名	間接アドレッシングモード活用最適化を抑止します。
section_forbid	セクション名 ファイル名 モジュール名	特定セクションの最適化を抑止します。入力ファイル名、もしくはライブラリモジュール名を同時に指定することで、最適化抑止対象をセクション全体だけでなく、特定ファイルに限定することも可能です。
absolute_forbid	アドレス[+サイズ]	アドレス＋サイズの範囲の最適化を抑止します。

例

```
symbol_forbid="f(int)"  ; C++関数 f(int) は参照回数 0 でも削除しません。
section_forbid=(P1)     ; P1 セクションへの全最適化を抑止します。
section_forbid=a.obj(P1,P2)
; a.obj 内の P1,P2 セクションへの全最適化を抑止します。
```

備　考　最適化を使用しないリンク処理では、本オプションは無効です。
パスを記述した入力ファイルを最適化抑止する場合、section_forbid オプションではファイル名にパスを記述してください。

4.2.5 セクションオプション

表 4.11 セクションカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	セクションアドレス	STARt= <sub>[,...] <sub> : [(]<セクション名> [{: | ,} <セクション名>[,...]] [)][,...] [/<アドレス>]	リンカ <セクション> [設定項目 :] [セクション]	セクションの開始アドレス指定
2	シンボルアドレスファイル	FSymbol = <セクション名>[,...]	リンカ <セクション> [設定項目 :] [シンボルアドレスファイル]	外部定義シンボルアドレスの定義ファイル出力
3	セクションアライメント指定	ALIGNED_SECTION= <セクション名>[,...]	リンカ <セクション> [設定項目 :] [セクションアライメント指定]	セクションアライメントを16バイトに変更

セクションアドレス

STARt

リンカ <セクション>[設定項目 :][セクション]

書 式

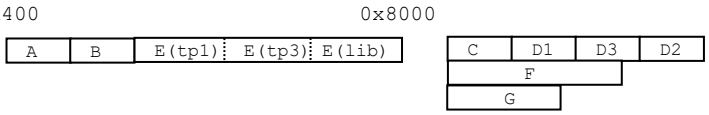
```
STARt = <sub> [,…]
<sub>: [(]<セクション名>[{:|,} <セクション名>[,…]][)][,…][/<アドレス>]
```

説 明

セクションの開始アドレスを指定します。アドレスは16進数で指定してください。
セクション名はワイルドカード"*"も指定できます。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、丸括弧"()"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できます。
同一セクション内オブジェクトは、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。
アドレスの指定がない場合は、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。

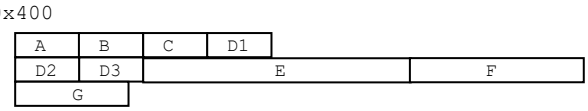
例　　下記順番でオブジェクトを入力する場合のセクション配置を例に示します。
(括弧内は各オブジェクトが持つセクション)
tp1.obj(A,D1,E) -> tp2.obj(B,D3,F) -> tp3.obj(C,D2,E,G) -> lib.lib(E)

(1) -start=A,B,E/400,C,D*:F:G/8000



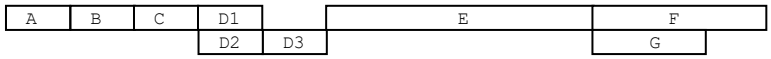
- ":"で区切ったC,F,Gセクションは、同一アドレスに割りつきます。
- ワイルドカードで記述したセクション(ここではDで始まる名前のセクション)は、入力した順番で割りつきます。
- 同名セクション内(ここではEセクション)は、入力したオブジェクトから順番に割りつきます。
- ライブラリ入力による同名セクション(ここではEセクション)は、入力オブジェクトの次に割りつきます。

(2) -start=A,B,C,D1:D2,D3,E,F:G/400



- ":"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、それぞれ先頭が同一アドレスに割りつきます。

(3) -start=A,B,C,(D1:D2,D3),E,(F:G)/400



- "()"で同一アドレス配置を括った場合、"()"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(この場合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　　form={object | relocate | library}またはstrip指定時、本オプションは無効です。
括弧"()"は、ネストして記述することはできません。
括弧"()"内では、少なくともひとつはコロン":"の記述が必要です。コロン":"を記述しない場合には、括弧"()"は記述できません。
括弧"()"を記述した場合、"()"外にコロン":"を記述することはできません。
括弧"()"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。

シンボルアドレスファイル

FSymbol

リンカ <セクション>[設定項目 :][シンボルアドレスファイル]

書　式　FSymbol = <セクション名>[,…]

説　明　指定したセクション内外部定義シンボルをアセンブラ制御命令形式でファイルに出力します。ファイル名は、<出力ファイル>.fsy です。

例

```
fsymbol=sct2,sct3
output=test.abs
```

セクション sct2,sct3 の外部定義シンボルを test.fsy に出力します。

[test.fsy の出力例]

```
;OPTIMIZING LINKAGE EDITOR GENERATED FILE 1999.11.26
;fsymbol = sct2, sct3

;SECTION NAME = sct2
 .export _f
_f: .equ h'00000000
 .export _g
_g: .equ h'00000016
;SECTION NAME = sct3
 .export _main
_main: .equ h'00000020
 .end
```

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
マイコン種別が H8,H8S,H8SX ファミリ, SuperH ファミリ, RX ファミリのときに使用できます。

セクションアライメント数を 16 バイトに変更

ALIGNED_SECTION

リンカ<セクション>[設定項目:][セクション]

書　式　ALIGNED_SECTION = <セクション名>[,…]

説　明　指定セクションのアライメント数を 16 byte に変更します。

備　考　form={object | relocate | library}および extract、strip 指定時、本オプションは無効です。

4.2.6　ベリファイオプション

表 4.12　ベリファイカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	アドレス整合性のチェック	CPu = { <cpu 情報ファイル名> | <メモリ種別> = 　<アドレス範囲>[,...] | STRIDE } <メモリ種別> = 　{ ROm | RAm 　| XROm | XRAm 　| YROm | YRAm | FIX} <アドレス範囲>: 　<先頭アドレス> 　　- <終了アドレス>	リンカ <ベリファイ> [アドレス整合チェック :]	セクションアドレスの割り付け可能範囲を指定 セクション名をセクション分割の対象に指定
2	物理空間上の重複チェック	PS_check=<sub>[:<sub>...] 　<sub>: <LS>,<LS>[,...] 　<LS>: <開始アドレス> 　　-<終端アドレス>	リンカ <ベリファイ> [物理空間上の重複チェック :]	物理空間上で重なり合うアドレス範囲を指定
3	セクション分割対象外指定	CONTIGUOUS_SECTION = <セクション名>[,…]	リンカ <ベリファイ> [分割対象がセクション :]	セクション名をセクション分割の対象外セクションに指定

アドレス整合性のチェック

CPu

リンカ <ベリファイ>[アドレス整合チェック :]

書　式

```
CPu = { <cpu 情報ファイル名>
      | <メモリ種別> = <アドレス範囲>[,…]
      | STRIDE}
＜メモリ種別＞ = { ROm | RAm | XROm | XRAm | YROm | YRAm | FIX }
＜アドレス範囲＞ : <先頭アドレス> - <終了アドレス>
```

説　明

cpu=stride 未指定時は、セクションの割り付けアドレスに対して、アドレス範囲に入らない場合は、エラーを出力します。
cpu=stride 指定時は、セクションの割り付けアドレスに対して、アドレス範囲に入らない場合は、次の同メモリ種別に配置、または、分割して配置します。

[例]サブオプション stride を指定しない場合
start=D1,D2/100
cpu=ROM=100-1FF,RAM=200-2FF
D1 が 100-1FF、D2 が 200-2FF の範囲に収まるとき、正常終了します。収まらないときエラーを出力します。
[例]サブオプション stride を指定した場合
start=D1,D2/100
cpu=ROM=100-1FF,RAM=200-2FF,ROM=300-3FF
cpu=stride
D1,D2 が ROM 属性の領域に(セクションを分割して/分割しないで)収まるとき、正常終了します。セクションを分割しても収まらないときリンクエラーになります。

xrom/xram は DSP の X メモリ領域、yrom/yram は DSP の Y メモリ領域を指定します。
セクション割り付けが可能なアドレス範囲を 16 進数で指定してください。ROM/RAM の属性は、モジュール間最適化で使用します。
メモリ種別”FIX”には、アドレス固定の領域(I/O エリア等)を指定します。
メモリ種別”FIX”と、それ以外のメモリ種別のアドレス範囲が重複した場合は、メモリ種別”FIX”を有効とします。

サブオプション stride は、メモリ種別が、ROM または RAM で、アドレス範囲にセクションが収まらなかった場合に、セクションを分割して同じメモリ種別の領域に割り付けます。
サブオプション stride で、セクションを分割する単位は、モジュール単位になります。
[例]
cpu=ROM=0-FFFF,RAM=10000-1FFFF
セクションアドレスが、0-FFFF または 10000-1FFFF の間に入っているかチェックします。
モジュール間最適化では、異なる属性間でのオブジェクトの移動は行いません。

cpu=ROM=100-1FF,ROM=400-4FF,RAM=500-5FF
cpu=stride
セクションアドレスが、100-1FF の間に収まらなかった場合に、セクションをモジュール単位で分割して 400-4FF に割り付けます

備　考　form={object | relocate | library}または strip 指定時、本オプションは無効です。
cpu=stride および memory=low 指定時、無効になります。
マイコン種別が SH2DSP,SH3DSP,SH4ALDSP 以外の場合は、メモリ種別が xrom,xram,yrom,yram の指定は無効となります。
cpu=stride および optimize=register が有効な場合、L2320 エラーが出力されることがあります。その場合には、optimize=register を無効にしてください。
cpu=stride を指定し、B セクションが分割された場合、0 初期化するための情報として 8 バイト×分割数分だけ C$BSEC セクションのサイズが増加します。

物理空間上の重複チェック

PS_check

リンカ <ベリファイ>[物理空間上の重複チェック :]

書　式

```
PS_check=<sub>[:<sub>...]
<sub>: <LS>,<LS>[,...]
<LS>: <開始アドレス>-<終端アドレス>
```

説　明　アドレス値では重なっていないが、実際にメモリ上に配置すると重なってしまうオブジェクトを検出するためのオプションです。
本オプションを使用することにより、SH3 や SH4 などで、論理アドレス上では重ならないが実メモリ上に配置する際に重なってしまうオブジェクトを検出することが可能です。
本オプションによって重複を検出した場合、エラーとしてリンク処理を終了します。
メモリ上で重なり合うアドレス範囲(書式の中の<LS>)をオプションに記述してください。
複数の物理メモリに対してチェックしたい場合には、':'で区切って記述することでチェック可能です。

例　SH4 は、MMU が無効状態の場合、4G バイトのアドレス空間は、512M バイト(29bit)の外部メモリ空間へマッピングします(4G バイトアドレスの上位 3bit を無視してマッピングします)。
たとえば、ユーザモードで使用可能な U0 領域(00000000～0x7fffffff)に対して、外部メモリ(512M)にマッピングする場合のオブジェクトの重なりは、下記記述で検出可能です。

```
-PS_check=00000000-1fffffff,20000000-3fffffff,40000000-
5fffffff,60000000-7fffffff
```

本オプション記述により、00000000,20000000,40000000,60000000 番地はすべて、実メモリ上では同じ場所に配置されることを表します。

備　考　本オプションは、SuperH ファミリのマイコンに対してのみ有効です。
出力形式(form オプション)が object,relocate,library の場合、本オプションは無効です。
absolute ファイルを入力する場合の処理は、本オプションは無効です。
マイコンのアドレス空間の仕様については、各マイコンのハードウェアマニュアルを参照してください。

セクション分割対象外指定

CONTIGUOUS_SECTION

リンカ<ベリファイ>[分割対象外セクション :]

書　式　　CONTIGUOUS_SECTION=<セクション名>[,...]

説　明　　cpu=stride が有効なときに、セクションを分割せずに同じメモリ種別の割り付け可能なアドレス領域に割り付けるセクションを指定します。

[例]
start=P,PA,PB/100
cpu=ROM=100-1FF,ROM=300-3FF,ROM=500-5FF
cpu=stride
contiguous_section=PA

セクション P を 100 番地に割り付けます。
contiguous_section 指定したセクション PA が、1FF 番地までに割り付けることができない場合、セクション PA を分割せずに、300 番地から割り付けます。
contiguous_section 指定してないセクション PB が、3FF 番地までに割り付けることができない場合、セクション PB を分割して、500 番地から割り付けます。

備　考　　cpu オプションのサブオプションの stride が無効なとき、本オプションは無効です。

4.2.7　その他オプション

表 4.13　その他カテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	終端コード	S9	リンカ <その他> [その他のオプション :] [S9 レコードを終端に出力]	S9 レコードを常に出力
2	スタック情報ファイル	STACk	リンカ <その他> [その他のオプション :] [スタック情報ファイル(sni)出力]	スタック使用量情報ファイル出力
3	デバッグ情報圧縮	COmpress NOCOmpress	リンカ <その他> [その他のオプション :] [デバッグ情報圧縮]	デバッグ情報を圧縮する デバッグ情報を圧縮しない
4	メモリ使用量削減指定	MEMory = [High｜Low]	リンカ <その他> [その他のオプション :] [入力ファイルロード時のメモリ使用量削減]	入力ファイルをロードする際のメモリ使用量指定
5	シンボル名変更	REName = <sub>[,...] <sub> : { [<ファイル名>] (<名前>=<名前>[,...]) | [<モジュール名>] (<名前>=<名前>[,...]) }	リンカ <その他> [ユーザ指定オプション :]	シンボル名、セクション名の変更
6	シンボル名削除	DELete = <sub>[,...] <sub> : { <モジュール名> | [<ファイル名>] (<名前>[,...]) }	リンカ <その他> [ユーザ指定オプション :]	シンボル名、モジュール名の削除
7	モジュールの置き換え	REPlace = <sub>[,...] <sub> : <ファイル> [(<モジュール>[,...])]	リンカ <その他> [ユーザ指定オプション :]	ライブラリファイル内同名 モジュールの置き換え
8	モジュールの抽出	EXTract = <モジュール>[,...]	リンカ <その他> [ユーザ指定オプション :]	ライブラリファイル内指定 モジュールの抽出
9	デバッグ情報削除	STRip	リンカ <その他> [ユーザ指定オプション :]	アブソリュートファイル、 ライブラリファイルのデバッグ情報削除

	項目	コマンドライン形式	ダイアログメニュー	指定内容
10	メッセージレベル	CHange_message = <sub>[,...] <sub>: {Information | Warning | Error} [=<エラー番号> [-<エラー番号>] [,...]]	リンカ <その他> [ユーザ指定オプション :]	メッセージレベルの変更
11	ローカルシンボル名秘匿指定	Hide	リンカ <その他> [ユーザ指定オプション :]	ローカルシンボル名情報を削除
12	合計セクションサイズの表示	Total_size	リンカ <その他> [その他オプション :] [合計セクションサイズの画面表示 :]	標準出力へ、リンク後の合計セクションサイズを表示できます。
13	エミュレータ向けの情報ファイル	RTs_file	リンカ<その他> [その他のオプション :] [関数出口情報ファイル(rts)出力]	エミュレータ向けの情報ファイルを出力します。 (SuperH ファミリ向け)

終端コード

S9

リンカ <その他>[その他のオプション :][S9 レコードを終端に出力]

書　式　S9

説　明　エントリアドレスが 0x10000 を超える場合でも、S9 レコードを終端に出力します。

備　考　form=stype 指定がないとき、本オプションは無効です。

スタック情報ファイル

STACk

リンカ <その他>[その他のオプション :][スタック情報ファイル(sni)出力]

書　式　STACk

説　明　スタック使用量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 になります。

備　考　form={object | relocate | library}および strip 指定時、本オプションは無効です。

デバッグ情報圧縮

COmpress
NOCOmpress

リンカ <その他>[その他のオプション :][デバッグ情報圧縮]

書　式　COmpress
NOCOmpress

説　明　デバッグ情報の圧縮有無を指定します。
compress オプションを指定した場合、デバッグ情報を圧縮します。
nocompress オプションを指定した場合、デバッグ情報を圧縮しません。
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。また、nocompress オプションを指定すると、リンク時間が短くなります。
本オプションの省略時解釈は、nocompress です。

備　考　form={object | relocate | library | hexadecimal | stype | binary}
または strip オプションを指定した場合、compress オプションは無効です。

メモリ使用量削減指定

MEMory

リンカ <その他>[その他のオプション :][入力ファイルロード時のメモリ使用量削減]

書　式　MEMory = [High | Low]

説　明　リンク時に使用するメモリ量を指定します。
memory=high オプションを指定した場合、従来通りの処理を行います。
memory=low オプションを指定した場合、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 memory=high オプション指定より処理が遅くなります。

大規模なプロジェクトをリンクした際、最適化リンケージエディタのメモリ使用量が稼動マシンの実装メモリ量を越えてしまい、動作が遅くなっているような場合には memory=low オプション指定をお試しください。

備　考　下記オプションを指定した場合、memory=low オプション指定は無効となります。
form=absolute,hexadecimal,stype,binary 指定時
　compress,delete,rename,map,stack,cpu=stride
　list と show[={reference | xreference}]を同時指定
form=library 指定時
　delete,rename,extract,hide,replace
form=object,relocate 指定時
　extract
　マイコン種別が NC ファミリ以外で optimize を指定時
また、入力ファイルや出力ファイル形式によっても無効となる組み合わせがあります。詳細は、**「4.2.2 出力オプション」**の**表 4.4** を参照してください。

シンボル名変更

REName

リンカ <その他>[ユーザ指定オプション :]

書　式

```
REName = <サブオプション>[,…]
<サブオプション> :  { [<ファイル>](<名前> = <名前>[,…])
                    | [<モジュール>](<名前> = <名前>[,…])}
```

説　明

外部シンボル名、セクション名を変更します。
特定のファイルまたは特定のライブラリ内モジュールに含まれるシンボル名、セクション名を変更することもできます。
C/C++変数名の場合、プログラム中での定義名先頭に_を付加します。
関数名を変更した場合の動作は保証できません。
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、シンボル名を優先します。
同一ファイル名、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。

例

```
rename=(_sym1=data)      ;_sym1 を data に変更します。
rename=lib1(P=P1)        ;ライブラリモジュール lib1 内の P セクションを
                         ;P1 セクションに変更します。
```

備　考

extract または strip 指定時、本オプションは無効です。
form=absolute 指定時、入力されたライブラリのセクション名を変更することができません。

シンボル名削除

DELete

リンカ <その他>[ユーザ指定オプション :]

書　式

```
DELete = <サブオプション>[,…]
<サブオプション> :  { [<ファイル>](<名前>[,…])
                    | <モジュール> }
```

説　明

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。
特定のファイル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/C++変数名、C 関数名はプログラム中での定義名先頭に_を付加します。C++関数の場合は、引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定します。但し引数が void の場合は、"関数名()"で指定します。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。

例

```
delete=(_sym1)           ; 全ファイル中のシンボル名_sym1 を削除します。
delete=file1.obj(_sym2) ;file1.obj 内のシンボル名_sym2 を削除します。
```

備　考

extract または strip 指定時、本オプションは無効です。
form=library のときに、モジュールを削除できます。
form={absolute|relocate|hexadecimal|stype|binary}のときに、外部シンボルを削除できます。

モジュールの置き換え

REPlace

リンカ <その他>[ユーザ指定オプション :]

書　式　　REPlace = <サブオプション>[,…]
<サブオプション> : <ファイル名>[(<モジュール名>[,…])]

説　明　　ライブラリモジュールを置換します。
指定したファイルまたはライブラリモジュールと library オプションで指定したライブラリ内同名モジュールを置換します。

例　　replace=file1.obj　　;モジュール file1 と file1.obj を置換します。
replace=lib1.lib(mdl1)　;モジュール mdl1 とライブラリファイル lib1.lib 内
;モジュール mdl1 を置換します。

備　考　　form={object | relocate | absolute | hexadecimal | stype | binary}
および extract、strip 指定時、本オプションは無効です。

モジュールの抽出

EXTract

リンカ <その他>[ユーザ指定オプション :]

書　式　　EXTract = <モジュール名>[,…]

説　明　　ライブラリモジュールを抽出します。
指定したライブラリモジュールを 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。

例　　extract=file1　　;モジュール file1 を抽出します。

備　考　　form={absolute | hexadecimal | stype | binary}および strip 指定時、本オプションは無効です。
form=library 指定時、モジュールを削除できます。
form={absolute|relocate|hexadecimal|stype|binary}指定時、外部シンボルを削除できます。

デバッグ情報削除

STRip

リンカ <その他>[ユーザ指定オプション :]

書　式　STRip

説　明　アブソリュートファイル、ライブラリファイルのデバッグ情報を削除します。
strip オプション指定時は、入力ファイルと出力ファイルは 1 対 1 対応になります。

例　input=file1.abs file2.abs file3.abs
strip
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。デバッグ情報削除前のファイルは、file1.abk, file2.abk, file3.abk にバックアップします。

備　考　form={object | relocate | hexadecimal | stype | binary}指定時、本オプションは無効です。

メッセージレベル

CHange_message

リンカ <その他>[ユーザ指定オプション :]

書　式　CHange_message = <サブオプション>[,…]
<サブオプション> : <エラーレベル>[=<エラー番号>[-<エラー番号>][,…]]
<エラーレベル>　: {Information | Warning | Error}

説　明　インフォメーション、ウォーニング、エラーレベルのメッセージレベルを変更します。
メッセージ出力時の実行継続/中断を変更できます。

例　change_message=warning=2310
L2310 をウォーニングレベルに変更し、L2310 出力時も処理を継続します。

change_message=error
全てのインフォメーション、ウォーニングメッセージをエラーレベルに変更します。
メッセージを一つでも出力すると、処理を中断します。

ローカルシンボル名秘匿指定

Hide

リンカ <その他>[ユーザ指定オプション :]

書　式　Hide

説　明　本オプションを指定した場合、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ますので、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は確認できなくなります。生成されるファイルの動作への影響は一切ありません。
ローカルシンボル名を機密扱いにしたい場合などに本オプションを指定してください。

秘匿対象となるシンボルの種類を以下に挙げます。
- ・ソースファイル：static 型修飾子を指定した変数名、関数名など
- ・ソースファイル：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

※ エントリ関数名は秘匿対象になりません。

例　ソースファイルで本オプションの機能が有効となる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　本オプションは出力ファイル形式が absolute,relocate,library の場合のみ有効です。
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本オプションを指定してください。
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。

合計セクションサイズの表示

Total_size

リンカ<その他>[その他のオプション :] [合計セクションサイズ画面表示]

書　式　Total_size

説　明　リンク後のセクションの合計サイズを、標準出力に表示するためのオプションです。
下記の 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

本オプションを使用することにより、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。

備　考　リンケージリストへ合計サイズを表示するには、別途 show=total_size オプションを使用する必要があります。
ROM 化支援機能(rom オプション)対象のセクションの場合、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。

エミュレータ向けの情報ファイル

RTs_file

リンカ<その他>[その他のオプション :][関数出口情報ファイル(rts)出力]

書　式　-RTs_file

説　明　エミュレータで使用するための情報、関数出口情報ファイル(.rts ファイル)を生成するオプションです。
お使いのエミュレータのマニュアルに従って、本オプションを使用してください。エミュレータの機種によって使用できない場合があります。
関数出口情報ファイルは、「<出力するロードモジュール名>.rts」というファイル名で生成されます。例えば、output オプションで指定する出力ファイル名を「test.abs」とした場合、関数出口情報ファイルは「test.rts」というファイル名で生成されます。
関数出口情報ファイルはロードモジュールと同じディレクトリに作成されます。

備　考　form={object | relocate | library}指定時、本オプションは無効です。
アブソリュートファイルを入力する場合、本オプションは無効です。
エミュレータのマニュアルに従って本オプションを使用してください。エミュレータの機種によって使用できない場合があります。
本オプションは、マイコン種別が SuperH ファミリのとき使用できます。

4.2.8　サブコマンドファイルオプション

表 4.14　サブコマンドファイルカテゴリオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	サブコマンドファイル	SUbcommand = <ファイル名>	リンカ <サブコマンドファイル> [サブコマンドファイルを指定]	サブコマンドファイルによるオプション指定

サブコマンドファイル

SUbcommand

リンカ<サブコマンドファイル> [サブコマンドファイルを指定]

書　式　　SUbcommand = <ファイル名>

説　明　　オプションをサブコマンドファイルで指定します。
サブコマンドファイルの書式は以下の通りです。

<オプション> {= | △} [<サブオプション> [,…]][△&] [;<コメント>]

オプションとサブオプションの区切りは、=の代わりに空白も指定できます。
input オプションの場合は、サブオプション区切りに空白を指定できます。
サブコマンドファイル内では 1 オプション/行で指定します。
サブオプションを 1 行に記述できない場合は、&を用いて継続指定できます。
サブコマンドファイル中に subcommand オプションは指定できません。

例　　コマンドライン指定：optlnk file1.obj -sub=test.sub file4.obj
サブコマンド指定　：input　file2.obj file3.obj ;ここはコメントです。
library　lib1.lib, &　;継続行を指定します。
lib2.lib
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サブコマンド指定位置に展開し、実行します。
ファイルの入力順序は、file1.obj, file2.obj, file3.obj, file4.obj になります。

4.2.9 マイコンオプション

表 4.15 CPU タブオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	SBR アドレス指定	SBr = { <SBR アドレス> | User}	CPU [SBR 値 :]	8bit 絶対領域の開始アドレスを指定(H8SX ファミリ向け)

8bit 絶対領域アドレス値指定

SBr

CPU [SBR 値 :]

書 式 SBr = { <アドレス>|User }

説 明 SBR のアドレス値を指定します。
本オプションでアドレス値を指定することにより、abs8 領域を用いた最適化が可能になります。本オプションで user を指定した場合は、abs8 領域への最適化は抑止されます。

備 考 本オプションはマイコン種別が H8SX ファミリの場合にのみ有効です。
ソース内、あるいはツールのオプション指定などで、複数の SBR アドレスが指定された場合には、本オプションは指定の如何に関わらず user が指定されたものとして扱われます。

4.2.10 残りのオプション

表 4.16 残りのオプション一覧

	項目	コマンドライン形式	ダイアログメニュー	指定内容
1	コピーライト	LOgo	-	出力あり
		NOLOgo	(常に NOLOgo が有効)	出力なし
2	継続指定	END	-	既入力オプション列を実行し、処理終了後は以降のオプション列を入力し、処理を継続
3	終了指定	EXIt	-	オプション入力の終了を指定

コピーライト

LOgo
NOLOgo

なし(常に nologo が有効)

書　式　　LOgo
NOLOgo

説　明　　コピーライトの出力有無を指定します。
logo オプション指定時はコピーライト表示を出力します。
nologo オプション指定時はコピーライト表示出力を抑止します。
本オプションの省略時解釈は、logo です。

継続処理

END

なし

書　式　　END

説　明　　END より前に指定したオプション列を実行します。リンケージ処理終了後、END 以降に指定したオプション列の入力、リンケージ処理を継続します。
本オプションは、コマンドライン上では指定できません。

例

```
input=a.obj,b.obj              ; 処理(1)
start=P,C,D/100,B/8000         ; 処理(2)
output=a.abs                   ; 処理(3)
end
input=a.abs                    ; 処理(4)
form=stype                     ; 処理(5)
output=a.mot                   ; 処理(6)
```

(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。

終了処理

EXIt

なし

書　式　　EXIt

説　明　　オプション指定の終了を指定します。
本オプションは、コマンドライン上では指定できません。

例　　コマンドライン指定：

```
optlnk -sub=test.sub -nodebug
test.sub:          input=a.obj,b.obj ; 処理(1)
                   start=P,C,D/100,B/8000      ; 処理(2)
                   output=a.abs                ; 処理(3)
                   exit
```

(1)～(3)の処理を実行し、a.abs を出力します。
Exit 実行後のコマンドライン指定の nodebug オプションは無効になります。

5.　標準ライブラリ構築ツール操作方法

5.1　オプション指定規則

5.1.1　コマンドラインの形式

標準ライブラリ構築ツールを起動するコマンドラインの形式は以下の通りです。

```
lbgsh[△<オプション列>...]
<オプション列>：-<オプション>[=<サブオプション>[,...]]
```

5.2　オプション解説

標準ライブラリ構築ツールのオプション、サブオプションは、コンパイラオプションに準拠します。以下にコンパイラオプションとの相違を示します。コンパイラオプションの詳細は、「**第 2 章 C/C++コンパイラ操作方法**」を参照して下さい。

コマンドライン形式の英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。また、統合開発環境の対応するダイアログメニューを、タブ名<カテゴリ名>[項目]で示します。

5.2.1　追加オプション

表 5.1　追加オプション一覧

項目	オプション	ダイアログメニュー	指定内容
1 対象ヘッダファイル	Head = <sub>[,...] <sub>:{ ALL | RUNTIME | CTYPE | MATH | MATHF | STDARG | STDIO | STDLIB | STRING | IOS | NEW | COMPLEX | CPPSTRING }	標準ライブラリ <標準ライブラリ> **[カテゴリ:]**	構築対象ファイルを指定 全てのライブラリ関数 ランタイムライブラリ ctype.h＋ランタイムライブラリ math.h＋ランタイムライブラリ mathf.h＋ランタイムライブラリ stdarg.h＋ランタイムライブラリ stdio.h＋ランタイムライブラリ stdlib.h＋ランタイムライブラリ string.h＋ランタイムライブラリ ios＋ランタイムライブラリ new＋ランタイムライブラリ complex＋ランタイムライブラリ string＋ランタイムライブラリ
2 出力ファイル	OUTPut = <ファイル名>	標準ライブラリ <オブジェクト> **[オブジェクト出力ディレクトリ:]**	出力ライブラリファイル名を指定
3 簡易入出力関数	NOFLoat	標準ライブラリ <オブジェクト> **[簡易入出力関数]**	簡易入出力関数の生成
4 リエントラントライブラリ	REent	標準ライブラリ <オブジェクト> **[リエントラントライブラリを生成]**	リエントラントライブラリを生成

対象ヘッダファイル

Head

標準ライブラリ<標準ライブラリ>[カテゴリ:]

書　式

```
Head = <sub>[,...]
    <sub>:{ ALL          |
            RUNTIME      |
            CTYPE        |
            MATH         |
            MATHF        |
            STDARG       |
            STDIO        |
            STDLIB       |
            STRING       |
            IOS          |
            NEW          |
            COMPLEX      |
            CPPSTRING    }
```

説　明

構築対象ファイルをヘッダファイル名で指定します。
各ヘッダファイルとライブラリ関数との関係は、「10.4 C/C++ライブラリ」を参照してください。ランタイムライブラリ(runtime)は常に構築対象ファイルになります。
本オプションの省略時解釈は、head=all です。

例

```
lbgsh -output=sh2.lib -head=mathf -cpu=sh2
```

mathf.h で定義されたライブラリ関数とランタイムライブラリを-cpu=sh2 でコンパイルし、ライブラリファイル sh2.lib を出力します。

出力ファイル

OUTPut

標準ライブラリ<オブジェクト>[オブジェクト出力ディレクトリ:]

書　式

```
OUTPut = <ファイル名>
```

説　明

出力ファイル名を指定します。
本オプションの省略時解釈は、output=stdlib.lib です。

例

```
lbgsh -output=sh2.lib -optimize=1 -speed -goptimize -cpu=sh2
```

全標準ライブラリ用ソースファイルを、-optimize=1 -speed -goptimize -cpu=sh2 でコンパイルし、ライブラリファイル sh2.lib を出力します。

簡易入出力関数

NOFLoat

標準ライブラリ<オブジェクト>[簡易入出力関数]

書　式　　NOFLoat

説　明　　浮動小数点変換(%f、%e、%E、%g、%G)をサポートしない、簡易入出力関数を生成します。浮動小数点変換を必要としないファイル入出力を行う場合、ROMサイズを削減することができます。
<対象関数>
fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf

備　考　　入出力関数で浮動小数点型を指定した場合に、本オプションを指定して作成したライブラリをリンクした時の動作は保証しません。

リエントラントライブラリ

REent

標準ライブラリ<オブジェクト>[リエントラントライブラリを生成]

書　式　　REent

説　明　　リエントラントライブラリを生成します。ただし、rand、srand関数はリエントラントではありません。

備　考　　リエントラントライブラリをリンクする場合は、プログラム内で標準インクルードファイルをインクルードする前に、_REENTRANTというマクロ名を#define文で定義(#define _REENTRANT)するか、コンパイル時にdefineオプションで_REENTRANTを定義してください。

5.2.2 指定不可オプション

標準ライブラリ構築ツールで指定できないオプションを表 5.2に示します。指定した場合は無視されます。

表 5.2 指定不可オプション一覧

	項目	オプション	コンパイラ解釈	内容
1	インクルードファイルフォルダ	include	なし	—
2	ファイル間インライン展開フォルダ指定	file_inline_path	なし	—
3	マクロ名の定義	define	なし	—
4	インフォメーションメッセージ	message nomessage	nomessage	出力なし
5	プリプロセッサ展開	preprocessor	なし	—
6	プリプロセッサ展開時出力制限	noline	なし	—
7	オブジェクト形式	code	code = machinecode	機械語プログラムを出力
8	デバッグ情報	debug nodebug	nodebug	出力なし
9	オブジェクトファイル出力指定	objectfile	なし	—
10	テンプレートインスタンス生成機能	template	なし	テンプレート機能は使用していません
11	リストファイル	listfile nolistfile	nolistfile	出力なし
12	リスト内容と形式	show	なし	—
13	ファイル間インライン展開	file_inline	なし	—
14	コメントのネスト	comment	なし	コメントネストは使用していません
15	MAC レジスタ保証	macsave	macsave = 1	MACH,MACL レジスタを保証
16	メッセージレベル	change_message	なし	—
17	C/C++言語の選択	lang	なし	ファイル拡張子に従います
18	コピーライト出力抑止	logo nologo	nologo	コピーライトの出力を抑止
19	文字列内の文字コード	euc sjis latin1	なし	文字コードは使用していません
20	オブジェクトコード内漢字変換	outcode	なし	文字コードは使用していません
21	TBR 相対関数呼び出し指定	tbr	なし	—
22	$G0,$G1 セクションの変数の配置指定	stuff_gbr	なし	—
23	C++インライン関数の展開抑止	cpp_noinline	なし	—
24	ポインタ指示先の型を考慮した最適化	alias	alias = noansi	ANSI 規約に基づくポインタ指示先の型を考慮した最適化を行わない

5.2.3 オプション指定時の注意事項

オプション指定時は、次の規則に従ってください。

(1) cpu、division、endian、fpu、round、denormalize、pic、double=float、rtti、pack オプションはコンパイル時と同じオプションを指定してください。

(2) #pragma global_register を使用する場合は、preinclude オプションで#pragma global_register 宣言を含むヘッダファイルをインクルード指定してください。統合開発環境で指定する場合は、その他 [ユーザ指定オプション :]で指定してください。

6. CallWalker 操作方法

CallWalkerは、最適化リンケージエディタが出力したスタック情報ファイル(*.sni)またはシミュレータ・デバッガが出力したプロファイル情報ファイル(*.pro)を読み込んでスタック使用量を表示します。

また、スタック情報ファイルに出力できないアセンブラでアセンブルしたアセンブリプログラムのスタック使用量は、編集機能を用いて情報を追加・修正することや、シンボルに対してスタック値を設定可能であり、システム全体のスタック使用量を求めることもできます。

編集したスタック使用量に関する情報は、呼び出し情報ファイル(*.cal)として保存・読込み可能です。

6.1 CallWalker の起動

CallWalker を起動するには、Windows®のスタートメニューより"ファイル名を指定して実行"を選択し、Call.exe を指定し実行してください。

また、ルネサス統合開発環境をご使用の場合は、Windows®のスタートメニューで"プログラム"を選び、ルネサス統合開発環境に登録されている"Call Walker"を選択してください。 ルネサス統合開発環境を起動後は、Tools メニューより起動することもできます。

操作方法については、CallWalker のヘルプを参照ください。

7.　環境変数

7.1　環境変数一覧

環境変数の一覧を表 7.1に示します。

表 7.1　環境変数

	環境変数	説明
1	path	実行ファイルの格納フォルダを指定します。 指定フォーマット：　C> path= <実行ファイルパス名>[;<既存パス名>;...]
2	SHC_LIB	コンパイラのロードモジュールを格納したフォルダを指定します。コマンドプロンプトでコンパイルする場合、この環境変数の指定は必須です。 指定フォーマット：　C> set SHC_LIB= <実行ファイルパス名>
3	SHCPU	コンパイラ・アセンブラの cpu オプションによるマイコン種別の指定を、環境変数によって指定します。 <CPU> SH1 SH2 SH2E SH2A SH2AFPU SH2DSP SH3 SH3DSP SH4 SH4A SH4ALDSP SHCPU 環境変数によるマイコンの指定と、cpu オプションによるマイコンの指定が異なる場合は、ウォーニングメッセージを出力し cpu オプションの指定を優先します。 コンパイラでは SHDSP を指定すると SH2DSP を指定したと解釈します。 指定フォーマット：　C> set SHCPU= <CPU>
4	SHC_INC*	コンパイラのインクルードファイル格納フォルダを指定します。 システムインクルードファイルの検索順序は、include オプション指定フォルダ、SHC_INC 指定フォルダ、システムフォルダ(SHC_LIB)となります。 ユーザインクルードファイルの検索順序は、カレントフォルダ、include オプション指定フォルダ、SHC_INC 指定フォルダ、システムフォルダ(SHC_LIB)となります。 省略時、値は設定されません。 指定フォーマット：　C> set SHC_INC = <インクルードパス名> [;<インクルードパス名> ;...]
5	SHC_TMP	コンパイラ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。コマンドプロンプトでコンパイルする場合は、この環境変数の指定は必須です。 指定フォーマット：C> set SHC_TMP= <テンポラリファイルパス名>

	環境変数	説明
6	HLNK_LIBRARY1 HLNK_LIBRARY2 HLNK_LIBRARY3	最適化リンケージエディタが使用するデフォルトライブラリ名を指定します。 library オプションで指定したライブラリを優先してリンクします。その後未解決のシンボルがある場合、1,2,3 の順にデフォルトライブラリを検索します。 指定フォーマット： C> set HLNK_LIBRARY1= <ライブラリ名 1> C> set HLNK_LIBRARY2= <ライブラリ名 2> C> set HLNK_LIBRARY3= <ライブラリ名 3>
7	HLNK_TMP	最適化リンケージエディタ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テンポラリファイルを作成します。 指定フォーマット：C> set HLNK_TMP= <テンポラリファイルパス名>
8	HLNK_DIR*	最適化リンケージエディタ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 指定フォルダになります。 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検索します。 指定フォーマット：C> set HLNK_DIR= <入力ファイルパス名>[;<入力ファイルパス名> ;...]

【注】* 複数フォルダを指定する場合は、" ; "(セミコロン)または" , "(カンマ)で区切ってください。

7.2 プリデファインドマクロ

コンパイラについては、オプション指定やバージョンに合わせて、以下のようなプリデファインマクロが定義されます。

表 7.2 プリデファインドマクロ

	オプション	プリデファインドマクロ
1	cpu=sh1	#define _SH1
	cpu=sh2	#define _SH2
	cpu=sh2e	#define _SH2E
	cpu=sh2a	#define _SH2A
	cpu=sh2afpu	#define _SH2AFPU
	cpu=sh2dsp	#define _SH2DSP
	cpu=sh3	#define _SH3
	cpu=sh3dsp	#define _SH3DSP
	cpu=sh4	#define _SH4
	cpu=sh4a	#define _SH4A
	cpu=sh4aldsp	#define _SH4ALDSP
2	pic=1	#define _PIC
3	endian=big	#define _BIG
4	endian=little	#define _LIT
5	double=float	#define _FLT, #define __FLT__
6	fpu=single	#define _FPS
7	fpu=double	#define _FPD
8	denormalize=on	#define _DON
9	round=nearest	#define _RON
10	dspc	#define _DSPC
11	fixed_const	#define _FXD
12	－	#define __HITACHI_VERSION__ *1
13	－	#define __HITACHI__ *2
14	－	#define _SH *2
15	－	#define __RENESAS_VERSION__ *1
16	－	#define __RENESAS__ *2

【注】*1 __HITACHI_VERSION__、__RENESAS_VERSION__の値は、次のように参照します。

・ソースプログラム内：__HITACHI_VERSION__==aabb

aa: version 部分

bb: revision 部分

・コンパイラ内での定義例

```
#define __HITACHI_VERSION__    0x0701    // V.7.1.00 の場合
#define __HITACHI_VERSION__    0x0900    // V.9.00.00 の場合
#define __RENESAS_VERSION__    0x0900    // V.9.00.00 の場合
```

*2 常に定義されます

8.　ファイル仕様

8.1　ファイル名の付け方

ファイル名指定時に拡張子を省略した場合、標準のファイル拡張子を付加したファイル名を使用します。統合開発環境で使用する標準のファイル拡張子を表 8.1に示します。

表 8.1　統合開発環境で使用する標準のファイル拡張子

No.	拡張子	意味
1	c	C ソースプログラムファイル
2	cpp,cc,cp	C++ソースプログラムファイル
3	h	インクルードファイル
4	lst	C プログラム用リストファイル
5	lpp	C++プログラム用リストファイル
6	p	C プログラム用プリプロセッサ展開ファイル
7	pp	C++プログラム用プリプロセッサ展開ファイル
8	src	アセンブリソースプログラムファイル
9	exp	アセンブリプログラム用プリプロセッサ展開ファイル
10	lis	アセンブリプログラム用リストファイル
11	obj	リロケータブルオブジェクトプログラムファイル
12	rel	リロケータブルロードモジュールファイル
13	abs	アブソリュートロードモジュールファイル
14	map	リンケージリストファイル
15	lib	ライブラリファイル
16	lbp	ライブラリリストファイル
17	mot	モトローラ S フォーマット
18	hex	インテル(拡張)HEX フォーマット
19	bin	バイナリファイル
20	fsy	最適化リンケージエディタ出力シンボルアドレスファイル
21	sni	スタック情報ファイル
22	pro	プロファイル情報ファイル
23	dbg	DWARF2 フォーマットデバッグ情報ファイル
24	rti	拡張子 td のファイルで指定された定義を含むオブジェクトファイル
25	cal	呼び出し情報ファイル
26	bls	外部シンボル割り付け情報ファイル

rti_ではじまるファイル名はシステム予約名ですので使用しないでください。
各プロジェクトで生成されるtpldirのフォルダの下に出力されるファイル拡張子を表 8.2に示します。

表 8.2　tpldir フォルダ出力ファイル

No.	拡張子	意味
1	td	tentative 定義の変数情報
2	ti	テンプレート情報ファイル
3	pi	パラメータ情報ファイル
4	ii	インスタンス情報ファイル

ファイル名の付け方の一般的な規則は、各ホストマシンに準じています。ご使用になるホストマシンのマニュアルを参照してください。

8.2 コンパイルリストの参照方法

本節では、コンパイルリストの内容と形式について説明します。

8.2.1 コンパイルリストの構成

コンパイルリストの構成と内容を表 8.3に示します。

表 8.3 コンパイルリストの構成と内容

No.	リストの作成	内容	サブオプション*1	オプション省略時
1	ソースリスト情報	ソースプログラムのリスト*2	show=source show=nosource	出力しない
		インクルードファイル展開後のソースプログラムのリスト*3	show=include show=noinclude	出力しない
		マクロ展開後のソースプログラムのリスト*3	show=expansion show=noexpansion	出力しない
2	オブジェクト情報	オブジェクトプログラムの機械語、アセンブリコード	show=object show=noobject	出力する
3	統計情報	エラーの総数、ソースプログラムの行数、セクションサイズ、シンボル数	show=statistics show=nostatistics	出力する
4	コマンド指定情報	コマンドで指定されたファイル名とオプションを表示	—	出力する

【注】 *1 すべてのオプションは listfile オプションを指定した場合に有効です。
*2 ソースプログラムのリストは show=object を同時に指定した場合、オブジェクト情報内に出力されます。
*3 インクルードファイル、マクロ展開後のソースプログラムのリストは show=source を指定した場合に有効です。

8.2.2 ソースリスト情報

ソースリスト情報の出力形式には、プリプロセッサを通す前のソースプログラムを出力する形式(show=noinclude,noexpansion を指定する場合)と、プリプロセッサを通した後のソースプログラムを出力する形式(show=include,expansion を指定する場合)があります。それぞれの出力形式を図 8.1、図 8.2 に示します。また、図 8.2 に相違点を網掛けで示します。

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

FILE NAME: m0260.c

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

図 8.1　show=noinclude,noexpansion のソースリスト情報

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

FILE NAME: m0260.c

  Seq        File     Line      0----+----1----+----2----+----3----+----4----+----5---
    1      m0260.c      1       #include "header.h"
    2      header.h     1       #define SML_INT          1
    3      header.h     2       #define LRG_INT        100   (4)
    4      m0260.c      2
    5      m0260.c      3       int sum2(void)
    6      m0260.c      4       {     int j;
    7      m0260.c      5
    8      m0260.c      6       #ifdef SMALL
    9      m0260.c      7(5) X        j=SML_INT;
   10      m0260.c      8       #else
   11      m0260.c      9(6) E        j=100;
   12      m0260.c     10       #endif
   13      m0260.c     11
   14      m0260.c     12             return j; /* continue12345678901234567890123456 7
   (1)       (2)       (3)     +2345678901234567890 */
   15      m0260.c     13           }
                                (7)
```

図 8.2　show=include,expansion のソースリスト情報

(1) リスト上の行番号
(2) ソースプログラムファイル名またはインクルードファイル名
(3) ソースプログラムまたはインクルードファイル内の行番号
(4) show=include 指定時、インクルードファイルの展開のあったソース行
(5) show=expansion 指定時、#ifdef 文、#elif 文等の条件コンパイル文でコンパイル対象とならないソース行
(6) show=expansion 指定時、#define 文によるマクロ置換のあったソース行
(7) ソースプログラムの 1 行がコンパイルリストの 1 行に入りきらず、複数行にまたがって表示されたソース行

8.2.3　オブジェクト情報

オブジェクト情報の出力例を図 8.3 に示します。

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

FILE NAME: m0251.c

  SCT    OFFSET    CODE        C LABEL    INSTRUCTION OPERAND        COMMENT
  (1)     (2)      (3)                                (4)                (5)
          m0251.c      1     extern int multipli(int);
          m0251.c      2
          m0251.c      3     int multipli(int x)
  P
      00000000                  _multipli:                          ; function: multipli
                                                                    ; frame size=16
                                                                                  (6)
      00000000    4F22                STS.L     PR,R15
      00000002    7FF4                ADD       #-12,R15
      00000004    1F42                MOV.L     R4,@(8,R15)
          m0251.c      4     {
          m0251.c      5             int i;
          m0251.c      6             int j;
          m0251.c      7
          m0251.c      8             j=1;
      00000006    E201                MOV       #1,R2
      00000008    2F22                MOV.L     R2,@R15
          m0251.c      9             for(i=1; i<=x; i++){
      0000000A    E301                MOV       #1,R3
      0000000C    1F31                MOV.L     R3,@(4,R15)
      0000000E    A009                BRA       L213
      00000010    0009                NOP
      00000012                  L214:
          m0251.c     10                      j*=i;
      00000012    50F1                MOV.L     @(4,R15),R0
      00000014    61F2                MOV       @R15,R1
      00000016    D30A                MOV.L     L216+2,R3           ; _ _muli
      00000018    430B                JSR       @R3
              .                               .
              .                               .
```

図 8.3　show=source,object のオブジェクト情報

(1)　各セクションのセクション名(P、C、D、B、C$INIT、C$VTBL)
(2)　各セクションの先頭からのオフセット
(3)　各セクションのオフセットアドレスの内容
(4)　機械語に対応するアセンブリコード
(5)　プログラムに対応するコメント
(6)　スタックフレームサイズ(バイト数)

8.2.4 統計情報

統計情報の出力例を図 8.4 に示します。

```
******** STATISTICS INFORMATION ********


********** ERROR INFORMATION ***********     (1)

NUMBER OF ERRORS:                0
NUMBER OF WARNINGS:              0
NUMBER OF INFORMATIONS:          0

******* SOURCE LINE INFORMATION ********  (2)

COMPILED SOURCE LINE:            13


******* SECTION SIZE INFORMATION *******   (3)

PROGRAM       SECTION(P):                 0x000044 Byte(s)
CONSTANT      SECTION(C):                 0x000000 Byte(s)
DATA          SECTION(D):                 0x000000 Byte(s)
BSS           SECTION(B):                 0x000000 Byte(s)

TOTAL PROGRAM    SECTION:  00000044 Byte(s)
TOTAL CONSTANT   SECTION:  00000000 Byte(s)
TOTAL DATA       SECTION:  00000000 Byte(s)
TOTAL BSS        SECTION:  00000000 Byte(s)

        TOTAL PROGRAM SIZE:   0x000044 Byte(s)


********** LABEL INFORMATION ***********      (4)

NUMBER OF EXTERNAL REFERENCE SYMBOLS:  1
NUMBER OF EXTERNAL DEFINITION SYMBOLS:  1
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:    6
```

図 8.4 統計情報

(1) レベル別メッセージの総数
(2) ソースファイルのコンパイルした行数
(3) 各セクションのサイズとその合計
(4) オブジェクトプログラムの外部参照シンボルの数、外部定義シンボルの数、内部ラベルと外部ラベルの合計数

【注】 message オプションが指定されていない場合には、レベル別メッセージ(1)の NUMBER OF INFORMATIONS は出力されません。noobject オプションを指定した場合とエラーレベル、フェータルレベルのエラーが発生した場合には、セクションサイズ情報(3)とラベル情報(4)は出力されません。また、code=asmcode を指定した場合は、セクションサイズ情報(3)は当該セクションの有無を 0 と 1 で示すようになります。

8.2.5　コマンド指定情報

コンパイラを起動したときのコマンドで指定されたファイル名とオプションを表示します。コマンド指定情報の出力例を図 8.5 に示します。

```
*** COMMAND PARAMETER ***
-listfile test.c
```

図 8.5　コマンド指定情報

8.3 アセンブルリストの参照方法

8.3.1 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表 8.4に示します。

表 8.4 アセンブルリストの構成と内容

No	リストの作成	内容	オプション*	オプション省略時
1	ソースリスト情報	ソースプログラムに関する情報を示します。	source	出力する
2	クロスリファレンスリスト情報	ソースプログラムのシンボルに関する情報を示します。	cross_reference	出力する
3	セクション情報リスト	ソースプログラムのセクションに関する情報を示します。	section	出力する

【注】 * すべてのオプションは list オプションを指定した場合に有効です。

8.3.2　ソースリスト情報

ソースリスト情報を出力します。ソースリスト情報の出力例を図 8.6 に示します。

```
PROGRAM NAME =                                  "SAMPLE"              (7)
  1                          1                  .HEADING """SAMPLE"""
  2                          2      POINT       .ASSIGNA 16
  3                          3      Parm1       .REG  (R0)
  4                          4      Parm2       .REG  (R1)
  5                          5      WORK1       .REG  (R2)
  6                          6      WORK2       .REG  (R3)
  7                          7      WORK3       .REG  (R4)
  8                          8      WORK4       .REG  (R5)
     ～
 20 00000000                 9 I1 FIX_MUL :
 21 00000000 2107           10 I1     DIV0S  Parm1,Parm2
 22 00000002 0229           11 I1     MOVT  WORK1
 23 00000004 4011           12 I1     CMP/PZ Parm1
 24 00000006 8900           13 I1     BT   MUL01
 25 00000008 600B           14 I1     NEG   Parm1,Parm1
(1)  (2)       (3)         (4)(5)                          (6)
     ～
231                                 ***** BEGIN-POOL *****
232 00000180 00018000               DATA FOR SOURCE-LINE 17          (8)
233 00000184 00024000               DATA FOR SOURCE-LINE 18
234 00000188 00030000               DATA FOR SOURCE-LINE 19
235 0000018C 00050000               DATA FOR SOURCE-LINE 20
236                                 ***** END-POOL *****
237                         35                    .END
****TOTAL ERRORS        0
****TOTAL WARNINGS      0
          (9)
```

図 8.6　ソースプログラムリスト

(1)　行番号(10 進)です。
(2)　ロケーションカウンタ値(16 進)です。
(3)　オブジェクトコード(16 進)です。オペレーションが.RES、.SRES、.SRESC、.SRESZ、.FRES のいずれかであるときは、確保する領域のサイズをバイト単位で示します。
(4)　ソース行番号(10 進)です。

(5) 展開区分
ファイルインクルード、条件つきアセンブリ機能、マクロ機能が展開した区分を示します。
In‥‥ファイルインクルード(n はインクルードのネストレベルを示します)
C‥‥条件つきアセンブルの成立、繰り返し展開、条件付き繰り返し展開
M‥‥マクロ展開
(6) ソースステートメントです。
(7) .HEADING で設定したヘッダです。
(8) リテラルプールです。
(9) エラーとウォーニングの総数です。エラーメッセージは、エラーが存在するソースステートメントの次の行に出力されます。

8.3.3 クロスリファレンスリスト

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図 8.7 に示します。

```
*** CROSS REFERENCE LIST
NAME              SECTION  ATTR VALUE           SEQUENCE
FIX_DIV           SAMPLE        00000088     91*  223
FIX_MUL           SAMPLE        00000000     19*  218
MUL01             SAMPLE        0000000A     23    25*
MUL02             SAMPLE        00000010     26    28*
MUL03             SAMPLE        00000082     87    89*
Parm1                      REG                3*   20    22    24    24
                                             28    29    29    31    32
                                             32    35    36    36    38
                                             40    45    49    55    57
                                             59    61    63    65    67
                                             69    71    73    75    77
                                             79    81    83    85    88
                                             88    93    94    99   101
Parm2                      REG                4*   20    25    27    27
                                             28    31    33    33    35
                                             38    41    43    44    46
                                             48    54    56    58    60
                                             62    64    66    68    70
     (1)           (2)      (3)      (4)                  (5)
```

図 8.7 クロスリファレンスリスト

(1) シンボルの名称です。
(2) シンボルが含まれるセクションの名称です(セクション名の最初の 8 文字)。
(3) シンボルの属性です。
EXPT‥‥‥外部定義シンボル
IMPT‥‥‥外部参照シンボル
SCT‥‥‥セクション名
REG‥‥‥.REG アセンブラ制御命令で定義したシンボル
FREG‥‥‥.FREG アセンブラ制御命令で定義したシンボル

ASGN・・・・・・.ASSIGN アセンブラ制御命令で定義したシンボル
EQU・・・・・・.EQU アセンブラ制御命令で定義したシンボル
MDEF・・・・・・二重定義したシンボル
UDEF・・・・・・未定義シンボル
表示なし・・・・・その他のシンボル

(4) シンボルの値です(16 進)。
(5) シンボルを定義、参照しているリスト行番号(10 進)です。
行番号の後ろのアスタリスク(*)は、定義行であることを示します。

8.3.4 セクション情報リスト

セクション情報を出力します。セクション情報の出力例を図 8.8 に示します。

```
*** SECTION DATA LIST
SECTION          ATTRIBUTE   SIZE          START
SAMPLE           REL-CODE    000000190
 (1)              (2)         (3)           (4)
```

図 8.8 セクション情報リスト

(1) セクションの名称です。
(2) セクションの種類です。
REL・・・・・相対アドレスセクション
ABS・・・・・絶対アドレスセクション
CODE・・・・・コードセクション
DATA・・・・・データセクション
STACK・・・・・スタックセクション
DUMMY・・・・・ダミーセクション
(3) セクションのサイズです(16 進数、単位はバイト)。
(4) 絶対アドレスセクションの先頭アドレスです。

8.4 リンケージリストの参照方法

最適化リンケージエディタが出力するリンケージリストの内容と形式について説明します。

8.4.1 リンケージリストの構成

リンケージリストの構成と内容を表8.5に示します。

表 8.5 リンケージリストの構成と内容

No.	リストファイルへ表示する情報	内容	show オプション* 指定	show オプション省略時
1	オプション情報	コマンドライン、サブコマンドで指定したオプション列を表示	なし	出力する
2	エラー情報	エラーメッセージを表示	なし	出力する
3	リンケージマップ情報	セクション名、先頭／最終アドレス、サイズ、種別を表示	なし	出力する
4	シンボル情報	静的定義シンボル名、アドレス、サイズ、種別をアドレス順に表示	show=symbol	出力しない
		show=reference を指定した場合は、各シンボルの参照回数、最適化実行有無も表示	show=reference	出力しない
5	シンボル削除最適化情報	最適化で削除したシンボルを表示	show=symbol	出力しない
6	クロスリファレンス情報	シンボルの参照情報を表示	show=xreference	出力しない
7	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を表示	show=total_size	出力しない
8	ベクタ情報	ベクタ番号とアドレスの情報を表示	show=vector	出力しない
9	CRC 情報	CRC の演算結果および出力位置アドレスを表示	なし	CRC オプション指定時は常に出力

【注】 * show オプションは list オプションを指定した場合に有効です。

8.4.2　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。オプション情報の出力例を図 8.9 に示します(optlnk -sub=test.sub -list -show 指定時)。

```
(test.subの内容)
INPUT test.obj
```

```
*** Options ***

-sub=test.sub
INPUT test.obj (2)
-list
-show
```

(1)

図 8.9　オプション情報の出力例（リンケージリスト）

(1) コマンドライン、サブコマンドで指定したオプション列を、指定順に出力します。
(2) サブコマンドファイルtest.sub内のサブコマンドです。

8.4.3　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図 8.10 に示します。

```
*** Error Information ***

** L2310 (E) Undefined external symbol "strcmp" referred to in "test.obj"
```

図 8.10　エラー情報の出力例（リンケージリスト）

(1) エラーメッセージを出力します。

8.4.4 リンケージマップ情報

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。リンケージマップ情報の出力例を図 8.11 に示します。

```
*** Mapping List ***

SECTION                              START      END         SIZE   ALIGN
  (1)                                 (2)       (3)          (4)    (5)

P
                                   00001000  00001000         1    1
C
                                   00001004  00001007         4    4
D_2
                                   00001008  000014dd       4d6    2
B_2
                                   000014de  000050b3      3bd6    2
```

図 8.11 リンケージマップ情報の出力例(リンケージリスト)

(1) セクション名を表示します。
(2) 先頭アドレスを表示します。
(3) 最終アドレスを表示します。
(4) セクションサイズを表示します。
(5) セクションのアライメント数を表示します。

8.4.5　シンボル情報

show=symbol を指定した場合、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、種別をアドレス順に出力します。また、show=reference を指定した場合は、各シンボルの参照回数、最適化実行の有無も出力します。シンボル情報の出力例を図 8.12 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)

SECTION=P
FILE=test.obj
                00000000        00000428            428
  _main
                00000000               2        func ,g              0
  _malloc
                00000000              32        func ,l              0
FILE=mvn3
                00000428        00000490             68
  $MVN#3
                00000428               0        none ,g              0
```

図 8.12　シンボル情報の出力例(リンケージリスト)

(1)　セクション名を表示します。
(2)　ファイル名を表示します。
(3)　(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表示します。
(4)　(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表示します。
(5)　(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表示します。
(6)　シンボル名を表示します。
(7)　シンボルアドレスを表示します。
(8)　シンボルサイズを表示します。
(9)　シンボル種別を次のように表示します。

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
	l	・・・・・・	内部定義

(10)　シンボル参照回数を表示します。show=referenceを指定した場合のみ表示します。参照回数を表示しないときは、*を表示します。
(11)　最適化有無を次のように表示します。

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

8.4.6 シンボル削除最適化情報

シンボル削除最適化(optimize=symbol_delete)によって削除されたシンボルのサイズ、種別を出力します。シンボル削除最適化情報の出力例を図 8.13 に示します。

```
*** Delete Symbols ***

SYMBOL                          SIZE    INFO
  (1)                           (2)     (3)
  _Version
                                   4    data ,g
```

図8.13 シンボル削除情報の出力例(リンケージリスト)

(1) 削除シンボル名を表示します。
(2) 削除シンボルサイズを表示します。
(3) 削除シンボルの種別を以下のように表示します。

データ種別：	func	･････････	関数名
	data	･････････	変数名
宣言種別 ：	g	･････････	外部定義
	l	･････････	内部定義

8.4.7　クロスリファレンス情報

show=xreference 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図 8.14 に示します。

```
*** Cross Reference List ***

No    Unit Name  Global.Symbol   Location    External Information
(1)      (2)          (3)          (4)              (5)
0001  a
      SECTION=P  _func
                                 00000100
                 _func1
                                 00000116
                 _main
                                 0000012c
                 _g
                                 00000136
      SECTION=B
                 _a
                                 00000190    0001(00000140:P)
                                             0002(00000178:P)
                                             0003(0000018c:P)
0002  b
      SECTION=P
                 _func01
                                 00000154    0001(00000148:P)
                 _func02
                                 00000166    0001(00000150:P)
0003  c
      SECTION=P
                 _func03
                                 00000184
```

図8.14　クロスリファレンス情報の出力例(リンケージリスト)

(1) Unit番号。オブジェクト単位の識別番号。
(2) オブジェクト名。リンク時の入力指定順になる。
(3) シンボル名。セクションごとに配置アドレスの昇順に出力される。
(4) シンボルの配置アドレス。
　form=relocate指定時は、セクション先頭からの相対値となる。
(5) 参照している外部シンボルのアドレスを表す。
　出力形式は以下のようになる。
　<Unit番号>(<アドレス or セクション内オフセット>:<セクション名>)

8.4.8 合計セクションサイズ

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。合計の出力例を図 8.15 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)
```

図8.15 合計セクションサイズの出力例(リンケージリスト)

(1) RAMデータセクションの合計サイズ。
(2) ROMデータセクションの合計サイズ。
(3) プログラムセクションの合計サイズ。

8.4.9 ベクタ情報

show=vector を指定した場合、可変ベクタテーブルの内容を表示します。合計の出力例を図 8.16 に示します。

```
*** Variable Vector Table List ***

NO.      SYMBOL/ADDRESS
(1)      (2)
  0      $fdummy
  1      $fa
  2      00ff8800
  3      $fdummy
    :
  <省略>
```

図8.16 ベクタ情報の出力例(リンケージリスト)

(1) ベクタ番号。
(2) シンボルを表示します。シンボルが定義されていない場合はアドレスで表示します。

8.4.10 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。

```
*** CRC Code ***

CODE    : cb0b
(1)
ADDRESS : 00007ffe
(2)
```

図8.17　CRC 情報の出力例(リンケージリスト)

(1) CRC演算結果
(2) CRCの演算結果の出力位置アドレス

8.5 ライブラリリストの参照方法

本節では、最適化リンケージエディタが出力するライブラリリストの内容と形式について説明します。

8.5.1 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表8.6に示します。

表8.6 ライブラリリストの構成と内容

No.	リストの作成	内容	サブオプション*	show オプション省略時
1	オプション情報	コマンドライン、サブコマンドで指定したオプション列を表示	—	出力する
2	エラー情報	エラーメッセージを表示	—	出力する
3	ライブラリ情報	ライブラリ情報を表示	—	出力する
4	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を表示	—	出力する
		show=symbol を指定した場合は、モジュール内シンボル名一覧も表示	show=symbol	出力しない
		show=section を指定した場合は、各モジュール内セクション名、シンボル名一覧も表示	show=section	出力しない

【注】 * すべてのオプションは、list オプションを指定した場合に有効です。

8.5.2 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。オプション情報の出力例を図 8.18 に示します(optlnk -sub=test.sub -list –show を指定した場合)。

```
 (test.subの内容)
form    library
in      adhry.obj
output  test.lib
```

```
*** Options ***

-sub=test.sub
form    library
in      adhry.obj
output  test.lib
-list
-show
```

(2)

(1)

図8.18 オプション情報の出力例(ライブラリリスト)

(1) コマンドライン、サブコマンドで指定したオプション列を、指定順に出力します。
(2) サブコマンドファイルtest.sub内のサブコマンドです。

8.5.3 エラー情報

エラー、ウォ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図 8.19 に示します。

```
*** Error Information ***

** L1200 (W) Backed up file "main.lib" into "main.lbk"    (1)
```

図8.19　エラー情報の出力例(ライブラリリスト)

(1) ウォーニングメッセージを出力します。

8.5.4 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図 8.20 に示します。

```
*** Library Information ***

LIBRARY NAME=test.lib   (1)
CPU=SuperH              (2)
ENDIAN=Big              (3)
ATTRIBUTE=system        (4)
NUMBER OF MODULE=1      (5)
```

図8.20　ライブラリ情報の出力例(ライブラリリスト)

(1) ライブラリ名を表示します。
(2) マイコン名を表示します。
(3) エンディアン種別を表示します。
(4) ライブラリファイルの属性がシステムライブラリかユーザライブラリかを表示します。
(5) ライブラリ内モジュール数を表示します。

8.5.5 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一覧を出力します。

show=symbol を指定した場合はモジュール内シンボル名一覧を、show=section を指定した場合はモジュール内セクション名、シンボル名一覧を出力します。

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図 8.21 に示します。

```
*** Library List ***

MODULE          LAST UPDATE
 (1)                (2)
  SECTION
    (3)
    SYMBOL
      (4)
adhry
                29-Feb-2000 12:34:56
  P
   _main
   _Proc0
   _Proc1
  C
  D
   _Version
  B
   _IntGlob
   _CharGlob
```

図8.21 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(ライブラリリスト)

(1) モジュール名を表示します。
(2) モジュールを登録した日付を表示します。モジュールが更新された場合は、最新の更新日付を表示します。
(3) モジュール内セクション名を表示します。
(4) セクション内をシンボル表示します。

9. プログラミング

9.1 プログラムの構造

9.1.1 セクション

コンパイラ、アセンブラが出力するオブジェクトプログラムの実行命令、データの各領域は、セクションを構成します。セクションは、メモリ上の配置を行う最小単位です。セクションの性質には、以下の項目があります。

- セクション属性
  code　実行命令を格納します。
  data　データを格納します。
  stack　スタック領域です。
- 形式種別
  相対アドレス形式・・・・・・・・最適化リンケージエディタで再配置可能なセクションです。
  絶対アドレス形式・・・・・・・・アドレス決定済みのセクションです。最適化リンケージエディタで再配置できません。
- 初期値
  プログラム実行開始時の初期値の有無です。同一セクション内で初期値があるデータと初期値がないデータは混在できません。一つでも初期値があると、初期値のない領域は0で初期化します。
- 書き込み操作
  プログラム実行時における書き込み操作の可/不可を示します。
- アライメント数
  セクションを割り付けるアドレスの補正値です。最適化リンケージエディタでは、アライメント数の倍数アドレスになるよう、アドレスを補正します。

9.1.2 C/C++プログラムのセクション

C/C++プログラム、標準ライブラリの使用メモリ領域の種類とセクションとの対応を**表 9.1**に示します。

表 9.1 メモリ領域の種類とその性質の概要

	名称	セクション		形式種別	初期値	アライメント数	内容
		名称	属性		書き込み操作		
1	プログラム領域	P*1	code	相対形式	有 不可	4byte*2	機械語を格納
2	定数領域	C*1*5	data	相対形式	有 不可	4byte	const 型のデータを格納
3	初期化データ領域	D*1*5	data	相対形式	有 可	4byte	初期値のあるデータを格納

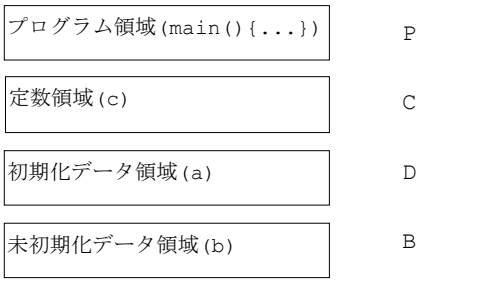
	名称	セクション 名称	セクション 属性	形式 種別	初期値 書き込み操作	アライメント数	内容
4	未初期化データ領域	B*1*5	data	相対形式	無 可	4byte	初期値のないデータを格納
5	X メモリ定数領域	$XC	data	相対形式	有 不可	4byte	const 型のデータを X メモリに格納
6	Y メモリ定数領域	$YC	data	相対形式	有 不可	4byte	const 型のデータを Y メモリに格納
7	X メモリ初期化データ領域	$XD	data	相対形式	有 可	4byte	初期値のあるデータを X メモリに格納
8	Y メモリ初期化データ領域	$YD	data	相対形式	有 可	4byte	初期値のあるデータを Y メモリに格納
9	X メモリ未初期化データ領域	$XB	data	相対形式	無 可	4byte	初期値のないデータを X メモリに格納
10	Y メモリ未初期化データ領域	$YB	data	相対形式	無 可	4byte	初期値のないデータを Y メモリに格納
11	GBR セクション	$G0*6	data	相対形式	有 可	4byte	#pragma gbr_base で指定された初期値のあるデータを格納 初期値のない場合は 0 を格納
12	GBR セクション	$G1*6	data	相対形式	有 可	4byte	#pragma gbr_base1 で指定された初期値のあるデータを格納 初期値のない場合は 0 を格納
13	C++初期処理／後処理データ領域	C$INIT	data	相対形式	有 不可	4byte	グローバルクラスオブジェクトに対して呼び出されるコンストラクタおよびデストラクタのアドレスを格納
14	C++仮想関数表領域	C$VTBL	data	相対形式	有 不可	4byte	クラス宣言中に仮想関数があるときに仮想関数をコールするためのデータを格納
15	スタック領域	S	stack	相対形式	無 可	4byte	プログラム実行に必要な領域 「9.2.1(2) 動的領域の割り付け」参照
16	ヒープ領域	—	—	相対形式	無 可	—	ライブラリ関数 malloc、realloc、calloc、new で使用する領域 「9.2.1(2) 動的領域の割り付け」参照
17	TBR テーブル領域	$TBR	data	相対形式	有 不可	4byte	TBR 相対関数呼び出しを行うためのデータを格納
18	絶対アドレス変数領域	$ADDRESS$ <section> <address>*3	data	絶対形式	有/無 可/不可 *4	—	#pragma address 指定した変数を格納

【注】 *1 section オプションまたは拡張子#pragma section でセクション名を切り替えることができます。
*2 align16 オプションを指定している場合は 16byte に、align32 オプションを指定している場合は 32byte になります。
*3 <section>は C,D,B のセクション名称、<address>は絶対アドレス値になります。
*4 初期値、書込み操作は<section>の属性に従います。
*5 stuff オプションにより、アライメント数が 4byte、2byte、1byte のセクションに分割されます。それぞれのセクションの説明については、「2.2.2 オブジェクトオプション」の stuff オプションの説明を参照ください。
*6 stuff_gbr オプションにより、アライメント数が 4byte、2byte、1byte のセクションに分割されます。それぞれのセクションの説明については、「2.2.2 オブジェクトオプション」の stuff_gbr オプションの説明を参照ください。

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

```
int a=1;
char b;
const int c=0;
void main(){
 ...
}
```

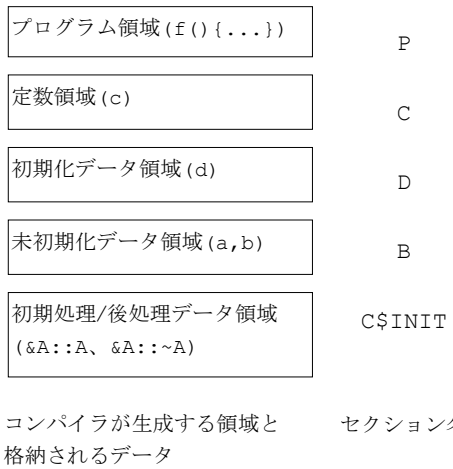
Cプログラム



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

```
class A{
  int m;
  A(int p);
  ~A();
};
A a(1);
int b;
extern const char c='a';
int d=1;
void f(){...}
```

C++プログラム



9.1.3 アセンブリプログラムのセクション

アセンブリプログラムでは、.SECTION 制御命令を用いてセクションの開始や属性、形式種別を宣言します。
.SECTION 制御命令の宣言形式は次のとおりです。詳細は「**11.4 アセンブラ制御命令**」を参照してください。

```
.SECTION <セクション名>[,<セクション属性>[,<形式種別>]]
    <形式種別>: 相対アドレス形式セクションの場合、ALIGN=<アライメント数>
                絶対アドレス形式セクションの場合、LOCATE=<アドレス値>
```

例：アセンブリプログラムのセクション宣言例を示します。

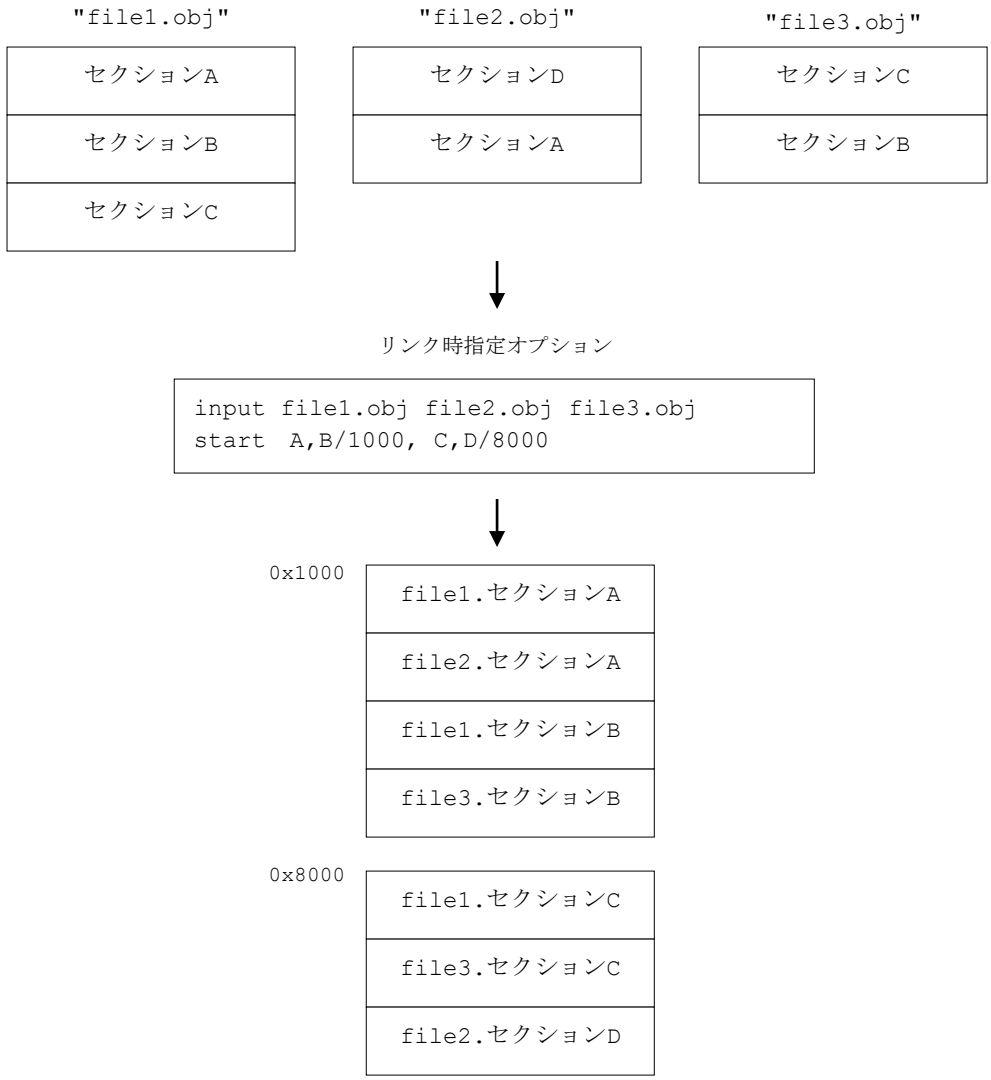
```
          .CPU       SH2
          .OUTPUT    DBG
SIZE:     .EQU       8

          .SECTION   A,CODE,ALIGN=4              ;(1)
START:
          MOV.L      LITERAL,R0
          MOV.L      LITERAL+4,R1
          MOV.L      #SIZE,R2
LOOP:
          CMP/PL     R2
          BF         EXIT
          MOV.B      @R0+,R3
          MOV.B      R3,@R1
          ADD        #-1,R2
          ADD        #1,R1
          BRA        LOOP
          NOP
EXIT:
          SLEEP
          NOP
LITERAL:
          .DATA.L    CONST
          .DATA.L    DATA
;
          .SECTION   B,DATA,LOCATE=H'00002000    ;(2)
CONST:
          .DATA.B    H'01,H'02,H'03,H'04,H'05,H'06,H'07,H'08
;
          .SECTION   C,STACK,ALIGN=4             ;(3)
DATA:
          .RES.B     8
          .END
```

(1) セクション名A、アライメント数4、相対アドレス形式のcodeセクションを宣言しています。
(2) セクション名B、割り付けアドレスH'2000、絶対アドレス形式のdataセクションを宣言しています。
(3) セクション名C、アライメント数4、相対アドレス形式のstackセクションを宣言しています。

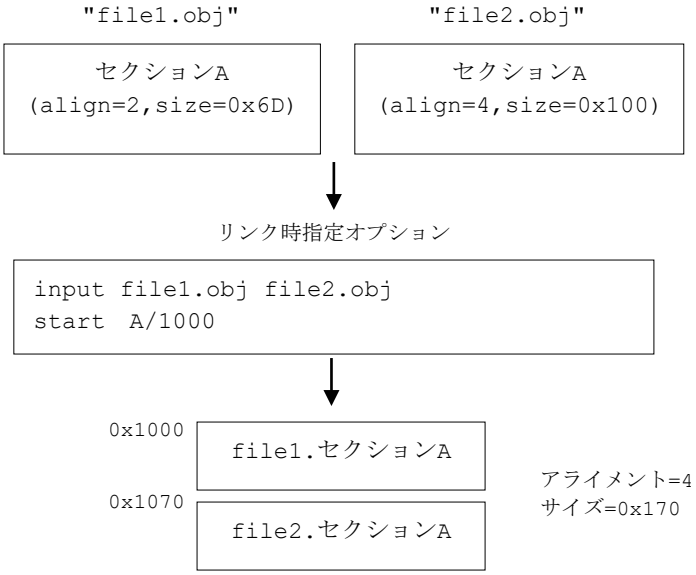
9.1.4 セクションの結合

最適化リンケージエディタでは、入力オブジェクトプログラム内の同一セクションを結合し、start オプションによって指定されたアドレスに割り付けます。

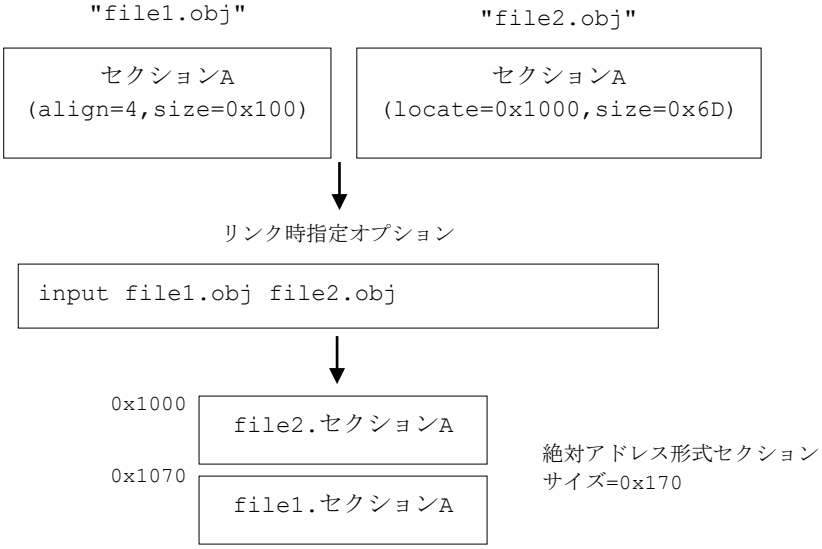
(1) 異なるファイルの同名セクションは、ファイルの入力順に連続して割り付けます。



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

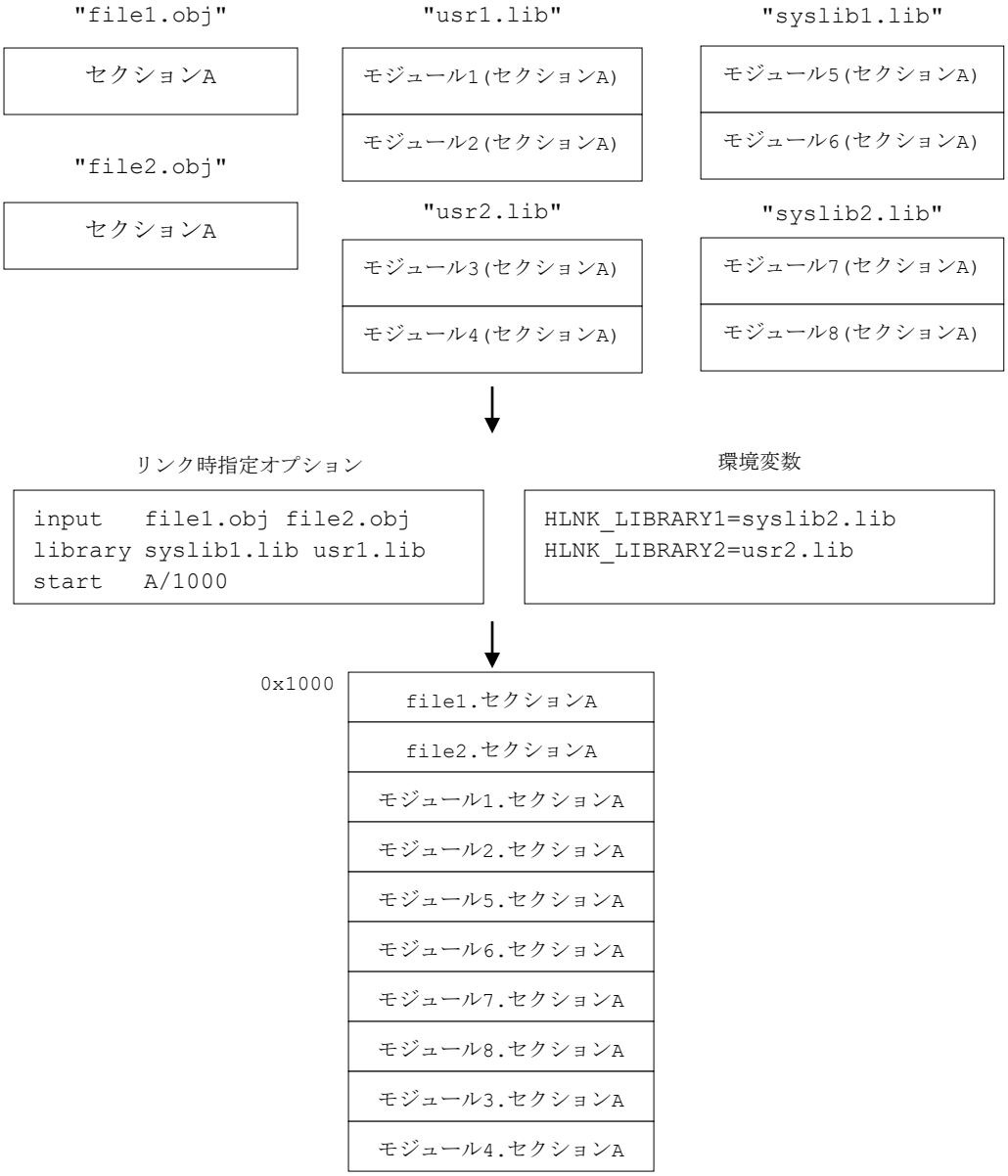


(3) 同名セクションに絶対アドレス形式と相対アドレス形式が含まれている場合、絶対アドレス形式オブジェクトの後に相対アドレス形式オブジェクトを結合します。リロケータブルファイル出力を指定した場合(form=relocate 指定)でも、当該セクションは絶対アドレス形式セクションになります。



(4) 同名セクション内オブジェクトの結合順序に関する規則は以下のとおりです。

- 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



9.2 初期設定プログラムの作成

本章では、プログラムを SuperH RISC engine マイコンを応用したシステムに組み込む方法を説明します。

プログラムをシステムに組み込むには、以下の準備が必要です。

- メモリの割り付け
  各セクション、スタック領域、ヒープ領域を、システム上の ROM、RAM のメモリ領域に割り当てる必要があります。
- プログラム実行環境の設定
  プログラムの実行環境を設定する処理には、レジスタの初期設定、メモリ領域の初期化、プログラムの起動があります。

また、入出力等の C/C++ライブラリ関数をご使用になる場合は、実行環境の設定時にライブラリの初期化を行う必要があります。特に入出力(stdio.h、ios、streambuf、istream、ostream)とメモリ割り付け(stdlib.h、new)の機能を使用する場合は、システムごとに、低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。
プログラムの終了処理を行う C ライブラリ関数(exit、atexit、abort 関数)を使用する場合も、別途ユーザシステムに合わせてこれらの関数を作成する必要があります。

9.2.1 ではプログラムのメモリ領域のアドレスを決定する考え方を説明し、実際にアドレスを決定するための最適化リンケージエディタのオプション指定方法について実例を挙げて説明します。

9.2.2 では実行環境設定の項目を説明し、設定プログラムの実例について説明します。
また、ライブラリ関数の初期設定処理、低水準ルーチンの作成方法および終了処理関数の作成例についても説明します。

9.2.1 メモリ領域の割り付け

本コンパイラの出力したオブジェクトプログラムをシステムに組み込むためには、プログラムの使用するメモリ領域のサイズを決定し、それぞれの領域を適切なメモリアドレスに割り付ける必要があります。

C/C++プログラムが使用するメモリ領域には、C/C++プログラム中の関数に対応する機械語や外部データ定義や静的データメンバで宣言したデータ領域のように静的に割り付ける領域とスタック領域のように動的に割り付ける領域があります。

以下、各領域の割り付け方を説明します。

(1) 静的領域の割り付け

(a) 静的領域の内容

スタック領域、ヒープ領域以外のセクションは静的領域に割り付けます。

C/C++プログラムの各セクション(プログラム領域、定数領域、初期化データ領域、未初期化データ領域、C++初期処理/後処理データ領域、C++仮想関数表領域)は静的領域に割り付けます。

(b) サイズの算出法

静的領域のサイズは、コンパイラ、アセンブラが生成するオブジェクトプログラムサイズとC/C++プログラムが使用するライブラリ関数のサイズの合計になります。

オブジェクトプログラムをリンクしたあと、リンケージリストのリンケージマップ情報にライブラリを含めた各セクションのサイズを出力しますので、静的領域のサイズを知ることができます。**図 9.1**にリンケージリスト内リンケージマップ情報の例を示します。

```
*** Mapping List ***

SECTION                          START      END        SIZE   ALIGN
  (1)                              (2)       (3)        (4)    (5)
P
                                 00000000  000004d6     4d6    2
C
                                 000004d6  00000533      5d    2
D
                                 00000534  0000053c       8    2
B
                                 0000053c  00004112    3bd6    2
```

図 9.1 リンケージリスト内リンケージマップ情報例

コンパイル、アセンブル単位のセクションサイズは、コンパイルリスト内統計情報およびアセンブルリスト内セクション情報に出力されます。**図 9.2**にコンパイルリスト内統計情報の例、**図 9.3**にアセンブルリスト内セクション情報の例を示します。

```
******* SECTION SIZE INFORMATION *******
PROGRAM   SECTION(P)        :0x00004A Byte(s)
CONSTANT  SECTION(C)        :0x000018 Byte(s)
DATA  SECTION(D)            :0x000004 Byte(s)
BSS   SECTION(B)            :0x000004 Byte(s)

TOTAL PROGRAM       SECTION:    0000004A Byte(s)
TOTAL CONSTANT      SECTION:    00000018 Byte(s)
TOTAL DATA          SECTION:    00000004 Byte(s)
TOTAL BSS           SECTION:    00000004 Byte(s)

TOTAL PROGRAM SIZE: 0x00006A Byte(s)
```

図 9.2 コンパイルリスト内統計情報例

```
*** SECTION DATA LIST

SECTION                      ATTRIBUTE   SIZE          START

P                           REL-CODE   000000604
D                           REL-DATA   000000008
C                           REL-DATA   00000005D
B                           REL-DATA   000003BD6
```

図 9.3 アセンブルリスト内セクション情報例

標準ライブラリを使用しない場合は、ファイル単位のセクションサイズの合計が静的領域のサイズになります。

標準ライブラリを使用している場合、各セクションのメモリ領域サイズにはライブラリ関数の使用するメモリ領域サイズが加算されます。コンパイラが提供する標準ライブラリの中には、C 言語仕様で規定したライブラリ関数や組み込み関数向け C++クラスライブラリ以外に、C/C++プログラムを実行する上で必要な算術演算を行うルーチン(ランタイムライブラリ)を含みます。そのため、C/C++プログラム上でライブラリ関数を使用しなくても、標準ライブラリが必要な場合がありますので注意してください。

C/C++プログラムで使用するランタイムライブラリは、本コンパイラ出力のアセンブリプログラム(code=asmcode 指定)に外部参照シンボルとして出力しますので、そのシンボル名を参照することによって C/C++プログラムで使用するランタイムライブラリ名を知ることができます。以下に具体例を示します。

- C/C++プログラム例

```
f(int a, int b)
{
    a /= b;
    return a;
}
```

- C コンパイル時に生成されるアセンブリプログラム例

```
        .IMPORT    _ _divls        ;(ランタイムライブラリの外部参照宣言)
        .EXPORT    _f
        .SECTION   P,CODE,ALIGN=4
_f:                                ;function: f
                                   ;frame size=4
        STS.L      PR,@-R15
        MOV        R5, R0
        MOV.L      L218,R3         ; _ _divls
        JSR        @R3
        MOV        R4, R1
        LDS.L      @R15+, PR
        RTS
        NOP
L218:
        .DATA.L    _ _divls
        .END
```

(c) ROM、RAM の割り付け

プログラムを ROM 化する場合、セクションの初期値の有無、書き込み操作の可/不可で、ROM に割り付けるか RAM に割り付けるかが決まります。

C/C++プログラムの各セクションを ROM 化する場合は、以下のように ROM と RAM に分けて割り付けます。

- プログラム領域　(セクション P)　→ ROM
- 定数領域　(セクション C、$G0、$G1*3)　→ ROM
- 未初期化データ領域　(セクション B、$G0、$G1*3)　→ RAM
- 初期化データ領域　(セクション D、$G0、$G1*3)　→ ROM、RAM((d)参照)
- 初期処理／後処理データ領域*1　(セクション C$INIT)　→ ROM
- 仮想関数表領域*2　(セクション C$VTBL)　→ ROM

【注】*1 C++プログラムでグローバルクラスオブジェクトがあるときにコンパイラが生成します。
*2 C++プログラムで仮想関数宣言があるときにコンパイラが生成します。
*3 $G0、$G1 はこれらの領域のいずれか 1 つのみに配置するようにしてください。

(d) 初期化データ領域の割り付け

初期化データ領域のように、初期値を持ち、プログラム実行時に値の変更が可能なセクションは、リンク時には ROM 上に置き、プログラムの実行開始時に RAM 上にコピーします。したがって、最適化リンケージエディタの rom オプションを用いて、ROM 上と RAM 上に、二重に領域をとる必要があります。指定例については、「(e)メモリの割り付け例とリンク時のアドレス指定方法」を参照してください。また ROM 上から RAM 上へ値をコピーするセクションの初期設定については、「9.2.2 (2)初期設定（PowerOn_Reset）」で説明します。

(e) メモリの割り付け例とリンク時のアドレス指定方法

アブソリュートロードモジュール作成時に、最適化リンケージエディタのオプションまたはサブコマンドで各セクション毎に割り付ける領域のアドレスを指定します。以下、静的領域のメモリ割り付け例とリンク時の指定方法について説明します。

図 9.4に、静的な領域の割り付け例を示します。

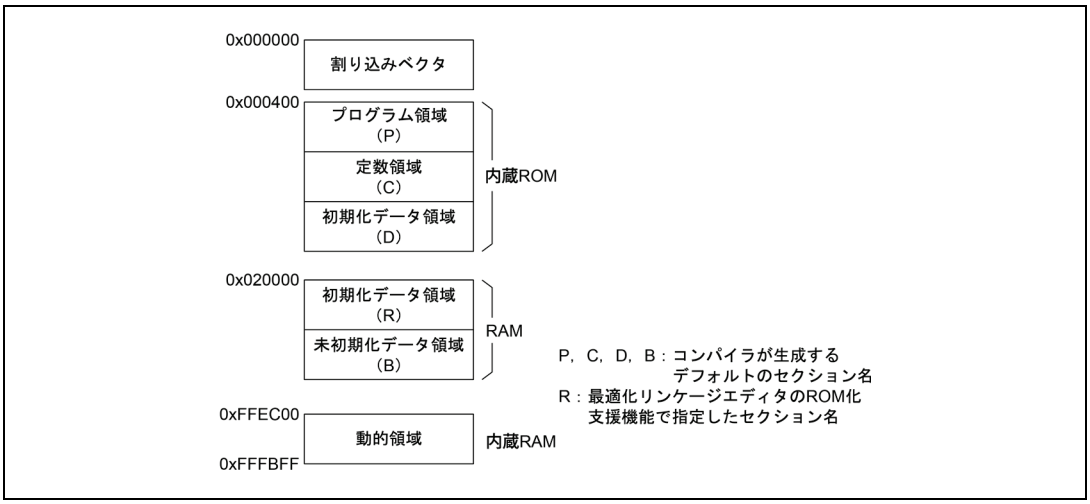


図 9.4　静的な領域の割り付け例

図 9.4に示すメモリ割り付けを行う場合、リンク時に以下のサブコマンドを指定します。

```
ROM△D=R                          ……[1]
START△P,C,D/400,R,B/20000        ……[2]
```

説明：[1] セクションDと同じ大きさのセクションRを出力ロードモジュールに確保します。また、セクションDに割り付けられたシンボルを参照している場合、セクションR上のアドレスとなるようリロケーションします。セクションDはROM上、セクションRはRAM上の初期化データセクション名になります。

[2] セクションP、C、Dを内蔵ROMのアドレス0x400から連続した領域に割り付けます。また、セクションR、BをRAMのアドレス0x20000から連続したアドレスに割り付けます。

(2) 動的領域の割り付け

(a) 動的領域の内容

C/C++プログラムで使用する動的領域には、以下の二つがあります。

- スタック領域
- ヒープ領域(メモリ割り付けライブラリ関数で使用)

(b) スタック領域サイズの算出法

C/C++プログラム、標準ライブラリの使用するスタック領域は、最適化リンケージエディタの stack オプションを指定してスタック情報ファイルを出力すると、CallWalker を用いて最大使用量を算出することができます。CallWalker の使用方法については、「6. CallWalker 操作方法」を参照してください。

V6 までのアセンブラでアセンブルしたプログラムの使用するスタック領域は、CallWalker では算出できません。以下の C/C++プログラムのスタック使用量算出法を参考にアセンブリプログラムのスタック使用量を算出し、CallWalker で算出したスタック使用量に加算してください。

- C/C++プログラムのスタック使用量計算の考え方

C/C++プログラムの使用するスタック領域は、関数の呼び出しのたびにスタック上に割り付け、関数のリターン時に解放します。スタック領域のサイズを算出するためには、まず各関数のスタック使用量を算出し、関数の呼び出し関係から実際のスタック使用量を算出します。

♦ 各関数の使用するスタック領域

各関数の使用するスタック領域は、コンパイラ出力のオブジェクトリスト中の frame size から分かります。

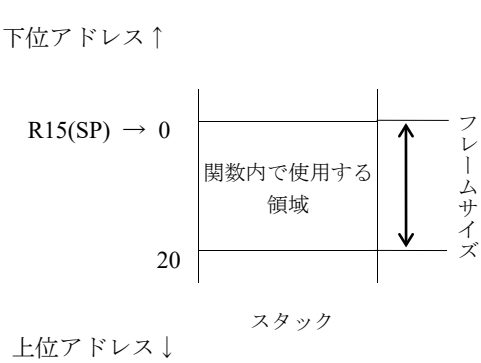
以下にオブジェクトリストとスタック上の割り付けの具体例を示し、そのスタック使用量の算出法について説明します。

例

次の C プログラムに対するオブジェクトリストとスタック使用量の算出法を示します。C++プログラムでも同様です。

```
extern int h(char, int *, double );
int h(char a, register int *b, double c)
{
    char *d;

    d= &a;
    h(*d,b,c);
    {
        register int i;

        i= *d;
        return i;
    }
}
```

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

  FILE NAME: m0251.c

  SCT   OFFSET    CODE     C LABEL  INSTRUCTION OPERAND  COMMENT
  P
        00000000           _h:                           ;function: h
                                                         ;frame size=20
        00000000  2FE6              MOV.L   R14,@-R15
        00000002  4F22              STS.L   PR,@-R15
                                                :
```



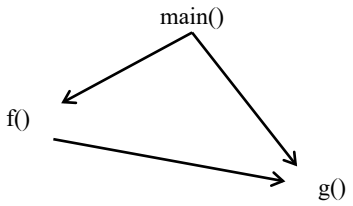
関数の使用するスタック領域サイズは、フレームサイズの値と同じです。したがって、上記の例で関数 h の使用するスタック領域サイズは、オブジェクトリスト中の項目 COMMENT の frame size の値 20 バイトとなります。

スタック上の引数領域に割り付けられる引数については、**「9.3.2(4) 引数とリターン値の設定、参照に関する規則」**を参照してください。

♦ スタック使用量の算出法

関数呼び出しの関係から使用するスタック領域のサイズを算出します。

例　関数呼び出しの関係と、各関数のスタック使用量の例を図 9.5に示します。



関数名	スタック使用量(バイト)
main	24
f	32
g	24

図 9.5　関数呼び出しの関係とスタック使用量の例

この場合、関数「f」を介して関数「g」が呼ばれた時のスタック領域のサイズは、表 9.2によって計算します。

表 9.2　スタックサイズの計算例

呼び出し経路	スタックサイズ計(バイト)
main(24)→f(32)→g(24)	80
main(24)→g(24)	48

このように、呼び出しレベルの一番深い関数についてスタック領域のサイズを計算し、その最大値(この場合 80 バイト)のスタック領域を最低限割り付ける必要があります。

【注】 C/C++プログラムの中で再帰呼び出しを行っている場合は、再帰的に呼び出す回数の最大値を算出してから、その関数のスタック領域のサイズに再帰的に呼び出す回数をかけて計算してください。

(c) ヒープ領域

ヒープ領域で使用するメモリ領域のサイズは、C/C++プログラム内でメモリ管理ライブラリ関数(calloc,malloc,realloc,new 関数)によって割り付ける領域の合計です。ただし、メモリ管理ライブラリ関数は、1 回の呼び出しのたびに管理用の領域として 4 バイト使用します。実際に確保する領域サイズにこの管理領域のサイズを加えて計算してください。

また、コンパイラはヒープ領域をユーザ指定のメモリサイズ(_sbrk_size)の単位で管理しています。_sbrk_size の指定方法は「9.2.2(4) C/C++ライブラリ関数の初期設定(_INITLIB)」を参照してください。ヒープ領域として確保する領域サイズ(HEAPSIZE)は次のように計算してください。

HEAPSIZE = _sbrk_size × n　(n≧1)
(メモリ管理ライブラリによって割り付ける領域サイズ) + 管理領域サイズ≦HEAPSIZE

入出力ライブラリ関数は、内部処理の中でメモリ管理ライブラリ関数を使用しています。入出力の中で割り付ける領域のサイズは、
516 バイト × (同時にオープンするファイルの数の最大値)
になります。

【注】 メモリ管理ライブラリ関数の free、または delete 関数で解放した領域は、再びメモリ管理ライブラリ関数で領域を確保するときに再利用しますが、割り付けを繰り返すことによって空き領域のサイズの合計は十分でも空き領域が小さな領域に分割しているために、新たに要求した大きなサイズの領域を確保できないという状況が生じることがあります。このような状況を避けるために、以下の注意に従ってヒープ領域を使用してください。
(ア) サイズの大きな領域は、なるべくプログラムの実行開始直後に確保してください。
(イ) 解放して再利用するデータ領域のサイズをなるべく一定にしてください。

- 動的領域の割り付け方

動的領域は RAM 上に割り付けます。

スタック領域は、ベクタテーブルにスタック領域の最上位アドレスを SP(スタックポインタ)として設定することにより割り付ける場所が決まります。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP では割り込み時の動作が SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP の場合と異なるので、割り込みハンドラが必要になります。

ヒープ領域は、低水準インタフェースルーチン(sbrk)の初期設定で割り付ける場所が決まります。

それぞれ、「9.2.2(1) ベクタテーブルの設定(VEC_TBL)」、「9.2.2(6) 低水準インタフェースルーチン」を参照してください。

9.2.2 実行環境の設定

本節では、プログラムの実行に必要な環境を設定するための処理について説明します。ただし、プログラムを実行する環境はユーザシステムごとに異なりますので、ユーザシステムの仕様に合わせて実行環境の設定プログラムを作成する必要があります。

図 9.6にプログラムの構成例を示します。

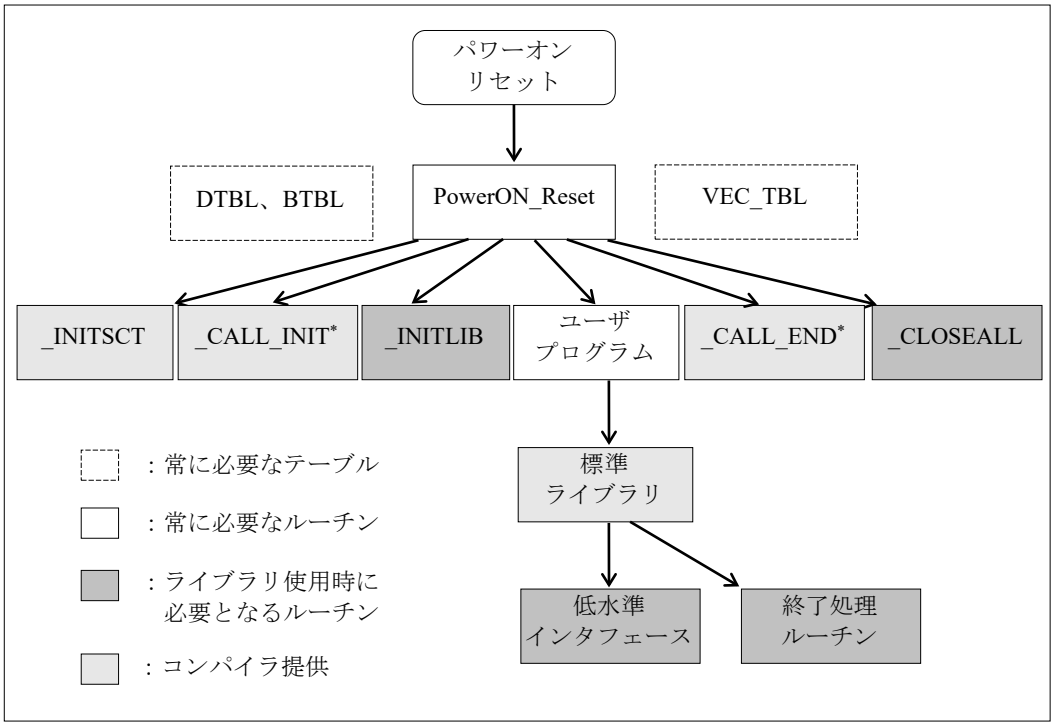


図 9.6 プログラムの構成例

【注】＊ グローバルクラスオブジェクトがあるときに必要になります。

各構成ルーチンの内容は以下のとおりです。

- ベクタテーブルの設定(VEC_TBL)
  パワーオンリセットでレジスタの初期設定プログラム(PowerON_Reset)が起動され、またスタックポインタ(SP)に値が設定されるように、ベクタテーブルを設定します。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSPでは割り込み時の動作がSH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSPの場合と異なるので、割り込みハンドラが必要になります。

- 初期設定(PowerON_Reset)
  レジスタの初期設定を行ったあと、初期設定ルーチンを順次呼び出します。

- セクション初期化用テーブル(DTBL、BTBL)
  セクションの初期化ルーチンで使用するセクションの先頭アドレスおよび最終アドレスを、セクションアドレス演算子を用いて設定します。

- セクションの初期化(_INITSCT)*1
  初期値が設定されていない静的変数領域(未初期化データ領域)をゼロで初期化します。また、初期化データ領域の初期値をROM上からRAM上にコピーします。

- グローバルクラスオブジェクト初期処理(_CALL_INIT)*1*2
  グローバルに宣言されたクラスオブジェクトに対してコンストラクタを呼び出します。

- グローバルクラスオブジェクト後処理(_CALL_END)*1*2
  main 関数の実行後、グローバルクラスオブジェクトに対してデストラクタを呼び出します。

- C/C++ライブラリの初期設定(_INITLIB)
  ライブラリ関数の中で、初期設定の必要なものについて、初期設定を行います。特に、標準入出力を行う準備をします。

- ファイルのクローズ(_CLOSEALL)
  オープンしているファイルをすべてクローズします。

- 低水準インタフェースルーチン
  標準入出力(stdio.h、ios、streambuf、istream、ostream)、メモリ管理ライブラリ(stdlib.h、new)を使用する場合に必要なライブラリ関数とユーザシステムとの間のインタフェースルーチンです。

- 終了処理ルーチン(exit、atexit、abort)*3
  プログラムの終了処理を行います。

【注】*1: 標準ライブラリとして提供しています。
*2: C++プログラム中にグローバルクラスオブジェクトの宣言がある時に必要な処理です。
*3: プログラムの終了処理を行う C ライブラリ関数 exit、atexit、abort 関数を使用する場合は、ユーザシステムにあわせてこれらの関数を作成する必要があります。C++プログラムを使用する場合、または C ライブラリ関数 assert マクロを使用する場合、abort 関数は必ず作成する必要があります。

以下、この構成に従って各処理の実現方法について解説します。

(1) ベクタテーブルの設定(VEC_TBL)

パワーオンリセットで、レジスタの初期設定を行う関数「PowerON_Reset」が呼び出されるようにするためには、ベクタテーブルの 0 番地に関数「PowerON_Reset」の先頭アドレスを設定します。また、スタックポインタ(SP)を設定するためには 4 番地にスタック領域の最上位アドレスを設定します。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP では割り込み時の動作が SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP の場合と異なるので割り込みハンドラが必要になります。

また、ユーザシステムで割り込み処理を使用する場合は、割り込みベクタの設定も本ルーチンで行います。以下にそのコーディング例を示します。

例 1　SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP 用ベクタテーブルの例

```
#pragma interrupt (IRQ0)

extern void Manual_Reset_PC(void);
extern void Manual_Reset_SP(void);

extern void IRQ0(void);

#pragma section VECTTBL  /* #pragma section宣言によりRESET_Vectorsを     */
                         /* CVECTTBLセクションに出力します。              */
                         /* リンク時にstartオプションでCVECTTBLセクションを */
                         /* 0x0番地に割り付けるよう指定します。           */
void (*const RESET_Vectors[])(void)={
   (void*) PowerON_Reset_PC,
   _ _secend("S"),
   (void*) Manual_Reset_PC,
   _ _secend("S")
};

#pragma section VECT2    /* #pragma section宣言によりvec_table2を        */
                         /* CVECT2セクションに出力します。                */
                         /* リンク時にstartアドレスでCVECT2セクションを   */
                         /* 指定番地に割り付けるよう指定します。          */

void (*const vec_table2[])(void)={IRQ0};
```

例 2　プログラムでバンク 0 を使用している場合の割り込みハンドラの例(SH7708)

```
;;;
```

```
.GLOBAL    INT_TLBInitial_Page
;H'0A0 TLB protect (load)
.GLOBAL    INT_TLBProtect_Load
;H'0C0 TLB protect (store)
.GLOBAL    INT_TLBProtect_Store
;H'0E0 Address error (load)
.GLOBAL    INT_Address_load
;H'100 Address error (store)
.GLOBAL    INT_Address_store
;H'120 Reserved
.GLOBAL    INT_Reserved1
;H'140 Reserved
.GLOBAL    INT_Reserved2
;H'160 TRAPA
.GLOBAL    INT_TRAPA
;H'180 Illegal code
.GLOBAL    INT_Illegal_code
;H'1A0 Illegal slot
.GLOBAL    INT_Illegal_slot
;H'1C0 NMI
.GLOBAL    INT_NMI
;H'1E0 User breakpoint trap
.GLOBAL    INT_User_Break
;H'200 External hardware interrupt
.GLOBAL    INT_Extern_0000
;H'220 External hardware interrupt
.GLOBAL    INT_Extern_0001
;H'240 External hardware interrupt
.GLOBAL     _INT_Extern_0010
;H'260 External hardware interrupt
.GLOBAL     _INT_Extern_0011
;H'280 External hardware interrupt
.GLOBAL     _INT_Extern_0100
;H'2A0 External hardware interrupt
.GLOBAL     _INT_Extern_0101
;H'2C0 External hardware interrupt
.GLOBAL     _INT_Extern_0110
;H'2E0 External hardware interrupt
.GLOBAL     _INT_Extern_0111
;H'300 External hardware interrupt
.GLOBAL     _INT_Extern_1000
;H'320 External hardware interrupt
.GLOBAL     _INT_Extern_1001
;H'340 External hardware interrupt
.GLOBAL     _INT_Extern_1010
;H'360 External hardware interrupt
.GLOBAL     _INT_Extern_1011
;H'380 External hardware interrupt
.GLOBAL     _INT_Extern_1100
;H'3A0 External hardware interrupt
.GLOBAL     _INT_Extern_1101
;H'3C0 External hardware interrupt
.GLOBAL     _INT_Extern_1110
;H'3E0 External hardware interrupt
.GLOBAL     _INT_Extern_1111
;H'400 TMU0 TUNI0
.GLOBAL     _INT_Timer_Under_0
```

```
        ;H'420 TMU1 TUNI1
        .GLOBAL     _INT_Timer_Under_1
        ;H'440 TMU2 TUNI2
        .GLOBAL     _INT_Timer_Under_2
        ;H'460 TMU2 TICPI2
        .GLOBAL     _INT_Input_Capture
        ;H'480 RTC ATI
        .GLOBAL     _INT_RTC_ATI
        ;H'4A0 RTC PRI
        .GLOBAL     _INT_RTC_PRI
        ;H'4C0 RTC CUI
        .GLOBAL     _INT_RTC_CUI
        ;H'4E0 SCI ERI
        .GLOBAL     _INT_SCI_ERI
        ;H'500 SCI RXI
        .GLOBAL     _INT_SCI_RXI
        ;H'520 SCI TXI
        .GLOBAL     _INT_SCI_TXI
        ;H'540 SCI TEI
        .GLOBAL     _INT_SCI_TEI
        ;H'560 WDT ITI
        .GLOBAL     _INT_WDT
        ;H'580 REF RCMI
        .GLOBAL     _INT_REF_RCMI
        ;H'5A0 REF ROVI
        .GLOBAL     _INT_REF_ROVI

;;;
```

```
        STC.L     R2_BANK,@-R15
        STC.L     R1_BANK,@-R15
        STC.L     R0_BANK,@-R15
        .ENDM
;
        .MACRO POP_EXP_BASE_REG
        LDC.L     @R15+,R0_BANK   ; RECOVER REGISTERS
        LDC.L     @R15+,R1_BANK
        LDC.L     @R15+,R2_BANK
        LDC.L     @R15+,R3_BANK
        LDC.L     @R15+,R4_BANK
        LDC.L     @R15+,R5_BANK
        LDC.L     @R15+,R6_BANK
        LDC.L     @R15+,R7_BANK
        LDS.L     @R15+,PR
        LDC.L     @R15+,SPC
        LDC.L     @R15+,SSR
        .ENDM

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                reset                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .SECTION   RSTHandler,CODE
_ResetHandler:
        MOV.L     #EXPEVT,R0
        MOV.L     @R0,R0
        SHLR2     R0
        SHLR      R0
        MOV.L     #_RESET_Vectors,r1
        ADD       R1,R0
        MOV.L     @R0,R0
        JMP       @R0
        NOP

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                        exceptional interrupt                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .SECTION  INTHandler,CODE
        .EXPORT   INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
        PUSH_EXP_BASE_REG
;
        MOV.L     #EXPEVT,R0      ; set event address
        MOV.L     @R0,R1          ; set exception code
        MOV.L     #_INT_Vectors,R0; set vector table address
        ADD       #-(H'40),R1     ; exception code - H'40
        SHLR2     R1
        SHLR      R1
        MOV.L     @(R0,R1),R3     ; set interrupt function addr
;
        MOV.L     #_INT_MASK,R0   ; interrupt mask table addr
        SHLR2     R1
        MOV.B     @(R0,R1),R1     ; interrupt mask
        EXTU.B    R1,R1
;
        STC       SR,R0           ; save SR
```

```
        MOV.L     #(RBBLclr&IMASKclr),R2
                                  ; RB,BL,mask clear data
        AND       R2,R0           ; clear mask data
        OR        R1,R0           ; set interrupt mask
        LDC       R0,SSR          ; set current status
;
        LDC.L     R3,SPC
        MOV.L     #_ _int_term,R0 ; set interrupt terminate
        LDS       R0,PR
;
        RTE
        NOP
;
        .POOL
;

;;;
```

```
        LDC       R0,SSR          ; set current status
;
        LDC.L     R3,SPC
        MOV.L     #_ _int_term,R0 ; set interrupt terminate
        LDS       R0,PR
;
        RTE
        NOP
;
        .POOL
;

;;;
```

(2) 初期設定(PowerON_Reset)

ライブラリ関数を使用する場合には、本関数でライブラリの初期設定を行う「_INITLIB」とファイルのクローズ処理を行う「_CLOSEALL」を呼び出します。以下に「PowerON_Reset」のコーディング例を示します。SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP では割り込み時の動作が SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP の場合と異なるので割り込みハンドラが必要になります。

例

```
#include <_h_c_lib.h>
#include <machine.h>

#pragma entry PowerON_Reset_PC
#pragma stacksize 0x100

#define SR_Init      0x000000F0 /* cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp    */
                                /* 指定時の初期値は 0x400000F0          */
#define FPSCR_Init   0x00040001*1 /* cpu=sh2afpu|sh4|sh4a 指定時のみ     */

#define INT_OFFSET   0x10

extern unsigned int INT_Vectors;
extern void PowerON_Reset_PC();
extern void main();
#ifdef _ _cplusplus
extern "C" {
#endif
extern void _INIT_IOLIB();
extern void _INIT_OTHERLIB();
extern void _CLOSEALL();
#ifdef _ _cplusplus
}
#endif

void PowerON_Reset_PC() {
   set_vbr((void *)(INT_Vectors - INT_OFFSET));
   set_fpscr(FPSCR_Init);     /* cpu=sh2afpu|sh4|sh4a 指定時のみ設定してください */
   _INITSCT();
   _INIT_IOLIB();
   _INIT_OTHERLIB();
#ifdef _ _cplusplus
   _CALL_INIT();
#endif
   set_cr(SR_Init);
   main();
#ifdef _ _cplusplus
   _CALL_END();
#endif
   _CLOSEALL();
   sleep();
}
```

【注】 *1 FPSCR の初期値は、以下のオプションに合わせて変更ください。FPSCR レジスタの詳細については、各マイコンのハードウェアマニュアルを参照ください。

- FPSCR の PR ビットは、-fpu=double を指定した場合は 1 に、それ以外の場合は 0に設定してください。
- FPSCR の RM ビットは、-round=nearest を指定した場合は 00 に、それ以外の場合は 01 に設定してください。
- FPSCR の DN ビットは、-cpu=sh4 または sh4a、かつ-denormalize=on を指定した場合は 0 に、それ以外の場合は 1 に設定してください。

(3) セクション初期化用テーブル(DTBL、BTBL)

セクションの初期化ルーチン(_INITSCT)では、未初期化データセクションをゼロで初期化し、初期化データセクションの ROM 上にある初期化データを RAM 上にコピーします。ここでは、_INITSCT 関数が使用するセクションの先頭アドレスおよび最終アドレスを、セクションアドレス演算子を用いて、セクションの初期化用テーブルに設定します。

セクション初期化用テーブルのセクション名は、未初期化データ領域を C$BSEC、初期化データ領域を C$DSEC で宣言します。

以下にコーディング例を示します。

例

```
#pragma section $DSEC          //セクション名を C$DSEC にします
static const struct {
   void *rom_s;                //初期化データセクションの ROM 上の先頭アドレスメンバ
   void *rom_e;                //初期化データセクションの ROM 上の最終アドレスメンバ
   void *ram_s;                //初期化データセクションの RAM 上の先頭アドレスメンバ
} DTBL[] = {_ _sectop("D"), _ _secend("D"), _ _sectop("R")};

#pragma section $BSEC          //セクション名を C$BSEC にします
static const struct {
   void *b_s;                  //未初期化データセクションの先頭アドレスメンバ
   void *b_e;                  //未初期化データセクションの最終アドレスメンバ
} BTBL[] = {_ _sectop("B"), _ _secend("B")};
```

(4) C/C++ライブラリ関数の初期設定(_INITLIB)

ここでは、C/C++ライブラリ関数の初期設定方法を説明します。

実際に使用する機能に合わせた必要最低限の初期設定を行うために、以下の指針を参考にしてください。

- <stdio.h>、<ios>、<streambuf>、<istream>、<ostream>の各関数と assert マクロを使用する場合、標準入出力の初期設定(_INIT_IOLIB)が必要です。
- 作成した低水準インタフェースルーチンの中で初期設定が必要な場合、低水準インタフェースルーチンの仕様に合わせた初期設定(_INIT_LOWLEVEL)が必要です。
- rand 関数、strtok 関数を使用する場合、標準入出力以外の初期設定(_INIT_OTHERLIB)が必要です。

ライブラリの初期設定を行うプログラム例を以下に示します。また、図 9.7に FILE 型データを示します。

```
#include <stdio.h>
#include <stdlib.h>
#define IOSTREAM 3
const size_t _sbrk_size = 520;        // ヒープ領域確保サイズの最小単位を指定します
                                      // (省略時：1024)
const int _nfiles = IOSTREAM;         // 入出力ファイル数を指定します(省略時：20)
struct _iobuf _iob[IOSTREAM];
```

```
unsigned char sml_buf[IOSTREAM];
extern char *_s1ptr;

#ifdef __cplusplus
extern "C" {
#endif
void _INITLIB (void)
{
    _INIT_LOWLEVEL();                   // 低水準インタフェースルーチンの初期設定をします
    _INIT_IOLIB();                      // 入出力ライブラリの初期設定をします
    _INIT_OTHERLIB();                   // rand 関数、strtok 関数の初期設定をします
}

void _INIT_LOWLEVEL (void)
{                                       // 低水準ライブラリで必要な初期設定をしてください
}

void _INIT_IOLIB(void)
{
    FILE *fp;
    for( fp = _iob; fp < _iob + _nfiles; fp++ )    // FILE 型データの初期設定です
    {
        fp->_bufptr = NULL;
        fp->_bufcnt = 0;
        fp->_buflen = 0;
        fp->_bufbase = NULL;
        fp->_ioflag1 = 0;
        fp->_ioflag2 = 0;
        fp->_iofd = 0;
    }
    if(freopen("stdin*1", "r", stdin)== NULL)    // 標準入力ファイルをオープンします
        stdin->_ioflag1 = 0xff;         // オープン失敗時ファイルアクセスを禁止します
    stdin->_ioflag1 |= _IOUNBUF;        // データのバッファリングなしに設定します*2
    if(freopen("stdout*1", "w", stdout)== NULL) // 標準出力ファイルをオープンします
        stdout->_ioflag1 = 0xff;        // オープン失敗時ファイルアクセスを禁止します
    stdout->_ioflag1 |= _IOUNBUF;       // データのバッファリングなしに設定します*2
    if(freopen("stderr*1", "w", stderr)== NULL) // 標準エラーファイルをオープンします
        stderr->_ioflag1 = 0xff;        // オープン失敗時ファイルアクセスを禁止します
    stderr->_ioflag1 |= _IOUNBUF;       // データのバッファリングなしに設定します*2
}

void _INIT_OTHERLIB(void)
{
    srand(1);                           // rand 関数を使用する場合の初期設定です
    _s1ptr=NULL;                        // strtok 関数を使用する場合の初期設定です
}
#ifdef __cplusplus
}
#endif
```

【注】*1　標準入出力ファイルのファイル名を指定します。この名前は、低水準インタフェースルーチン「open」で使用します。
*2　コンソール等対話的な装置の場合、バッファリングを行わないためのフラグを立てます。

```
                /* ファイル型データのC言語での宣言 */

struct _iobuf{
              unsigned char  *_bufptr;   /* バッファへのポインタ       */
              long           _bufcnt;    /* バッファカウント           */
              unsigned char  *_bufbase;  /* バッファへのベースポインタ */
              long           _buflen;    /* バッファ長                 */
              char           _ioflag1;   /* I/Oフラグ                  */
              char           _ioflag2;   /* I/Oフラグ                  */
              char           _iofd;      /* I/Oフラグ                  */
}iob[_nfiles];
```

図 9.7　FILE 型データ

(5)　ファイルのクローズ(_CLOSEALL)

通常ファイルへの出力は、メモリ領域上のバッファにためておき、バッファが一杯になったときに実際の外部記憶装置への書き出しを行います。したがってファイルのクローズを行わないと、ファイルへの出力内容が外部記憶装置へ書き出されないことがあります。

機器組み込み用のプログラムの場合、通常プログラムが終了することはありません。しかし、プログラムの誤りなどにより main 関数が終了する場合、オープンしているファイルは、すべてクローズしなければなりません。

本処理は、main 関数終了時にオープンしているファイルのクローズを行います。

ファイルのクローズを行うプログラム例を以下に示します。

```
#include <stdio.h>

#ifdef _ _cplusplus
extern "C"
#endif
void _CLOSEALL(void)
{
int i;

    for( i=0; i < _nfiles; i++ )

        // ファイルがオープンしているかどうかチェックします
        if( _iob[i]._ioflag1 & (_IOREAD | _IOWRITE | _IORW ) )
            fclose( &_iob[i] );    // ファイルをクローズします
}
```

(6) 低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリを C/C++プログラムで使用する場合は、低水準インタフェースルーチンを作成しなければなりません。表 9.3に C ライブラリ関数で使用している低水準インタフェースルーチンの一覧を示します。

表 9.3 低水準インタフェースルーチンの一覧

	名称	機能
1	open	ファイルのオープン
2	close	ファイルのクローズ
3	read	ファイルからの読み込み
4	write	ファイルへの書き出し
5	lseek	ファイルの読み込み／書き出しの位置の設定
6	sbrk	メモリ領域の確保
7	sbrk_ _X	X メモリ領域の確保
8	sbrk_ _Y	Y メモリ領域の確保
9	errno_addr*	errno アドレスの取得
10	wait_sem*	セマフォの確保
11	signal_sem*	セマフォの解放

【注】* リエントラントライブラリを使用する場合に必要です。

低水準インタフェースルーチンで必要な初期化は、プログラム起動時に行う必要があります。これは、「9.2.2(4) C/C++ライブラリ関数の初期設定(_INITLIB)」の中の「_INIT_LOWLEVEL」という関数の中で行ってください。

以下、低水準入出力の基本的な考え方を説明したあと、各インタフェースルーチンの仕様を説明します。

【注】関数名 open、close、read、write、lseek、sbrk、sbrk_ _X、sbrk_ _Y、errno_addr、wait_sem、signal_sem は低水準インタフェースルーチンの予約済み識別子です。ユーザプログラム中では使用しないでください。

(a) 入出力の考え方

標準入出力ライブラリでは、ファイルを FILE 型のデータによって管理しますが、低水準インタフェースルーチンでは、実際のファイルと 1 対 1 に対応する正の整数を与え、これによって管理します。この整数をファイル番号といいます。

open ルーチンでは、与えられたファイル名に対してファイル番号を与えます。open ルーチンでは、この番号によってファイルの入出力ができるように、以下の情報を設定する必要があります。

- ファイルのデバイスの種類(コンソール、プリンタ、ディスクファイル等)。
  (コンソールやプリンタ等の特殊なデバイスに対しては、特別なファイル名をシステムで決めておいて open ルーチンで判定する必要があります)
- ファイルのバッファリングをする場合はバッファの位置、サイズ等の情報。
- ディスクファイルならば、ファイルの先頭から次に読み込み/書き出しを行う位置までのバイトオフセット。

open ルーチンで設定した情報に基づいて、以後、入出力(read、write ルーチン)、読み込み/書き出し位置の設定(lseek ルーチン)を行います。

close ルーチンでは、出力ファイルのバッファリングを行っている場合はバッファの内容を実際のファイルに書き出し、open ルーチンで設定したデータの領域が再使用できるようにしてください。

(b)　低水準インタフェースルーチンの仕様

本項では低水準インタフェースルーチンを作成するための仕様を説明します。以下、各ルーチンごとに、ルーチンを呼び出す際のインタフェースとその動作および実現上の注意事項を示します。

各ルーチンのインタフェースは以下の形式で示します。なお、低水準インタフェースルーチンは必ず関数原型してください。また C++プログラム内で宣言する場合は「extern "C"」を付加してください。

凡例：

簡易説明

(ルーチン名)

説　明	(ルーチンの機能概要を示します)	
リターン値	正常：	(正常に終了した場合のリターン値の意味を示します)
	異常：	(エラーが生じた場合のリターン値を示します)
引　数	(名前)	(意味)
	(インタフェースに示す引数名です)	(引数として渡される値の意味を示します)

ファイルオープン

*int open(char *name, int mode, int flg)*

説　明　第 1 引数のファイル名に対応するファイルを操作するための準備をします。
open ルーチンでは、後で読み込み/書き出しを行うために、ファイルの種類 (コンソール、プリンタ、ディスクファイル等) を決定しなければなりません。ファイルの種類は、以後 open ルーチンで返したファイル番号を用いて読み込み/書き出しを行うたびに参照する必要があります。
第 2 引数の mode は、ファイルをオープンする時の処理の指定です。このデータの各ビットの意味について以下に示します。

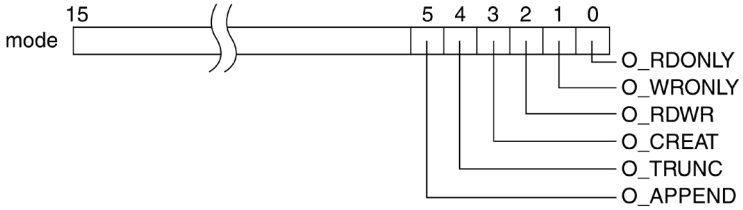


表 9.4　open ルーチン mode ビット説明

mode ビット	説明
O_RDONLY (0 ビット)	ビットが 1 のとき、ファイルを読み込み専用にオープン
O_WRONLY (1 ビット)	ビットが 1 のとき、ファイルを書き出し専用にオープン
O_RDWR (2 ビット)	ビットが 1 のとき、ファイルを読み込み、書き出し両用にオープン
O_CREAT (3 ビット)	ビットが 1 のとき、ファイル名で示すファイルが存在しない場合にファイルを新規に作成
O_TRUNC (4 ビット)	ビットが 1 のとき、ファイル名で示すファイルが存在する場合にファイルの内容を捨て、ファイルのサイズを 0 に更新
O_APPEND (5 ビット)	次に読み込み/書き出しを行うファイル内の位置を設定 ビットが 0 のとき：ファイルの先頭に設定 ビットが 1 のとき：ファイルの最後に設定

mode で示したファイルの処理の指定と実際のファイルの性質が矛盾する場合はエラーにしてください。正常にファイルがオープンできた場合は、以後の read、write、lseek、close ルーチンで使用されるファイル番号 (0～127) を返してください。ファイル番号と実際のファイルの対応は低水準インタフェースルーチンで管理する必要があります。オープンに失敗した場合は-1 を返してください。

リターン値　正常：　正常オープンしたファイルのファイル番号
　　異常：　-1

引　数　name　ファイルのファイル名を指す文字
　　mode　ファイルをオープンするときの処理の指定
　　flg　ファイルをオープンするときの処理の指定 (常に 0777)

ファイルクローズ

int close(int fileno)

説　明　open ルーチンで得られたファイル番号が引数として渡されます。
open ルーチンで設定したファイル管理情報の領域を、再び使用できるように解放してください。また、低水準インタフェースルーチン内で出力ファイルのバッファリングを行っている場合は、バッファの内容を実際のファイルに書き出してください。
ファイルを正常にクローズできた場合は 0、失敗した場合は-1 を返してください。

リターン値　正常：　0
異常：　-1

引　数　fileno　クローズするファイル番号

データ読み込み

*int read(int fileno, char *buf, unsigned int count)*

説　明　第 1 引数(fileno)で示すファイルから、第 2 引数(buf)の指す領域へデータを読み込みます。読み込むデータのバイト数は第 3 引数(count)で示します。
ファイルが終了した場合、count で示されたバイト数以下のバイト数しか読み込むことができません。
ファイルの読み込み/書き出しの位置は、読み込んだバイト数だけ先に進みます。
正常に読み込みができた場合は、実際に読み込んだバイト数を返してください。読み込みに失敗した場合は-1 を返してください。

リターン値　正常：　実際に読み込んだバイト数
異常：　-1

引　数　fileno　読み込みの対象となるファイル番号
buf　読み込んだデータを格納する領域
count　読み込むバイト数

データ書き出し

*int write(int fileno, char *buf, unsigned int count)*

説　明　第 2 引数(buf)の指す領域から、第 1 引数(fileno)の示すファイルにデータを書き出します。書き込むデータのバイト数は第 3 引数(count)で示します。
ファイルを書き出そうとしているデバイス(ディスク等)が満杯の時は、count で示されたバイト数以下のバイト数しか書き出すことができません。実際に書き出すことのできたバイト数が何度か連続して 0 バイトの場合、ディスクが満杯であると判断してエラー(-1)を返すように実現することをお勧めします。
ファイルの読み込み/書き出しの位置は、書き出したバイト数だけ先に進みます。
正常に書き出しができた場合は、実際に書き出したバイト数を返してください。書き出しに失敗した場合は-1 を返してください。
引数 count が 0 の場合、返却値が 0 となるように実装してください。

リターン値　正常：　実際に書き出されたバイト数
　　異常：　-1

引　数　fileno　書き出しの対象となるファイル番号
　　buf　書き出すデータ領域
　　count　書き出すバイト数

ファイル内位置設定

long lseek(int fileno, long offset, int base)

説　明　ファイルの読み込み/書き出しを行うファイル内の位置を、バイト単位で設定します。
新しいファイル内の位置は、第 3 引数(base)によって、以下の方法で計算し設定してください。
(1) base が 0 のとき　ファイルの先頭から offset バイトの位置に設定します。
(2) base が 1 のとき　現在の位置に offset バイトを加えた位置に設定します。
(3) base が 2 のとき　ファイルのサイズに offset バイトを加えた位置に設定します。
ファイルがコンソールやプリンタ等の対話的なデバイスの場合や、新しいオフセットの値が負になったり、(1)(2)のときファイルのサイズをこえる場合はエラーにします。
正しくファイル位置を設定できた場合は、新しい読み込み/書き出し位置のファイルの先頭からのオフセットを、そうでない場合は-1 を返してください。

リターン値　正常：　新しいファイルの読み込み/書き出し位置のファイルの先頭からのオフセット(バイト単位)
　　異常：　-1

引　数　fileno　対象となるファイル番号
　　offset　読み込み/書き出しの位置を示すオフセット(バイト単位)
　　base　オフセットの起点

メモリ領域割り付け

*char *sbrk(size_t size)*

説　明　メモリ領域を割り付けるサイズが引数として渡されます。
連続して sbrk ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。割り付けるメモリ領域が不足した場合はエラーにしてください。
正常に割り付けができた場合は、割り付けた領域の先頭のアドレスを、割り付けに失敗した場合は、「(char *)-1」を返してください。

リターン値　正常：　割り付けた領域の先頭アドレス
異常：　(char *)-1

引　数　size　割り付けるデータのサイズ

X メモリ領域割り付け

*char _ _X *sbrk_ _X(size_t size)*

説　明　X メモリ領域を割り付けるサイズが引数として渡されます。
連続して sbrk_ _X ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。割り付けるメモリ領域が不足した場合はエラーにしてください。
正常に割り付けができた場合は、割り付けた領域の先頭のアドレスを、割り付けに失敗した場合は、「(char _ _X *)-1」を返してください。

リターン値　正常：　割り付けた領域の先頭アドレス
異常：　(char _ _X *)-1

引　数　size　割り付けるデータのサイズ

Y メモリ領域割り付け

*char_ _Y *sbrk_ _Y(size_t size)*

説　明　Y メモリ領域を割り付けるサイズが引数として渡されます。
連続して sbrk_ _Y ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。割り付けるメモリ領域が不足した場合はエラーにしてください。
正常に割り付けができた場合は、割り付けた領域の先頭のアドレスを、割り付けに失敗した場合は、「(char _ _Y *)-1」を返してください。

リターン値　正常：　割り付けた領域の先頭アドレス
異常：　(char _ _Y *)-1

引　数　size　割り付けるデータのサイズ

errno アドレス取得

*int *errno_addr(void)*

説　明　現在のタスクが持つエラー番号のアドレスを返却します。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　現在のタスクが持つエラー番号のアドレス

セマフォ確保

int wait_sem(int semnum)

説　明　semnum で示されたセマフォを確保します。
確保できた場合は 1、確保できなかった場合は 0 を返してください。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　正常：　1
異常：　0

引　数　semnum　セマフォ ID

セマフォ解放

int signal_sem(int semnum)

説　明　semnum で示されたセマフォを解放します。
解放できた場合は 1、解放できなかった場合は 0 を返してください。
標準ライブラリ構築ツールで reent オプションを指定して作成した標準ライブラリを使用する場合に、本関数は必要になります。

リターン値　正常：　1
異常：　0

引　数　semnum　セマフォ ID

(c) 低水準インタフェースルーチンコーディング例

```
/*****************************************************************************/
/*                                lowsrc.c:                                  */
/*---------------------------------------------------------------------------*/
/*     SuperH RISC engine シリーズ  シミュレータ・デバッガ  インタフェースルーチン      */
/*            - 標準入出力(stdin,stdout,stderr)だけをサポートしています -          */
/*****************************************************************************/
#include <string.h>

/*  ファイル番号  */
#define STDIN  0                          /*  標準入力  (コンソール)  */
#define STDOUT 1                          /*  標準出力  (コンソール)  */
#define STDERR 2                          /*  標準エラー出力 (コンソール)*/

#define FLMIN 0                           /*  最小のファイル番号*/
#define FLMAX 3                           /*  ファイル数の最大値*/

/*  ファイルのフラグ  */
#define O_RDONLY 0x0001                   /*  読み込み専用*/
#define O_WRONLY 0x0002                   /*  書き込み専用*/
#define O_RDWR 0x0004                     /*  読み書き両用*/

/*  特殊文字コード  */
#define CR 0x0d                           /*  復帰*/
#define LF 0x0a                           /*  改行*/

/*  sbrk で管理する領域サイズ  */
#define HEAPSIZE 1024

/*****************************************************************************/
/*                              参照関数の宣言：                              */
/*     シミュレータ・デバッガでコンソールへの文字入出力を行うアセンブリプログラムの参照      */
/*****************************************************************************/
extern void charput(char);                /*  一文字入力処理*/
extern char charget(void);                /*  一文字出力処理*/

/*****************************************************************************/
/*                              静的変数の定義：                              */
/*                低水準インタフェースルーチンで使用する静的変数の定義                 */
/*****************************************************************************/
char flmod[FLMAX];                        /*  オープンしたファイルのモード設定場所  */

union HEAP_TYPE{
   long dummy;                            /*  4 バイトアライメントにするためのダミー*/
   char heap[HEAPSIZE];                   /*  sbrk で管理する領域の宣言*/
};

static union HEAP_TYPE heap_area;
static _ _X union HEAP_TYPE heap_area_ _X;
static _ _Y union HEAP_TYPE heap_area_ _Y;

static char *brk=(char*)&heap_area;       /*  sbrk で割り付けた領域の最終アドレス  */
static _ _X char *brk_ _X=(char _ _X *)&heap_area_ _X;
                                       /*  sbrk_ _X で割り付けた領域の最終アドレス  */
```

```
static _ _Y char *brk_ _Y=(char _ _Y *)&heap_area_ _Y;
                                        /*  sbrk_ _Y で割り付けた領域の最終アドレス  */
/*****************************************************************************/
/*                          open:ファイルのオープン                              */
/*                        リターン値：ファイル番号 (成功)                          */
/*                                  -1          (失敗)                           */
/*****************************************************************************/
int open(char *name,                    /*  ファイル名        */
         int mode)                      /*  ファイルのモード   */
{
    /*  ファイル名に従ってモードをチェックし、ファイル番号を返す  */

    if (strcmp(name,"stdin")==0) {      /*  標準入力ファイル  */
        if ((mode&O_RDONLY)==0) {
            return (-1);
        }
        flmod[STDIN]=mode;
        return (STDIN);
    }

    else if (strcmp(name,"stdout")==0) {/*  標準出力ファイル  */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDOUT]=mode;
        return (STDOUT);
    }

    else if (strcmp(name,"stderr")==0){/* 標準エラー出力ファイル  */
        if ((mode&O_WRONLY)==0) {
            return (-1);
        }
        flmod[STDERR]=mode;
        return (STDERR);
    }

    else {
        return (-1);                    /* エラー  */
    }
}

/*****************************************************************************/
/*                          close:ファイルのクローズ                             */
/*                        リターン値： 0          (成功)                          */
/*                                  -1          (失敗)                           */
/*****************************************************************************/
int close(int fileno)                   /*  ファイル番号  */
{
    if (fileno<FLMIN || FLMAX<fileno) {/*  ファイル番号の範囲チェック  */
        return -1;
    }

    flmod[fileno]=0;                    /*  ファイルのモードリセット  */

    return 0;
}
```

```
/*****************************************************************************/
/*                          read:データの読み込み                              */
/*                  リターン値： 実際に読み込んだ文字数 (成功)                  */
/*                              -1                  (失敗)                    */
/*****************************************************************************/
int read(int fileno,                    /*  ファイル番号          */
        char *buf,                      /*  転送先バッファアドレス  */
        unsigned int  count)            /*  読み込み文字数        */
{
   unsigned int i;

   /*  ファイル名に従ってモードをチェックし、一文字づつ入力してバッファに格納  */

   if (flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR) {
      for (i=count;i>0;i--) {
         *buf=charget();
         if (*buf==CR) {                /*  改行文字の置き換え  */
            *buf=LF;
         }
      buf++;
      }
      return count;
   }

   else {
      return -1;
   }
}

/*****************************************************************************/
/*                          write:データの書き出し                             */
/*                  リターン値： 実際に書き出した文字数 (成功)                  */
/*                              -1                  (失敗)                    */
/*****************************************************************************/
int write(int fileno,                   /*  ファイル番号  */
         char *buf,                     /*  転送元バッファアドレス  */
         unsigned int  count)           /*  書き出し文字数  */
{
   unsigned int i;
   char c;

   /*  ファイル名に従ってモードをチェックし、一文字づつ出力  */

   if (flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR) {
      for (i=count; i>0; i--) {
         c=*buf++;
         charput(c);
      }
      return count;
   }

   else {
      return -1;
   }
}
```

```
/*****************************************************************************/
/*                    lseek:ファイルの読み込み／書き出し位置の設定                */
/*         リターン値：読み込み／書き出し位置のファイル先頭からのオフセット (成功)      */
/*                    -1                                          (失敗)      */
/*                (コンソール入出力では、lseek はサポートしていません)                */
/*****************************************************************************/
long lseek(int fileno,                  /*  ファイル番号            */
         long offset,                   /*  読み込み／書き出し位置   */
         int  base)                     /*  オフセットの起点         */
{
   return -1;
}

/*****************************************************************************/
/*                          sbrk:メモリ領域の割り付け                          */
/*                  リターン値：割り付けた領域の先頭アドレス  (成功)              */
/*                             -1                          (失敗)              */
/*****************************************************************************/
char *sbrk(size_t size)                 /*  割り付ける領域のサイズ  */
{
   char *p;

   /*  空き領域のチェック  */

   if (brk+size>heap_area.heap+HEAPSIZE) {
      return (char *)-1;
   }

   p=brk;                               /* 領域の割り付け     */
   brk+=size;                           /* 最終アドレスの更新  */
   return p;
}

/*****************************************************************************/
/*                          sbrk_ _X:_ _X メモリ領域の割り付け                  */
/*                  リターン値：割り付けた領域の先頭アドレス  (成功)              */
/*                             -1                          (失敗)              */
/*****************************************************************************/
char _ _X *sbrk_ _X(size_t size)        /*  割り付ける領域のサイズ  */
{
   _ _X char *p;

   /*  空き領域のチェック  */

   if (brk_ _X+size>heap_area_ _X.heap+HEAPSIZE) {
      return (char _ _X *)-1;
   }

   p=brk_ _X;                           /* 領域の割り付け     */
   brk_ _X+=size;                       /* 最終アドレスの更新  */
   return p;
}
/*****************************************************************************/
/*                          sbrk_ _Y:_ _Y メモリ領域の割り付け                  */
/*                  リターン値：割り付けた領域の先頭アドレス  (成功)              */
/*                             -1                          (失敗)              */
/*****************************************************************************/
```

```
char _ _Y *sbrk_ _Y(size_t size)       /*  割り付ける領域のサイズ  */
{
    _ _Y char *p;

    /*  空き領域のチェック  */

    if (brk_ _Y+size>heap_area_ _Y.heap+HEAPSIZE) {
        return (char _ _Y *)-1;
    }

    p=brk_ _Y;                          /* 領域の割り付け      */
    brk_ _Y+=size;                      /* 最終アドレスの更新  */
    return p;
}


;;;
```

```
;;;
```

(d) リエントラントライブラリ用低水準インタフェースルーチン例

リエントラントライブラリ用低水準インタフェース例を示します。標準ライブラリ構築ツールで reent オプションを指定して作成したライブラリを使用する場合に必要になります。

wait_sem 関数、signal_sem 関数で NG が返った場合、errno に以下を設定し、ライブラリ関数からリターンします。

wait_sem	EMALRESM	malloc 用セマフォ資源確保に失敗しました
	ETOKRESM	strtok 用セマフォ資源確保に失敗しました
	EIOBRESM	_iob 用セマフォ資源確保に失敗しました
signal_sem	EMALFRSM	malloc 用セマフォ資源解放に失敗しました
	ETOKFRSM	strtok 用セマフォ資源解放に失敗しました
	EIOBFRSM	_iob 用セマフォ資源解放に失敗しました

割り込みに関しては、セマフォ確保後により優先度の高い割り込みが発生し、セマフォ確保を行うとデッドロックが発生するため、リソースを共有するような処理が割り込みでネストしないようにしてください。

```
#define MALLOC_SEM            1     /* malloc 用セマフォ No.         */
#define STRTOK_SEM            2     /* strtok 用セマフォ No.         */
#define FILE_TBL_SEM          3     /* fopen 用セマフォ No.          */
#define MALLOC_SEM_ _X        4     /* malloc_ _X 用セマフォ No.     */
#define MALLOC_SEM_ _Y        5     /* malloc_ _Y 用セマフォ No.     */
#define IOB_SEM               6     /* iob 用セマフォ No.            */
#define SEMSIZE               26    /* IOB_SEM_ + _nfiles (nfiles = 20 の場合)*/
#define TRUE                  1
#define FALSE                 0
#define OK                    1
#define NG                    0

extern int *errno_addr(void);
extern int wait_sem(int);
extern int signal_sem(int);

int sem_errno;
int force_fail_signal_sem = FALSE;
static int semaphore[SEMSIZE];

/*****************************************************************************/
/*                        errno_addr:errno アドレスの取得                    */
/*                           リターン値：errno アドレス                      */
/*****************************************************************************/
int *errno_addr(void)
{
    /* 現在のタスクの errno アドレスを返してください */
    return (&sem_errno);
}

/*****************************************************************************/
/*                        wait_sem:指定されたセマフォの確保                  */
/*                           リターン値：OK(=1) (成功)                       */
/*                                       NG(=0) (失敗)                       */
/*****************************************************************************/
int wait_sem(int semnum)      /* セマフォ ID      */
{
    if((0 < semnum) && (semnum < SEMSIZE)) {
          if(semaphore[semnum] == FALSE) {
              semaphore[semnum] = TRUE;
              return(OK);
          }
    }
    return(NG);
}
```

```
/*****************************************************************************/
/*                      signal_sem:指定されたセマフォの解放                    */
/*                           リターン値：OK(=1) (成功)                         */
/*                                       NG(=0) (失敗)                         */
/*****************************************************************************/
int signal_sem(int semnum)    /* セマフォ ID      */
{
    if(!force_fail_signal_sem) {
          if((0 <= semnum) && (semnum < SEMSIZE)) {
              if( semaphore[semnum] == TRUE ) {
             semaphore[semnum] = FALSE;
             return(OK);
              }
          }
    }
    return(NG);
}
```

(7) 終了処理ルーチン

(a) 終了処理の登録と実行(atexit)ルーチンの作成例

終了処理の登録を行うライブラリ atexit 関数の作成法を示します。

atexit 関数では、引数として渡された関数のアドレスを、終了処理のテーブルに登録します。登録された関数の個数が限界値(ここでは、登録できる個数を 32 個とします)を超えた場合、あるいは同じ関数が二度以上登録された場合はリターン値として NULL を返します。そうでなければ NULL 以外の値(ここでは、登録した関数のアドレス)を返します。

以下にプログラム例を示します。

例:

```
#include <stdlib.h>
typedef void *atexit_t ;

int _atexit_count=0 ;

atexit_t (*_atexit_buf[32])(void) ;

#ifdef __cplusplus
extern "C"
#endif
atexit_t atexit(atexit_t (*f)(void))
{
    int i;

    for(i=0; i<_atexit_count ; i++)              // 既に登録されていないかチェックします
        if(_atexit_buf[i]==f)
            return NULL ;
    if (_atexit_count==32)                       // 登録数の限界値をチェックします
        return NULL ;
    else {
        _atexit_buf[_atexit_count++]=f;          // 関数のアドレスを登録します
        return f;
    }
}
```

(b) プログラムの終了(exit)ルーチンの作成例

プログラムの終了処理を行うライブラリ exit 関数の作成法を示します。プログラムの終了処理は、ユーザシステムによって異なりますので、以下のプログラム例を参考に、ユーザシステムの仕様に従った終了処理を作成してください。

exit 関数は、引数として渡されたプログラムの終了コードに従ってプログラムの終了処理を行い、プログラム起動時の環境に戻ります。ここでは、終了コードを外部変数に設定して、main 関数を呼び出す直前に setjmp 関数で退避した環境に戻ることによって実現します。プログラム実行前の環境に戻るためには、次の関数「callmain」を作成し、初期設定関数「PowerON_Reset」から関数「main」を呼び出す代わりに、関数「callmain」を呼び出してください。

以下にプログラム例を示します。

```
#include <setjmp.h>
#include <stddef.h>

typedef void * atexit_t ;
extern int _atexit_count ;
extern atexit_t (*_atexit_buf[32])(void) ;
#ifdef __cplusplus
extern "C"
#endif
void _CLOSEALL(void);
int main(void);
extern jmp_buf _init_env ;
int _exit_code ;

#ifdef __cplusplus
extern "C"
#endif
void exit(int code)
{
   int i;
   _exit_code=code ;                    // _exit_code にリターンコードを設定します
   for(i=_atexit_count-1; i>=0; i--)    // atexit 関数で登録した関数を順次実行します
      (*_atexit_buf[i])();
   _CLOSEALL();                         // オープンした関数を全てクローズします
   longjmp(_init_env, 1) ;              // setjmp で退避した環境にリターンします
}
#ifdef __cplusplus
extern "C"
#endif
void callmain(void)
{
    // setjmp を用いて現在の環境を退避し、main 関数を呼び出します
    if(!setjmp(_init_env))
       _exit_code=main();               // exit 関数からのリターン時には処理を終了します
}
```

(c) 異常終了(abort)ルーチンの作成例

異常終了の場合は、ご使用になっているユーザシステムの仕様に従って、プログラムを異常終了させる処理を行ってください。

C++プログラムを使用する場合、以下のときにも abort 関数を呼び出します。

- 例外処理が正しく動作しなかった場合
- 純粋仮想関数自体をコールした場合
- dynamic_cast に失敗した場合
- typeid に失敗した場合
- クラス配列の delete 時に情報が取れなかった場合
- クラスオブジェクトのデストラクタコール情報登録時に矛盾が発生した場合

以下、標準出力装置にメッセージを出力したあと、ファイルをクローズしてから無限ループしてリセットを待つプログラム例を示します。

例:

```
#include <stdio.h>

#ifdef _ _cplusplus
extern "C"
#endif
void _CLOSEALL(void);
#ifdef _ _cplusplus
extern "C"
#endif
void abort(void)
{
        printf("program is abort !!¥n");   // メッセージを出力します
        _CLOSEALL();                       // ファイルをクローズします
        while(1)    ;                      // 無限ループします
}
```

9.3 C/C++プログラムとアセンブリプログラムとの結合

C/C++プログラムとアセンブリプログラムの結合時に留意すべき以下の内容について述べます。

- 外部名の相互参照方法
- 関数呼び出し規約

9.3.1 外部名の相互参照方法

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

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

(1) アセンブリプログラムの外部名を C/C++プログラムで参照する方法

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

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

アセンブリプログラム(定義する側)

```
     .EXPORT  _a, _b
     .SECTION D,DATA,ALIGN=4
 _a: .DATA.L 1
 _b: .DATA.L 1
     .END
```

C/C++プログラム(参照する側)

```
extern int a,b;

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

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

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

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

C/C++プログラム(定義する側)

```
int a;
```

アセンブリプログラム(参照する側)

```
        .IMPORT    _a
        .SECTION   P,CODE,ALIGN=4
        MOV.L      A_a,R1
        MOV.L      @R1,R0
        ADD        #1,R0
        RTS
        MOV.L      R0,@R1
        .ALIGN     4
A_a:    .DATA.L    _a
        .END
```

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

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

C++プログラム(呼び出される側)

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

アセンブリプログラム(呼び出す側)

```
          .IMPORT  _sub
          .SECTION P,CODE,ALIGN=4
                  :




          STS.L   PR,@-R15
          MOV.L   R1,@(1,R15)
          MOV     R3,R12
          MOV.L   A_sub,R0
          JSR     @R0
          NOP
          LDS.L   @R15+,PR
                  :
A_sub: .DATA.L _sub
          .END
```

9.3.2 関数呼び出し規約

C/C++プログラムとアセンブリプログラム間で相互に関数呼び出しを行うときに、アセンブリプログラム側で守るべき次の 4 つの規則について説明します。

(1) スタックポインタに関する規則
(2) スタックフレームの割り付け、解放に関する規則
(3) レジスタに関する規則
(4) 引数とリターン値の設定、参照に関する規則

(1) スタックポインタに関する規則

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

(2) スタックフレームの割り付け、解放に関する規則

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

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

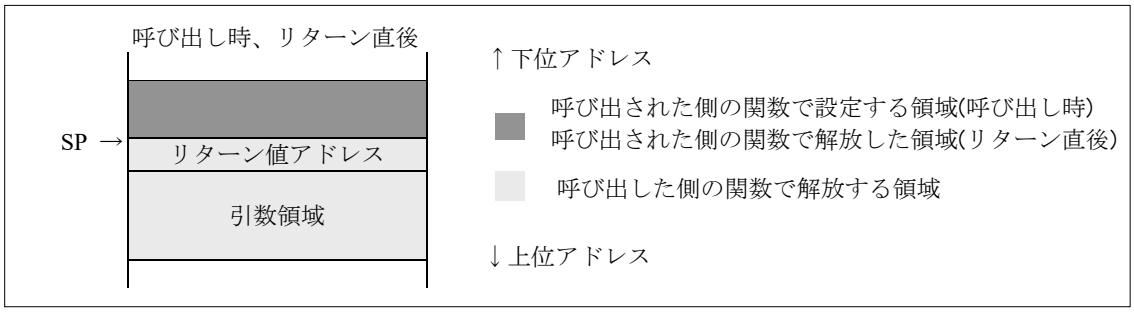


図 9.8 スタックフレームの割り付け、解放に関する規則

(3) レジスタに関する規則

関数呼び出し前後においてレジスタの値が同一であることを保証するレジスタと保証しないレジスタがあります。レジスタの保証規則を表 9.5に示します。

表 9.5 関数呼び出し前後のレジスタ保証規則

項目	対象レジスタ	プログラミングにおける注意点
1 保証しないレジスタ	R0～R7, FR0～FR11[*1] DR0～DR10[*2] FPUL[*1,*2], FPSCR[*1,*2,*4] A0[*3], A0G[*3], A1[*3], A1G[*3], M0[*3], M1[*3], X0[*3],X1[*3], Y0[*3], Y1[*3], DSR[*3], MOD[*3], RS[*3], RE[*3]	関数呼び出し時に対象レジスタに有効な値があれば、呼び出し側で値を退避する。呼び出される側の関数では退避せずに使用できる。ただし fpscr=safe を指定した場合、FPSCR は保証するレジスタ。
2 保証するレジスタ	R8～R15 MACH, MACL, PR FR12～FR15[*1] DR12～DR14[*2]	対象レジスタのうち関数内で使用するレジスタの値を退避し、リターン時に回復する。ただし、macsave = 0 を指定した場合は MACH、MACL は保証しないレジスタ。また、gbr=auto を指定した場合、GBR は保証するレジスタ。

【注】 *1 SH-2E、SH2A-FPU、SH-4、SH-4A の単精度浮動小数点用レジスタです。
*2 SH2A-FPU、SH-4、SH-4A の倍精度浮動小数点用レジスタです。
*3 SH2-DSP、SH3-DSP、SH4AL-DSP の DSP レジスタです。
*4 関数先頭における精度モードは以下となります。
・fpu=double オプションを使用した場合は倍精度となります。
・fpu=single オプションを使用した場合、および fpu オプションを使用しない場合は単精度となります。なお、割り込み関数の場合は実際には倍精度モード中に呼び出されることがあるため精度モードの設定が必要になる場合があります。詳細は「9.4.1(6) マイコン種別が SH2A-FPU|SH4|SH4A の場合の割り込み関数について」を参照してください。

以下、レジスタの保証規則の具体例を示します。

(a) アセンブリプログラムのサブルーチンを C/C++プログラムから呼び出す場合

アセンブリプログラム(呼び出される側)

```
        .EXPORT  _sub
        .SECTION P,CODE,ALIGN=4
_sub:   MOV.L    R14,@-R15          ; 関数内で使用するレジスタの退避
        MOV.L    R13,@-R15
        ADD      #-8,R15
                                    ; 関数本体の処理
         :                          ; (R0～R7は保証しないレジスタのため、
                                    ; 関数内では退避せずに使用可能)
        ADD      #8,R15
        MOV.L    @R15+,R13          ; 退避したレジスタの回復
        RTS
        MOV.L    @R15+,R14
        .END
```

C/C++プログラム(呼び出す側)

```
#ifdef _ _cplusplus
extern "C"
#endif
void sub();

void f()
{
      sub();
}
```

(b) C/C++プログラムの関数をアセンブリプログラムから呼び出す場合

C/C++プログラム(呼び出される側)

```
void sub()
{
    :
}
```

アセンブリプログラム(呼び出す側)

```
        .IMPORT  _sub
        .SECTION P,CODE,ALIGN=4
              :

        STS.L   PR,@-R15
        MOV.L   R1,@(4,R15)
        MOV     R3,R12
        MOV.L   A_sub,R0
        JSR     @R0
        NOP
        LDS.L   @R15+,PR
              :
A_sub:  .DATA.L _sub
        .END
```

呼び出す関数名の先頭に下線"_"を付けたものを.IMPORTで宣言(C)
コンパイラが関数宣言/定義から生成した外部名*を.IMPORTで宣言(C++)

関数呼び出しする場合は、PRレジスタ(リターンアドレス格納レジスタ)を退避
レジスタR0～R7に有効な値があれば空きレジスタ(R8～R14)またはスタックに退避
関数「sub」の呼び出し
PRレジスタの復帰
関数「sub」のアドレスデータ

【注】 * 関数名、静的データメンバから生成する外部名は、C++コンパイルのとき一定の規則で変換を行っています。コンパイラが生成した外部名を知る必要があるときは、code=asmcode または listfile オプションにてコンパイラが生成する外部名を参照してください。また、C++の関数を「extern "C"」を付与して関数定義を行えば、外部名は C の関数と同様の生成規則になります。ただし、その関数を多重定義できなくなります。

(4) 引数とリターン値の設定、参照に関する規則

以下、引数とリターン値の設定、参照方法について説明します。解説では、まず引数とリターン値に対する一般的な規則について述べたあと、引数の割り付け方とリターン値の設定場所について述べます。

(a) 引数とリターン値に対する一般的な規則

- 引数の渡し方

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

- **型変換の規則**

引数を渡す場合、またはリターン値を返す場合、自動的に型変換を行う場合があります。以下、この型変換の規則について説明します。

♦ **型の宣言された引数の型変換**

関数原型によって型が宣言されている引数は、宣言された型に変換します。

♦ **型の宣言されていない引数の型変換**

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

- (signed)char 型、unsigned char 型、(signed)short 型、unsigned short 型の引数は、(signed)int 型に変換します。
- float 型の引数は、double 型に変換します。
- 上記以外の引数は、変換しません。

♦ **リターン値の型変換**

リターン値は、その関数の返す型に変換します。

例

(1)

```
long f( );
long f( )
{
    float x;
    return x;
}
```

← 関数原型にしたがってリターン値はlong型に変換されます。

(2)

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

→ cは、対応する引数の型宣言がないので、int型に変換されます。

→ 1.0は、対応する引数の型がint型なので、int型に変換されます。

(b) 引数の割り付け領域

引数は、レジスタに割り付ける場合とレジスタに割り付けられないときスタック上の引数領域に割り付ける場合があります。引数の割り付け領域を**図 9.9**に、引数割り付け領域の一般規則を**表 9.6**にそれぞれ示します。C++プログラムの非静的関数メンバの this ポインタは、R4 に割り付けられます。

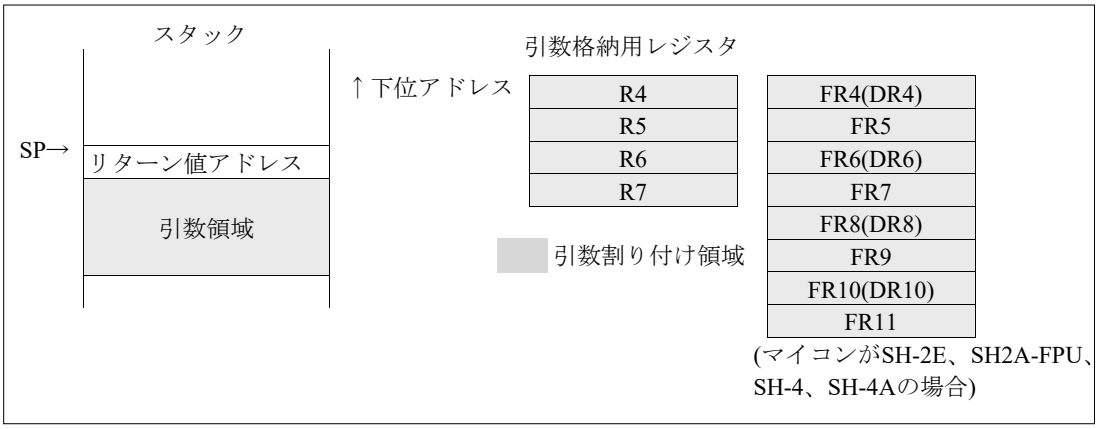


図 9.9 引数の割り付け領域

表 9.6 引数割り付け領域の一般規則

割り付け規則		
レジスタで渡される引数		スタック渡しになる引数
引数格納用レジスタ	対象の型	
R4～R7	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, float(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A 以外の場合)、ポインタ,データメンバへのポインタ,リファレンス	(1) 引数の型がレジスタ渡しの対象の型以外のもの (2) 関数原型により可変個の引数を持つ関数として宣言しているもの*3 (3) 他の引数がすでに R4～R7 に割り付いている場合 (4) 他の引数がすでに FR4(DR4)～FR11(DR10)に割り付いている場合 (5) long long,unsigned long long 型の引数 (6) _ _fixed 型,long _ _fixed 型, _ _accum 型,long _ _accum 型の引数
FR4～FR11*1	SH-2E のとき ・引数が float 型 ・引数が double 型かつ double=float 指定 SH2A-FPU、SH-4、SH-4A のとき ・引数型が float 型かつ fpu=double オプション指定なし ・引数型が double 型または long double 型、かつ fpu=single 指定	
DR4～DR10*2	SH2A-FPU、SH-4、SH-4A のとき ・引数型が double 型または long double 型、かつ fpu=single 指定なし ・引数型が float 型かつ fpu=double 指定	

【注】 *1 SH-2E、SH2A-FPU、SH-4、SH-4A の単精度浮動小数点用のレジスタです。
*2 SH2A-FPU、SH-4、SH-4A の倍精度浮動小数点用レジスタです。
*3 関数原型により可変個の引数をもつ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタック渡しになります。

例

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

(c) 引数の割り付け

- 引数格納用レジスタへの割り付け

引数格納用レジスタには、ソースプログラムの宣言順に番号の小さいレジスタから割り付けます。引数格納用レジスタの割り付け例を図 9.10に示します。

```
f(char a, int b)
{
    ・・・
}
```

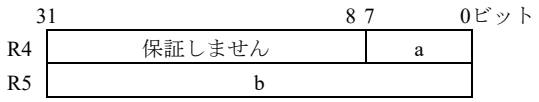


図 9.10 引数格納用レジスタの割り付け例

- スタック上の引数領域への割り付け

スタック上の引数領域には、ソースプログラム上で宣言した順に下位アドレスから割り付けます。

【注】 構造体、共用体、クラス型の引数に関する注意
構造体、共用体、クラス型の引数を設定する場合は、その型の本来のアライメントにかかわらず 4 バイトアライメントに割り付けられ、しかもその領域として 4 の倍数バイトの領域が使用されます。これは、SuperH RISC engine マイコンのスタックポインタが 4 バイト単位で変化するためです。
「9.3.3 引数割り付けの具体例」に、引数割り付けの具体例がありますので、あわせて参照してください。

(d) リターン値の設定場所

関数のリターン値の型によっては、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は表 9.7を参照してください。

関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスをリターン値アドレスに設定してから関数を呼び出します(図 9.11参照)。関数のリターン値が void 型の場合、リターン値を設定しません。

表 9.7 リターン値の型と設定場所

	リターン値の型	リターン値の設定場所
1	(singed)char, unsigned char, (singed)short, unsigned short, (singed)int, unsigned int, long, unsigned long, float, ポインタ, bool, リファレンス, データメンバへのポインタ	R0 : 32 ビット (signed)char,unsigned char の上位 3 バイト、(signed)short,unsigned short の上位 2 バイトの内容は保証しません。ただし、rtnext オプションを指定した場合は(signed)char,(singed)short 型は符号拡張、unsigned char,unsigned short 型はゼロ拡張を行います。 FR0 : 32 ビット (1) SH-2E のとき ・リターン値が float 型 ・リターン値が double 型かつ double=float 指定 (2) SH2A-FPU、SH-4、SH-4A のとき ・リターン値が float 型かつ fpu=double 指定なし ・リターン値が浮動小数点型かつ fpu=single 指定
2	double, long double, 構造体, 共用体, クラス型, 関数メンバへのポインタ	リターン値設定領域(メモリ) DR0:64 ビット SH2A-FPU、SH-4、SH-4A のとき ・リターン値が double 型かつ fpu=single 指定なし ・リターン値が浮動小数点型かつ fpu=double 指定
3	(signed)long long, unsigned long long	リターン値設定領域(メモリ)
4	_ _fixed, long _ _fixed, _ _accum, long _ _accum	リターン値設定領域(メモリ)

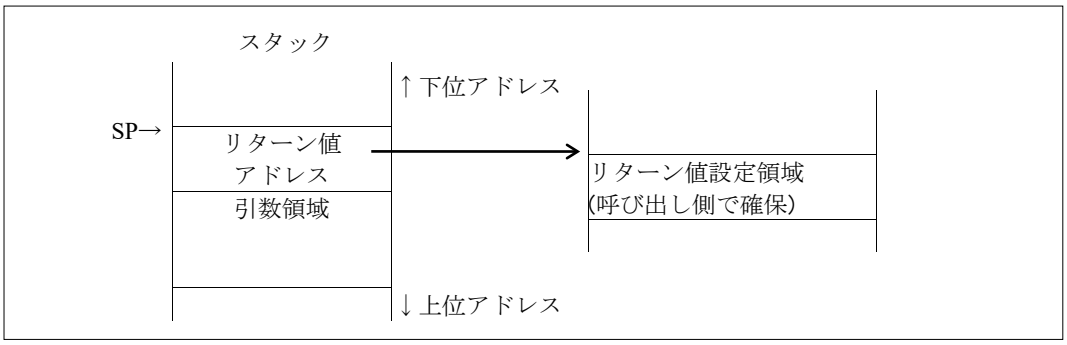
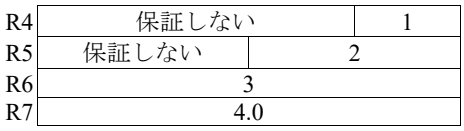


図 9.11 リターン値をメモリに設定する場合のリターン値の設定領域

9.3.3 引数割り付けの具体例

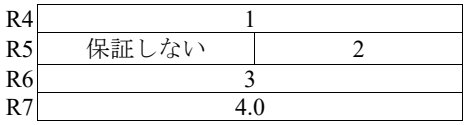
例１．レジスタ渡しの対象の型である引数は、宣言順にレジスタ R4～R7 に割り付けます。

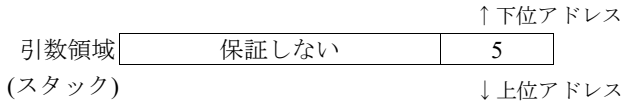
```
int f(char,short,int,float);
    :
f(1,2,3,4.0);
```



例２．レジスタに割り付けることができなかった引数は、スタック渡しになります。また、引数の型が（unsigned)char 型、または、(unsigned)short 型でスタック上の引数領域に割り付く場合、4 バイトに拡張して割り付けます。

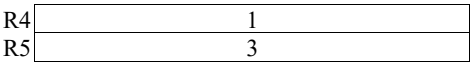
```
int f(int,short,long,float,char);
    :
f(1,2,3,4.0,5);
```

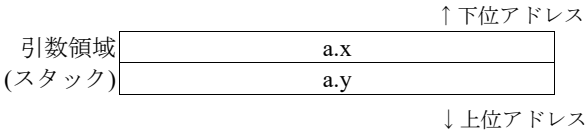




例３．レジスタに割り付けられない型の引数は、スタック渡しになります。

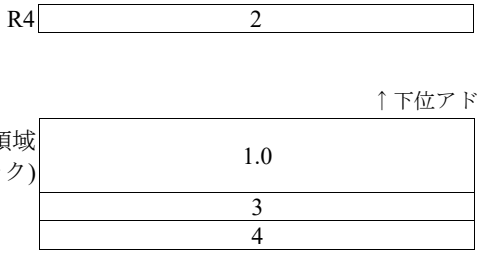
```
struct s{int x,y;}a;
int f(int,struct s,int);
    :
f(1,a,3);
```





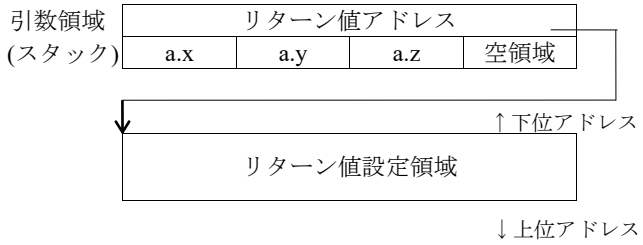
例４．関数原型により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタック渡しになります。

```
int f(double, int, int...)
    :
f(1.0, 2, 3, 4)
```



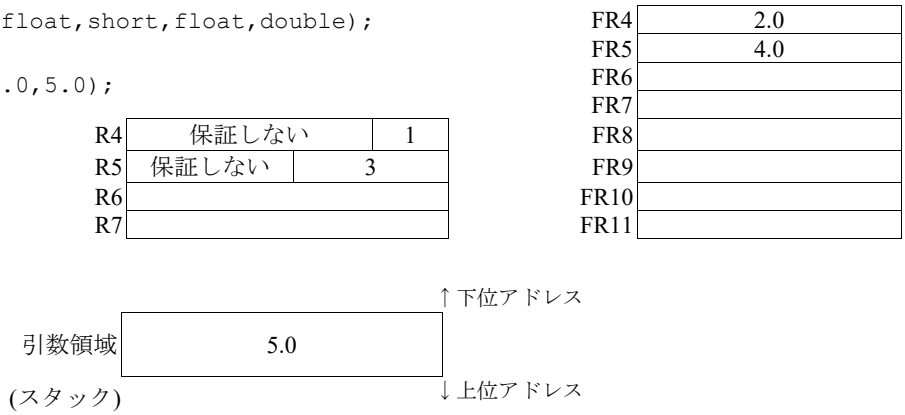
例５．関数の返す型が４バイトをこえる場合またはクラスの場合、引数領域の直前にリターン値アドレスを設定します。また、クラスのサイズが４の倍数バイトでないとき、空領域が生じます。

```
struct s{char x,y,z;}a;
double f(struct s);
    :
f(a);
```



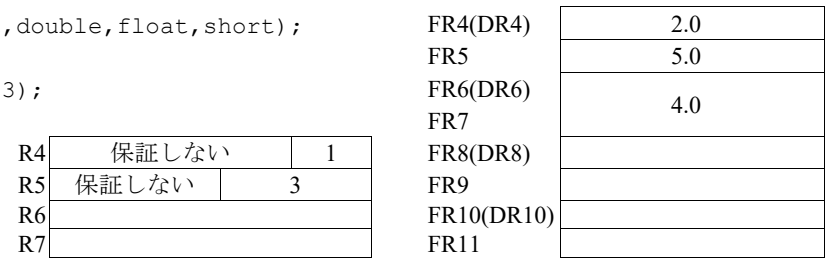
例６．マイコンが SH-2E の場合、float 型の引数は FPU レジスタに割り付けます。

```
int f(char,float,short,float,double);
    :
f(1,2.0,3,4.0,5.0);
```



例７．マイコンが SH2A-FPU、SH-4、SH-4A の場合、float/double 型の引数は FPU レジスタに割り付けます。

```
int f(char,float,double,float,short);
    :
f(1,2.0,4.0,5.0,3);
```



9.3.4　レジスタとスタック領域の使用法

コンパイラのレジスタ、スタック領域の使用法を示します。

関数内でのレジスタ、スタック領域はすべてコンパイラが操作しますので、ユーザが特にこの領域の使用法に留意する必要はありません。

レジスタとスタック領域の使用法を図 9.12に示します。

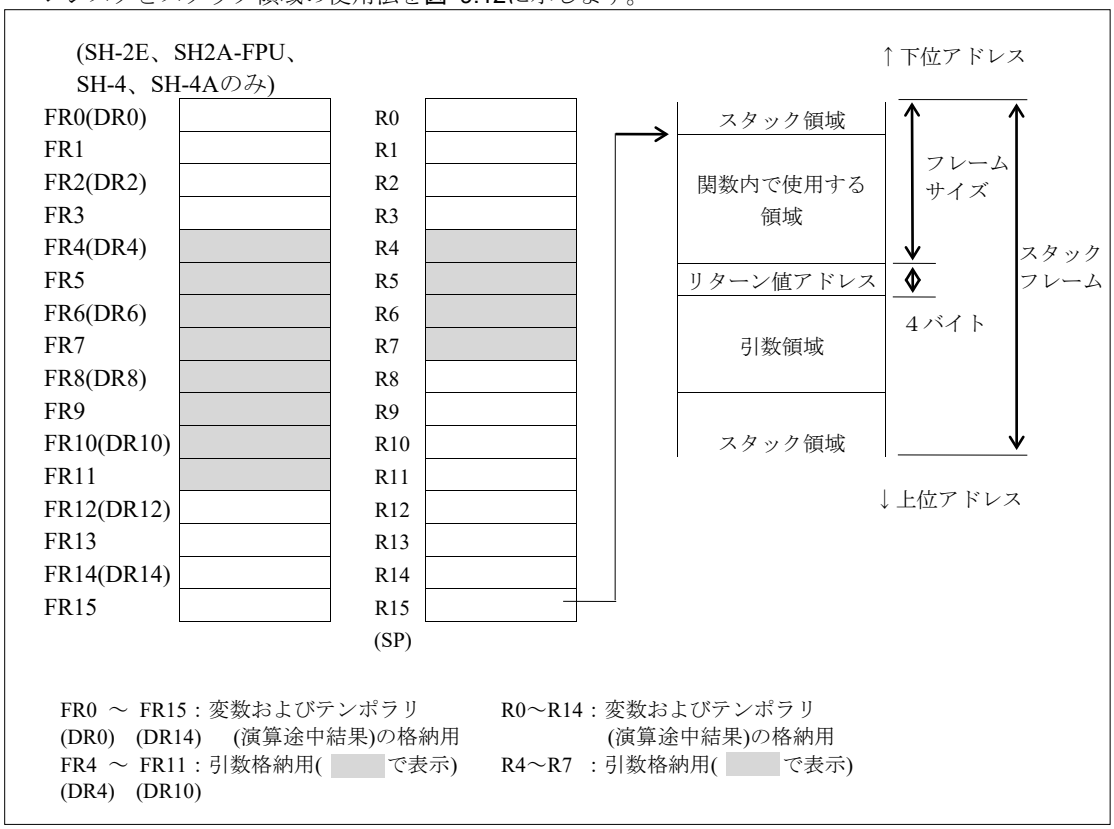


図 9.12　レジスタとスタック領域の使用法

9.4 プログラム作成上の注意事項

本節では、コンパイラにおけるコーディング上の注意事項と、コンパイルからデバッグまでのプログラム開発上の注意事項を述べます。

9.4.1 コーディング上の注意事項

(1) float 型パラメータをもつ関数

float 型のパラメータをもつ関数は、必ず関数原型を行うか、float 型を double 型に変更してください。関数原型のない float 型パラメータの受け渡しによるデータの値は保証しません。

例

```
void f(float);
void g()
{
float a;
...
f(a);
}
void f(float x)
{...}
```

関数 f は、float 型のパラメータをもつ関数です。必ず関数原型を行ってください。

(2) C/C++言語で評価順序を規定していない式

C/C++言語で評価順序を規定していない式で、評価順序で結果が変わるようなコーディングをした場合、その動作は保証しません。

例

`a[i]=a[++i];` 代入式の右辺を先に評価するか後に評価するかで左辺の値が変わります。

`sub(++i, i);` 関数の第 1 引数を先に評価するか後に評価するかで第 2 引数の値が変わります。

(3) オーバフロー演算、ゼロ除算

オーバフロー演算や浮動小数点のゼロ除算があっても、エラーメッセージを出力しません。ただし、以下のいずれかがある場合はエラーメッセージを出力します。

- 絶対値が unsigned long long 型の範囲外の値の整数定数
- 接尾語 f または F を持ち、かつ float 型の範囲外の値の浮動小数点定数
- 接尾語を持たない、または接尾語 l または L を持ち、かつ double 型の範囲外の値の浮動小数点定数
- 整数定数、または浮動小数点定数によるゼロ除算

例

```
unsigned long long la,lb,lc,ld,le=0;
float fa,fb,fc,fc,fd,fe=0.0f;
double da,db,dc,dd,de=0.0;
void main()
{
  la = 32767;
  fa = 3.5e+37f;
```

```
  da = 1.8e+307;

  lb = 18446744073709551616;  /* (W) 整数定数のオーバフローを検出 */
  fb = 3.5e+38f;          /* (W) 浮動小数点定数のオーバフローを検出 */
  db = 1.8e+308;          /* (W) 浮動小数点定数のオーバフローを検出 */

  lc = la + 18446744073709551615;        /* メッセージ出力なし */
  fc = fa + 3.4e+38f;                    /* メッセージ出力なし */
  dc = da + 1.7e+308;                    /* メッセージ出力なし */

  ld /= 0;                  /* (W) 整数定数によるゼロ除算を検出 */
  fd /= 0.0f;          /* (W) 浮動小数点定数によるゼロ除算を検出 */
  dd /= 0.0;           /* (W) 浮動小数点定数によるゼロ除算を検出 */

  ld /= le;                              /* メッセージ出力なし */
  fd /= fe;                              /* メッセージ出力なし */
  dd /= de;                              /* メッセージ出力なし */
}
```

(4) const 型変数への書き込み

const 型の変数を宣言していても、型変換で const 型でない型に変換して代入した場合や、分割コンパイルしたプログラムの間で、型を統一して扱っていない場合は、const 型変数への書き込みをコンパイラでチェックできませんので、注意が必要です。

例

```
const char *p;       /* ライブラリ関数 strcat の第 1 引数は char 型への     */
 :                   /* ポインタ型なので、引数の指す領域が書き換わる        */
strcat(p, "abc");    /* ことがあります。                                   */

  ファイル 1
const int i;

  ファイル 2
extern int i;        /* 変数 i は、ファイル 2 では const 型で宣言していま   */
 :                   /* せんのでファイル 2 の中で書き込んでもエラーに       */
i=10;                /* なりません                                         */
```

(5) 数学関数ライブラリの精度について

acos(x)、asin(x)関数では x≒ 1 で誤差が大きくなりますので注意が必要です。

誤差範囲は以下のとおりです。

acos(1.0 - ε)における絶対誤差	倍精度 2^{-39} ($\varepsilon = 2^{-33}$)
	単精度 2^{-21} ($\varepsilon = 2^{-19}$)
asin(1.0 - ε)における絶対誤差	倍精度 2^{-39} ($\varepsilon = 2^{-28}$)
	単精度 2^{-21} ($\varepsilon = 2^{-16}$)

(6) マイコン種別が SH2A-FPU | SH4 | SH4A の場合の割り込み関数について

浮動小数点型精度モードを持つマイコン（SH2A-FPU | SH4 | SH4A）において、fpu オプション未指定時、あるいは fpu=single 指定時は、全ての関数の関数先頭において精度モードは単精度モード(FPSCR レジスタの PR ビットが 0)であることを想定して浮動小数点演算コードが生成されます。ただし、割り込み関数に関しては、実際には倍精度モード中に呼び出される可能性があります。その為、単精度浮動小数点演算を行う割り込み関数においては、必ず関数内で FPSCR に対し以下の設定を行ってください。fpu=double 指定時は設定は不要です。

[fpu オプション未指定時]

割り込み関数の入口にて、FPSCR の精度モードを単精度(0)に設定してください。

設定例：

```
set_fpscr(get_fpscr()&0xFFF7FFFF);
```

[fpu=single 指定時]

割り込み関数の入口にて、FPSCR の PR ビットの状態を記録したのち、精度モードを単精度(0)に設定してください。そして割り込み関数の出口にて、PR ビットを元の状態に戻してください。

設定例：

関数入口

```
int original_fpscr = get_fpscr();
set_fpscr(original_fpscr&0xFFF7FFFF); //  単精度に設定
```

関数出口

```
set_fpscr(original_fpscr); //  元の精度に戻す
return;
```

ただし、fpu=single 指定がされたプログラムで倍精度モードになり得るのは以下の条件を共に満たす標準ライブラリ関数の実行中のみです。
・fprintf(), printf(), sprintf(), vfprintf(), vprintf(), vsprintf()のいずれかの関数
・上記関数におけるフォーマット指定に"%g, %G, %f, %e, %E"のいずれかを使用している
プログラム内でこの条件を満たすライブラリ関数を使用していない場合は、単精度浮動小数点演算を行う割り込み関数があっても上記設定を行う必要はありません。

9.4.2 C プログラムを C++コンパイラでコンパイルするときの注意事項

(1) 関数の関数原型

関数を使用する前に関数原型が必要です。そのときには、仮引数の型も必ず宣言してください。

```
extern void func1();
void g()
{
    func1(1); // エラー
}
```

```
extern void func1(int);
void g()
{
    func1(1); // OK
}
```

(2) const オブジェクトのリンケージ

const オブジェクトのリンケージは、C プログラムでは外部結合であるのに対し、C++プログラムでは内部結合になります。また、const オブジェクトは初期値を必要とします。

```
const cvalue1;      // エラー

const cvalue2 = 1; // 内部結合
```

```
const cvalue1=0;
   // 初期値を与えます

extern const cvalue2 = 1;
   // Cプログラムと同様に外部結合に
   // なります
```

(3) void*からの代入

C++プログラムでは、明示的なキャストを用いないと他のオブジェクト型へのポインタ(関数へのポインタ、メンバへのポインタを除く)へ代入できません。

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

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

9.4.3 プログラム開発上の注意事項

プログラムの作成からデバッグまでのプログラム開発上の注意事項を示します。

(1) マイコンの選択に関する注意事項

(a) コンパイル、アセンブル時に指定するマイコン種別は統一してください

コンパイル、アセンブル時に cpu オプションで指定するマイコン種別は、必ず統一してください。異なったマイコン種別で作成したオブジェクトプログラムを一緒にリンクした場合、オブジェクトプログラム実行時の動作は保証しません。

(b) アセンブル時はコンパイル時のマイコンと同じマイコン種別を指定してください

コンパイラが生成したアセンブリプログラムをアセンブルするとき、コンパイル時と同じマイコン種別を cpu オプションで指定してください。

(c) リンク時にはマイコン種別に合わせた標準ライブラリを指定してください

マイコン種別に対応するライブラリを指定してください。それ以外のライブラリを指定した場合の動作は保証しません。

(2) オプションに関する注意事項

以下のオプションは、コンパイル時、ライブラリ構築時に必ず統一して下さい。異なるオプションを用いて作成したオブジェクトプログラムを一緒にリンクした場合、オブジェクトプログラム実行時の動作は保証しません。

- endian = big | little (SH-2,SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP)
- pic = 0 | 1 (SH-1 以外)
- fpu = single | double (SH2A-FPU,SH-4,SH-4A)
- fpscr = safe | aggressive (SH2A-FPU,SH-4,SH-4A)
- round = zero | nearest (SH2A-FPU,SH-4,SH-4A)
- denormalize = on | off (SH-4,SH-4A)
- double = float (SH2A-FPU,SH-4,SH-4A 以外)
- exception | noexception
- rtti = on | off
- pack = 1 | 4
- rtnext | nortnext
- macsave
- gbr = auto | user
- bit_order = left | right
- auto_enum

10.　C/C++言語仕様

10.1　言語仕様

10.1.1　コンパイラの仕様

言語仕様で規定していない処理系定義項目について、コンパイラの仕様を示します。

(1)　環境

表 10.1　環境の仕様

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

(2)　識別子

表 10.2　識別子の仕様

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

(3)　文字

表 10.3　文字の仕様

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

(4) 整数

表 10.4 整数の仕様

	項 目	コンパイラの仕様
1	整数型の表現方法とその値	表10.5 に示します。
2	整数の値がより短いサイズの符号付き整数型、または符号なし整数型を同一のサイズの符号付き整数型に変換したときの値(結果の値が変換先の型で表現できない場合)	整数の値の下位 8 バイト(変換後の型のサイズが 8 バイトの場合)、下位 4 バイト(変換後の型のサイズが 4 バイトの場合)、下位 2 バイト(変換後の型のサイズが 2 バイトの場合)あるいは下位 1 バイト(変換後の型のサイズが 1 バイトの場合)が変換後の値となります。
3	符号付き整数に対するビットごとの演算の結果	符号付きの値になります。
4	整数除算における剰余の符号	被除数の符号と同符号になります。
5	負の値を持つ符号付きスカラ型の右シフトの結果	符号ビットを保持します。

表 10.5 整数型とその値の範囲

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

(5) 浮動小数点

表 10.6 浮動小数点の仕様

	項 目	コンパイラの仕様
1	浮動小数点型の表現方法とその値	浮動小数点型には、float 型、double 型と long double 型があります。浮動小数点型の内部表現や変換仕様、演算仕様等の性質は「10.1.3 浮動小数点型の仕様」で説明します。表10.7 に、浮動小数点型の表現可能な値の限界値を示します。
2	整数を本来の値に正確に表現することができない浮動小数点型に変換したときの切り捨て方向	
3	浮動小数点型をより狭い浮動小数点型に変換したときの切り捨てまたは丸め方法	

表 10.7　浮動小数点型の限界値

	項　目		限界値	
			10 進数表現*1	16 進数表現
1	float 型の最大値		3.4028235677973364e+38f (3.4028234663852886e+38f)	7f7fffff
2	float 型の正の最小値		1.0000000000000000E-45f (1.4012984643248171e-45f)	00000001
3	double long double	型の最大値	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffffff
4	double long double	型の正の最小値	4.9406564584124655e-324 (4.9406564584124654e-324)	0000000000000001

【注】 *1　10 進数表現の限界値は 0 または無限大にならない限界値です。また、( )内は理論値を示します。
*2　double=float を指定した場合、double 型は float 型と同じ値となります。
fpu=single を指定した場合、double、long double 型は float 型と同じ値になります。
fpu=double を指定した場合、float 型は double 型と同じ値になります。

(6)　配列とポインタ

表 10.8　配列とポインタの仕様

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

(7) レジスタ

表 10.9 レジスタの仕様

	項 目		コンパイラの仕様
1	レジスタに割り付けることができる変数の最大数	7 個	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, ポインタ
		4 個	float*2
		2 個	double*3
2	レジスタに割り付けることができる変数の型	char, unsigned char, bool, short, unsigned short,int, unsigned int, long, unsigned long, float*2, double*3, ポインタ	

【注】 *1 変数へのレジスタ割り付けは register 記憶クラス宣言を行っているかどうかは影響しません。ただし、enable_register オプション指定時は register 記憶クラス宣言を行った変数を優先的にレジスタに割り付けます。
*2 マイコンが SH-2E、SH2A-FPU、SH-4 または SH-4A の場合のみです。
*3 マイコンが SH2A-FPU、SH-4 または SH-4A の場合のみです。

(8) クラス、構造体、共用体、列挙型、ビットフィールド

表 10.10 クラス、構造体、共用体、列挙型、ビットフィールドの仕様

	項 目	コンパイラの仕様
1	異なる型のメンバでアクセスされる共用体型のメンバ参照	参照はできますが、値は保証しません。
2	クラス・構造体メンバのアライメント	クラス・構造体メンバ中のアライメント数の最大値がそのクラス・構造体のアライメント数になります。割り付け方の詳細な仕様は「10.1.2(2) 構造体/共用体、クラス型」を参照してください。
3	単なる int 型のビットフィールドの符号	signed int 型
4	int 型のサイズ内のビットフィールドの割り付け順序	上位ビットから割り付けます。*1*2
5	int 型のサイズ内にビットフィールドが割り付けられているとき、次に割り付けるビットフィールドのサイズが int 型内の残っているサイズを超えたときの割り付け方	次の int 型の領域に割り付けます。*1
6	ビットフィールドで許される型指定子	char、unsigned char、bool、short、unsigned short、int、unsigned int、long、unsigned long、enum、long long、unsigned long long
7	列挙型の値を表現する整数型	int 型です。ただし、列挙型サイズオプション指定時は、「2.2.5 その他オプション」を参照してください。

【注】 *1 ビットフィールドの割り付け方の詳細については、「10.1.2(3) ビットフィールド」を参照してください。
*2 bit_order=right オプションを指定すると下位ビットから割り付けられます。

(9)　型修飾子

表 10.11　型修飾子の仕様

	項　目	コンパイラの仕様
1	volatile 型データへのアクセスの種類	規定しません。

(10)　宣言

表 10.12　宣言の仕様

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

基本型を修飾する型の数の数え方を、以下に例を用いて示します。

例：

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

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

(11)　文

表 10.13　文の仕様

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

(12)　プリプロセッサ

表 10.14　プリプロセッサの仕様

	項　目	コンパイラの仕様
1	条件コンパイルの定数式内の単一文字の文字定数と実行環境文字集合の対応	プリプロセッサ文の文字定数と実行環境文字集合は一致します。
2	インクルードファイルの読み込み方法	「<」、「>」で囲まれたファイルは include オプションで指定されたパスから読み込みます。 ファイルが見つからない場合、環境変数 SHC_INC 指定フォルダ、システムフォルダ(SHC_LIB)の順序で各フォルダを検索します。*1
3	二重引用符で囲まれたインクルードファイルのサポートの有無	サポートします。インクルードファイルをカレントフォルダから読み込みます。カレントフォルダになければ、本表 2 項の読み込み方法に従います。
4	ソースファイルの文字の並びの対応(マクロ展開後の文字列の空白文字)	空白文字列は、空白文字 1 文字として展開します。
5	#pragma の動作	「10.3.1 #pragma」を参照してください。
6	__DATE__、__TIME__の値	コンパイル開始時のホストマシンのタイマに基づく値が設定されます。

10.1.2 データの内部表現

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

- データのサイズ
  データの占有する領域のサイズです。
- データのアライメント数
  データを割り付けるアドレスに関する制約です。任意のアドレスに割り付ける1バイトアライメント、偶数バイトに割り付ける2バイトアライメント、4の倍数バイトに割り付ける4バイトアライメントがあります。
- 値の範囲
  スカラ型(C言語)、基本型(C++言語)の値のとり得る範囲を示します。
- データの割り付け例
  構造体/共用体(C言語)、クラス型(C++言語)の要素となるデータの割り付け方を示します。

(1) スカラ型 (C 言語)、基本型 (C++言語)

C 言語におけるスカラ型および、C++言語における基本型の内部表現を**表 10.15**に示します。

表 10.15 スカラ型・基本型の内部表現

	型名	サイズ (byte)	アライメント数 (byte)	符号の有無	値の範囲 最小値	値の範囲 最大値
1	char	1	1	有	-2^7 (-128)	2^7-1 (127)
2	signed char	1	1	有	-2^7 (-128)	2^7-1 (127)
3	unsigned char	1	1	無	0	2^8-1 (255)
4	short	2	2	有	-2^{15} (-32768)	2^{15}-1 (32767)
5	unsigned short	2	2	無	0	2^{16}-1 (65535)
6	int	4	4	有	-2^{31}(-2147483648)	2^{31}-1 (2147483647)
7	unsigned int	4	4	無	0	2^{32}-1 (4294967295)
8	long	4	4	有	-2^{31}(-2147483648)	2^{31}-1 (2147483647)
9	unsigned long	4	4	無	0	2^{32}-1 (4294967295)
10	long long	8	4	有	-2^{63} (-9223372036854775808)	2^{63}-1 (9223372036854775807)
11	unsigned long long	8	4	無	0	2^{64}-1 (18446744073709551615)
12	enum*1	4	4	有	-2^{31}(-2147483648)	2^{31}-1 (2147483647)
13	float	4*4	4	有	-∞	+∞
14	double long double	8*2,*4	4	有	-∞	+∞
15	ポインタ	4	4	無	0	2^{32}-1 (4294967295)
16	bool*3	4	4	有	—	—
17	リファレンス*3	4	4	無	0	2^{32}-1 (4294967295)
18	データメンバへのポインタ*3	4	4	有	0	2^{32}-1 (4294967295)
19	関数メンバへのポインタ*3 *5	12	4	—	—	—

【注】 *1　auto_enum オプションを指定した場合、列挙型のサイズは可変になります。
*2　double=float を指定した場合、double 型のサイズは 4 バイトになります。
*3　C++コンパイルのみ有効です。
*4　cpu=sh2afpu|sh4|sh4a かつ fpu=single を指定した場合、double、long double 型を 4 バイト(float 型)として扱います。また、cpu=sh2afpu|sh4|sh4a かつ fpu=double を指定した場合、float 型を 8 バイト(double 型)として扱います。
*5　関数メンバ・仮想関数メンバへのポインタは、以下のデータ構造で表現しています。

```
class PMF{
  public:
    long d;            // オブジェクトのオフセット値
    long i;            // 対象メンバ関数が仮想関数のときの仮想関数表中での
                       // インデックス
    union{
      void (*f)();     // 対象メンバ関数が非仮想関数のときの関数のアドレス
      long offset;     // 対象メンバ関数が仮想関数のときの仮想関数表のオブジェクト
                       // 中のオフセット
    };
};
```

(2)　構造体/共用体 (C 言語)、クラス型 (C++言語)

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

表 10.16 に構造体/共用体、クラス型の内部表現を示します。

表 10.16　構造体/共用体、クラス型の内部表現

	型名	アライメント数(byte)	サイズ(byte)	データの割り付け例		
1	配列型	配列要素のアライメント数	配列要素の数×要素サイズ	char a[10];	アライメント数 サイズ	1byte 10byte

2	構造体型	構造体メンバのアライメント数のうち最大値	メンバのサイズの和
			「(a) 構造体データの割り付け方」参照
				struct {
				char a,b;
				};
					アライメント数
					サイズ
						1byte
						2byte

3	共用体型	共用体メンバのアライメント数のうち最大値	最大メンバのサイズ
			「(b) 共用体データの割り付け方」参照
				union {
				char a,b;
				};
					アライメント数
					サイズ
						1byte
						1byte

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

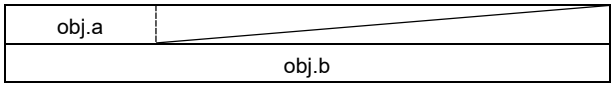
以下の例でサイズを明記していない □ は、4 バイトを表します。 ／ はパディングを表します。

(a) 構造体データの割り付け方

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

  例：

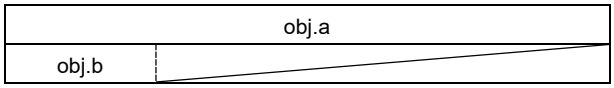
```
struct {
  char a;
  int b;
} obj;
```



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

  例：

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

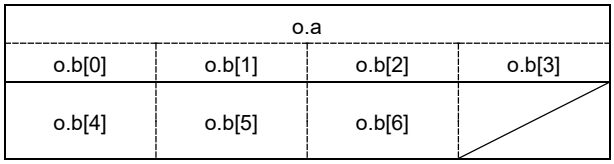


(b) 共用体データの割り付け方

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

  例：

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

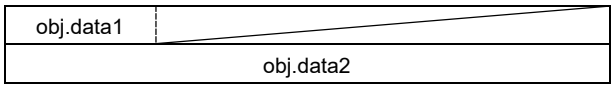


(c)　クラスデータの割り付け方

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

例：

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



・　アライメント数が1の基底クラスから派生したクラスの先頭メンバが1byteデータの場合、パディングを作らないようにデータメンバを割り付けます。

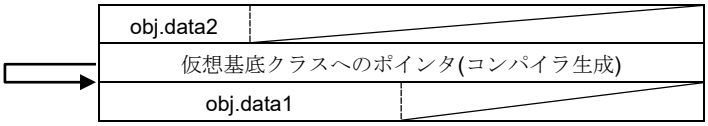
例：

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

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

例：

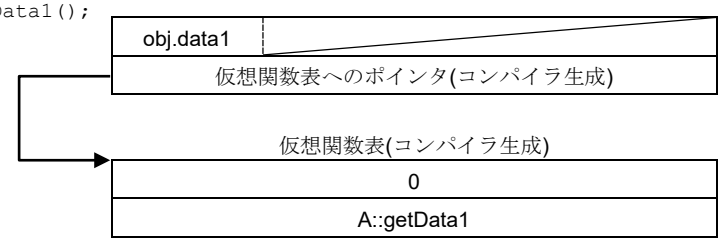
```
class A{
    short data1;
};
class B: virtual protected A{
    char data2;
}obj;
```



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

例：

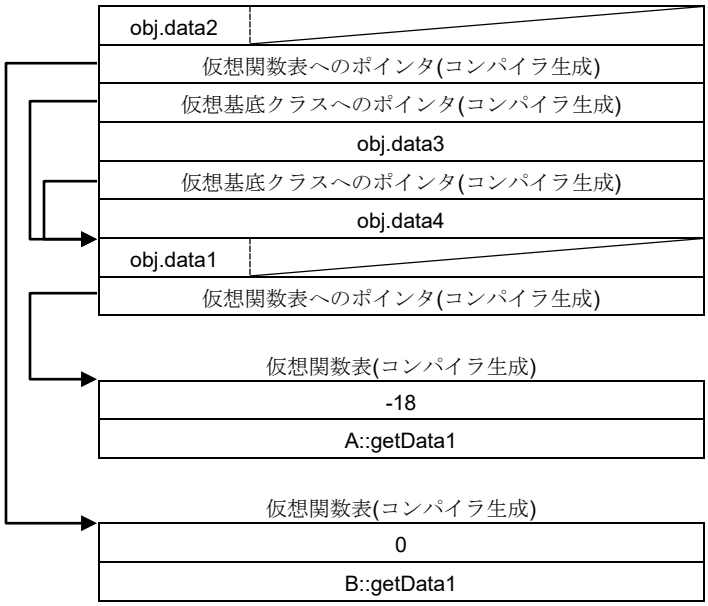
```
class A{
   char data1;
  public:
   virtual char getData1();
}obj;
```



・ 仮想基底クラス、基底クラス、仮想関数があるクラスの例を示します。

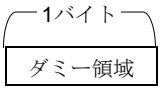
例：

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



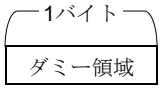
- 空クラスの場合、1バイトのダミー領域を割り付けます。
  例：

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



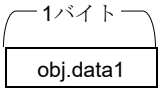
- 空クラスを基底クラスに持つ空クラスの場合でも、ダミー領域は1バイトになります。
  例：

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



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

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



(3) ビットフィールド

ビットフィールドは、構造体、共用体、クラスの中にビット幅を指定して割り付けるメンバです。本項では、ビットフィールド特有の割り付け規則について説明します。

(a) ビットフィールドのメンバ

表 10.17 にビットフィールドメンバの仕様を示します。

表 10.17 ビットフィールドメンバの仕様

	項目	仕様
1	ビットフィールドで許される型指定子	(signed)char、unsigned char、bool[*1]、 (signed)short、unsigned short、enum、 (signed)int、unsigned int、 (signed)long、unsigned long、 (signed)long long、unsigned long long
2	宣言された型に拡張するときの符号の扱い[*2]	符号なし(unsigned を指定した型) →ゼロ拡張[*3] 符号あり(unsigned を指定しない型) →符号拡張[*4]

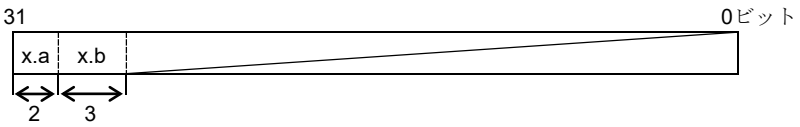
【注】 *1 C++プログラムのみ bool を指定できます。
*2 ビットフィールドのメンバを使用する場合、ビットフィールドに格納したデータを宣言した型に拡張して使用します。符号付き(signed)で宣言されたサイズが 1 ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は 0 と-1 だけになります。0 と 1 を表現する場合には、必ず符号なし(unsigned)で宣言してください。
*3 ゼロ拡張：拡張するときに上位のビットにゼロを補います。
*4 符号拡張：拡張するときにビットフィールドデータの最上位ビットを符号として解釈し、データより上位のビット全てに符号ビットを補います。

(b) ビットフィールドの割り付け方

ビットフィールドは、以下の 5 つの規則に従って割り付けます。

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

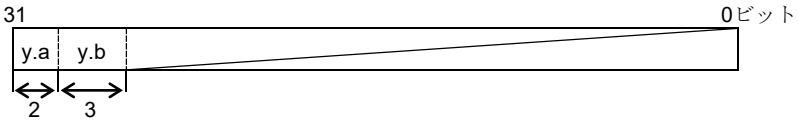
```
struct b1 {
  int a:2;
  int b:3;
} x;
```



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

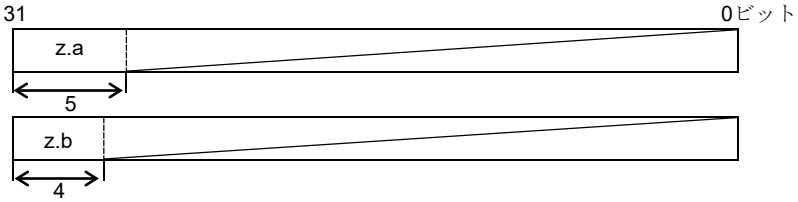
例：

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



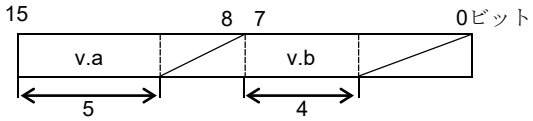
・　異なるサイズの型指定子で宣言されたメンバは、次の領域に割り付けます。
例：

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



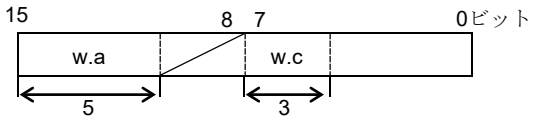
・　同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。
例：

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



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

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



【備考】ビットフィールドメンバを下位ビット側から詰め込むことも可能です。
詳細は、「2.2オプション解説」の `bit_order` **オプション、「10.3.1　#pragma」の**`#pragma bit_order` **を参照してください。**

(4) Little Endian のメモリ割り付け

Little Endian でのメモリ上のデータ配列は以下のとおりです。

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

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

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

2 バイトデータの中のバイト並び順は、上位、下位のバイトが逆になります。

例

0x100番地に2バイトデータ0x1234がある場合

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

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

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

例

0x100番地に4バイトデータ0x12345678がある場合

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

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

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

例

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

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

(e) 構造体/共用体、クラス型データ

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

例

0x100番地に、

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

がある場合

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

(f) ビットフィールド

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

例

0x100番地に、

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

がある場合

Big Endian:	0x100番地:	0x00	Little Endian:	0x100番地:	0x02
	0x101番地:	0x01		0x101番地:	0x00
	0x102番地:	0x00		0x102番地:	0x01
	0x103番地:	0x02		0x103番地:	0x00
	0x104番地:	0x08		0x104番地:	0x00
	0x105番地:	0x00		0x105番地:	0x08
	0x106番地:	パディング		0x106番地:	パディング
	0x107番地:	パディング		0x107番地:	パディング

10.1.3 浮動小数点型の仕様

(1) 浮動小数点型の内部表現

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

(a) 内部表現の形式

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

(b) 浮動小数点データフォーマット

float 型および double 型と long double 型の浮動小数点データフォーマットを図 10.1に示します。

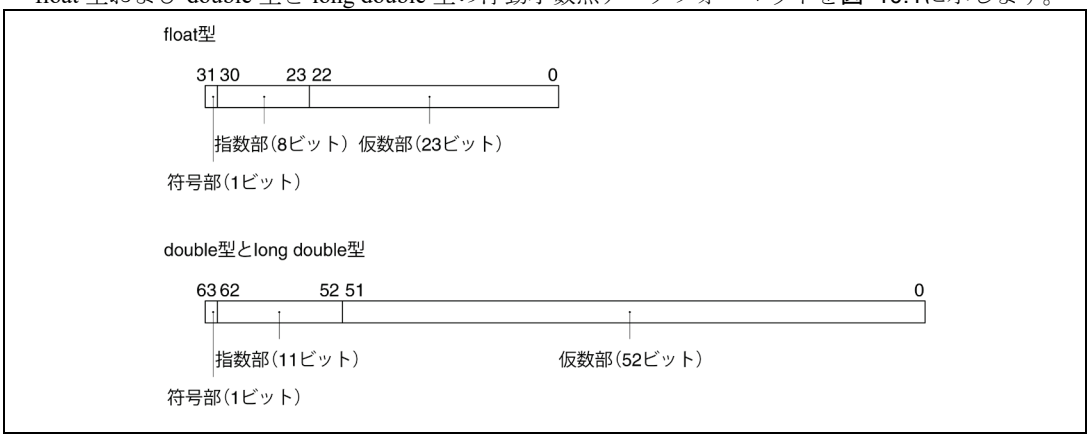


図 10.1 浮動小数点データフォーマット

【注】 double=float を指定した場合、double 型は float 型と同じ内部表現となります。
cpu=sh2afpu|sh4|sh4a かつ fpu=single を指定した場合、double 型、long double 型は float 型と同じ内部表現となります。
cpu=sh2afpu|sh4|sh4a かつ fpu=double を指定した場合、float 型は double 型と同じ内部表現になります。

内部表現の各構成要素の意味を以下に示します。

(i) 符号部
浮動小数点型の符号を示します。0のとき正、1のとき負を示します。

(ii) 指数部
浮動小数点型の指数を2のべき乗で示します。

(iii) 仮数部
浮動小数点型の有効数字に対応するデータです。

(c) 表現する値の種類

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

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

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

(iii) ゼロ
指数部および仮数部が0の場合です。値0.0を表現します。

(iv) 無限大

指数部が全ビット1で仮数部が0の場合です。無限大を表現します。

(v)　非数

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

浮動小数点型の表現する値を決定する条件を**表 10.18**に示します。

表 10.18　浮動小数点型の表現する値の種類

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

【注】 非正規化数は、正規化数で表現できない範囲の絶対値の小さな浮動小数点型を表現しますが、正規化数と比較して有効桁数が少なくなっています。したがって、演算の結果あるいは途中結果が非正規化数となる場合、結果の有効桁数は保証しません。
cpu=sh4|sh4a かつ denormalize=off を指定した場合、非正規化数は 0 として扱います。
cpu=sh4|sh4a かつ denormalize=on を指定した場合、非正規化数は非正規化数のまま扱います。

(2)　float 型

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

(i)　正規化数

符号部は、0(正)または1(負)で、値の符号を示します。

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

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

正規化数の表現する値は、

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

となります。

例：

31	30　23	22　0
1	10000000	11000000000000000000000

符号：　－

指数：　10000000$_{(2)}$－127＝1

仮数：　1.11$_{(2)}$＝1.75

値　：　－1.75×2^1＝－3.5

(2)は2進数を意味します。

(ii) 非正規化数

符号部は0(正)または1(負)で、値の符号を示します。
指数部は0で、実際の指数は-126になります。
仮数部は、1～2^{23}-1で、実際の仮数は、2^{23}のビットを0と仮定し、その直後に小数点があるものとして解釈します。
非正規化数の表現する値は、

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

となります。
例：

31	30　　23	22　　0
0	00000000	11000000000000000000000

符号：　＋
指数：　－126
仮数：　0.11(2)＝0.75
値　：　0.75×2^{-126}

(2)は2進数を意味します。

(iii) ゼロ

符号部は0(正)または1(負)で、それぞれ+0.0、-0.0を示します。
指数部、仮数部はともに0です。
+0.0、-0.0は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては「**10.1.3(4) 浮動小数点演算の仕様**」を参照してください。

(iv) 無限大

符号部は0(正)または1(負)で、それぞれ+∞、-∞を示します。
指数部は255(2^8-1)です。
仮数部は0です。

(v) 非数

指数部は255(2^8-1)です。
仮数部は0以外の値です。

【注】 マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合、仮数部の最上位ビットが 0 の非数を qNaN、仮数部の最上位ビットが 1 の非数を sNaN と呼びます。
その他の仮数フィールドの値、および符号部については規定していません。

(3) double 型と long double 型

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

(i) 正規化数

符号部は0(正)または1(負)で、値の符号を示します。
指数部は、1～2046(2^{11}-2)の値をとります。実際の指数は、この値から1023を引いた値で、その範囲は-1022～1023です。
仮数部は、0～2^{52}-1の値となります。実際の仮数は、2^{52}のビットを1と仮定し、その直後に小数点があるものとして解釈します。
正規化数の表現する値は、

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

となります。

例：

63	62　52	51　0
0	01111111111	1110000000000000000000000000000000000000000000000000

符号：　＋

指数：　1111111111(2)－1023＝0

仮数：　1.111(2)＝1.875　　(2)は2進数を意味します。

値　：　1.875×2^0＝1.875

(ii)　非正規化数

符号部は0(正)または1(負)で、値の符号を示します。

指数部は0で、実際の指数は-1022になります。

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

非正規化数が表現する値は、

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

となります。

例：

63	62　52	51　0
1	00000000000	1110000000000000000000000000000000000000000000000000

符号：　－

指数：　－1022

仮数：　0.111(2)＝0.875　　(2)は2進数を意味します。

値　：　0.875×2^{-1022}

(iii)　ゼロ

符号部は0(正)または1(負)で、それぞれ+0.0、-0.0を示します。

指数部、仮数部はともに0です。

+0.0、-0.0は、ともに値としては0.0を示します。ゼロの符号による、各演算での機能の違いについては「**10.1.3(4) 浮動小数点演算の仕様**」を参照してください。

(iv)　無限大

符号部は0(正)または1(負)で、それぞれ+∞、-∞を示します。

指数部は2047(2^{11}-1)です。

仮数部は0です。

(v)　非数

指数部は2047(2^{11}-1)です。

仮数部は0以外の値です。

【注】　マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合、仮数部の最上位ビットが 0 の非数を qNaN、仮数部の最上位ビットが 1 の非数を sNaN と呼びます。
その他の仮数フィールドの値、および符号部については規定していません。

(4) 浮動小数点演算の仕様

本項では、C/C++言語の機能として表現されている浮動小数点の四則演算、およびコンパイル時やCライブラリ関数の処理で生じる浮動小数点の10進表現と内部表現の間の変換の仕様について解説します。

(a) 四則演算の仕様

(i) 結果の値の丸め方

浮動小数点の四則演算の結果の正確な値が、内部表現の仮数の有効数字を超えた場合は、以下の規則に従って丸めを行います。

[1] マイコンがSH-2Eで、かつ精度が単精度の場合、有効数字を超える部分を切り捨てます。

[2] マイコンがSH2A-FPU、SH-4またはSH-4Aで、かつround=zeroを指定した場合は、有効数字を超える部分を切り捨てます。

[3] それ以外の場合、近似する二つの浮動小数点型の内部表現のうち、近い方に向かって丸めます。
ちょうど中央になる場合は、仮数の最後の桁が0となる方向に丸めます。

(ii) オーバフロー、アンダフロー、無効演算時の処理

実行時のオーバフロー、アンダフロー、無効演算に対しては、以下の処理を行います。

[1] オーバフローの場合は、結果の符号に従って正または負の無限大になります。

[2] アンダフローの場合は、マイコンの指定により以下のようになります。

[2-1] マイコンがSH-2Eの場合、float型の場合は結果の符号に従って正または負のゼロになり、double型またはlong double型の場合は非正規化数となります。

[2-2] マイコンがSH2A-FPUの場合、結果の符号に従って正または負のゼロになります。

[2-3] マイコンがSH-4またはSH-4Aの場合、denormalize=onを指定した場合は非正規化数となり、denormalize=offを指定した場合は結果の符号に従って正または負のゼロになります。

[2-4] マイコンが上記以外の場合、非正規化数となります。

[3] 無効演算は、符号が逆の無限大を加算した場合、符号が同じ無限大を減算した場合、ゼロと無限大を乗算した場合、ゼロをゼロで、あるいは無限大を無限大で除算した場合に生じます。
これらの場合、結果は非数になります。

[4] 浮動小数点型から整数へ変換したときにオーバフローが生じた場合、結果の値は保証しません。

【注】 定数式に関しては、コンパイル時に演算を行います。この時にオーバフロー、アンダフロー、無効演算を検出した場合は、ウォーニングレベルのエラーになります。

(iii) 特殊な値(ゼロ、無限大、非数)の演算に関する注意事項

[1] 正のゼロと負のゼロの和は正のゼロとなります。

[2] 同符号のゼロの差は正のゼロになります。

[3] 被演算子の一方あるいは両方に非数を含む演算の結果は、常に非数になります。

[4] 比較演算においては、正のゼロと負のゼロは等しいものとして扱います。

[5] 被演算子の一方あるいは両方が非数であるような比較演算、等値演算の結果は、「!=」については常に真、その他は常に偽となります。

(b) 10進表現と内部表現の間の変換

本項ではソースプログラム上の浮動小数点定数と内部表現の間の変換、あるいはCライブラリ関数によるASCII文字列による浮動小数点型の10進表現と、内部表現の間の変換の仕様について解説します。

(i)　10進表現から内部表現に変換する場合、まず10進表現を10進表現の正規形に変換します。
10進表現の正規形は、「$\pm M \times 10^{\pm N}$」の形式で、M、Nの範囲は以下のとおりです。

[1]　float型の正規形
$0 \leqq M \leqq 10^9-1$
$0 \leqq N \leqq 99$

[2]　double型とlong double型の正規形
$0 \leqq M \leqq 10^{17}-1$
$0 \leqq N \leqq 999$

正規形に変換できない10進表現については、オーバフロー、またはアンダフローになります。また、10進表現が正規形よりも多くの有効数字を含んでいる場合は、下位の桁は切り捨てます。これらの場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数errnoに設定します。
また、正規形に変換するためには、もとの10進表現のASCII文字列としての長さが511文字以下でなければなりません。そうでない場合、コンパイル時にはエラーになり、実行時には対応するエラーの番号を変数errnoに設定します。
内部表現から10進表現に変換する場合には、一度10進表現の正規形に変換してから、指定した書式に従ってASCII文字列に変換します。

(ii)　10進表現の正規形と内部表現の間の変換
10進表現の正規形と内部表現の間の変換は、指数が大きいときや小さいときには、正確な変換ができません。以下に、正確な変換ができる範囲と、その範囲外の場合での誤差の限界値について解説します。

[1]　正確な変換ができる範囲
以下に示す指数の範囲の浮動小数点型については、**「(a)(i) 結果の値の丸め方」**に示す丸めが正確に行われます。この範囲ではオーバフロー、アンダフローは生じません。
float型の場合：　$0 \leqq M \leqq 10^9-1$、$0 \leqq N \leqq 13$
double型とlong double型の場合：　$0 \leqq M \leqq 10^{17}-1$、$0 \leqq N \leqq 27$

[2]　誤差の限界値
[1]で示す範囲に入っていない値を変換する場合の誤差と、正確な丸めを行ったときの誤差の差は、有効数字の最小位桁の0.47倍を超えません。
　また、[1]で示した範囲を超えている場合、変換の際にオーバフローやアンダフローが生じる場合があります。この場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数 errno に設定します。

10.1.4 演算子の評価順序

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

各演算子の優先順位と結合性を**表 10.19**に示します。

表 10.19 演算子の優先順位と結合性

優先順位	演算子	結合性	適用される式
1	++ -- (後置) ( ) [ ] -> .	左	後置式
2	++ -- (前置) ! ~ + - * & sizeof	右	単項式
3	(型名)	右	キャスト式
4	* / %	左	乗除式
5	+ -	左	加減式
6	<< >>	左	ビット単位のシフト式
7	< <= > >=	左	関係式
8	== !=	左	等価式
9	&	左	ビット単位の AND 式
10	^	左	ビット単位の排他 OR 式
11	|	左	ビット単位の OR 式
12	&&	左	論理 AND 式
13	||	左	論理 OR 式
14	?:	右	条件式
15	= += -= *= /= %= <<= >>= &= |= ^=	右	代入式
16	,	左	カンマ式

10.2　DSP-C 仕様

DSP-C 言語について、コンパイラの仕様を示します。DSP-C は dspc オプションを指定した場合に有効です。

10.2.1　固定小数点型

表 10.20　固定小数点型の内部表現

型	サイズ(メモリ上のサイズ)	アライメント数(byte)	値の範囲		接尾語
			最小値	最大値	
_ _fixed	16bit(16bit)	2	-1.0	$1.0-2^{-15}$ (0.999969482421875)	r
long _ _fixed	32bit(32bit)	4	-1.0	$1.0-2^{-31}$ (0.9999999995343387126922607421875)	R
_ _accum	24bit(32bit) *1	4	-256.0	$256.0-2^{-15}$ (255.999969482421875)	a
long _ _accum	40bit(64bit) *1	4	-256.0	$256.0-2^{-31}$ (255.9999999995343387126922607421875)	A

【注】*1 メモリに格納される場合は右詰めとなり、先頭部分は符号拡張されます。

例：

(i) (_ _accum)128.5a　→ 00 40 40 00

(ii) (long _ _accum)(-256.0A)　→ FF FF FF 80 00 00 00 00

10.2.2 型修飾子

(1) メモリ型修飾子

X/Y メモリへの格納を明示的に指定するために以下の型修飾子を使用します。

_ _X : X メモリにデータを格納
_ _Y : Y メモリにデータを格納

メモリ型修飾子とセクションの対応を表 10.21 に示します。

表 10.21 メモリ型修飾子の仕様

	名称	セクション	内容
1	定数領域	$XC	const 型のデータ(X メモリに格納)
		$YC	const 型のデータ(Y メモリに格納)
2	初期化データ領域	$XD	初期値のあるデータ(X メモリに格納)
		$YD	初期値のあるデータ(Y メモリに格納)
3	未初期化データ領域	$XB	初期値のないデータ(X メモリに格納)
		$YB	初期値のないデータ(Y メモリに格納)

【注】 *1 2 つのメモリ型修飾子を同じ変数に指定することはできません。指定した場合は、エラーを出力します。
*2 メモリ型修飾子をつけた変数は、#pragma section 指定によるセクション切り替えの対象外となります。
*3 メモリ型修飾子を使用して関数本体のメモリ格納先を変更することはできません。
*4 static 指定のない局所変数にメモリ型修飾子を指定した場合、ウォーニングを出力し、メモリ型修飾子指定は無効となります。ただし、メモリ型修飾子つきデータへのポインタとしての指定は有効です。

以下に_ _X/_ _Y 型修飾子使用時のメモリ格納例を示します。

- _ _X int a; // X メモリに格納
- int _ _X b; // X メモリに格納
- _ _Y int *c; // Y メモリ上の int 型へのポインタ(メモリは不定)
- int _ _Y * d; // Y メモリ上の int 型へのポインタ(メモリは不定)
- int *_ _Y e; // int 型へのポインタ(Y メモリに格納)
- _ _X int * _ _Y f; // X メモリ上の int 型へのポインタ(Y メモリに格納)

(2) 飽和型修飾子

飽和演算を指定するために以下の型修飾子を使用します。

_ _sat

_ _sat 型修飾子は_ _fixed 型、long _ _fixed 型データのみに使用することができます。それ以外の型に指定した場合はエラーになります。

式の中に 1 つ以上 _ _sat 指定があれば飽和演算を行ないます。

例 :

```
_ _fixed a;
_ _sat _ _fixed b;
_ _fixed c;

a=-0.75r;
b=-0.75r;
```

```
c=a+b;              // c=-1.0r になります。
```

(3) 循環型修飾子

モジュロアドレッシングを指定するために以下の型修飾子を使用します。

__circ

モジュロアドレッシングは、メモリ型修飾子(__X/__Y)指定のある__fixed 型の 1 次元配列およびポインタのみに指定することができます。それ以外の条件で使用した場合はエラーになります。

モジュロアドレッシングは、組み込み関数 set_circ_x()または set_circ_y()と clr_circ()の間にある 1 次元配列およびポインタが対象となります。組み込み関数の仕様については、「10.3.3 組み込み関数」を参照してください。

複数の配列を同時にモジュロアドレッシング指定をした場合、及び上記組み込み関数の間以外で__circ 指定のある配列またはポインタを参照した場合は動作は保証しません。

負方向へのモジュロアドレッシングを指定した場合は動作は保証しません。

モジュロアドレッシングを行なうデータは、リンク時にアドレスの上位 16 ビットが同じになるように配置する必要があります。

配列内容の直接参照はできません。

【注】以下のいずれかに該当する場合は動作保証しません(ウォーニングを出力する場合もあります)。

- optimize=0 または optimize=debug_only を指定している
- __circ ポインタを局所変数以外に指定している
- __circ ポインタに volatile を指定している
- __circ ポインタの更新のみで参照がない
- 組み込み関数 set_circ_x() / set_circ_y()と clr_circ()の間に関数呼び出しがある

10.2.3 定数

数値に**表 10.20** で示した接尾語を付けることによって、明示的に固定小数点定数であることを表すことができます。ただし、接尾語 r、R をつけた定数は、数値が整数部分を含む場合、それぞれ__accum 型、long__accum 型の定数として扱います。

接尾語を省略した場合は、double 型定数として扱います。fixed_const オプション指定時は、固定小数点定数として扱います。

飽和型の固定小数点定数はありませんが、明示的に型変換することによって飽和型として扱えます。

例：

```
__fixed a;
__fixed b;

a=-0.75r;
b=a + (__sat __fixed)(-0.75r);   // b=-1.0r になります。
```

単項マイナス演算子は固定小数点定数の一部ではないので、-1.0r は__fixed 型として有効な値ではありません。-1.0r は(-0.5r-0.5r)と記述する必要があります(本マニュアル上では、簡略化のため-1.0r と表記して__fixed 型の-1.0 を表しているところがあります)。

小数部分が各型で表現できる精度を超えた場合、もしくは符号を除く整数部分が__accum 型、long__accum 型の最大値の整数部分の値である 255 を超えた場合は、ウォーニングを出力し、小数部分は精度を超えた部分を切り捨て、整数部分は溢れた部分の最下位 bit を符号として扱い、その他の溢れた部分は切り捨てます。

10.2.4 型変換

型変換の規則について、表 10.22 に示します。

表 10.22 型変換の規則

変換	仕様
_ _fixed → long _ _fixed _ _accum → long _ _accum	下位16bitゼロで初期化。 値は変わらない。
long _ _fixed → _ _fixed long _ _accum → _ _accum	下位16bit切り捨て。 小数の精度が落ちる。
_ _fixed → _ _accum long _ _fixed → long _ _accum	上位8bit符号拡張。 値は変わらない。
_ _fixed → long _ _accum	上位8bit符号拡張。下位16bitゼロで初期化。 値は変わらない。
long _ _fixed → _ _accum	上位8bit符号拡張。下位16bitは切り捨て。 小数の精度が落ちる。
_ _accum → _ _fixed long _ _accum → long _ _fixed	上位8bit切り捨て。9bit目を符号bitとする。 整数部分が0の時は値は不変。
_ _accum → long _ _fixed	上位8bit切り捨て。9bit目を符号bitとする。 整数部分が0の時は値は不変。
long _ _accum → _ _fixed	上位8bit切り捨て。下位16bit切り捨て。 9bit目を符号bitとする。整数部分が0の時は値は不変。 小数の精度が落ちる。
_ _fixed → 符号付き整数型 long _ _fixed → 符号付き整数型	-1.0r、-1.0Rの場合は-1、それ以外の場合は0。
_ _accum → 符号付き整数型 long _ _accum → 符号付き整数型	小数部切り捨て。 変換後の値は-256～255の間の整数。
_ _fixed → 符号なし整数型 long _ _fixed → 符号なし整数型	-1.0r、-1.0Rの場合は変換後の型の最大値、 それ以外の場合は0。
_ _accum → 符号なし整数型 long _ _accum → 符号なし整数型	小数部切り捨て。 正の値の場合、変換後の値は0～255の整数。 負の値の場合、(変換前の値+1+変換後の型の最大値)。
符号付き整数型 → _ _fixed 符号付き整数型 → long _ _fixed	変換前の最上位bitを変換後の最上位bitビットにする。 その他のbitは全て0。
符号付き整数型 → _ _accum 符号付き整数型 → long _ _accum	値の下位9bitを整数部とする。 小数部は0とする。
符号なし整数型 → _ _fixed 符号なし整数型 → long _ _fixed	変換後の全てのbitを0とする。
符号なし整数型 → _ _accum 符号なし整数型 → long _ _accum	値の下位9bitを整数部とする。 小数部は0とする。
固定小数点 → 浮動小数点	変換後の型で表現可能な値は、元の値と同じ。 表現できない場合は、もっとも近い値に丸める。

変換	仕様
浮動小数点 → 固定小数点	小数部分は固定小数点 → 浮動小数点変換と同じ。 整数部分は浮動小数点 → 整数変換と同じ。 整数部分が固定小数点の表現可能範囲内の場合は、そのままの値を保つようにする。範囲外の場合は溢れた部分の最下位bitを符号bitとする。 また変換後の型に飽和処理指定があっても飽和処理は行わない。

10.2.5　算術変換

演算の 2 つのオペランドの型が異なる場合は、図 10.2で上側にある方の型に合わせて演算します。

また図 10.2で上下の間の関係にない型(整数型 ⇔ 固定小数点型、_ _accum ⇔ long _ _fixed)間の演算を記述した場合はエラーになります。演算が必要な場合は、明示的にキャストを行って型を合わせる必要があります。

ただし結果の値が保証される範囲では、効率などの面から上記変換ルールに従わないで演算する場合もあります。

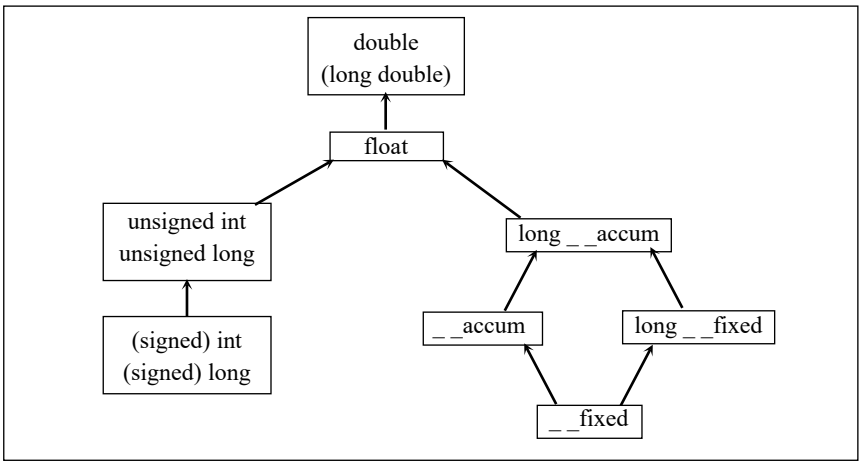


図 10.2　算術変換の規則

10.2.6 ポインタ変換

(a) __circ 型修飾子

__circ 指定ありのポインタを__circ 指定なしの型に変換した場合、ウォーニングを出力します。モジュロアドレッシングは使用できません。

__circ 指定なしのポインタに__circ 指定した場合はウォーニングを出力しません。ただしモジュロアドレッシングは行いません。

(b) メモリ型修飾子(__X/__Y)

別のメモリ型修飾子へ変換した場合はエラーになります。

10.2.7 演算子

以下の演算子は固定小数点型に対しては指定できません。指定した場合はエラーになります。

- 1 の補数演算子(~)
- ビットごとの AND 演算子(&, &=)
- ビットごとの OR 演算子(|, |=)
- ビットごとの XOR 演算子(^, ^=)
- シフト演算子(<<, >>, <<=, >>=)
- 剰余演算子(%, %=)

sizeof 演算子の値は表 10.23 の通りです。

表 10.23 sizeof 演算子の値

	型	値
1	__fixed	2
2	long __fixed	4
3	__accum	4
4	long __accum	8

10.2.8　ライブラリ

(a)　fixed.h

固定小数点型の限界値を<fixed.h>で定義します。詳細については「10.4.1　標準 C ライブラリ」を参照してください。

(b)　stdio.h

固定小数点用の変換指定子を表 10.24 に示します。

表 10.24　固定小数点変換指定子

	変換指定子	意味
1	%r	_ _fixed値
2	%lr	long _ _fixed値
3	%a	_ _accum値
4	%la	long _ _accum値
5	%P	_ _circポインタ値

固定小数点の変換に関しては%f 変換(浮動小数点変換)に準拠します。%P は%p(ポインタ変換)と同じ変換をします。

(c)　stdlib.h

固定小数点用の関数を表 10.25 に示します。詳細については「10.4.1　標準 C ライブラリ」を参照してください。

表 10.25 関数一覧

	種類	関数名
1	文字列数値変換関数	long __fixed atolfixed(const char * nptr);
		long __accum atolaccum(const char * nptr);
		long __fixed strtolfixed(const char * nptr, char ** endptr);
		long __accum strtolaccum(const char * nptr, char ** endptr);
2	記憶域管理関数	void __X *calloc__X(size_t nelem, size_t elsize);
		void free__X(void __X *ptr);
		void __X *malloc__X(size_t size);
		void __X *realloc__X(void __X *ptr, size_t size);
		void __Y *calloc__Y(size_t nelem, size_t elsize);
		void free__Y(void __Y *ptr);
		void __Y *malloc__Y(size_t size);
		void __Y *realloc__Y(void __Y *ptr, size_t size);

【注】 *1 低水準インタフェースルーチンも X/Y メモリ用に用意する必要があります。

char __X *sbrk__X(int size);

char __Y *sbrk__Y(int size);

詳細については「9.2.2(6) 低水準インタフェースルーチン」を参照してください。

(d) string.h

固定小数点用の関数を表 10.26 に示します。詳細については「10.4.1 標準 C ライブラリ」を参照してください。

表 10.26 関数一覧

	種類	関数名
1	記憶域内容複写関数	void __X * memcpy__X__X(void __X * s1, const void __X * s2, size_t n);
		void __X * memcpy__X__Y(void __X * s1, const void __Y * s2, size_t n);
		void __Y * memcpy__Y__X(void __Y * s1, const void __X * s2, size_t n);
		void __Y * memcpy__Y__Y(void __Y * s1, const void __Y * s2, size_t n);

(e) DSP ライブラリ

dspc オプションを指定した場合、short 型配列、ポインタのパラメータに__fixed 型配列、ポインタを指定することができます。詳細については「10.4.5 DSP ライブラリ」を参照してください。

10.3　拡張機能

コンパイラの拡張機能として、以下の機能をサポートしています。

- #pragma
- セクションアドレス演算子
- 組み込み関数

10.3.1　#pragma

#pragma の一覧を示します。

なお、最適化に関わる#pragma は、条件により、適用されない場合もあります。当該最適化が適用されたかどうかは出力コードで確認してください。

表 10.27　メモリ配置に関する拡張機能

	#pragma	機能
1	#pragma section	セクションの切り替え指定
2	#pragma abs16 #pragma abs20 #pragma abs28 #pragma abs32	アドレス領域の指定
3	#pragma stacksize	スタックセクションの作成

表 10.28　関数に関する拡張機能

	#pragma	機能
1	#pragma interrupt	割り込み関数の作成
2	#pragma inline	関数のインライン展開を指定
3	#pragma inline_asm	アセンブリ記述関数のインライン展開
4	#pragma regsave #pragma noregsave #pragma noregalloc	レジスタの退避/回復コード出力の制御
5	#pragma entry	エントリ関数の作成
6	#pragma ifunc	浮動小数点レジスタの退避/回復抑止
7	#pragma tbr	TBR レジスタ相対関数呼び出し指定
8	#pragma align4	分岐先アドレスの 4 バイト整合

表 10.29　その他の拡張機能

	#pragma	機能
1	#pragma global_register	大域変数のレジスタ割り付け
2	#pragma gbr_base #pragma gbr_base1	GBR ベース変数の指定
3	#pragma bit_order	ビットフィールドの並び順指定
4	#pragma pack #pragma unpack	構造体、共用体、クラスのアライメント数指定
5	#pragma address	変数に絶対アドレスを指定

上記拡張機能には、データメンバ、メンバ関数が指定可能な機能があります。指定方法は(クラス名::メンバ名)です。指定可能なメンバの種類は、各機能の記述方法を参照してください。

(1) メモリ配置に関する拡張機能

セクション切り替え指定

#pragma section

書　式　　#pragma section [{<名前>|<数値>}]

説　明　　コンパイラの出力するセクション名を切り替えます。
デフォルト、切り替え後のセクション名は表 10.30 のとおりです。

表 10.30 セクション切り替え機能とセクション名

	対象領域	指定方法	デフォルト名	切り替え後
1	プログラム領域	#pragma section <xx>	P*1	P<xx>
2	定数領域		C*1	C<xx>
3	初期化データ領域		D*1	D<xx>
4	未初期化データ領域		B*1	B<xx>

【注】*1　section オプションでデフォルトセクション名を変更できます。
<名前>や<数値>を省略すると、以降はデフォルトのセクション名になります。

例

```
#pragma section abc
int a;              /* a はセクション Babc に割り付きます      */
const int c=1;      /* c はセクション Cabc に割り付きます      */
void f(void)        /* f はセクション Pabc に割り付きます      */
{
    a=c;
}
#pragma section
int b;              /* b はセクション B に割り付きます         */
void g(void)        /* g はセクション P に割り付きます         */
{
    b=c;
}
```

備　考　　#pragma section は関数定義の外で宣言しなければなりません。
1 ファイルで宣言できるセクション名は最大 2045 個です。
メモリ型修飾子(_ _X/_ _Y)と同時に指定された場合は、#pragma section 指定は無効になります。
C++言語を使用している時は、コンパイル単位の最初の宣言に対して#pragma section の指定が必要となります。

例

```
#pragma section _A
extern int N;
#pragma section

int N;

/* 変数Nは B_A セクションに配置されます */
----------------------------------------
extern int N;

#pragma section _A
int N;
#pragma section

/* 変数Nは B セクション(デフォルト)に配置されます */
----------------------------------------
```

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

例

```
class A
{
   private:
   // 初期値なし
   #pragma section DATA
   static int data_;
   #pragma section

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

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

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

アドレス領域指定

#pragma abs16
#pragma abs20
#pragma abs28
#pragma abs32

書　式

```
#pragma abs16 [(]<識別子>[,...][)]
#pragma abs20 [(]<識別子>[,...][)]
#pragma abs28 [(]<識別子>[,...][)]
#pragma abs32 [(]<識別子>[,...][)]
```

説　明

`#pragma abs16|abs20|abs28|abs32` で宣言した変数または関数を以下のメモリ空間に配置したものとして扱います。これによってプログラムサイズを削減できます。
識別子には、変数、グローバル関数、静的データメンバ及びメンバ関数を指定できます。

表 10.31 アドレス範囲

#pragma 拡張子	アドレス範囲	
	下位	上位
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*1
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

【注】*1　0x07FFFF7F となることに注意。

備　考

`#pragma abs16|abs20|abs28|abs32` は、自動オブジェクトや非静的データメンバを指定できません。
`#pragma abs16|abs20|abs28|abs32` で宣言された変数および関数は、必ず指定アドレス範囲内に配置してください。
同じ識別子に複数のアドレス範囲を指定できません。
`abs16|abs20|abs28|abs32` オプションと同時に指定された場合は、
`#pragma abs16|abs20|abs28|abs32` が有効になります。
`#pragma gbr_base|gbr_base1` と同時に指定された場合は、
`#pragma abs16|abs20|abs28|abs32` は無効になります。

スタックセクション作成

#pragma stacksize

書　式　　#pragma stacksize <定数>

説　明　　セクション名 S、サイズ<定数>のスタックセクションを作成します。

例　　#pragma stacksize 100

<コード展開例>

```
.SECTION S,STACK,ALIGN=4
.RES.B    100
```

備　考　　サイズ<定数>は必ず 4 の倍数を指定してください。
#pragma stacksize はファイル内で一回しか指定できません。

(2) 関数に関する拡張機能

割り込み関数作成

#pragma interrupt

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

説　明　`#pragma interrupt` を用いて割り込み関数となる関数を宣言します。
関数名には、グローバル関数および静的メンバ関数を指定できます。
割り込み仕様の一覧を表 10.32 に示します。

表 10.32 割り込み仕様の一覧

	項目	形式	オプション	指定内容
1	スタック切り替え指定	sp=	{ <変数> | &<変数> | <定数> | <変数>+<定数> | &<変数>+<定数> }	新しいスタックのアドレスを変数または定数で指定 <変数> ：変数(ポインタ型) &<変数> ：変数(オブジェクト型)のアドレス <定数> ：定数値
2	トラップ命令リターン指定	tn=	<定数>	終了を TRAPA 命令で指定 定数値(トラップベクタ番号)
3	レジスタバンク指定	resbank	なし	レジスタ R0～R14、GBR、MACH、MACL、および PR の退避コードの出力抑止 tn 指定がない場合は、RTE 命令の直前に RESBANK 命令を生成
4	レジスタバンク切り替え RTS 命令リターン指定	sr_rts	なし	RTS 命令で終了する 関数内で使用したレジスタのみ退避コード生成 関数の最後に SR レジスタの RB ビット、BL ビットをセット
5	割り込みハンドリング関数指定	bank	なし	sr_jsr()組込み関数が存在する場合、SSR/SPC レジスタの退避コード生成 R0～R7 レジスタの退避コード出力抑止 その他関数内で使用したレジスタは退避コード生成
6	RTS 命令リターン指定	rts	なし	RTS 命令で終了する SSR/SPC、R0～R7 レジスタの退避コード出力抑止 その他関数内で使用したレジスタは退避コード生成

#pragma interrupt を用いて宣言した関数は、関数の処理の前後で全レジスタを保証(関数入口/出口において関数内で使用する全レジスタを退避・回復)し、通常 RTE 命令でリターンし、トラップ命令リターンを指定した場合は TRAPA 命令でリターンします。割り込み仕様を指定しない場合は単純な割り込み関数として処理します。また、スタック切り換え指定とトラップ命令リターン指定は重複して指定できます。

例

```
extern int STK[100];
#pragma interrupt(f(sp=STK+400, tn=10),A::g)
class A{
public:
  static void g();
};
```

説明

(a) スタック切り替え指定
STK+400 を割り込み関数「f」で使用するスタックポインタとして設定します。

(b) トラップ命令リターン指定
割り込み関数終了時に TRAPA #10 でトラップ例外処理を開始します。トラップ例外処理開始時の SP は図 10.3 のようになっています。トラップルーチンの側で RTE 命令を使用して PC(プログラムカウンタ)、SR(ステータスレジスタ)を回復し、割り込み関数から復帰してください。

(c) C++プログラムで指定可能なメンバ関数は、静的メンバ関数です。例では、クラス A の静的メンバ関数 g を割り込み関数として指定しています。非静的メンバ関数は、指定できないので注意してください。

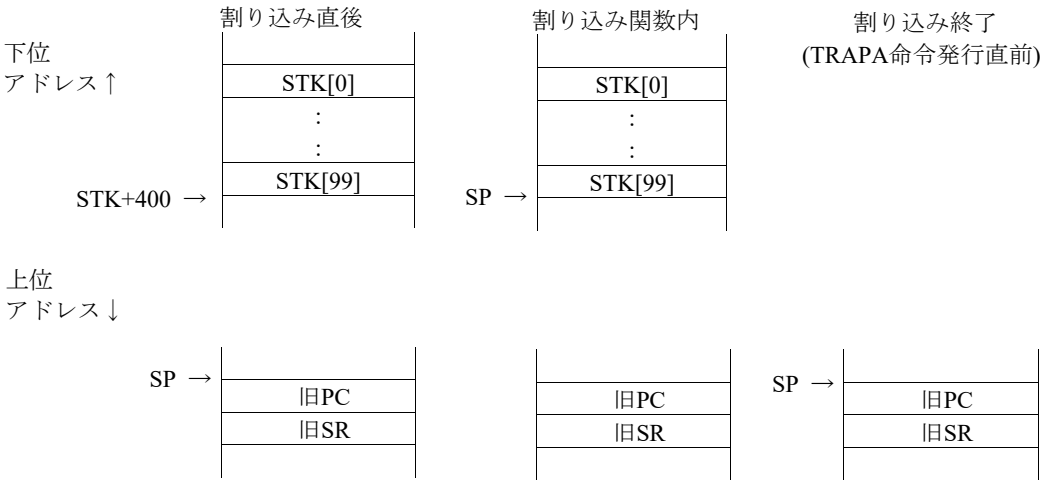


図 10.3 割り込み関数によるスタック使用例

sr_rts/bank 指定および組み込み関数 sr_jsr() を使用することにより、多重割り込み関数を作成することができます。

例

```
#include <machine.h>

// ハンドリング関数宣言
#pragma interrupt (func(bank))
void func();

// 割り込み処理関数宣言
#pragma interrupt (sub(sr_rts))
void sub();
void func() {
      :
    sr_jsr(sub,8);   // RB=0、BL=0、
                     // 割り込みレベルを 8 にして
                     // sub()を呼び出す
      :
}

void sub() {
      :
}
```

```
func:
                    ;
    MOV.L       R14,@-R15
    STS.L       PR,@-R15
    STC         SSR,@-R15
    STC         SPC,@-R15
          :
    STC         SR,R6
    MOV.L       L12+6,R1    ; H'CFFFFF0F
    MOV         #-128,R4    ; H'FFFFFF80
    EXTU.B      R4,R4
    MOV.L       L12+10,R14 ; _sub
    AND         R1,R6
    OR          R4,R6
    LDC         R6,SR
    JSR         @R14

    NOP
          :
    LDC         @R15+,SPC
    LDC         @R15+,SSR
    LDS.L       @R15+,PR
    MOV.L       @R15+,R14
    RTE
    NOP

_sub:

    MOV.L       R0,@-R15
    MOV.L       R1,@-R15
          :
    STC         SR,R0
    MOV.L       L12+2,R1    ; H'30000000
    OR          R1,R0
    MOV.L       @R15+,R1
    LDC         R0,SR
    RTS
    LDC.L       @R15+,R0_BANK
```

RB=1, BL=1

関数中で使用したR0～R7
以外のレジスタと
SPC/SSRの退避

RB=0,BL=0
IMASK=8に変更

sub()関数呼び出し
RB=1,BL=1に変更されて戻ってくる

関数中で使用したR0～R7以外の
レジスタとSPC/SSRの回復
IMASK=8に変更

RB=0, BL=0

関数内で使用した
レジスタの退避

RB=1,BL=1に変更し、
関数内で使用した
レジスタの回復

RB=1に変更後のため

rts/bank 指定を使用することにより、レジスタバンクを活用した効率の良い割り込み関数を作成することができます。

例

```
#include <machine.h>

// ハンドリング関数宣言
#pragma interrupt (func(bank))
void func();

// 割り込み処理関数宣言
#pragma interrupt (sub(rts)
void sub();
void func() {
                :
    sub();
                :
}

void sub() {
                :
}


_func:
                ←――――――――――――――――――― RB=1, BL=1
                ;
    STS.L       PR,@-R15                 } 関数中で使用したR0～R7以外のレジスタの退避
          :
    MOV.L       L12,R14; _sub
    JSR         @R14
    NOP
          :
    LDS.L       @R15+,PR                 } 関数中で使用したR0～R7以外のレジスタの回復
    RTE
    NOP

_sub:

    MOV.L       R14,@-R15                } 関数中で使用したR0～R7以外のレジスタの退避
    MOV.L       R13,@-R15
          :
    MOV.L       @R15+,R13                } 関数中で使用したR0～R7以外のレジスタの回復
    RTS
    MOV.L       @R15+,R14
```

備　考　resbank 指定は cpu=sh2a|sh2afpu 指定時のみ有効です。
resbank 指定関数の割り込み発生前にレジスタバンク使用を許可してください。
resbank、tn とも指定した場合は、レジスタ退避コード、RESBANK 命令とも生成しません。この場合は、トラップルーチン側で RESBANK 命令を生成するようにしてください。
resbank 指定を行った関数から回復する時は、#pragma global_register で指定した変数の値は、割り込み処理中に書き換えた場合も割り込み前の値になります。
SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP では割り込み時の動作が SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP の場合と異なるので、割り込みハンドラが必要になります。
また、SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP で#pragma interrupt と#pragma noregsave に同じ関数を指定した場合、その関数内で使用した関数呼び出し前後で保証するレジスタのみ退避・回復を行います。
repeat オプションを指定した場合は、RS/RE レジスタの退避・回復を行います。
dspc オプションを指定した場合は、DSP レジスタ(X0、X1、Y0、Y1、M0、M1、A0、A0G、A1、A1G)、DSR レジスタ、MOD レジスタの退避・回復を行います。
割り込み関数の定義に対して指定できる関数は、グローバル関数(C/C++プログラム)と静的メンバ関数(C++プログラム)です。
また、関数の返却値の型は void のみです。return 文のリターン値を指定することはできません。指定があった場合はエラーを出力します。

例

```
#pragma interrupt (f1(sp=100), f2)
void f1(){…}..............(a)
int f2(){…}...............(b)
```

説明

(a)は正しい宣言になります。
(b)は関数の返却値の型が void ではないので誤った宣言です。エラーメッセージを出力します。

sr_rts/bank/rts 指定は cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp 指定時のみ有効です。
sr_rts/bank/rts 指定と他の割り込み仕様との同時指定が有効な組み合わせは以下の通りとなります。

	#pragma interrupt					#pragma noregsave
	sp	tn	sr_rts	bank	rts	
sr_rts	エラー	エラー	エラー	エラー	エラー	エラー
bank	有効	エラー	エラー	エラー	エラー	有効
rts	エラー	エラー	エラー	エラー	エラー	エラー

以下のいずれかに該当する場合、エラーとなります。
- sr_rts 指定した関数を組み込み関数 sr_jsr()以外から呼び出した場合
- bank 指定した関数を呼び出した場合
- rts 指定した関数を bank 指定も rts 指定もしていない関数から呼び出した場合

割り込み関数として宣言した関数をプログラムの中で呼び出すことはできません。呼び出しがあった場合はエラーメッセージを出力します。ただし、割り込み関数として定義した関数を割り込み関数の宣言のないプログラム内で呼び出した場合は、エラーメッセージは出力しません。この場合、実行時の動作は保証しません。

例 1

割り込み関数宣言のある場合

```
#pragma interrupt (f1)
void f1(){…}
int f2(){f1();}.......................(a)
```

説明

関数「f1」は割り込み関数として宣言しているのでプログラム中で呼び出すことはできません。(a)に対してエラーメッセージを出力します。

例 2

割り込み関数宣言がない場合

```
int f1();
int f2(){f1();}......................(b)
```

説明

関数「f1」は割り込み関数としての宣言がないので非割り込み関数 int f1();としてオブジェクトを生成します。関数「f1」が別コンパイル単位で割り込み関数として宣言された場合、実行時の動作は保証しません。

注　意　浮動小数点型精度モードを持つマイコン（SH2A-FPU | SH4 | SH4A）において、fpu オプション未指定時、あるいは fpu=single 指定時に、割り込み関数内で単精度浮動小数点演算を行う時は精度モードの設定が必要になる場合があります。詳細は「9.4.1(6) **マイコン種別が SH2A-FPU|SH4|SH4A の場合の割り込み関数について**」を参照してください。

関数インライン展開

#pragma inline

書　式　#pragma inline [(]<関数名>[,...][)]

説　明　#pragma inline を用いて、インライン展開する関数を宣言します。
関数名には、グローバル関数および静的メンバ関数を指定できます。
#pragma inline で指定した関数名の関数と関数指定子 inline(C++言語)を指定した関数は、その関数を呼び出したところにインライン展開されます。

例　ソースプログラム

```
#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;
}
```

備　考　以下の場合はインライン展開しません。
－#pragma inline 指定より前に関数の定義がある。
－可変パラメータを持つ関数である。
－関数内でパラメータのアドレスを参照している。
－展開対象関数のアドレスを介して呼び出しを行っている。
－再帰呼び出しを行っている。
#pragma inline は、関数本体の定義の前に指定してください。
#pragma inline で指定した関数に対しても外部定義を生成します。各プログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に static を指定してください。static を指定した場合は、外部定義を生成しません。inline(C++言語)指定された関数は、外部定義を生成しません。
scope オプションを指定した場合、インライン展開されないことがあります。

アセンブラ記述関数インライン展開

#pragma inline_asm

書　式　　#pragma inline_asm [(]<関数名>[(size=数値)] [,...][)]

説　明　　#pragma inline_asm で宣言したアセンブリ記述関数をインライン展開します。
関数名には、グローバル関数のみ指定できます。メンバ関数は指定できません。
アセンブラ埋め込みインライン関数のパラメータは、通常の関数呼び出しと同様にレジスタ、あるいはスタックに設定されますので、inline_asm 関数から参照することができます。アセンブラ埋め込みインライン関数のリターン値は R0、SH-2E、SH2A-FPU、SH-4、SH-4A のとき単精度浮動小数点型のリターン値は FR0、SH2A-FPU、SH-4、SH-4A のとき倍精度浮動小数点型のリターン値は DR0 に設定してください。オプションとの組み合わせによりリターン値を設定するレジスタは異なります。詳細は**表 9.7** を参照してください。
(size=数値)指定で、アセンブラ埋め込みインライン関数のサイズが指定できます。

例　　ソースプログラム

```
#pragma inline_asm(sub)
static int sub(int a)
{
  ROTL  R4
  MOV   R4,R0
}
int x;
main()
{
  x=0x55555555;
  x=sub(x);
}
```

出力結果(一部)

```
          :
_main                                   ;function main
                                        ;frame size = 4
         MOV.L    R14,@-R15
         MOV.L    L220+2,R14            ;x
         MOV.L    L220+6,R3             ;H'55555555
         MOV.L    R3,@R14
         MOV      R3,R4
         BRA      L219
         NOP
L220:
         .RES.W   1
         .DATA.L  x
         .DATA.L  H’55555555
L219:
         ROTL     R4
         MOV      R4,R0
         .ALIGN   4
         MOV.L    R0,@R14
         RTS
         MOV.L    @R15+,R14
         .SECTION B,DATA,ALIGN=4
```

```
_x:                                     ;static: x
        .RES.L   1
        .END
```

備　考　#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ラベルを使用する場合、必ずローカルラベルを使用してください。
アセンブラ埋め込みインライン関数内で関数の出入口で保証するレジスタ(**表 9.5** 参照)を使用する場合は、アセンブラ埋め込みインライン関数の先頭と最後でこれらのレジスタの退避・回復が必要です。また、FR12 から FR15(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合)、DR12 から DR14(マイコンが SH2A-FPU、SH-4、SH-4A の場合)のレジスタを使用する場合もアセンブラ埋め込みインライン関数の先頭と最後でこれらのレジスタの退避・回復が必要です。
アセンブラ埋め込みインライン関数の最後に RTS を記述しないでください。
本機能を使用する際は、code=asmcode を指定してコンパイルしてください。
(size=数値)で指定する数値は、実際のオブジェクトサイズ以上の値を指定してください。オブジェクトサイズより小さい値を指定した場合、動作は保証しません。また、数値が浮動小数点型または 0 以下の数値の場合、エラーとなります。
#pragma global_register 機能で指定したレジスタを本関数内で使用する場合も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退避・回復が必要です。
関数名にメンバ関数を指定することはできません。
リテラルプールを生成するような記述は使用しないでください(MOV.L #100000,R0 等)。

レジスタ退避/回復コード制御機能

#pragma regsave
#pragma noregsave
#pragma noregalloc

書　式

```
#pragma regsave [(]<関数名>[,...][)]
#pragma noregsave [(]<関数名>[,...][)]
#pragma noregalloc [(]<関数名>[,...][)]
```

説　明

関数名には、グローバル関数及びメンバ関数を指定できます。

#pragma regsave で指定された関数は、関数の出入口で保証するレジスタ(**表 9.5 参照**)の退避・回復を行います。また関数呼び出しを越えて R8-R14(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合はさらに FR12-FR15)を割り付けないオブジェクトを生成します。

#pragma noregsave で指定された関数は、関数の出入口で保証するレジスタの退避・回復を行いません。

#pragma noregalloc で指定された関数は、関数の出入口で保証するレジスタの退避・回復を行いません。また、関数呼び出しを越えて R8-R14(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合はさらに FR12-FR15)を割り付けないオブジェクトを生成します。

#pragma regsave と#pragma noregalloc は同一関数に対して重複指定できます。このとき、関数の出入口で保証するレジスタ R8-R14(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合はさらに FR12-FR15)を全て退避・回復し、関数呼び出しを越えて、レジスタ R8-R14(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合はさらに FR12-FR15)を割り付けないオブジェクトを生成します。

#pragma noregsave が指定された関数は、以下の条件で使用することができます。

－他の関数から呼び出されることなく、最初に起動する関数として使用する。
－#pragma regsave を指定した関数から呼び出す。
－#pragma regsave を指定した関数から、さらに#pragma noregalloc を介して呼び出す。

例

```
#pragma noregsave(f,A::j)
#pragma noregalloc(g)
#pragma regsave(h)
class A{
public:
  static void j();
};
void f();
void g();
void h();
void h() {
  g();
  f();  /* #pragma regsave 関数(h)から#pragma noregsave 関数(f)の直後 */
        /* の呼び出し                                                  */
}
void g() {
  f();  /* #pragma regsave 関数(h)から#pragma noregsave 関数(f,A::j) */
        /* の#pragma noregalloc 関数(g)を介した呼び出し               */
  A::j();
}
void f(){}
```

備　考

上記以外の方法で#pragma noregsave を指定した関数を呼び出した場合の結果は保証しませんので注意が必要です。

エントリ関数作成

#pragma entry

書　式　#pragma entry [(]<関数名>[(sp=<定数>)][)]

説　明　<関数名>で指定した関数をエントリ関数として扱います。
エントリ関数では、レジスタの退避・回復コードを一切作成しません。
マイコンが SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP の場合では、sp=<定数>指定、もしくは#pragma stacksize 宣言があると、関数先頭でスタックポインタの初期設定コードを出力します。

例 1

```
#pragma entry INIT(sp=0x10000)
void INIT() {
        :
}
<コード展開例>
        .SECTION P, CODE
_INIT:
        MOV.L    L1, R15
                :
L1:     .DATA.L  H'00010000
                :
```

例 2

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

<コード展開例>
        .SECTION S, STACK
        .RES.B   100
        .SECTION P, CODE
_INIT:
        MOV.L    L1, R15
                :
L1:     .DATA.L  STARTOF S + SIZEOF S
                :
```

備　考　#pragma entry 指定は、関数の宣言前に行ってください。
エントリ関数はロードモジュール全体で 1 つまでしか指定できません。
<定数>は必ず 4 の倍数を指定してください。
cpu=sh1|sh2|sh2e｜sh2a|sh2afpu|sh2dsp を指定した場合、sp=<定数>指定は無効になります。

浮動小数点レジスタ退避/回復抑止

#pragma ifunc

書　式　　#pragma ifunc [(]<関数名>[)]

説　明　　<関数名>で指定した関数での浮動小数点レジスタの退避/回復を抑止します。

備　考　　#pragma ifunc 指定は、関数の宣言前に行ってください。
#pragma ifunc 指定は cpu=sh2e|sh2afpu|sh4|sh4a を指定した場合のみ有効です。
#pragma ifunc 指定された関数内で浮動小数点を使用した場合はエラーとなります。

例

```
float f;
#pragma ifunc(func)
void func(void) {
  f=0.0f;      /* エラー */
}
```

TBR レジスタ相対

#pragma tbr

書　式　　#pragma tbr [(]<関数名>[({sn=<セクション名>|ov=<オフセット値>})][,...][)]

説　明　　#pragma tbr で TBR レジスタ相対で呼び出す関数を宣言します。
仕様の詳細は以下のとおりです。

(a) #pragma tbr <関数名>

<関数名>の関数呼び出しを行った場合、関数呼び出しを TBR レジスタ相対で行います。
また<関数名>の定義が存在する場合、$TBR セクションに func のアドレスを出力します。

例

```
#pragma tbr func
void func() {}
#pragma section AA
void func2() {
    func();
}
```

<コード展開例>

```
_func:
    RTS/N

_func2:
    STS.L  PR,@-R15
    JSR/N  @@($_func-(START OF $TBR),TBR)
    LDS.L  @R15+,PR
    RTS/N

    .SECTION $TBR,DATA,ALIGN=4
$_func:
    .DATA.L _func
```

(b) #pragma tbr <関数名>(sn=<セクション名>)

<関数名>の関数呼び出しを行った場合、関数呼び出しを TBR レジスタ相対で行います。

また<関数名>の定義が存在する場合、$TBR<セクション名>セクションに func のアドレスを出力します。

例

```
#pragma tbr func(sn=_A)
void func() {}
#pragma section AA
void func2() {
    func();
}

<コード展開例>
_func:
    RTS/N

_func2:
    STS.L  PR,@-R15
    JSR/N  @@($_func-(START OF $TBR_A),TBR)
    LDS.L  @R15+,PR
    RTS/N

    .SECTION $TBR_A,DATA,ALIGN=4
$_func:
    .DATA.L _func
```

(c) #pragma tbr <関数名>(ov=<オフセット値>)

<関数名>の関数呼び出しを行った場合、関数呼び出しを TBR レジスタ相対で行います。

disp 値は<オフセット値>になります。

<オフセット値>は 0～1020 の範囲の 4 の倍数を指定してください。

TBR アドレステーブルはコンパイラでは作成しません。

例

```
#pragma tbr func(ov=32)
void func() {}
#pragma section AA
void func2() {
    func();
}

<コード展開例>
_func:
    RTS/N

_func2:
    STS.L  PR,@-R15
    JSR/N  @@(32,TBR)
    LDS.L  @R15+,PR
    RTS/N
```

備　考　　#pragma tbr は cpu=sh2a|sh2afpu 指定時のみ有効です。
tbr オプションと同時に指定した場合は、#pragma tbr 指定が有効になります。
pic=1 を指定した場合、#pragma tbr 指定は無効になります。
同じ関数に sn と ov を同時に指定した場合、最初に指定したものが有効になります。
関数呼び出し前に組み込み関数 set_tbr() で TBR に当該セクションの先頭アドレスを設定してください。
#pragma tbr 指定できる関数は各セクションにつき 255 個までです。

分岐先アドレスの 4 バイト整合

#pragma align4

書　式　　#pragma align4 [(]<関数名>=<種別>[,...][)]

説　明　　<関数名>で指定した関数の分岐先アドレスを 4 バイト整合します。
種別の一覧を表 10.33 に示します。

表 10.33　種別の一覧

項目	種別	指定内容
1　すべて	all	指定した関数内のすべての分岐先アドレスを 4 バイト整合します
2　全ループ先頭	loop	指定した関数内のすべてのループの先頭アドレスを 4 バイト整合します
3　最内側ループ先頭	inmostloop	指定した関数内の最内側ループの先頭アドレスを 4 バイト整合します

備　考　　#pragma align4 を指定した関数の先頭アドレスは常に 4 バイト整合されます。
#pragma align4 を指定した関数は、リンク時最適化の対象外になります。
align4 オプションと#pragma align4 を同時に指定した場合は、#pragma align4 で指定した種別が有効になります。
align16 オプションまたは align32 オプションと#pragma align4 を同時に指定した場合は、関数内の分岐先アドレスを 4 バイト整合します。また、関数先頭アドレスについては、直前に#pragma align4 指定された関数があれば 4 バイト整合、そうでなければ 16 バイトまたは 32 バイト整合します。

(3) その他の拡張機能

大域変数レジスタ割り付け

#pragma global_register

書　式　#pragma global_register [(]<変数名>=<レジスタ名>[,…][)]

説　明　<変数名>で指定した変数に、<レジスタ名>で指定したレジスタを割り付けます。
変数名には、大域変数及び静的データメンバを指定できます。

例
```
#pragma global_register(a=R8,A::b=R9)
class A{
   public:
   static int b;
};
int a;
void g()
{
   a=A::b;
}
```

備　考　大域変数で、単純型またはポインタ型の変数に使用できます。(unsigned)long long 型の変数は指定できません。また、double=float を指定した場合を除き、double 型の変数は指定できません(マイコンが SH2A-FPU、SH-4、SH-4A の場合を除く)。指定可能なレジスタは、R8-R14、FR12-FR15(マイコンが SH-2E、SH2A-FPU、SH-4、SH-4A の場合)、DR12-DR14 (マイコンが SH2A-FPU、SH-4、SH-4A の場合)です。
初期値の設定はできません。また、アドレスの参照もできません。
指定された変数の、リンク先からの参照は保証しません。
静的データメンバの指定は可能ですが、非静的データメンバの指定は不可能です。

FR12-FR15 に設定可能な変数の型
- SH-2E の場合
  - ・float 型変数
  - ・double 型変数(double=float 指定)
- SH2A-FPU、SH-4、SH-4A の場合
  - ・float 型変数(fpu=double 指定なし)
  - ・double 型変数(fpu=single 指定)

DR12～DR14 に設定可能な変数の型
- SH2A-FPU、SH-4、SH-4A の場合
  - ・float 型変数(fpu=double 指定)
  - ・double 型変数(fpu=single 指定なし)

GBR ベース変数指定

#pragma gbr_base
#pragma gbr_base1

書　式　　#pragma gbr_base [(]変数名[,...][)]
#pragma gbr_base1 [(]変数名[,...][)]

説　明　　変数を GBR レジスタからのオフセットでアクセスすることを指定します。
変数名に、変数及び静的データメンバを指定できます。
#pragma gbr_base で指定した変数は、セクション$G0 に割り付けられます。
#pragma gbr_base1 で指定した変数は、セクション$G1 に割り付けられます。
#pragma gbr_base は、変数が GBR レジスタの指すアドレスからオフセット 0-127 バイトにあることを指定します。
#pragma gbr_base1 は、#pragma gbr_base でアクセス可能でない範囲(GBR レジスタの指すアドレスからオフセット 128 バイト以上)の変数に対して指定できます。GBR レジスタの指すアドレスからのオフセットが、char 型、unsigned char 型の場合は最大 255 バイト、short 型、unsigned short 型の場合は、最大 510 バイト、int 型、unsigned int 型、long 型、unsigned long 型、float 型、double 型の場合は最大 1020 バイトであることを指定します。
コンパイラは、これらの指定に基づき、変数の参照、設定に対して、最適な GBR 相対アドレッシングでオブジェクトプログラムを生成します。また、セクション$G0 内の char 型、unsigned char 型のデータに対して、GBR 間接アドレッシングで最適なビット命令を生成します。

備　考　　セクション$G0 のリンク後のサイズ合計が 128 バイトを超えた場合は動作を保証しません。
また、セクション$G1 内に、上記の#pragma gbr_base1 の制約で各型名に示した以上のオフセットを持つデータがある場合、動作を保証しません。
セクション$G1 は、リンク時にセクション$G0 の 128 バイト後に必ず配置してください。
本機能を使用する場合は、プログラム実行開始時に、GBR レジスタにセクション$G0 の先頭を設定してください。
静的データメンバは指定可能ですが、非静的データメンバは指定できません。
gbr=auto を指定した場合、#pragma gbr_base、#pragma gbr_base1 指定は無効になります。

ビットフィールド並び順指定

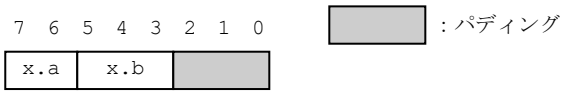
#pragma bit_order

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

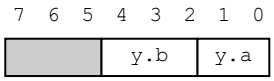
説　明　　ビットフィールドの並び順の切り替えを指定します。
`left` を指定した場合は上位ビット側から、`right` を指定した場合は下位ビット側から、それぞれメンバが割り付けられます。
デフォルトの設定は `left` です。
`left|right` を省略すると、以降はオプションに従います。

例

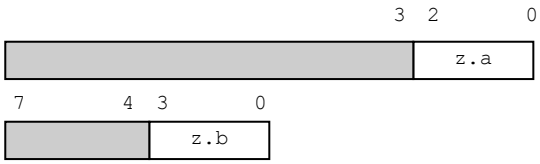
```
#pragma bit_order left
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}x;
```



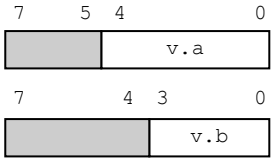

```
#pragma bit_order right
typedef struct{
  unsigned char a:2;
  unsigned char b:3;
}y;
```




```
// 異なるサイズのメンバの場合
#pragma bit_order right
typedef struct{
  unsigned short a:2;
  unsigned char  b:3;
}z;
```




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



備　考　　並び順の切り替えをしない限り、指定したビットフィールドの並び順は有効です。
ビットフィールドの詳細については「10.1.2(3) **ビットフィールド**」を参照してください。

構造体、共用体、クラスのアライメント数指定

#pragma pack
#pragma unpack

書　式

```
#pragma pack {1|4}
#pragma unpack
```

説　明

ソースプログラム中の指定位置以降の構造体、共用体、クラスメンバのアライメント数を指定します。
本拡張子が指定されていない場合または#pragma unpack 指定位置以降で宣言された構造体、共用体、クラスメンバのアライメント数はpack オプションの指定に従います。#pragma pack 拡張子とアライメント数の関係を表 10.33 に示します。

表 10.33　#pragma pack とメンバのアライメント数

拡張子／メンバの型	#pragma pack 1	#pragma pack 4	#pragma unpack または指定なし
(unsigned)char	1	1	1
(unsigned)short、_ _fixed	1	2	pack オプションに従う
(unsigned)int、(unsigned)long、(unsigned)long long、long _ _fixed、_ _accum、long _ _accum、浮動小数点型、ポインタ型	1	4	pack オプションに従う
アライメント数が 1 の構造体、共用体、クラス	1	1	1
アライメント数が 2 の構造体、共用体、クラス	1	2	pack オプションに従う
アライメント数が 4 の構造体、共用体、クラス	1	4	pack オプションに従う

例

```
#pragma pack 1
struct S1 {
    char a;         /* offset:0 */
    int b;          /* offset:1 */
    char c;         /* offset:5 */
} ST1;
#pragma pack 4
struct S2 {
    char a;         /* offset:0 */
                    /* gap:3 byte */
    int b;          /* offset:4 */
                    /* gap:0 byte */
    char c;         /* offset:8 */
                    /* gap:3 byte */
} ST2;
```

備　考　pack=1 もしくは#pragma pack 1 を指定した構造体、共用体、クラス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(ポインタを使用したメンバ関数内でのアクセスを含みます)。
構造体メンバのアドレスを代入文、関数の実引数、または関数戻り値で使用した場合、ウォーニングを出力します。

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 */
}
```

構造体、共用体、クラスメンバのアライメント数は pack オプションでも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。
ひとつの構造体、共用体、クラスの中に、アライメント数が異なるメンバを記述することはできません。該当した場合、ウォーニングを出力します。

例

```
struct X {
    int m;
} x;
#pragma pack 1
struct S {
    char c;
    struct X a[2];  // アライメントが 4 → ウォーニング出力
};
```

ルネサス統合開発環境で作成された iodefine.h を使用する場合、#pragma もしくはオプションでアライメント数を 1 にすると、I/O レジスタ用構造体のメンバが示すアドレスが正しいアドレスを指しません。iodefine.h の先頭に、#pragma pack 4 を、iodefine.h の最後に、#pragma unpack を記述してください。

なお、以下の場合は、コンパイル時にウォーニングが出力されないことがありますのでご注意ください。

・構造体、共用体、クラスのメンバにポインタを用いてアクセスしている場合
・メンバ関数内でポインタを用いてアクセスしている場合

絶対アドレス指定

#pragma address

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

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

例　　(1) スカラ変数

```
#pragma address A=0x100
int A;
void func() {
    A=0;
}

<コード展開例>
_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    RTS
    MOV.L   R4,@R2

    .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_A:
    .RES.L  1
```

(2) 構造体

```
#pragma address ST=0x100
struct {
    int a;
    int b;
} ST;
void func() {
    ST.b=0;
}

<コード展開例>
_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    RTS
    MOV.L   R4,@(4,R2)

    .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_ST:
    .RES.L  2
```

(3) 複数の変数を連続領域に割り付け

```
#pragma address A=0x100,B=0x104
int A,B;
void func() {
    A=0;
    B=0;
}
```

<コード展開例>

```
_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    MOV.L   R4,@R2
    RTS
    MOV.L   R4,@(4,R2)

    .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_A:
    .RES.L  1
_B:
    .RES.L  1
```

(4) 複数の変数を非連続領域に割り付け

```
#pragma address A=0x100,B=0x108
int A,B;
void func() {
    A=0;
    B=0;
}
```

<コード展開例>

```
_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    MOV.L   R4,@R2
    RTS
    MOV.L   R4,@(8,R2)

    .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_A:
    .RES.L  1

    .SECTION $ADDRESS$B108,DATA,LOCATE=H'108
_B:
    .RES.L  1
```

(5) 異なる属性の変数を連続領域に割り付け

```
#pragma address A=0x100,B=0x104
int A;
const int B=0;
void func() {
    A=0;
}
```

<コード展開例>

```
_func:
    MOV     #1,R2
    SHLL8   R2
    MOV     #0,R4
    RTS
    MOV.L   R4,@R2

    .SECTION $ADDRESS$B100,DATA,LOCATE=H'100
_A:
    .RES.L  1

    .SECTION $ADDRESS$C104,DATA,LOCATE=H'104
_B:
    .DATA.L H'00000000
```

備　考　#pragma address 指定は、変数の宣言前に行ってください。
構造体/共用体のメンバ、もしくは変数以外を指定した場合はエラーとなります。
アライメント数 2 の変数、構造体に奇数アドレスを指定した場合、アライメント数 4 の変数、構造体に 4 の倍数以外のアドレスを指定した場合は、エラーとなります。
#pragma address を同一の変数に対して複数回指定した場合はエラーとなります。
異なる変数に対して同一アドレスを指定した場合、もしくは変数のアドレスが重なった場合はエラーとなります。
同一の変数に対して以下の#pragma を同時に指定した場合はエラーとなります。

```
#pragma section
#pragma abs16/abs20/abs28/abs32
#pragma gbr_base/gbr_base1
#pragma global_register
```

10.3.2　セクションアドレス演算子

セクションアドレス演算子

_ _sectop
_ _secend
_ _secsize

書　式

```
_ _sectop("<セクション名>")
_ _secend("<セクション名>")
_ _secsize("<セクション名>")
```

説　明

_ _sectop で指定した<セクション名>の先頭アドレスを参照します。
_ _secend で指定した<セクション名>の末尾+1 アドレスを参照します。
_ _secsize で指定した<セクション名>のサイズを生成します。

例

```
<_ _sectop, _ _secend>
#include <machine.h>
#pragma section $DSEC
static const struct {
    void *rom_s;   /* 初期化データセクションの ROM 上の先頭アドレス  */
    void *rom_e;   /* 初期化データセクションの ROM 上の最終アドレス  */
    void *ram_s;   /* 初期化データセクションの RAM 上の先頭アドレス  */
} DTBL[]={_ _sectop("D"), _ _secend("D"), _ _sectop("R")};
#pragma section $BSEC
static const struct {
    void *b_s;     /* 未初期化データセクションの先頭アドレス        */
    void *b_e;     /* 未初期化データセクションの最終アドレス        */
} BTBL[]={_ _sectop("B"), _ _secend("B")};
#pragma section
#pragma stacksize 0x100
#pragma entry INIT
void main(void);
void INIT(void)
{
    _INITSCT();
    main();
    sleep();
}

<_ _secsize>
unsigned int size = _ _secsize("NAME");
          ↓
_size:
      .DATA.L    SIZEOF NAME
```

10.3.3 組み込み関数

C/C++言語で記述できない以下の機能を、組み込み関数として提供します。

- ステータスレジスタの設定、参照
- ベクタベースレジスタの設定、参照
- グローバルベースレジスタを利用した I/O 機能
- C/C++言語で使用するレジスタ資源と競合しないシステム命令
- 浮動小数点ユニットを利用したマルチメディア命令、コントロールレジスタの設定、参照

組み込み関数は、通常の関数と同様に関数呼び出し形式で記述します。
組み込み関数の一覧を表 10.34 に示します。

表 10.34 組み込み関数の一覧

	項目	仕様	機能
1	ステータスレジスタ(SR)	void set_cr(int cr)	SR の設定
2		int get_cr(void)	SR の参照
3		void set_imask(int mask)	割り込みマスクの設定
4		int get_imask(void)	割り込みマスクの参照
5	ベクタベースレジスタ(VBR)	void set_vbr(void *base)	VBR の設定
6		void *get_vbr(void)	VBR の参照
7	グローバルベースレジスタ(GBR)	void set_gbr(void *base)	GBR の設定
8		void *get_gbr(void)	GBR の参照
9		unsigned char gbr_read_byte(int offset)	GBR ベースのバイト参照
10		unsigned short gbr_read_word(int offset)	GBR ベースのワード参照
11		unsigned long gbr_read_long(int offset)	GBR ベースのロングワード参照
12		void gbr_write_byte(int offset, unsigned char data)	GBR ベースのバイト設定
13		void gbr_write_word(int offset, unsigned short data)	GBR ベースのワード設定
14		void gbr_write_long(int offset, unsigned long data)	GBR ベースのロングワード設定
15		void gbr_and_byte(int offset, unsigned char mask)	GBR ベースのバイト AND
16		void gbr_or_byte(int offset, unsigned char mask)	GBR ベースのバイト OR
17		void gbr_xor_byte(int offset, unsigned char mask)	GBR ベースのバイト XOR
18		int gbr_tst_byte(int offset, unsigned char mask)	GBR ベースのバイト TEST
19	特殊命令	void sleep (void)	SLEEP 命令
20		int tas (char *addr)	TAS 命令
21		int trapa (int trap_no)	TRAPA 命令
22		int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)	OS システムコール
23		void prefetch (void *p)	PREF 命令
24		void trace(long v)	TRACE 命令
25		void ldtlb(void)	LDTLB 命令
26		void nop(void)	NOP 命令

	項目	仕様	機能
27	64bit 乗算	long dmuls_h(long data1, long data2)	符号付き 64bit 乗算の上位 32bit
28		unsigned long dmuls_l(long data1, long data2)	符号付き 64bit 乗算の下位 32bit
29		unsigned long dmulu_h(unsigned long data1, unsigned long data2)	符号なし 64bit 乗算の上位 32bit
30		unsigned long dmulu_l(unsigned long data1, unsigned long data2)	符号なし 64bit 乗算の下位 32bit
31	データ上位・下位交換	unsigned short swapb(unsigned short data)	SWAP.B 命令
32		unsigned long swapw(unsigned long data)	SWAP.W 命令
33		unsigned long end_cnvl(unsigned long data)	4 バイトデータの上位・下位交換
34	積和演算	int macw(short *ptr1, short *ptr2, unsigned int count) int macwl(short *ptr1, short *ptr2, unsigned int count, unsigned int mask)	MAC.W 命令
35		int macl(int *ptr1, int *ptr2, unsigned int count) int macll (int *ptr1, int *ptr2, unsigned int count, unsigned int mask)	MAC.L 命令
36	浮動小数点ユニット	void set_fpscr(int cr)	FPSCR の設定
37		int get_fpscr(void)	FPSCR の参照
38	単精度浮動小数点ベクタ演算	float fipr(float vect1[4], float vect2[4])	FIPR 命令
39		void ftrv(float vec1[4], float vec2[4])	FTRV 命令
40		void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	4 次元ベクタの 4×4 行列による変換と 4 次元ベクタとの和
41		void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	4 次元ベクタの 4×4 行列による変換と 4 次元ベクタとの差
42		void add4(float vec1[4], float vec2[4], float vec3[4])	4 次元ベクタの和
43		void sub4(float vec1[4], float vec2[4], float vec3[4])	4 次元ベクタの差
44		void mtrx4mul(float mat1[4][4], float mat2[4][4])	4×4 行列の乗算
45		void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	4×4 行列の乗算と和
46		void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	4×4 行列の乗算と差
47	拡張レジスタへのアクセス	void ld_ext(float mat[4][4])	拡張レジスタへのロード
48		void st_ext(float mat[4][4])	拡張レジスタからのストア
49	DSP 命令	long _ _fixed pabs_lf(long _ _fixed data) long _ _accum pabs_la(long _ _accum data)	絶対値
50		_ _fixed pdmsb_lf(long _ _fixed data) _ _fixed pdmsb_la(long _ _accum data)	MSB 検出
51		long _ _fixed psha_lf(long _ _fixed data, int count) long _ _accum psha_la(long _ _accum data, int count)	算術シフト
52		_ _accum rndtoa(long _ _accum data) _ _fixed rndtof(long _ _fixed data)	丸め演算
53		long _ _fixed long_as_lfixed(long data) long lfixed_as_long(long _ _fixed data)	ビットパターンコピー

	項目	仕様	機能
54	DSP 命令	void set_circ_x(_ _X _ _circ _ _fixed array[], size_t size) void set_circ_y(_ _Y _ _circ _ _fixed array[], size_t size)	モジュロアドレッシング設定
55		void clr_circ(void)	モジュロアドレッシング解除
56		void set_cs(unsigned int mode)	CS ビット(DSR レジスタ)設定
57	正弦・余弦	void fsca(long angle, float *sinv, float *cosv)	正弦・余弦の計算
58	平方根の逆数	float fsrra(float data)	平方根の逆数
59	命令キャッシュ無効化	void icbi(void *p)	命令キャッシュブロックの無効化
60	キャッシュブロック操作	void ocbi(void *p) void ocbp(void *p) void ocbwb(void *p)	キャッシュブロックの無効化 キャッシュブロックのパージ キャッシュブロックの書き戻し
61	命令キャッシュプリフェッチ	void prefi(void *p)	命令キャッシュブロックのプリフェッチ
62	システム同期	void synco(void)	データ操作の同期
63	T ビット参照・設定	int movt(void)	T ビットの参照
64		void clrt(void)	T ビットのクリア
65		void sett(void)	T ビットのセット
66	連結レジスタの中央切り出し	unsigned long xtrct(unsigned long data1, unsigned long data2)	連結した 64 ビットの内容から中央の 32 ビットを切り出し
67	キャリー付き加算	long addc(long data1, long data2)	2 つのデータと T ビットを加算し、キャリーを T ビットに反映
68		int ovf_addc(long data1, long data2)	2 つのデータと T ビットを加算し、キャリーを参照
69		long addv(long data1, long data2)	2 つのデータを加算し、キャリーを T ビットに反映
70		int ovf_addv(long data1, long data2)	2 つのデータを加算し、キャリーを参照
71	ボロー付き減算	long subc(long data1, long data2)	data1 から data2 と T ビットを減算し、ボローを T ビットに反映
72		int unf_subc(long data1, long data2)	data1 から data2 と T ビットを減算し、ボローを参照
73		long subv(long data1, long data2)	data1 から data2 を減算し、ボローを T ビットに反映
74		int unf_subv(long data1, long data2)	data1 から data2 を減算し、ボローを参照
75	符号反転	long negc(long data)	0 から data と T ビットを減算し、ボローを T ビットに反映

項目		仕様	機能
76	1 ビット除算	unsigned long div1(unsigned long data1, unsigned long data2)	data1/data2 の 1 ステップ除算を行い、結果を T ビットに反映
77		int div0s(long data1, long data2)	data1/data2 の符号付き除算の初期設定をし、T ビットを参照
78		void div0u(void)	符号なし除算の初期設定
79	回転	unsigned long rotl(unsigned long data)	データを 1 ビット左方向に回転し、オペランドの外へ出たビットを T ビットに反映
80		unsigned long rotr(unsigned long data)	データを 1 ビット右方向に回転し、オペランドの外へ出たビットを T ビットに反映
81		unsigned long rotcl(unsigned long data)	データを 1 ビット左方向に T ビットを含めて回転し、オペランドの外へ出たビットを T ビットに反映
82		unsigned long rotcr(unsigned long data)	データを 1 ビット右方向に T ビットを含めて回転し、オペランドの外へ出たビットを T ビットに反映
83	シフト	unsigned long shll(unsigned long data)	データを 1 ビット左シフトし、オペランドの外へ出たビットを T ビットに反映
84		unsigned long shlr(unsigned long data)	データを論理的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映
85		long shar(long data)	データを算術的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映
86	飽和減算	long clipsb(long data)	符号付き 1 バイトデータ飽和演算
87		long clipsw(long data)	符号付き 2 バイトデータ飽和演算
88		unsigned long clipub(unsigned long data)	符号なし 1 バイトデータ飽和演算
89		unsigned long clipuw(unsigned long data)	符号なし 2 バイトデータ飽和演算
90	TBR 設定・参照	void set_tbr(void *data)	TBR に data を設定
91		void *get_tbr(void)	TBR の値を参照
92	多重割り込み	void sr_jsr(void *func, int imask);	SR.RB と SR.BL を 0 クリアし、SR.I0-I3 に imask を設定し、func 関数を呼び出す
93	メモリ上のビット操作	void bset(unsigned char *addr, unsigned char bit_num);	指定アドレス(addr)の指定ビット(bit_num)に 1 を設定
94		void bclr(unsigned char *addr, unsigned char bit_num);	指定アドレス(addr)の指定ビット(bit_num)に 0 を設定

	項目	仕様	機能
95	メモリ上のビット操作	void bcopy(unsigned char *from_addr, unsigned char from_bit_num, unsigned char *to_addr, unsigned char to_bit_num);	指定アドレス 1(from_addr)の指定ビット 1(from_bit_num)を、T ビットと指定アドレス 2(to_addr)の指定ビット 2(to_bit_num)に設定
96		void bnotcopy(unsigned char *from_addr, unsigned char from_bit_num, unsigned char *to_addr, unsigned char to_bit_num);	指定アドレス 1(from_addr)の指定ビット 1(from_bit_num)の反転を、T ビットと指定アドレス 2(to_addr)の指定ビット 2(to_bit_num)に設定

組み込み関数を使用する場合は、必ず<machine.h>、または<umachine.h>か<smachine.h>をインクルードしてください。

SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP の実行モードに対応し<machine.h>の内容を表 10.35 のように分割しています。

表 10.35 組み込み関数用インクルードファイル

	インクルードファイル	内容
1	<machine.h>	組み込み関数全体
2	<smachine.h>	特権モードでのみ使用可能な組み込み関数
3	<umachine.h>	2 以外の組み込み関数

SR 設定

void set_cr(int cr)

説　明　ステータスレジスタに cr(32 ビット)を設定します。

ヘッダ　<machine.h>または<smachine.h>

引　数　cr　設定値

例
```
#include <machine.h>
void main(void)
{
    set_cr(0x60000000);  /* Supervisor, RBank=1, BL=0, Imask=0 */
}
```

SR 参照

int get_cr(void)

説　明　ステータスレジスタの値を参照します。

ヘッダ　<machine.h>または<smachine.h>

リターン値　ステータスレジスタの値

例
```
#include <machine.h>
void main(void)
{
    set_cr(get_cr()|0x1000000);      /* set BL bit */
}
```

割り込みマスク設定

void set_imask(int mask)

説　明　割り込みマスク(4 ビット)に mask(4 ビット)を設定します。

ヘッダ　<machine.h>または<smachine.h>

引　数　mask　設定値(4 ビット)

例
```
#include <machine.h>
void main(void)
{
    set_imask(15);
}
```

割り込みマスク参照

int get_imask(void)

説　明　割り込みマスク(4 ビット)を参照します。

ヘッダ　<machine.h>または<smachine.h>

リターン値　割り込みマスクの値

例
```
#include <machine.h>
void main(void)
{
    int mask;
    mask=get_imask();
}
```

VBR 設定

*void set_vbr(void *base)*

説　明　ベクタベースレジスタ(VBR)にbase(32ビット)を設定します。

ヘッダ　<machine.h>または<smachine.h>

引　数　base　設定値

例
```
#include <machine.h>
#define VBR 0x0000FC00
void main(void)
{
    set_vbr((void *)VBR);
}
```

VBR 参照

*void *get_vbr(void)*

説　明　ベクタベースレジスタ(VBR)を参照します。

ヘッダ　<machine.h>または<smachine.h>

リターン値　ベクタベースレジスタの値

例
```
#include <machine.h>
void main(void)
{
    void *vbr;
    vbr=get_vbr();
}
```

GBR 設定

*void set_gbr(void *base)*

説　明　グローバルベースレジスタ(GBR)にbase(32ビット)を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　base　設定値

例
```
#include <machine.h>
#define IOBASE 0x05fffec0
void main(void)
{
    set_gbr((void *)IOBASE);
}
```

備　考　GBRはコントロールレジスタのため、本コンパイラでは関数ごとに内容を保証していません。
GBRの設定を変えるときには注意が必要です。
gbr=autoを指定した場合、本関数は使用できません。

GBR 参照

*void *get_gbr(void)*

説　明　グローバルベースレジスタ(GBR)の値を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　グローバルベースレジスタの値

例
```
#include <machine.h>
void main(void)
{
    void *gbr;
    gbr=get_gbr();
}
```

備　考　gbr=autoを指定した場合、本関数は使用できません。

GBR ベースバイト参照

unsigned char gbr_read_byte(int offset)

説　明　GBR 相対 offset のバイトデータ (8 ビット) を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　バイトデータ (8 ビット) の参照値

引　数　offset　オフセットアドレス

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    if(gbr_read_byte(BDATA)!=0)
        :
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースワード参照

unsigned short gbr_read_word(int offset)

説　明　GBR 相対 offset のワードデータ (16 ビット) を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　ワードデータ (16 ビット) の参照値

引　数　offset　オフセットアドレス

例

```
#include <machine.h>
#define WDATA 0
void main(void)
{
    if(gbr_read_word(WDATA)!=0)
        :
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+510 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースロングワード参照

unsigned long gbr_read_long(int offset)

説　明　GBR 相対 offset のロングワードデータ (32 ビット) を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　ロングワードデータ (32 ビット) の参照値

引　数　offset　オフセットアドレス

例

```
#include <machine.h>
#define LDATA 0
void main(void)
{
    if(gbr_read_long(LDATA)!=0)
        :
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+1020 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースバイト設定

void gbr_write_byte(int offset, unsigned char data)

説　明　GBR 相対 offset へ data (8 ビット) を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　offset　オフセットアドレス
data　設定値 (8 ビット)

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_write_byte(BDATA,0);
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースワード設定

void gbr_write_word(int offset, unsigned short data)

説　明　GBR 相対 offset へ data(16 ビット)を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数

offset	オフセットアドレス
data	設定値(16 ビット)

例

```
#include <machine.h>
#define WDATA 0
void main(void)
{
    gbr_write_word(WDATA,0);
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+510 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースロングワード設定

void gbr_write_long(int offset, unsigned long data)

説　明　GBR 相対 offset へ data(32 ビット)を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数

offset	オフセットアドレス
data	設定値(32 ビット)

例

```
#include <machine.h>
#define LDATA 0
void main(void)
{
    gbr_write_long(LDATA,0);
}
```

備　考　offset は定数でなければなりません。
offset 指定可能範囲は+1020 バイトまでです。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースバイト AND

void gbr_and_byte(int offset, unsigned char mask)

説　明　GBR 相対 offset のバイトデータと mask の AND をとり、offset に設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数

offset	オフセットアドレス
mask	データ(8 ビット)

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_and_byte(BDATA,0x01);
}
```

備　考
mask は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
mask 指定可能範囲は 0～+255 です。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースバイト OR

void gbr_or_byte(int offset, unsigned char mask)

説　明　GBR 相対 offset のバイトデータと mask の OR をとり、offset に設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数

offset	オフセットアドレス
mask	データ(8 ビット)

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_or_byte(BDATA,0x01);
}
```

備　考
mask は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
mask 指定可能範囲は 0～+255 です。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースバイト XOR

void gbr_xor_byte(int offset, unsigned char mask)

説　明　GBR 相対 offset のバイトデータと mask の XOR をとり、offset に設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数

offset	オフセットアドレス
mask	データ(8 ビット)

例

```
#include <machine.h>
#define BDATA 0
void main(void)
{
    gbr_xor_byte(BDATA,0x01);
}
```

備　考　mask は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
mask 指定可能範囲は 0～+255 です。
gbr=auto を指定した場合、本関数は使用できません。

GBR ベースバイト TEST

int gbr_tst_byte(int offset, unsigned char mask)

説　明　GBR 相対 offset のバイトデータと mask の AND をとり、その値を 0 と判定し結果を T ビットに設定します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　T ビットの反転値

引　数

offset	オフセットアドレス
mask	データ(8 ビット)

例

```
#include <machine.h>
#define BDATA 0
int a;
void main(void)
{
    if(gbr_tst_byte(BDATA,0))
        a = 0;
}
```

備　考　mask は定数でなければなりません。
offset 指定可能範囲は+255 バイトまでです。
mask 指定可能範囲は 0～+255 です。
gbr=auto を指定した場合、本関数は使用できません。

GBR 組み込み関数の使用例

```
#include <machine.h>
#define CDATA1 0
#define CDATA2 1
#define CDATA3 2
#define SDATA1 4
#define IDATA1 8
#define IDATA2 12

struct {
   char    cdata1;                    /*offset 0                    */
   char    cdata2;                    /*offset 1                    */
   char    cdata3;                    /*offset 2                    */
   short   sdata1;                    /*offset 4                    */
   int     idata1;                    /*offset 8                    */
   int     idata2;                    /*offset 12                   */
} table;
void f();

void f()
{
   set_gbr(&table);                   /* GBR に table の先頭アドレス      */
       :                              /* を設定                          */
   gbr_write_byte(CDATA2,10);         /* table.cdata2 に 10 を設定       */
   gbr_write_long(IDATA2,100);        /* table.idata2 に 100 を設定      */
       :
   if(gbr_read_byte(CDATA2)!=10)      /* table.cdata2 の値を参照         */
      gbr_and_byte(CDATA2,10);        /* table.cdata2 の値と 10 の AND   */
                                      /* をとって table.cdata2 に設定    */
   gbr_or_byte(CDATA2,0x0F);          /* table.cdata2 の値と 0x0f の OR  */
       :                              /* をとって table.cdata2 に設定    */
   sleep();                           /* SLEEP 命令に展開                */
}
```

GBR 組み込み関数の有効な使用法

- 頻繁にアクセスするオブジェクトをメモリに割り付け、そのオブジェクトの先頭アドレスを GBR に設定する。
- 論理演算を多用するバイトデータをできるだけ構造体の先頭から 128 バイトまでに宣言する。

これにより、構造体アクセスに必要な先頭アドレスロードと、論理演算に必要なメモリロード、ストアに対する命令が削減できます。

SLEEP 命令

void sleep(void)

説　明　低消費電力状態遷移命令 SLEEP に展開します。

ヘッダ　<machine.h>または<smachine.h>

例

```
#include <machine.h>
void main(void)
{
    sleep();
}
```

TAS 命令

*int tas (char *addr)*

説　明　TAS.B @Rn に展開します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　TAS 命令実行結果の T ビット値

引　数　addr　TAS 命令で指定するアドレス

例

```
#include <machine.h>
char a;
void main(void)
{
    tas(&a);
}
```

TRAPA 命令

int trapa (int trap_no)

説　明　　TRAPA #trap_no に展開します。

ヘッダ　　<machine.h>または<umachine.h>

引　数　　trap_no　　トラップ番号

例

```
#include <machine.h>
void main(void)
{
    trapa(0);
}
```

備　考　　trap_no は 0 以上 255 以下の定数でなければなりません。

OS システムコール

int trapa_svc (int trap_no, int code, type1 para1, type2 para2, type3 para3, type4 para4)

説　明　　HI7000 をはじめ、各種 OS のシステムコールを発行します。trapa_svc を実行すると、R0 に code、R4～R7 に para1～para4 を設定し、
TRAPA #trap_no
命令を実行します。

ヘッダ　　<machine.h>または<umachine.h>

引　数

trap_no	トラップ番号
code	機能コード
para1～para4	パラメータ (0～4 個の可変) 型 type1～type4 は、スカラ型またはポインタ型です。

例

```
#include <machine.h>
#define SIG_SEM 0xffc8
void main(void)
{
    trapa_svc(63,SIG_SEM,0x05);
}
```

備　考　　trap_no は 0 以上 255 以下の定数でなければなりません。

PREF 命令

*void prefetch (void *p)*

説　明　p の指す領域(16 バイト、ただし、領域は(int)p&0xfffffff0)からの 16 バイト(cpu=sh4|sh4a|sh4aldsp の場合は(int)p&0xffffffe0 から 32 バイト)のデータをキャッシュに読み込みます。

ヘッダ　<machine.h>または<umachine.h>

引　数　p　プリフェッチを行うアドレス

例

```
#include <machine.h>
char a[1200];
void main(void)
{
    char *pa = a;
    prefetch(pa);
}
```

備　考　cpu= sh2a|sh2afpu|sh3|sh3dsp|sh4|sh4a|sh4aldsp を指定した場合のみ使用可能です。
プログラムの論理的な動作には影響を与えません。

TRACE 命令

void trace (long v)

説　明　一部のエミュレータが持つソフトウェアトレース機能をサポートします。

ヘッダ　<machine.h>または<umachine.h>

引　数　v　指定する変数

例

```
#include <machine.h>
void main(void)
{
    long v;
    trace(v);
}
```

備　考　cpu=sh1 以外を指定した場合に使用可能です。
ソフトウェアトレース機能の詳細は、ご使用のエミュレータのユーザーズマニュアルを参照してください。
本関数は、エミュレータを接続してデバッグを行っている段階でのみ使用できます。
エミュレータを接続しない状態では本関数は使用しないでください。

LDTLB 命令

void ldtlb (void)

説　明　LDTLB 命令に展開します。

ヘッダ　<machine.h>または<smachine.h>

例
```
#include <machine.h>
void main(void)
{
    ldtlb();
}
```

備　考　cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp を指定した場合のみ使用可能です。

NOP 命令

void nop (void)

説　明　NOP 命令に展開します。

ヘッダ　<machine.h>または<umachine.h>

例
```
#include <machine.h>
void main(void)
{
    int a;
    if (a) {
        nop();
    }
}
```

64bit 乗算

long dmuls_h (long data1, long data2)

説　明　符号付き 32 ビット×符号付き 32 ビット→符号付き 64 ビットの乗算を行い、結果の上位 32 ビットを参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　乗算結果の上位 32 ビット

例

```
#include <machine.h>
extern long data1, data2;
extern long result;
void main(void)
{
    result = dmuls_h(data1, data2);
}
```

備　考　cpu=sh1 を指定した場合、本関数は使用できません。

64bit 乗算

unsigned long dmuls_l (long data1, long data2)

説　明　符号付き 32 ビット×符号付き 32 ビット→符号付き 64 ビットの乗算を行い、結果の下位 32 ビットを参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　乗算結果の下位 32 ビット

例

```
#include <machine.h>
extern long data1, data2;
extern unsigned long result;
void main(void)
{
    result = dmuls_l(data1, data2);
}
```

備　考　cpu=sh1 を指定した場合、本関数は使用できません。

64bit 乗算

unsigned long dmulu_h (unsigned long data1, unsigned long data2)

説　明　符号なし 32 ビット×符号なし 32 ビット→符号なし 64 ビットの乗算を行い、結果の上位 32 ビットを参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　乗算結果の上位 32 ビット

例

```
#include <machine.h>
extern unsigned long data1, data2;
extern unsigned long result;
void main(void)
{
    result = dmulu_h(data1, data2);
}
```

備　考　cpu=sh1 を指定した場合、本関数は使用できません。

64bit 乗算

unsigned long dmulu_l (unsigned long data1, unsigned long data2)

説　明　符号なし 32 ビット×符号なし 32 ビット→符号なし 64 ビットの乗算を行い、結果の下位 32 ビットを参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　乗算結果の下位 32 ビット

例

```
#include <machine.h>
extern unsigned long data1, data2;
extern unsigned long result;
void main(void)
{
    result = dmulu_l(data1, data2);
}
```

備　考　cpu=sh1 を指定した場合、本関数は使用できません。

SWAP.B 命令

unsigned short swapb(unsigned short data)

説　明　2 バイトデータの上位・下位 1 バイトを交換します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　2 バイトデータの上位・下位 1 バイト交換結果

例

```
#include <machine.h>
extern unsigned short data;
extern unsigned short result;
void main(void)
{
    result = swapb(data);
    /* 例えば、data=0x1234 の場合、
       result=0x3412 になります。 */
}
```

SWAP.W 命令

unsigned long swapw(unsigned long data)

説　明　4 バイトデータの上位・下位 2 バイトを交換します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　4 バイトデータの上位・下位 2 バイト交換結果

例

```
#include <machine.h>
extern unsigned long data;
extern unsigned long result;
void main(void)
{
    result = swapw(data);
    /* 例えば、data=0x12345678 の場合、
       result=0x56781234 になります。 */
}
```

4 バイトデータ並べ替え

unsigned long end_cnvl(unsigned long data)

説　明　4 バイトデータを 1 バイトごとに上位・下位を逆順に並べ替えます。

ヘッダ　<machine.h>または<umachine.h>

リターン値　4 バイトデータの並び替え結果

例

```
#include <machine.h>
extern unsigned long data;
extern unsigned long result;
void main(void)
{
    result = end_cnvl(data);
    /* 例えば、data=0x12345678 の場合、
       result=0x78563412 になります。 */
}
```

MAC.W 命令

*int macw(short *ptr1, short *ptr2, unsigned int count)*
*int macwl(short *ptr1, short *ptr2, unsigned int count, unsigned int mask)*

説　明　二つのデータテーブルの内容の積和を求めます。

ヘッダ　<machine.h>または<umachine.h>

リターン値　演算結果

引　数

ptr1	積和演算するデータの先頭アドレス
ptr2	積和演算するデータの先頭アドレス
count	積和演算を実行する回数
mask	リングバッファ対応のアドレスマスク

例

```
#include <machine.h>
short tbl1[]={a1,a2,a3,a4};
short tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
    result1=macw(tbl1,tbl2,3);      /* a1*b1+a2*b2+a3*b3 を求めます。 */

    result2=macwl(tbl1,tbl2,4,0xfffffffb);
                                    /* a1*b1+a2*b2+a3*b1+a4*b2 を求めます。 */
}
```

備　考　パラメータのチェックを行いませんので、次のことに注意してください。
- ptr1、ptr2 の指すテーブルは、2 バイトアライメントでなければなりません。
- macwl の ptr2 の指すテーブルはリングバッファマスク×2 のサイズのアライメントでなければなりません。

MAC.L 命令

*int macl(int *ptr1, int *ptr2, unsigned int count)*
*int macll(int *ptr1, int *ptr2, unsigned int count,unsigned int mask)*

説　明　二つのデータテーブルの内容の積和を求めます。

ヘッダ　<machine.h>または<umachine.h>

リターン値　演算結果

引　数

ptr1	積和演算するデータの先頭アドレス
ptr2	積和演算するデータの先頭アドレス
count	積和演算を実行する回数
mask	リングバッファ対応のアドレスマスク

例

```
#include <machine.h>
int tbl1[]={a1,a2,a3,a4};
int tbl2[]={b1,b2,b3,b4};
int result1,result2;
void main(void)
{
    result1=macl(tbl1,tbl2,3);      /* a1*b1+a2*b2+a3*b3 を求めます。 */

    result2=macll(tbl1,tbl2,4,0xfffffff7);
                                    /* a1*b1+a2*b2+a3*b1+a4*b2 を求めます。 */
}
```

備　考　cpu=sh1 を指定した場合、本関数は使用できません。
パラメータのチェックを行いませんので、次のことに注意してください。
・ptr1、ptr2 の指すテーブルは、4 バイトアライメントでなければなりません。
・macll の ptr2 の指すテーブルはリングバッファマスク×2 のサイズのアライメントでなければなりません。

FPSCR 設定

void set_fpscr(int cr)

説　明　浮動小数点ステータス制御レジスタ(FPSCR)に cr(32 ビット)を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　cr　設定値(32 ビット)

例

```
#include <machine.h>
void main(void)
{
    set_fpscr(0);
}
```

備　考　cpu=sh2e|sh2afpu|sh4|sh4a を指定した場合のみ使用可能です。

FPSCR 参照

int get_fpscr(void)

説　明　　浮動小数点ステータス制御レジスタ(FPSCR)を参照します。

ヘッダ　　<machine.h>または<umachine.h>

リターン値　FPSCR の値

例

```
#include <machine.h>
int cr;
void main(void)
{
    cr=get_fpscr();
}
```

備　考　　cpu=sh2e|sh2afpu|sh4|sh4a を指定した場合のみ使用可能です。

FIPR 命令

float fipr(float vect1[4], float vect2[4])

説　明　　2 つのベクタの内積を求めます。

ヘッダ　　<machine.h>または<umachine.h>

リターン値　演算結果

引　数　　vect1　　ベクタ
　　vect2　　ベクタ

例

```
#include <machine.h>
extern float data1[4],data2[4];
float result;
void main(void)
{
    result=fipr(data1,data2);
}
```

備　考　　cpu=sh4|sh4a を指定した場合のみ使用可能です。

FTRV 命令

void ftrv(float vec1[4], float vec2[4])

説　明　vec1(ベクタ)を tbl(4×4 行列)で変換した結果を vec2(ベクタ)に格納します。
tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数　vec1　ベクタ
vec2　ベクタ

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4],data2[4];
void main(void)
{
    ld_ext(tbl);
    ftrv(data1,data2);
    /* i=0,1,2,3 として
        data2[i]=data1[0]*tbl[0][i]+data1[1]*tbl[1][i]
           +data1[2]*tbl[2][i]+data1[3]*tbl[3][i]
       が data2 の結果になります。 */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
ld_ext()関数および st_ext()関数は、浮動小数点ステータス制御レジスタ(FPSCR)の浮動小数点レジスタバンクビット(FR)を変更して拡張レジスタにアクセスします。割り込み関数内で ld_ext() 関数または st_ext()関数を使用しているときは、ベクタ演算組み込み関数の前後で割り込みマスクを変更してください。以下に例を示します。
例：

```
extern float mat1[4][4];
extern float vec1[4],vec2[4];
#pragma interrupt (intfunc)
void intfunc(){
        :
    ld_ext();
        :
}
void normfunc(){
        :
    int maskdata=get_imask();
    set_imask(15);
    ld_ext(mat1);
    ftrv(vec1,vec2);
    set_imask(maskdata);
        :
}
```

4 次元ベクタの 4×4 行列による変換と 4 次元ベクタとの和

void ftrvadd(float vec1[4], float vec2[4], float vec3[4])

説　明　vec1(ベクタ)を tbl(4×4 行列)で変換した結果と vec2(ベクタ)の和を vec3(ベクタ)に格納します。tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数

vec1	ベクタ
vec2	ベクタ
vec3	ベクタ

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    ld_ext(tbl);
    ftrvadd(data1,data2,data3);      /* data3=data1×tbl＋data2 */
    /* i=0,1,2,3 として
        data3[i]=data1[0]*tbl[0][i]
                 +data1[1]*tbl[1][i]
                 +data1[2]*tbl[2][i]
                 +data1[3]*tbl[3][i]
                 +data2[i]
       が data3 の結果になります。 */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。

4 次元ベクタの 4×4 行列による変換と 4 次元ベクタとの差

void ftrvsub(float vec1[4], float vec2[4], float vec3[4])

説　明　vec1(ベクタ)を tbl(4×4 行列)で変換した結果と vec2(ベクタ)の差を vec3(ベクタ)に格納します。tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数

vec1	ベクタ
vec2	ベクタ
vec3	ベクタ

例

```
#include <machine.h>
extern float tbl[4][4];
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    ld_ext(tbl);
    ftrvsub(data1,data2,data3);      /* data3=data1×tbl－data2 */
    /* i=0,1,2,3 として
          data3[i]=data1[0]*tbl[0][i]
                   +data1[1]*tbl[1][i]
                   +data1[2]*tbl[2][i]
                   +data1[3]*tbl[3][i]
                   -data2[i]
        が data3 の結果になります。 */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。

4 次元ベクタの和

void add4(float vec1[4], float vec2[4], float vec3[4])

説　明　vec1(ベクタ)と vec2(ベクタ)の和を vec3(ベクタ)に格納します。

ヘッダ　<machine.h>または<umachine.h>

引　数

vec1	ベクタ
vec2	ベクタ
vec3	ベクタ

例

```
#include <machine.h>
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    add4(data1,data2,data3);        /* data3=data1+data2 */
}
```

備　考　cpu=sh2afpu|sh4|sh4a を指定した場合のみ使用可能です。

4 次元ベクタの差

void sub4(float vec1[4], float vec2[4], float vec3[4])

説　明　vec1(ベクタ)と vec2(ベクタ)の差を vec3(ベクタ)に格納します。

ヘッダ　<machine.h>または<umachine.h>

引　数

vec1	ベクタ
vec2	ベクタ
vec3	ベクタ

例

```
#include <machine.h>
extern float data1[4];
extern float data2[4];
extern float data3[4];
void main(void)
{
    sub4(data1,data2,data3);        /* data3=data1-data2 */
}
```

備　考　cpu=sh2afpu|sh4|sh4a を指定した場合のみ使用可能です。

4×4 行列の乗算

void mtrx4mul(float mat1[4], float mat2[4])

説　明　mat1(4×4 行列)を tbl(4×4 行列)で変換した結果 mat2 に格納します。
tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数　mat1　4×4 行列
mat2　4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4mul(tbl1,tbl2);            /* tbl2=tbl1×tbl */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
本関数は行列の演算のため非可換です。
例：

```
extern float matA[][];
extern float matB[][];
int judge(){
   float data1[4][4], data2[4][4];
   set_imask(15);
   ld_ext(matA);
   mtrx4mul(matB,data1);/* data1=matB×matA */
   ld_ext(matB);
   mtrx4mul(matA,data2);/* data2=matA×matB */
   /* この時の data1[][]と data2[][]の各要素は必ずしも一致しません。 */
}
```

4×4 行列の乗算と和

void mtrx4muladd(float mat1[4], float mat2[4], float mat3[4])

説　明　mat1(4×4 行列)を tbl(4×4 行列)で変換した結果と mat2(4×4 行列)との和を mat3 に格納します。tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数

mat1	4×4 行列
mat2	4×4 行列
mat3	4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4muladd(tbl1,tbl2,tbl3);     /* tbl3=tbl1×tbl＋tbl2 */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
本関数は行列の演算のため非可換です。

4×4 行列の乗算と差

void mtrx4mulsub(float mat1[4] , float mat2[4] , float mat3[4])

説　明　mat1(4×4 行列)を tbl(4×4 行列)で変換した結果と mat2(4×4 行列)との差を mat3 に格納します。tbl は組み込み関数 ld_ext()でロードした行列です。

ヘッダ　<machine.h>または<umachine.h>

引　数

mat1	4×4 行列
mat2	4×4 行列
mat3	4×4 行列

例

```
#include <machine.h>
extern float tbl[4][4];
extern float tbl1[4][4];
extern float tbl2[4][4];
extern float tbl3[4][4];
void main(void)
{
    ld_ext(tbl);
    mtrx4mulsub(tbl1,tbl2,tbl3);     /* tbl3=tbl1×tbl－tbl2 */
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
本関数は行列の演算のため非可換です。

拡張レジスタへのロード

void ld_ext(float mat[4][4])

説　明　mat(4×4 行列)を拡張レジスタにロードします。

ヘッダ　<machine.h>または<umachine.h>

引　数　mat　4×4 行列

例
```
#include <machine.h>
extern float tbl[4][4];
void main(void)
{
    ld_ext(tbl);
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
本関数は、浮動小数点ステータス制御レジスタ(FPSCR)の浮動小数点レジスタバンクビット(FR)を変更して拡張レジスタにアクセスします。割り込み関数内で本関数を使用しているときは、ベクタ演算組み込み関数の前後で割り込みマスクを変更してください。

拡張レジスタからのストア

void st_ext(float mat[4][4])

説　明　拡張レジスタの内容を mat(4×4 行列)にストアします。

ヘッダ　<machine.h>または<umachine.h>

引　数　mat　4×4 行列

例
```
#include <machine.h>
extern float tbl[4][4];
void main(void)
{
    st_ext(tbl);
}
```

備　考　cpu=sh4|sh4a を指定した場合のみ使用可能です。
本関数は、浮動小数点ステータス制御レジスタ(FPSCR)の浮動小数点レジスタバンクビット(FR)を変更して拡張レジスタにアクセスします。割り込み関数内で本関数を使用しているときは、ベクタ演算組み込み関数の前後で割り込みマスクを変更してください。

絶対値

long _ _fixed pabs_lf(long _ _fixed data)
long _ _accum pabs_la(long _ _accum data)

説　明　絶対値を求めます。

ヘッダ　<machine.h>または<umachine.h>

リターン値　演算結果

引　数　data　絶対値を求めるデータ

例
```
#include <machine.h>
long _ _fixed result;
long _ _fixed ptr;
void main(void)
{
    result=pabs_lf(ptr);
}
```

備　考　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
data の絶対値を求めた結果、リターン値の型(pabs_lf()は long _ _fixed 型、pabs_la()は long _ _accum 型)の値として表現できない時の動作は保証しません。

MSB 検出

_ _fixed pdmsb_lf(long _ _fixed data)
_ _fixed pdmsb_la(long _ _accum data)

説　明　MSB 検出します(データを正規化するためのシフト量を求めます)。

ヘッダ　<machine.h>または<umachine.h>

リターン値　演算結果

引　数　data　MSB を検出するデータ

例
```
#include <machine.h>
_ _fixed result;
long _ _fixed ptr;
void main(void)
{
    result=pdmsb_lf(ptr);
}
```

備　考　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

算術シフト

long _ _fixed psha_lf(long _ _fixed data, int count)
long _ _accum psha_la(long _ _accum data, int count)

説　明　算術シフトを行います。

ヘッダ　<machine.h>または<umachine.h>

リターン値　演算結果

引　数

data	算術シフトを行うデータ
count	算術シフト数

例

```
#include <machine.h>
long _ _fixed result;
long _ _fixed ptr;
int count;
void main(void)
{
    result=psha_lf(ptr,count);
}
```

備　考　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
count 指定可能範囲は-32～+32 です。正の値を指定した場合は左シフト、負の値を指定した場合はその絶対値分だけ右シフトを行います。範囲外の値を指定した場合、動作は保証しません。

ビットパターンコピー

long _ _fixed long_as_lfixed(long data)
long lfixed_as_long(long _ _fixed data)

説　明　ビットパターンコピーを行います(汎用レジスタ⇔DSP レジスタ間コピー)。

ヘッダ　<machine.h>または<umachine.h>

リターン値　コピー結果

引　数

data	コピーを行うデータ

例

```
#include <machine.h>
long _ _fixed result;
long ptr;
void main(void)
{
    result=long_as_lfixed(ptr);
}
```

備　考　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

丸め演算

_ _accum rndtoa(long _ _accum data)
_ _fixed rndtof(long _ _fixed data)

説　明　　丸め演算を行います。

ヘッダ　　<machine.h>または<umachine.h>

リターン値　　演算結果

引　数　　data　　丸め演算を行うデータ

例

```
#include <machine.h>
_ _accum result;
long _ _accum ptr;
void main(void)
{
    result=rndtoa(ptr);
}
```

備　考　　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

モジュロアドレッシング設定

void set_circ_x(_ _X_ _circ _ _fixed array[ ], size_t size)
void set_circ_y(_ _Y_ _circ _ _fixed array[ ], size_t size)

説　明　モジュロアドレッシングの設定を行います。

ヘッダ　<machine.h>または<smachine.h>

引　数　array[]　モジュロアドレッシングを行うデータ
size　データのサイズ

例

```
#include <machine.h>
_ _circ _ _X _ _fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
_ _Y _ _fixed output[8];

void main(void)
{
    int i;
    set_circ_x(input, sizeof(input));  /* モジュロアドレッシングの設定 */
    for (i = 0; i < 8; i++) {
        output[i] = input[i];
    }
    clr_circ();                        /* モジュロアドレッシングの解除 */
}
```

備　考　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

モジュロアドレッシング解除

void clr_circ()

説　明　　モジュロアドレッシングの解除を行います。
SR の右から 10、11 ビット目をゼロで初期化します。

ヘッダ　　<machine.h>または<smachine.h>

例

```
#include <machine.h>
_ _circ _ _X _ _fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
_ _Y _ _fixed output[8];
void main(void)
{
    int i;
    set_circ_x(input, sizeof(input));  /* モジュロアドレッシングの設定 */
    for (i = 0; i < 8; i++) {
        output[i] = input[i];
    }
    clr_circ();                        /* モジュロアドレッシングの解除 */
}
```

備　考　　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

CS ビットの設定

void set_cs(unsigned int mode)

説　明　　CS ビットの設定を行います。

ヘッダ　　<machine.h>または<umachine.h>

引　数　　mode　　設定するモード(0～5)

設定する値	モード
0	キャリ/ボローモード
1	負値モード
2	ゼロ値モード
3	オーバフローモード
4	符号付き大モード
5	符号付き以上モード

例

```
#include <machine.h>
#define MODE 1
void main(void)
{
    set_cs(MODE);
}
```

備　考　　cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。

正弦・余弦の計算

*void fsca(long angle, float *sinv,float *cosv)*

説　明　angle で指定された角度から、正弦・余弦の近似値を計算し、その結果を sinv、cosv の指す領域に設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　angle　正弦・余弦を求める角度
(32bit 領域のうち、2 の 16 乗 bit の右側に小数点のある固定小数点データとして表現した時の、ビットイメージを angle に指定してください)
sinv　演算結果を格納するアドレス(正弦)
cosv　演算結果を格納するアドレス(余弦)

例

```
#include <machine.h>
long angle = (45<<16)/360;     /* 45 度 */
float sinv;
float cosv;
void main(void)
{
    fsca(angle,&sinv,&cosv);
}
```

備　考　cpu=sh4a を指定した場合のみ使用可能です。

平方根の逆数

float fsrra(float data)

説　明　平方根の逆数の近似値を求めます。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　平方根の逆数を求める値

リターン値　演算結果

例

```
#include <machine.h>
float data;
float result;
void main(void)
{
    result=fsrra(data);
}
```

備　考　cpu=sh4a を指定した場合のみ使用可能です。

命令キャッシュ無効化

*void icbi(void *p)*

説　明　命令キャッシュの無効化を行います。

ヘッダ　<machine.h>または<umachine.h>

引　数　p　変数または関数のアドレス

例
```
#include <machine.h>
extern int *p;
void main(void)
{
    icbi(p);
}
```

備　考　cpu=sh4a|sh4aldsp を指定した場合のみ使用可能です。

キャッシュブロック操作

*void ocbi(void *p)*
*void ocbp(void *p)*
*void ocbwb(void *p)*

説　明　キャッシュブロックの操作を行います。
ocbi:　キャッシュブロックの無効化
ocbp:　キャッシュブロックのパージ
ocbwb:　キャッシュブロックの書き戻し

ヘッダ　<machine.h>または<umachine.h>

引　数　p　変数または関数のアドレス

例
```
#include <machine.h>
extern int *p;
void main(void)
{
    ocbi(p);
}
```

備　考　cpu=sh4|sh4a|sh4aldsp を指定した場合のみ使用可能です。

命令キャッシュプリフェッチ

*void prefi(void *p)*

説　明　32 バイト境界で始まる 32 バイトの命令ブロックを命令キャッシュに読み込みます。

ヘッダ　<machine.h>または<umachine.h>

引　数　p　プリフェッチを行うアドレス

例
```
#include <machine.h>
void *pa;
void main(void)
{
    prefi(pa);
}
```

備　考　cpu=sh4a|sh4aldsp を指定した場合のみ使用可能です。

システム同期

void synco(void)

説　明　SYNCO 命令に展開します。
SYNCO 命令はデータ操作の同期に使用します。SYNCO 命令を実行すると SYNCO 命令より前のデータ操作を完了してから SYNCO 命令より後の命令を開始します。

ヘッダ　<machine.h>または<umachine.h>

例
```
#include <machine.h>
void main(void)
{
    synco();
}
```

備　考　cpu=sh4a|sh4aldsp を指定した場合のみ使用可能です。

T ビットの参照

int movt (void)

説　明　SR レジスタの T ビットの値を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　T ビットの値

例

```
#include <machine.h>
extern int sr_t;
void main(void)
{
     sr_t = movt();
}
```

T ビットのクリア

void clrt (void)

説　明　SR レジスタの T ビットをクリアします。

ヘッダ　<machine.h>または<umachine.h>

例

```
#include <machine.h>
void main(void)
{
     clrt();
}
```

T ビットのセット

void sett (void)

説　明　SR レジスタの T ビットをセットします。

ヘッダ　<machine.h>または<umachine.h>

例
```
#include <machine.h>
void main(void)
{
     sett();
}
```

連結レジスタの中央切り出し

unsigned long xtrct (unsigned long data1, unsigned long data2)

説　明　data1 と data2 を連結した 64 ビットの内容から中央の 32 ビットを切り出します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　データ上位 32bit
　　data2　データ下位 32bit

リターン値　(data1 の下位 16bit):(data2 の上位 16bit)

例
```
#include <machine.h>
extern unsigned long result,data1,data2;
void main(void)
{
     result = xtrct(data1,data2);
}
```

キャリー付き加算

long addc (long data1, long data2)

説　明　2 つのデータと T ビットを加算し、キャリーを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　加算データ 1
　　data2　加算データ 2

リターン値　加算結果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     result = addc(data1,data2);
}
```

備　考　本組み込み関数は T ビットの内容を参照しますので、直前には正しい T ビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われません。

```
if (a) {...}                                /* 比較結果を T ビットに設定*/
result[1] = addc(data1[1], data2[1]);       /* 比較結果が加算される*/
result[0] = addc(data1[0], data2[0]);       /* 前の演算結果を反映*/
```

キャリー付き加算

int ovf_addc (long data1, long data2)

説　明　2 つのデータと T ビットを加算し、キャリーを参照します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　加算データ 1
　　data2　加算データ 2

リターン値　キャリー

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     if (ovf_addc(data1,data2)) {
          result = 0;
     }
}
```

備　考　本組み込み関数は T ビットの内容を参照しますので、直前には正しい T ビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われません。

```
if (a) {...}                          /* 比較結果を T ビットに設定*/
if (ovf_addc(data1,data2)) {          /* 比較結果が加算される*/
```

キャリー付き加算

long addv (long data1, long data2)

説　明　2 つのデータを加算し、キャリーを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　加算データ 1
　　data2　加算データ 2

リターン値　加算結果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     result = addv(data1,data2);
}
```

キャリー付き加算

int ovf_addv (long data1, long data2)

説　明　2 つのデータを加算し、キャリーを参照します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　加算データ 1
　　data2　加算データ 2

リターン値　キャリー

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     if (ovf_addv(data1,data2)) {
          result = 0;
     }
}
```

ボロー付き減算

long subc (long data1, long data2)

説　明　data1 から data2 と T ビットを減算し、ボローを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　減算データ 1
　　data2　減算データ 2

リターン値　減算結果

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     result = subc(data1,data2);
}
```

備　考　本組み込み関数は T ビットの内容を参照しますので、直前には正しい T ビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われません。

```
if (a) {...}                                /* 比較結果を T ビットに設定*/
result[0] = subc(data1[0], data2[0]);       /* 比較結果が減算される*/
result[1] = subc(data1[1], data2[1]);       /* 前の演算結果を反映*/
```

ボロー付き減算

int unf_subc (long data1, long data2)

説　明　data1 から data2 と T ビットを減算し、ボローを参照します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　減算データ 1
　　data2　減算データ 2

リターン値　ボロー

例

```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     if (unf_subc(data1,data2)) {
          result = 0;
     }
}
```

備　考　本組み込み関数は T ビットの内容を参照しますので、直前には正しい T ビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われません。

```
if (a) {...}                          /* 比較結果を T ビットに設定*/
if (unf_subc(data1,data2)) {          /* 比較結果が減算される*/
```

ボロー付き減算

long subv (long data1, long data2)

説　明　data1 から data2 を減算し、ボローを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　減算データ 1
　　data2　減算データ 2

リターン値　減算結果

例
```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     result = subv(data1,data2);
}
```

ボロー付き減算

int unf_subv (long data1, long data2)

説　明　data1 から data2 を減算し、ボローを参照します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　減算データ 1
　　data2　減算データ 2

リターン値　ボロー

例
```
#include <machine.h>
extern long result,data1,data2;
void main()
{
     if (unf_subv(data1,data2)) {
          result = 0;
     }
}
```

符号反転

long negc (long data)

説　明　0 から data と T ビットを減算し、ボローを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　符号反転結果

例
```
#include <machine.h>
extern long result,data;
void main()
{
     result = negc(data);
}
```

備　考　本組み込み関数は T ビットの内容を参照しますので、直前には正しい T ビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われません。
```
 if (a) {...}
 result[0] = negc(data[0]);          /* 比較結果が減算される*/
 result[1] = negc(data[1]);          /* 前の演算結果を反映*/
```

1 ビット除算

unsigned long div1 (unsigned long data1, unsigned long data2)

説　明　data1/data2 の 1 ステップ除算を行い、結果を T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　被除数
　　data2　除数

リターン値　被除数の更新値

例
```
#include <machine.h>
extern unsigned long data1,data2;
void main(void)
{
     div0u();
     data1 = div1(data1,data2);
}
```

備　考　本関数を繰り返し使用することで除算を行うことができますが、繰り返しの間は M、Q、T ビットを書き換えないでください(比較やシフトなどを記述しても T ビットを書き換えます)。
本関数の直前で必ず div0s()または div0u()を使用して M、Q、T ビットの初期化を行ってください。

1 ビット除算

int div0s (long data1, long data2)

説　明　data1/data2 の符号付き除算の初期設定をし、T ビットを参照します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data1　被除数
　　data2　除数

リターン値　T ビットの値

例
```
#include <machine.h>
extern long data1,data2;
void main(void)
{
     (void)div0s(data1,data2);
     data1 = div1(data1,data2);
}
```

1 ビット除算

void div0u (void)

説　明　符号なし除算の初期設定をします。

ヘッダ　<machine.h>または<umachine.h>

例
```
#include <machine.h>
extern unsigned long data1,data2;
void main(void)
{
     div0u();
     data1 = div1(data1,data2);
}
```

回転

unsigned long rotl (unsigned long data)

説　明　データを左方向に 1 ビットローテートします。
オペランドの外へ出たビットを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　左方向に 1 ビットローテートした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = rotl(data);
}
```

回転

unsigned long rotr (unsigned long data)

説　明　データを右方向に 1 ビットローテートします。
オペランドの外へ出たビットを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　右方向に 1 ビットローテートした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = rotr(data);
}
```

回転

unsigned long rotcl (unsigned long data)

説　明　データを左方向にＴビットを含めて１ビットローテートします。
オペランドの外へ出たビットをＴビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　　データ

リターン値　左方向に１ビットローテートした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = rotcl(data);
}
```

備　考　本組み込み関数はＴビットの内容を参照しますので、直前には正しいＴビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果がＴビットに反映されるため、本関数の動作が正しく行われません。

```
  if (a) {...}                          /* 比較結果をＴビットに設定*/
  result[1] = rotcl(data[1]);           /* 比較結果がローテートされる*/
```

回転

unsigned long rotcr (unsigned long data)

説　明　データを右方向にＴビットを含めて１ビットローテートします。
オペランドの外へ出たビットをＴビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　　データ

リターン値　右方向に１ビットローテートした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = rotcr(data);
}
```

備　考　本組み込み関数はＴビットの内容を参照しますので、直前には正しいＴビットの値になるように記述してください。直前に比較やシフトなどを記述した場合、その演算結果がＴビットに反映されるため、本関数の動作が正しく行われません。

```
  if (a) {...}                          /* 比較結果をＴビットに設定*/
  result[1] = rotcr(data[1]);           /* 比較結果がローテートされる*/
```

左シフト

unsigned long shll (unsigned long data)

説　明　データを 1 ビット左シフトします。
オペランドの外へ出たビットを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　1 ビット左シフトした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = shll(data);
}
```

右シフト

unsigned long shlr (unsigned long data)

説　明　データを論理的に 1 ビット右シフトします。
オペランドの外へ出たビットを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　1 ビット右シフトした結果

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = shlr(data);
}
```

右シフト

long shar (long data)

説　明　データを算術的に 1 ビット右シフトします。
オペランドの外へ出たビットを T ビットに反映します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値　1 ビット右シフトした結果

例
```
#include <machine.h>
extern long result,data;
void main()
{
     result = shar(data);
}
```

飽和演算

long clipsb (long data)

説　明　データが-128～127 の範囲内の場合はその値を、範囲外の場合は上限値もしくは下限値を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値
```
-128 (data < -128)
data (-128 <= data <= 127)
 127 ( 127 <  data)
```

例
```
#include <machine.h>
extern long result,data;
void main()
{
     result = clipsb(data);
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

飽和演算

long clipsw (long data)

説　明　データが-32768～32767 の範囲内の場合はその値を、範囲外の場合は上限値もしくは下限値を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値

```
-32768 ( data < -32768)
  data (-32768 <= data <= 32767)
 32767 (32767 <  data)
```

例

```
#include <machine.h>
extern long result,data;
void main()
{
     result = clipsw(data);
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

飽和演算

unsigned long clipub (unsigned long data)

説　明　データが 0～255 の範囲内の場合はその値を、範囲外の場合は上限値を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　データ

リターン値

```
data (data <= 255)
 255 ( 255 <  data)
```

例

```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = clipub(data);
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

飽和演算

unsigned long clipuw (unsigned long data)

説　明　データが 0～65535 の範囲内の場合はその値を、範囲外の場合は上限値を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　　データ

リターン値　data　(data <= 65535)
65535 (65535 < data)

例
```
#include <machine.h>
extern unsigned long result,data;
void main()
{
     result = clipuw(data);
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

TBR 設定

*void set_tbr (void *data)*

説　明　TBR に data を設定します。

ヘッダ　<machine.h>または<umachine.h>

引　数　data　　設定値

例
```
#include <machine.h>
void *data;
void main(void)
{
     set_tbr(data);
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

TBR 参照

*void *get_tbr (void)*

説　明　TBR の値を参照します。

ヘッダ　<machine.h>または<umachine.h>

リターン値　TBR の値

例

```
#include <machine.h>
void *result;
void main(void)
{
     result = get_tbr();
}
```

備　考　本関数は cpu=sh2a|sh2afpu 指定時のみ有効です。

多重割り込み

*void *sr_jsr (void *func, int imask)*

説　明　SR レジスタの RB ビット、BL ビットをクリアし、割り込みマスクを imask に設定したあとに func 関数を呼び出します。imask の指定可能範囲は 0 から 15 です。ただし、0 を指定した場合は割り込みマスクの設定は行わず、RB ビット、BL ビットのクリアのみを行います。

ヘッダ　<machine.h>または<smachine.h>

リターン値　なし

例

```
#include <machine.h>
#pragma interrupt func1(bank)
extern void func2(void);
void func1(void)
{
     sr_jsr(func2, 15);
}

_func1:
        MOV.L      R14,@-R15
        STS.L      PR,@-R15
        STC        SSR,@-R15
        STC        SPC,@-R15
        STC        SR,R4
        MOV.L      L11,R1     ; H'CFFFFF0F
        MOV        #-16,R5    ; H'FFFFFFF0
        AND        R1,R4      ; RB, BL ビットをクリア
        EXTU.B     R5,R5
        MOV.L      L11+4,R14  ; _func2
        OR         R5,R4      ; 割り込みマスクを 15 に設定
        LDC        R4,SR
```

```
JSR        @R14
NOP
LDC        @R15+,SPC
LDC        @R15+,SSR
LDS.L      @R15+,PR
MOV.L      @R15+,R14
RTE
NOP
```

備　考　cpu=sh3|sh3dsp|sh4|sh4a|sh4aldsp を指定した場合のみ使用可能です。
func に、引数またはリターン値を持つ関数、またはその関数ポインタ以外を指定することはできません。
sr_jsr() 使用時に、#pragma global_register で、R8～R14 の全てを指定した場合エラーとなります。
sr_jsr() を、割り込み仕様 bank 指定した関数以外で使用した場合エラーとなります。
0 が設定された変数を imask に指定した場合、割り込みマスクは 0 に設定されます。

メモリ上のビット操作

*void bset(unsigned char *addr, unsigned char bit_num)*

説　明　指定アドレス(addr)の指定ビット(bit_num)に 1 を設定します。bit_num の指定可能範囲は 0～7 です。

ヘッダ　<machine.h>または<umachine.h>

引　数　*addr　指定アドレス
bit_num　指定ビット

例

```
#include <machine.h>
void func1(void)
{
   bset((unsigned char *)(0xfffe3886),0);
}
```

コンパイル結果：

```
MOVI20  #-116602,R14   ; H'FFFE3886
BSET.B  #0,@(0,R14)
```

備　考　cpu=sh2a|sh2afpu を指定した場合のみ使用可能です。

メモリ上のビット操作

*void bclr(unsigned char *addr, unsigned char bit_num)*

説　明　　指定アドレス(addr)の指定ビット(bit_num)に 0 を設定します。bit_num の指定可能範囲は 0～7 です。

ヘッダ　　<machine.h>または<umachine.h>

引　数　　*addr　　指定アドレス
bit_num　指定ビット

例

```
#include <machine.h>
void func1(void)
{
    bclr((unsigned char *)(0xfffe3886),0);
}
```

コンパイル結果：

```
MOVI20  #-116602,R14   ; H'FFFE3886
BCLR.B  #0,@(0,R14)
```

備　考　　cpu=sh2a|sh2afpu を指定した場合のみ使用可能です。

メモリ上のビット操作

*void bcopy(unsigned char *from_addr, unsigned char from_bit_num, unsigned char *to_addr, unsigned char to_bit_num)*

説　明　　指定アドレス１(from_addr)の指定ビット１(from_bit_num)を、T ビットと指定アドレス２(to_addr)の指定ビット２(to_bit_num)に設定します。from_bit_num および to_bit_num の指定可能範囲は 0～7 です。

ヘッダ　　<machine.h>または<umachine.h>

引　数　　*from_addr　指定アドレス１(コピー元)
from_bit_num　指定ビット１(コピー元)
*to_addr　指定アドレス２(コピー先)
to_bit_num　指定ビット２(コピー先)

例　　異なるアドレスの異なるビットをコピーする

```
#include <machine.h>
void func1(void)
{
    bcopy((unsigned char *)(0xfffe3886),
          0,
          (unsigned char *)(0xfffd3886),
          1);
}
```

コンパイル結果：

```
MOVI20    #-116602,R14   ; H'FFFE3886
BLD.B     #0,@(0,R14)
MOVI20    #-182138,R14   ; H'FFFD3886
BST.B     #1,@(0,R14)
RTS/N
```

同じアドレスの異なるビットをコピーする

```
#include <machine.h>
void func1(void)
{
    bcopy((unsigned char *)(0xfffe3886),
          0,
          (unsigned char *)(0xfffe3886),
          1);
}
```

コンパイル結果：

```
MOVI20    #-116602,R14   ; H'FFFE3886
BLD.B     #0,@(0,R14)
BST.B     #1,@(0,R14)
RTS/N
```

備　考　　cpu=sh2a|sh2afpu を指定した場合のみ使用可能です。

メモリ上のビット操作

*void bnotcopy(unsigned char *from_addr, unsigned char from_bit_num, unsigned char *to_addr, unsigned char to_bit_num)*

説　明　　指定アドレス１(from_addr)の指定ビット１(from_bit_num)を反転した値を、T ビットと指定アドレス２(to_addr)の指定ビット２(to_bit_num)に設定します。from_bit_num および to_bit_num の指定可能範囲は 0～7 です。

ヘッダ　　<machine.h>または<umachine.h>

引　数　　*from_addr　指定アドレス１(コピー元)
from_bit_num　指定ビット１(コピー元)
*to_addr　指定アドレス２(コピー先)
to_bit_num　指定ビット２(コピー先)

例　　異なるアドレスの異なるビットを反転してコピーする

```
#include <machine.h>
void func1(void)
{
    bnotcopy((unsigned char *)(0xfffe3886),
             0,
             (unsigned char *)(0xfffd3886),
             1);
}
```

コンパイル結果：

```
MOVI20     #-116602,R14   ; H'FFFE3886
BLDNOT.B   #0,@(0,R14)
MOVI20     #-182138,R14   ; H'FFFD3886
BST.B      #1,@(0,R14)
RTS/N
```

特定のアドレスの特定のビットを反転する

```
#include <machine.h>
void func1(void)
{
    bnotcopy((unsigned char *)(0xfffe3886),
             1,
             (unsigned char *)(0xfffe3886),
             1);
}
```

コンパイル結果：

```
MOVI20     #-116602,R14   ; H'FFFE3886
BLDNOT.B   #1,@(0,R14)
BST.B      #1,@(0,R14)
RTS/N
```

備　考　　cpu=sh2a|sh2afpu を指定した場合のみ使用可能です。

10.4　C/C++ライブラリ

10.4.1　標準 C ライブラリ

(1)　ライブラリの概要

C/C++言語の中で標準的に利用できる C ライブラリ関数の仕様について説明します。ここでは、ライブラリの構成を概説し、本節の読み方および用語について説明します。以降ではライブラリの構成に従って各ライブラリ関数の仕様を説明します。

(a)　ライブラリの種類

ライブラリとは、入出力、文字列操作等の標準的な処理を C/C++言語の関数の形式で実現したものです。また、これらのライブラリは、各処理単位ごとに対応した標準インクルードファイルを取り込むことによって使用可能となります。

標準インクルードファイルには、対応するライブラリの宣言とそれらを使用するために必要なマクロ名が定義されています。

表 10.36 にライブラリの種類と対応する標準インクルードファイルを示します。

表 10.36　ライブラリの種類と対応する標準インクルードファイル

	ライブラリの種類	内容	標準インクルードファイル
1	プログラム診断用ライブラリ	プログラムの診断情報の出力を行うライブラリです。	<assert.h>
2	文字操作用ライブラリ	文字の操作およびチェックを行うライブラリです。	<ctype.h>
3	数値計算用ライブラリ	三角関数等の数値計算を行うライブラリです。	<math.h> <mathf.h>
4	プログラムの制御移動用ライブラリ	関数間の制御の移動をサポートするライブラリです。	<setjmp.h>
5	可変個の実引数アクセス用ライブラリ	可変個の実引数を持つ関数に対し、その実引数へのアクセスをサポートするライブラリです。	<stdarg.h>
6	入出力用ライブラリ	入出力操作を行うライブラリです。	<stdio.h>
7	標準処理用ライブラリ	記憶域管理等の C プログラムでの標準的処理を行うライブラリです。	<stdlib.h>
8	文字列操作用ライブラリ	文字列の比較、複写等を行うライブラリです。	<string.h>

また、以上の標準インクルードファイルの他にプログラムの作成作業の効率向上を図るため表 10.37 に示すマクロ名の定義だけからなる標準インクルードファイルがあります。

表 10.37　マクロ名定義からなる標準インクルードファイル

	標準インクルードファイル	内容
1	<stddef.h>	各標準インクルードファイルで共通に使用するマクロ名を定義します。
2	<float.h>	浮動小数点型の内部表現に関する各種制限値を定義します。
3	<limits.h>	コンパイラの内部処理に関する各種制限値を定義します。
4	<errno.h>	ライブラリ関数においてエラーが発生した時に errno に設定する値を定義します。
5	<fixed.h>	固定小数点数の内部表現に関する各種制限値を定義します。

(b) ライブラリの説明形式

ライブラリの各関数を標準インクルードファイルごとに分類し、その標準インクルードファイルごとに説明します。その各分類は、まず、標準インクルードファイルの中で定義されているマクロ名や関数宣言に対する説明を行い(図 10.4参照)、その後、各関数ごとの説明を行う(図 10.5参照)という形式で構成されています。

図 10.4に標準インクルードファイルの説明形式、図 10.5に関数の説明形式を示します。

項番 <標準インクルードファイル名>

・本インクルードファイルがもつ全体的な機能の概要を説明します。

・本インクルードファイル内で定義・宣言される名前を名前種別(【型】、【定数】、【変数】、【関数】)に分類して説明します。マクロである場合、名前種別のタイトル(【】内)または名前の説明箇所に(マクロ)と表記しています。

・処理系定義仕様がある場合や、本インクルードファイル内で宣言されている関数に共通する注意事項がある場合、説明を補足します。

図 10.4 標準インクルードファイルの説明形式

機能の概要を示します。

ライブラリ関数の型(リターン値および引数)を示します。

説 明	ライブラリ関数の機能を説明します。
ヘッダ	宣言元の標準インクルードファイル名です。
リターン値	正常： ライブラリ関数が正常終了したときの値です。 異常： ライブラリ関数が異常終了したときの値です。
引 数	引数の意味を説明します。
例	呼び出し手順を説明します。
エラー条件	ライブラリ関数の処理でリターン値からでは、判断できないエラーが発生する条件を示します。このようなエラーが発生したとき、エラーの種類に対応する、コンパイラごとに定義された値が **errno*** に設定されます。
備 考	補足説明、または使用上の注意事項です。
処理系定義仕様	本コンパイラの処理方法です。

図 10.5 関数の説明形式

【注】* errno は、ライブラリ関数実行中に生じたエラーの種類を格納する変数です。詳細については「10.4.1(2) <stddef.h>」を参照してください。

(c) ライブラリ関数の説明で使用する用語

(i) ストリーム入出力

データの入出力において、1 文字ごとの入出力関数の呼び出しのたびに入出力装置を駆動したり、OS の機能を呼び出していたのでは、効率が悪くなります。そこで、通常はバッファと呼ばれる記憶域を用意しておき、バッファ内のデータを一括して入出力を行います。

一方、プログラムの側から見ると、1 文字ごとに入出力関数を呼び出せた方が便利です。

ライブラリ関数では、バッファの管理を自動的に行うことにより、プログラム内でバッファの状態を意識することなしに、1 文字単位ごとの入出力を効率よく行うことができます。

このように、データの入出力を効率よく実現するために詳細な手段を意識せず、入出力をひとつのデータの流れ(ストリーム)と考えてプログラムを作ることのできる機能をストリーム入出力といいます。

(ii) FILE 構造体およびファイルポインタ

ストリーム入出力に必要なバッファやその他の情報は、一つの構造体の中に記憶されており、標準インクルードファイル<stdio.h>の中で FILE という名前で定義されています。

ストリーム入出力においては、ファイルはすべて FILE 構造体のデータ構造を持つものとして扱います。このようなファイルをストリームファイルと呼びます。このファイル構造体へのポインタをファイルポインタと呼び、入出力ファイルを指定するために用います。

ファイルポインタは、

```
FILE *fp;
```

と定義します。

fopen 関数等でファイルをオープンすると、ファイルポインタが得られますが、オープン処理が失敗すると NULL が返ってきます。NULL ポインタを、他のストリーム入出力関数に指定すると、その関数は異常終了しますので、注意が必要です。ファイルをオープンした時は、必ずファイルポインタの値をチェックするようにしてください。

(iii) 関数とマクロ

ライブラリ関数の実現方法としては、関数とマクロの二通りがあります。

関数は、通常のユーザ作成の関数と同じインタフェースを持ち、リンク時に取り込みます。

マクロは、その関数に関連した標準インクルードファイルの中で#define 文を用いて定義されています。

マクロについては、以下の点に注意する必要があります。

- マクロは、プリプロセッサによって自動的に展開されてしまうので、ユーザが同じ名前の関数を宣言してもマクロを無効にすることはできません。
- マクロのパラメータとして副作用のある式(代入式、インクリメント、デクリメント)を指定した時、その効果は保証しません。

例：

パラメータの絶対値を求める MACRO を以下のようにマクロ定義します。

```
#define MACRO(a) ((a)>=0?(a):-(a))
```

と定義されている時、

```
X=MACRO(a++)
```

がプログラム内にあると、

```
X=((a++)>=0?(a++):-(a++))
```

と展開され、a は 2 回インクリメントされることになり、また結果の値も最初の a の値の絶対値とは異なります。

(iv) EOF

getc 関数、getchar 関数、fgetc 関数等のファイルからデータを入力する関数において、ファイル終了(End Of File)時に返される値です。EOF は、標準インクルードファイル<stdio.h>の中で定義されています。

(v) NULL

ポインタが何も指していない時の値です。NULL は、標準インクルードファイル<stddef.h>の中で定義されています。

(vi) ヌル文字

C/C++言語における文字列の終わりは、文字"¥0"によって示されることになっています。
ライブラリ関数における文字列のパラメータも、すべてこの約束に従っていなければなりません。
この文字列の終わりを示す文字"¥0"を以下ヌル文字と呼びます。

(vii) リターンコード

ライブラリ関数の中には、リターン値によって、指定された処理が成功したか、失敗したか等の結果を判断するものがあります。
このような場合のリターン値を特にリターンコードと呼びます。

(viii) テキストファイルとバイナリファイル

多くのシステムでは、データを格納するために特殊なファイル形式を持っています。
これをサポートするために、ライブラリ関数にはテキストファイルとバイナリファイルの 2 種類のファイル形式があります。

- テキストファイル
  テキストファイルは、通常のテキストを格納するためのファイルで、行の集まりとして構成されています。テキストファイルの入力の時、行の区切りとして改行文字("¥n")が入力されます。また、出力の時、改行文字を出力することにより、現在の行の出力を終了します。テキストファイルは、処理系ごとの標準的なテキストを格納するファイルの入出力を行うためのファイルです。テキストファイルでは、ライブラリ関数で入出力する文字は必ずしもファイル内の物理的なデータの並びと対応していません。

- バイナリファイル
  バイナリファイルは、バイトデータの列として構成されているファイルです。ライブラリ関数で入出力するデータは、ファイル内の物理的なデータの並びと対応しています。

(ix) 標準入出力ファイル

入出力のライブラリ関数で、ファイルのオープン等の準備を行わずに標準的に使用できるファイルを標準入出力ファイルといいます。標準入出力ファイルには、標準入力ファイル(stdin)、標準出力ファイル(stdout)、標準エラー出力ファイル(stderr)があります。

- 標準入力ファイル (stdin)
  プログラムへの入力となる標準的なファイルです。

- 標準出力ファイル (stdout)
  プログラムからの出力となる標準的なファイルです。

- 標準エラー出力ファイル (stderr)
  プログラムからのエラーメッセージ等の出力を行うための標準的なファイルです。

(x) **浮動小数点型**

浮動小数点型は、実数を近似して表現したものです。C/C++言語のソースプログラム上では浮動小数点型を 10 進数で表現していますが、計算機の内部では通常 2 進数で表現されます。
2 進数の場合の浮動小数点型の表現は次のようになります。

$2^n \times m$　(n：整数、m：2 進小数)

ここで n を浮動小数点型の指数部、m を仮数部といいます。浮動小数点型を一定のデータサイズで表現するために、n と m のビット数は通常固定されています。
以下、浮動小数点型に関する用語を説明します。

- 基数
  浮動小数点型が何進数で表現されているかを示す整数値です。通常、基数は2です。

- 丸め
  浮動小数点型よりも精度の高い演算の途中結果を浮動小数点型に格納する場合に丸めが行われます。丸めには、切り上げ、切り捨て、四捨五入(2進小数の場合は、0捨1入となります)があります。

- 正規化
  浮動小数点型を、$2^n \times m$の形式で表現する場合、同一の数値を表す異なる表現が可能です。

例：

$2^5 \times 1.0_{(2)}$　($_{(2)}$は 2 進数を示します)
$2^6 \times 0.1_{(2)}$
どちらも同じ値です。

通常は、有効桁数を確保するために、先頭の桁が0でないような表現を用います。これを正規化された浮動小数点型といい、浮動小数点型をこのような表現に変換する操作を正規化といいます。

- ガードビット
  浮動小数点型の演算の途中結果を保持する場合、通常は、丸めを行うために実際の浮動小数点型よりも1ビット多いデータを用意します。しかし、これだけでは桁落ち等が生じた時に正確な結果を求めることができません。このために、もう1ビット設けて演算の途中結果を保持する手法があります。このビットをガードビットといいます。

(xi) **ファイルアクセスモード**

ファイルをオープンする時にどのような処理をファイルに行うかを示す文字列です。文字列の種類には**表 10.38** に示す 12 種類があります。

表 10.38　ファイルアクセスモードの種類

アクセスモード		意味
1	"r"	テキストファイルを読み込み用にオープンします。
2	"w"	テキストファイルを書き出し用にオープンします。
3	"a"	テキストファイルを追加用にオープンします。
4	"rb"	バイナリファイルを読み込み用にオープンします。
5	"wb"	バイナリファイルを書き出し用にオープンします。
6	"ab"	バイナリファイルを追加用にオープンします。
7	"r+"	テキストファイルを読み込み用でかつ更新用にオープンします。
8	"w+"	テキストファイルを書き出し用でかつ更新用にオープンします。
9	"a+"	テキストファイルを追加用でかつ更新用にオープンします。
10	"r+b"	バイナリファイルを読み込み用でかつ更新用にオープンします。
11	"w+b"	バイナリファイルを書き出し用でかつ更新用にオープンします。
12	"a+b"	バイナリファイルを追加用でかつ更新用にオープンします。

(xii) **処理系定義**

コンパイラが異なることによって定義が異なるという意味です。

(xiii) **エラー指示子、ファイル終了指示子**

ストリームファイルごとに、ファイルの入出力の際にエラーが生じたかどうかを示すエラー指示子と、入力ファイルが終了したかどうかを示すファイル終了指示子というデータを保持しています。

これらのデータは、それぞれ ferror 関数、feof 関数によって参照することができます。

ストリームファイルを扱う関数のうち、そのリターン値だけからではエラーの発生やファイルの終了の情報が得られないものがあります。エラー指示子とファイル終了指示子は、このような関数の実行後にファイルの状態を調べるために使用することができます。

(xiv) **位置指示子**

ディスク上のファイル等、ファイル内の任意の位置からの読み書きができるストリームファイルは、現在読み書きしているファイル内の位置を示すデータを保持しています。これを位置指示子といいます。端末装置等、ファイル内の読み書きの位置を変更できないストリームファイルでは、位置指示子は使用しません。

(d) **ライブラリ使用時の注意事項**

ライブラリの中で定義されているマクロの内容は、コンパイラごとに異なります。

ライブラリを使用する場合、これらのマクロの内容を再定義した場合、動作は保証しません。

ライブラリは、すべての場合についてエラーを検出しているわけではありません。以降の説明に示す以外の形式でライブラリ関数を呼び出した場合、動作は保証しません。

(2)　<stddef.h>

標準インクルードファイルの中で共通に使用されるマクロ名を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
型 (マクロ)	ptrdiff_t	二つのポインタを減算した結果の型です。
	size_t	sizeof 演算子による演算結果の型です。
定数 (マクロ)	NULL	ポインタが何も指していない時の値です。 これは、0 と等値演算子(==)による比較結果が真になるような値です。
変数 (マクロ)	errno	ライブラリ関数の処理中にエラーが発生した場合、そのライブラリごとに定義されたエラーコードがこの errno に設定されます。 ライブラリ関数を呼び出す前に errno に 0 を設定しておき、ライブラリ関数の処理終了後に errno に設定されているコードを調べることによってライブラリ関数の処理中に発生したエラーをチェックすることができます。
関数 (マクロ)	offsetof	構造体メンバの構造体先頭からのオフセット値をバイト単位で求めます。

処理系定義仕様

項目	
1　マクロ NULL の値	void 型へのポインタ型の値 0 です
2　マクロ ptrdiff_t の内容	int 型

(3) <assert.h>

プログラム中に診断機能を付け加えます。

種別	定義名	説明
関数(マクロ)	assert	プログラム中に診断機能を付け加えます。

<assert.h>で定義される診断機能を無効にするためには、<assert.h>を取り込む前に NDEBUG というマクロ名を#define 文で定義してください(#define NDEBUG)。

【注】 assert というマクロ名に対して#undef 文を使用すると、それ以降の assert の呼び出しの効果は保証しません。

診断

void assert(int expression)

説　明　　プログラム中に診断機能を付け加えます。

ヘッダ　　<assert.h>

引　数　　expression　　評価する式

例

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

備　考　　assert マクロは、expression が真の時は値を返さずに処理を終了します。expression が偽の時は、診断情報をコンパイラによって定義された書式で標準エラーファイルに出力し、その後 abort 関数を呼び出します。
診断情報の中には、パラメータのプログラムテキスト、ソースファイル名、ソース行番号が含まれています。

処理系定義仕様　　assert(expression)において、expression が偽の時メッセージを出力します。
ASSERTION FAILED:Δ式ΔFILEΔ<ファイル名>,lineΔ<行番号>

(4)　<ctype.h>

文字に対して、その種類の判定や変換を行います。

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

上記の関数において、入力パラメータの値が unsigned char 型で表現できる範囲に含まれず、なおかつ EOF でない場合、その関数の動作は保証しません。

文字の種類の一覧を表 10.39 に示します。

表 10.39　文字の種類

	文字の種類	内容
1	英大文字	以下の 26 文字のいずれかの文字です。 'A'、'B'、'C'、'D'、'E'、'F'、'G'、'H'、'I'、'J'、'K'、'L'、'M'、 'N'、'O'、'P'、'Q'、'R'、'S'、'T'、'U'、'V'、'W'、'X'、'Y'、'Z'
2	英小文字	以下の 26 文字のいずれかの文字です。 'a'、'b'、'c'、'd'、'e'、'f'、'g'、'h'、'i'、'j'、'k'、'l'、'm'、 'n'、'o'、'p'、'q'、'r'、's'、't'、'u'、'v'、'w'、'x'、'y'、'z'
3	英字	英大文字と英小文字のいずれかの文字です。
4	10 進数字	以下の 10 文字のいずれかの文字です。 '0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'
5	印字文字	空白(' ')を含む、ディスプレイ上に表示される文字のことです。 ASCII コードの 0x20～0x7E に対応します。
6	制御文字	印字文字以外の文字のことです。
7	空白類文字	以下の 6 文字のいずれかの文字です。 空白(' ')、書式送り('¥f')、改行('¥n')、復帰('¥r')、水平タブ('¥t')、垂直タブ('¥v')
8	16 進数字	以下の 22 文字のいずれかの文字です。 '0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'、 'A'、'B'、'C'、'D'、'E'、'F'、'a'、'b'、'c'、'd'、'e'、'f'
9	特殊文字	空白(' ')、英字、および 10 進数字を除く任意の印字文字のことです。

処理系定義仕様

	項目	コンパイラの仕様
1	isalnum 関数、isalpha 関数、iscntrl 関数、islower 関数、isprint 関数、isupper 関数で判定される文字集合	unsigned char 型で表現できる文字集合です。判定の結果、真になる文字を表 10.40 に示します。

表 10.40 真となる文字の集合

	関数名	真となる文字
1	isalnum	'0'～'9', 'A'～'Z', 'a'～'z'
2	isalpha	'A'～'Z', 'a'～'z'
3	iscntrl	'¥x00'～'¥x1f', '¥x7f'
4	islower	'a'～'z'
5	isprint	'¥x20'～'¥x7E'
6	isupper	'A'～'Z'

英字、10 進数字判定

int isalnum(int c)

説　明　　文字が英字または 10 進数字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英字または 10 進数字の時　　：0 以外
文字 c が英字または 10 進数字以外の時　　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
   ret=isalnum(c);
```

英字判定

int isalpha(int c)

説　明　　文字が英字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英字の時　　：0 以外
文字 c が英字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
   ret=isalpha(c);
```

制御文字判定

int iscntrl(int c)

説　明　　文字が制御文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が制御文字の時　　：0 以外
文字 c が制御文字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
   ret=iscntrl(c);
```

10 進数字判定

int isdigit(int c)

説　明　　文字が 10 進数字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が 10 進数字の時　　：0 以外
文字 c が 10 進数字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=isdigit(c);
```

空白を除く印字文字判定

int isgraph(int c)

説　明　　文字が空白(' ')を除く任意の印字文字かどうかを判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が空白を除く印字文字の時　　：0 以外
文字 c が空白を除く印字文字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=isgraph(c);
```

英小文字判定

int islower(int c)

説　明　　文字が英小文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英小文字の時　　：0 以外
文字 c が英小文字以外の時　　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
   ret=islower(c);
```

印字文字判定

int isprint(int c)

説　明　　文字が空白文字(' ')を含む印字文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が空白文字を含む印字文字の時　　：0 以外
文字 c が空白文字を含む印字文字以外の時　　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
   ret=isprint(c);
```

特殊文字判定

int ispunct(int c)

説　明　　文字が特殊文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が特殊文字の時　　：0 以外
文字 c が特殊文字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=ispunct(c);
```

空白類文字判定

int isspace(int c)

説　明　　文字が空白類文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が空白類文字の時　　：0 以外
文字 c が空白類文字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=isspace(c);
```

英大文字判定

int isupper(int c)

説　明　　文字が英大文字であるかどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英大文字の時　　：0 以外
文字 c が英大文字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=isupper(c);
```

16 進数字判定

int isxdigit(int c)

説　明　　文字が 16 進数字かどうか判定します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が 16 進数字の時　　：0 以外
文字 c が 16 進数字以外の時　：0

引　数　　c　　判定する文字

例

```
#include <ctype.h>
int c, ret;
    ret=isxdigit(c);
```

英小文字変換

int tolower(int c)

説　明　　英大文字を対応する英小文字に変換します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英大文字の時　　：文字 c に対応する英小文字
文字 c が英大文字以外の時　：文字 c

引　数　　c　　変換する文字

例

```
#include <ctype.h>
int c, ret;
    ret=tolower(c);
```

英大文字変換

int toupper(int c)

説　明　　英小文字を対応する英大文字に変換します。

ヘッダ　　<ctype.h>

リターン値　　文字 c が英小文字の時　　：文字 c に対応する英大文字
文字 c が英小文字以外の時　：文字 c

引　数　　c　　変換する文字

例

```
#include <ctype.h>
int c, ret;
    ret=toupper(c);
```

(5)　<float.h>

浮動小数点型の内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数 (マクロ)	FLT_RADIX	2	指数部表現における基数です。
	FLT_ROUNDS	1	加算演算結果を丸めるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・演算結果を丸める場合　：正の値 ・演算結果を切り捨てる場合：0 ・特に規定しない場合　：-1 丸め、切り捨ての方法は、処理系定義です。
	FLT_GUARD	1	乗算演算結果においてガードビットを用いるかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・ガードビットを用いる場合　：1 ・ガードビットを用いない場合　：0
	FLT_NORMALIZE	1	浮動小数点値を正規化するかどうかを示します。本マクロの定義の意味は以下のとおりです。 ・正規化する場合　：1 ・正規化しない場合　：0
	FLT_MAX	3.4028235677973364e+38F	float 型が浮動小数点値として表現できる最大値です。
	DBL_MAX	1.7976931348623158e+308	double 型が浮動小数点値として表現できる最大値です。
	LDBL_MAX	1.7976931348623158e+308	long double 型が浮動小数点値として表現できる最大値です。
	FLT_MAX_EXP	127	float 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	DBL_MAX_EXP	1023	double 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	LDBL_MAX_EXP	1023	long double 型が浮動小数点値として表現できる基数のべき乗の最大値です。
	FLT_MAX_10_EXP	38	float 型が浮動小数点値として表現できる 10 のべき乗の最大値です。
	DBL_MAX_10_EXP	308	double 型が浮動小数点値として表現できる 10 のべき乗の最大値です。
	LDBL_MAX_10_EXP	308	long double 型が浮動小数点値として表現できる 10 のべき乗の最大値です。
	FLT_MIN	• -cpu=sh4|sh4a かつ -denormalize=off の場合 1.1754943508222875e-38F • 上記以外の場合 1.4012984643248171e-45F	float 型が浮動小数点値として表現できる正の値での最小値です。

種別	定義名	定義値	説明
定数 (マクロ)	DBL_MIN	• -cpu=sh4|sh4a かつ -denormalize=off の場合 2.2250738585072014e-308 • 上記以外の場合 4.9406564584124654e-324	double 型が浮動小数点値として表現できる正の値での最小値です。
	LDBL_MIN	• -cpu=sh4|sh4a かつ -denormalize=off の場合 2.2250738585072014e-308 • 上記以外の場合 4.9406564584124654e-324	long double 型が浮動小数点値として表現できる正の値での最小値です。
	FLT_MIN_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -126 • 上記以外の場合 -149	float 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	DBL_MIN_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -1022 • 上記以外の場合 -1074	double 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	LDBL_MIN_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -1022 • 上記以外の場合 -1074	long double 型が正の値として表現できる浮動小数点値の基数のべき乗の最小値です。
	FLT_MIN_10_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -38 • 上記以外の場合 -44	float 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	DBL_MIN_10_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -308 • 上記以外の場合 -323	double 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	LDBL_MIN_10_EXP	• -cpu=sh4|sh4a かつ -denormalize=off の場合 -308 • 上記以外の場合 -323	long double 型が正の値として表現できる浮動小数点値の 10 のべき乗の最小値です。
	FLT_DIG	6	float 型の浮動小数点値の 10 進精度の最大桁数です。
	DBL_DIG	15	double 型の浮動小数点値の 10 進精度の最大桁数です。
	LDBL_DIG	15	long double 型の浮動小数点値の 10 進精度の最大桁数です。

種別	定義名	定義値	説明
定数 (マクロ)	FLT_MANT_DIG	24	float 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。
	DBL_MANT_DIG	53	double 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。
	LDBL_MANT_DIG	53	long double 型の浮動小数点値を基数に合わせて表現した時の仮数部の最大桁数です。
	FLT_EXP_DIG	8	float 型の浮動小数点値を基数に合わせて表現した時の指数部の最大桁数です。
	DBL_EXP_DIG	11	double 型の浮動小数点値を基数に合わせて表現した時の指数部の最大桁数です。
	LDBL_EXP_DIG	11	long double 型の浮動小数点値を基数に合わせて表現した時の指数部の最大桁数です。
	FLT_POS_EPS FLT_EPSILON	5.9604648328104311e-8F	float 型において、1.0＋x≠1.0 である最小の浮動小数点値 x を示します。
	DBL_POS_EPS DBL_EPSILON	1.1102230246251567e-16	double 型において、1.0＋x≠1.0 である最小の浮動小数点値 x を示します。
	LDBL_POS_EPS LDBL_EPSILON	1.1102230246251567e-16	long double 型において、1.0＋x≠1.0 である最小の浮動小数点値 x を示します。
	FLT_NEG_EPS	2.9802324164052156e-8F	float 型において、1.0－x≠1.0 である最小の浮動小数点値 x を示します。
	DBL_NEG_EPS	5.5511151231257834e-17	double 型において、1.0－x≠1.0 である最小の浮動小数点値 x を示します。
	LDBL_NEG_EPS	5.5511151231257834e-17	long double 型において、1.0－x≠1.0 である最小の浮動小数点値 x を示します。
	FLT_POS_EPS_EXP	-23	float 型において、1.0＋(基数)^n≠1.0 となる最小の整数 n を示します。
	DBL_POS_EPS_EXP	-52	double 型において、1.0＋(基数)^n≠1.0 となる最小の整数 n を示します。
	LDBL_POS_EPS_EXP	-52	long double 型において、1.0＋(基数)^n≠1.0 となる最小の整数 n を示します。
	FLT_NEG_EPS_EXP	-24	float 型において、1.0－(基数)^n≠1.0 となる最小の整数 n を示します。
	DBL_NEG_EPS_EXP	-53	double 型において、1.0－(基数)^n≠1.0 となる最小の整数 n を示します。
	LDBL_NEG_EPS_EXP	-53	long double 型において、1.0－(基数)^n≠1.0 となる最小の整数 n を示します。

(6) <limits.h>

整数型データの内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数(マクロ)	CHAR_BIT	8	char 型が何ビットから構成されるかを示します。
	CHAR_MAX	127	char 型の変数が値として持つことができる最大値です。
	CHAR_MIN	-128	char 型の変数が値として持つことができる最小値です。
	SCHAR_MAX	127	signed char 型の変数が値として持つことができる最大値です。
	SCHAR_MIN	-128	signed char 型の変数が値として持つことができる最小値です。
	UCHAR_MAX	255U	unsigned char 型の変数が値として持つことができる最大値です。
	SHRT_MAX	32767	short 型の変数が値として持つことができる最大値です。
	SHRT_MIN	-32768	short 型の変数が値として持つことができる最小値です。
	USHRT_MAX	65535U	unsigned short 型の変数が値として持つことができる最大値です。
	INT_MAX	2147483647	int 型の変数が値として持つことができる最大値です。
	INT_MIN	-2147483647-1	int 型の変数が値として持つことができる最小値です。
	UINT_MAX	4294967295U	unsigned int 型の変数が値として持つことができる最大値です。
	LONG_MAX	2147483647L	long 型の変数が値として持つことができる最大値です。
	LONG_MIN	-2147483647L-1L	long 型の変数が値として持つことができる最小値です。
	ULONG_MAX	4294967295U	unsigned long 型の変数が値として持つことができる最大値です。
	LLONG_MAX	9223372036854775807LL	long long 型の変数が値として持つことができる最大値です。
	LLONG_MIN	–9223372036854775807LL-1LL	long long 型の変数が値として持つことができる最小値です。
	ULLONG_MAX	18446744073709551615ULL	unsigned long long 型の変数が値として持つことができる最大値です。

(7)　<errno.h>

ライブラリ関数においてエラーが発生したときに errno に設定する値を定義します。

以下は、すべて処理系定義です。

種別	定義名	説明
変数 (マクロ)	errno	int 型変数です。ライブラリ関数においてエラーが発生したときにエラー番号が設定されます。
定数 (マクロ)	ERANGE	「12.3　C 標準ライブラリ関数のエラーメッセージ」を参照してください。
	EDOM	
	ESTRN	
	PTRERR	
	ECBASE	
	ETLN	
	EEXP	
	EEXPN	
	EFLOATO	
	EFLOATU	
	EDBLO	
	EDBLU	
	ELDBLO	
	ELDBLU	
	NOTOPN	
	EBADF	
	ECSPEC	
	EFIXEDO	
	EFIXEDU	
	EACCUMO	
	EACCUMU	
	ELFIXEDO	
	ELFIXEDU	
	ELACCUMO	
	ELACCUMU	
	EMALRESM	
	EMALFRSM	
	ETOKRESM	
	ETOKFRSM	
	EIOBRESM	
	EIOBFRSM	

(8) <fixed.h>

固定小数点数の内部表現に関する各種制限値を定義します。

以下はすべて処理系定義です。

種別	定義名	定義値	説明
定数(マクロ)	FIXED_BIT	16	__fixed 型が何ビットから構成されるかを示します。
	FIXED_MIN	(-0.5r-0.5r)	__fixed 型の変数が値として持つことができる最小値です。
	FIXED_MAX	0.999969482421875r	__fixed 型の変数が値として持つことができる最大値です。
	FIXED_EPSILON	0.000030517578125r	__fixed 型で表現可能な 0.0r より大きい値のうちで最小の値と 0.0r との差です。
	LFIXED_BIT	32	long __fixed 型が何ビットから構成されるかを示します。
	LFIXED_MIN	(-0.5R-0.5R)	long __fixed 型の変数が値として持つことができる最小値です。
	LFIXED_MAX	0.9999999995343387126922607421875R	long __fixed 型の変数が値として持つことができる最大値です。
	LFIXED_EPSILON	0.0000000004656612873077392578125R	long __fixed 型で表現可能な 0.0R より大きい値のうちで最小の値と 0.0R との差です。
	ACCUM_BIT	24	__accum 型が何ビットから構成されるかを示します。
	ACCUM_MIN	(-128.0a-128.0a)	__accum 型の変数が値として持つことができる最小値です。
	ACCUM_MAX	255.999969482421875a	__accum 型の変数が値として持つことができる最大値です。
	ACCUM_EPSILON	0.000030517578125a	__accum 型で表現可能な 0.0a より大きい値のうちで最小の値と 0.0a との差です。
	LACCUM_BIT	40	long __accum 型が何ビットから構成されるかを示します。
	LACCUM_MIN	(-128.0A-128.0A)	long __accum 型の変数が値として持つことができる最小値です。
	LACCUM_MAX	255.9999999995343387126922607421875A	long __accum 型の変数が値として持つことができる最大値です。
	LACCUM_EPSILON	0.0000000004656612873077392578125A	long __accum 型で表現可能な 0.0A より大きい値のうちで最小の値と 0.0A との差です。

(9)　<math.h>

各種の数値計算を行います。

以下の定数(マクロ)はすべて処理系定義です。

種別	定義名	説明
定数(マクロ)	EDOM	関数に入力するパラメータの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が double 型の値として表わせない時、あるいはオーバフロー/アンダフローとなった時、errno に設定する値です。
	HUGE_VAL	関数の計算結果がオーバフローした時に、関数のリターン値として返す値です。
関数	acos	浮動小数点値の逆余弦を計算します。
	asin	浮動小数点値の逆正弦を計算します。
	atan	浮動小数点値の逆正接を計算します。
	atan2	浮動小数点値どうしを除算した結果の値の逆正接を計算します。
	cos	浮動小数点値のラジアン値の余弦を計算します。
	sin	浮動小数点値のラジアン値の正弦を計算します。
	tan	浮動小数点値のラジアン値の正接を計算します。
	cosh	浮動小数点値の双曲線余弦を計算します。
	sinh	浮動小数点値の双曲線正弦を計算します。
	tanh	浮動小数点値の双曲線正接を計算します。
	exp	浮動小数点値の指数関数を計算します。
	frexp	浮動小数点値を[0.5,1.0)の値と 2 のべき乗の積に分解します。
	ldexp	浮動小数点値と 2 のべき乗の乗算を計算します。
	log	浮動小数点値の自然対数を計算します。
	log10	浮動小数点値の 10 を底とする対数を計算します。
	modf	浮動小数点値を整数部分と小数部分に分解します。
	pow	浮動小数点値のべき乗を計算します。
	sqrt	浮動小数点値の正の平方根を計算します。
	ceil	浮動小数点値の小数点以下を切り上げた整数値を求めます。
	fabs	浮動小数点値の絶対値を計算します。
	floor	浮動小数点値の小数点以下を切り捨てた整数値を求めます。
	fmod	浮動小数点値どうしを除算した結果の余りを計算します。

エラーが発生した時の動作を以下に説明します。

(1) 定義域エラー
関数に入力するパラメータの値が関数内で定義している値の範囲を超えている時、定義域エラーが発生します。この時errnoにはEDOMの値が設定されます。また、関数のリターン値は、処理系定義です。

(2) 範囲エラー
関数における計算結果がdouble型の値として表わせない時には範囲エラーが発生します。この時、errnoにはERANGEの値が設定されます。また、計算結果がオーバフローの時は、正しく計算が行われた時と同様の符号のHUGE_VALの値をリターン値として返します。逆に計算結果がアンダフローの時は、0をリターン値として返します。

【注】1. <math.h>の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず errno をチェックしてから用いてください。

例：

```
        .
        .
        .
1   x=asin(a);
2   if (errno==EDOM)
3      printf("error¥n");
4   else
5      printf("result is : %lf¥n",x);
        .
        .
        .
```

1 行目で、asin 関数を使って逆正弦値を求めます。このとき、引数 a の値が、asin 関数の定義域[-1.0, 1.0]の範囲を超えていると、errno に値 EDOM が設定されます。2 行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、3 行目で、error を出力します。定義域エラーが生じなければ 5 行目で、逆正弦値を出力します。

2. 範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点型の内部表現形式によって異なります。例えば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように<math.h>のライブラリ関数を実現することができます。

3. cpu=sh2afpu|sh4|sh4a を指定し、かつ fabs 関数および sqrt 関数を使用した場合、もしくは cpu=sh2e かつ double=float を指定し、かつ fabs 関数を使用した場合は、エラーが発生しても errno は設定されません。

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力引数値が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「10.1.3 浮動小数点型の仕様」を参照してください。
2	数学関数でアンダフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmod 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

逆余弦

double acos(double d)

説　明　浮動小数点値の逆余弦を計算します。

ヘッダ　<math.h>

リターン値　正常：d の逆余弦値
異常：定義域エラーの時は、非数を返します

引　数　d　逆余弦を求める浮動小数点値

例
```
#include <math.h>
double d, ret;
    ret=acos(d);
```

エラー条件　d の値が[-1.0,1.0]の範囲を超えている時、定義域エラーになります。

備　考　acos 関数のリターン値の範囲は[0,π]です。

逆正弦

double asin(double d)

説　明　浮動小数点値の逆正弦を計算します。

ヘッダ　<math.h>

リターン値　正常：d の逆正弦値
異常：定義域エラーの時は、非数を返します

引　数　d　逆正弦を求める浮動小数点値

例
```
#include <math.h>
double d, ret;
    ret=asin(d);
```

エラー条件　d の値が[-1.0,1.0]の範囲を超えている時、定義域エラーになります。

備　考　asin 関数のリターン値の範囲は[-π/2,π/2]です。

逆正接

double atan(double d)

説　明　　浮動小数点値の逆正接を計算します。

ヘッダ　　<math.h>

リターン値　　d の逆正接値

引　数　　d　　逆正接を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
   ret=atan(d);
```

備　考　　atan 関数のリターン値の範囲は(-π/2,π/2)です。

除算後の逆正接

double atan2(double y, double x)

説　明　　浮動小数点値どうしを除算した結果の値の逆正接を計算します。

ヘッダ　　<math.h>

リターン値　　正常：y を x で除算したときの逆正接値
異常：定義域エラーの時は、非数を返します

引　数

x	除数
y	被除数

例

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

エラー条件　　x, y の値がともに 0.0 の時、定義域エラーになります。

備　考　　atan2 関数のリターン値の範囲は(-π,π)です。atan2 関数の示す意味を図 10.6 に示します。図に示すように、atan2 関数の結果は、点(x, y)と原点を通る直線と x 軸をなす角を求めます。y=0.0 で x が負の時、結果はπとなります。
x=0.0 の時、y の値の正負に従って結果は±π/2 となります。但し、マイコンの設定によりゼロ除算例外が発生します。

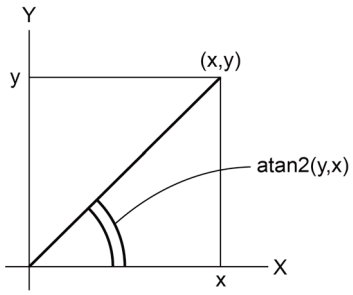


図 10.6　atan2 関数の意味

余弦

double cos(double d)

説　明　　浮動小数点値のラジアン値の余弦を計算します。

ヘッダ　　<math.h>

リターン値　　d の余弦値

引　数　　d　　余弦を求めるラジアン値

例

```
#include <math.h>
double d, ret;
    ret=cos(d);
```

正弦

double sin(double d)

説　明　　浮動小数点値のラジアン値の正弦を計算します。

ヘッダ　　<math.h>

リターン値　　d の正弦値

引　数　　d　　正弦を求めるラジアン値

例

```
#include <math.h>
double d, ret;
    ret=sin(d);
```

正接

double tan(double d)

説　明　浮動小数点値のラジアン値の正接を計算します。

ヘッダ　<math.h>

リターン値　d の正接値

引　数　d　正接を求めるラジアン値

例
```
#include <math.h>
double d, ret;
    ret=tan(d);
```

双曲線余弦

double cosh(double d)

説　明　浮動小数点値の双曲線余弦を計算します。

ヘッダ　<math.h>

リターン値　d の双曲線余弦値

引　数　d　双曲線余弦を求める浮動小数点値

例
```
#include <math.h>
double d, ret;
    ret=cosh(d);
```

双曲線正弦

double sinh(double d)

説　明　　浮動小数点値の双曲線正弦を計算します。

ヘッダ　　<math.h>

リターン値　　d の双曲線正弦値

引　数　　d　　双曲線正弦を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=sinh(d);
```

双曲線正接

double tanh(double d)

説　明　　浮動小数点値の双曲線正接を計算します。

ヘッダ　　<math.h>

リターン値　　d の双曲線正接値

引　数　　d　　双曲線正接を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=tanh(d);
```

指数関数

double exp(double d)

説　明　浮動小数点値の指数関数を計算します。

ヘッダ　<math.h>

リターン値　d の指数関数値

引　数　d　指数関数を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=exp(d);
```

浮動小数点値を仮数、指数に分解

*double frexp(double value, int *exp)*

説　明　浮動小数点値を[0.5,1.0)の値と 2 のべき乗の積に分解します。

ヘッダ　<math.h>

リターン値　value が 0.0 の時　：0.0
value が 0.0 でない時　：ret*2^(exp の指している領域の値)=value で定義される ret の値

引　数　value　[0.5,1.0)の値と 2 のべき乗の積に分解する浮動小数点値
exp　2 のべき乗値を格納する記憶域へのポインタ

例

```
#include <math.h>
double ret, value;
int *exp;
    ret=frexp(value,exp);
```

備　考　frexp 関数は、value を[0.5,1.0)の値と 2 のべき乗の積に分解します。exp の指す領域には、分解した結果の 2 のべき乗値を設定します。
リターン値 ret の値の範囲は[0.5,1.0)または 0.0 になります。
value が 0.0 ならば、exp の指す int 型の記憶域の内容と ret の値は 0.0 になります。

仮数、指数を浮動小数点値に変換

double ldexp(double e, int f)

説　明　　浮動小数点値と 2 のべき乗の積を計算します。

ヘッダ　　<math.h>

リターン値　　$e*2^f$ の演算結果の値

引　数　　e　　2 のべき乗値を求める浮動小数点値
f　　2 のべき乗値

例

```
#include <math.h>
double ret, e;
int f;
   ret=ldexp(e, f);
```

自然対数

double log(double d)

説　明　　浮動小数点値の自然対数を計算します。

ヘッダ　　<math.h>

リターン値　　正常：d の自然対数の値
異常：定義域エラーの時は、非数を返します

引　数　　d　　自然対数を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
   ret=log(d);
```

エラー条件　　d の値が負の時、定義域エラーになります。
d の値が 0.0 の時、範囲エラーになります。

常用対数

double log10(double d)

説　明　浮動小数点値の 10 を底とする対数を計算します。

ヘッダ　<math.h>

リターン値　正常：d は 10 を底とする対数値
異常：定義域エラーの時は、非数を返します

引　数　d　10 を底とする対数を求める浮動小数点値

例
```
#include <math.h>
double d, ret;
    ret=log10(d);
```

エラー条件　d の値が負の値の時、定義域エラーになります。
d の値が 0.0 の時、範囲エラーになります。

浮動小数点値を整数部、小数部に分解

*double modf(double a, double *b)*

説　明　浮動小数点値を整数部分と小数部分に分解します。

ヘッダ　<math.h>

リターン値　a の小数部分

引　数　a　整数部分と小数部分に分解する浮動小数点値
b　整数部分を格納する記憶域を指すポインタ

例
```
#include <math.h>
double a, *b, ret;
    ret=modf(a,b);
```

べき乗

double pow(double x, double y)

説　明　　浮動小数点値のべき乗を計算します。

ヘッダ　　<math.h>

リターン値　　正常：x の y 乗の値
異常：定義域エラーの時は、非数を返します

引　数　　x　　べき乗される値
y　　べき乗する値

例

```
#include <math.h>
double x, y, ret;
    ret=pow(x,y);
```

エラー条件　　x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

平方根

double sqrt(double d)

説　明　　浮動小数点値の正の平方根を計算します。

ヘッダ　　<math.h>

リターン値　　正常：d の正の平方根の値
異常：定義域エラーの時は、非数を返します

引　数　　d　　正の平方根を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=sqrt(d);
```

エラー条件　　d の値が負の値の時、定義域エラーになります。

切り上げ

double ceil(double d)

説　明　　浮動小数点値の小数点以下を切り上げた整数値を求めます。

ヘッダ　　<math.h>

リターン値　　d の小数点以下を切り上げた整数値

引　数　　d　　小数点以下を切り上げる浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=ceil(d);
```

備　考　　ceil 関数は、d の値より大きいかまたは等しい最小の整数値を double 型の値として返す関数です。したがって d の値が負の値の時は小数点以下を切り捨てた時の値を返します。

絶対値

double fabs(double d)

説　明　　浮動小数点値の絶対値を計算します。

ヘッダ　　<math.h>

リターン値　　d の絶対値

引　数　　d　　絶対値を求める浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=fabs(d);
```

切り捨て

double floor(double d)

説　明　　浮動小数点値の小数点以下を切り捨てた整数値を求めます。

ヘッダ　　<math.h>

リターン値　　d の小数点以下を切り捨てた整数値

引　数　　d　　小数点以下を切り捨てる浮動小数点値

例

```
#include <math.h>
double d, ret;
    ret=floor(d);
```

備　考　　floor 関数は、d の値を超えない範囲の整数の最大値を、double 型の値として返す関数です。
したがって d の値が負の値の時は小数点以下を切り上げた時の値を返します。

余り

double fmod(double x, double y)

説　明　　浮動小数点値どうしを除算した結果の余りを計算します。

ヘッダ　　<math.h>

リターン値　　y の値が 0.0 の時　　：x
y の値が 0.0 でない時　：x を y で除算した結果の余り

引　数　　x　　被除数
y　　除数

例

```
#include <math.h>
double x, y, ret;
    ret=fmod(x,y);
```

備　考　　fmod 関数では、引数 x、y、リターン値 ret の間には、次に示す関係が成立します。
x=y*i+ret　(ただし i は整数値)
また、リターン値 ret の符号は x の符号と同じ符号になります。
x/y の商を表現できない場合、結果の値は、保証しません。
y の値が 0.0 の時、マイコンの設定によりゼロ除算例外が発生します。

(10)　<mathf.h>

各種の数値計算を行います。

<mathf.h>では ANSI 規格規定外の単精度形式の数学関数の宣言とマクロの定義をしています。

各関数は float 型の引数を受け取り、float 型の値を返します。

以下の定数(マクロ)はすべて処理系定義です。

種別	定義名	説明
定数(マクロ)	EDOM	関数に入力するパラメータの値が関数内で定義している値の範囲を超える時、errno に設定する値です。
	ERANGE	関数の計算結果が float 型の値として表わせない時、あるいはオーバフロー/アンダフローとなった時、errno に設定する値です。
	HUGE_VALF	関数の計算結果がオーバフローした時に、関数のリターン値として返す値です。
関数	acosf	浮動小数点値の逆余弦を計算します。
	asinf	浮動小数点値の逆正弦を計算します。
	atanf	浮動小数点値の逆正接を計算します。
	atan2f	浮動小数点値どうしを除算した結果の値の逆正接を計算します。
	cosf	浮動小数点値のラジアン値の余弦を計算します。
	sinf	浮動小数点値のラジアン値の正弦を計算します。
	tanf	浮動小数点値のラジアン値の正接を計算します。
	coshf	浮動小数点値の双曲線余弦を計算します。
	sinhf	浮動小数点値の双曲線正弦を計算します。
	tanhf	浮動小数点値の双曲線正接を計算します。
	expf	浮動小数点値の指数関数を計算します。
	frexpf	浮動小数点値を[0.5, 1.0)の値と 2 のべき乗の積に分解します。
	ldexpf	浮動小数点値と 2 のべき乗の乗算を計算します。
	logf	浮動小数点値の自然対数を計算します。
	log10f	浮動小数点値の 10 を底とする対数を計算します。
	modff	浮動小数点値を整数部分と小数部分に分解します。
	powf	浮動小数点値のべき乗を計算します。
	sqrtf	浮動小数点値の正の平方根を計算します。
	ceilf	浮動小数点値の小数点以下を切り上げた整数値を求めます。
	fabsf	浮動小数点値の絶対値を計算します。
	floorf	浮動小数点値の小数点以下を切り捨てた整数値を求めます。
	fmodf	浮動小数点値どうしを除算した結果の余りを計算します。

エラーが発生した時の動作を以下に説明します。

(1) 定義域エラー
関数に入力するパラメータの値が関数内で定義している値の範囲を超えている時、定義域エラーが発生します。この時errnoにはEDOMの値が設定されます。また、関数のリターン値は、処理系定義です。

(2) 範囲エラー
関数における計算結果がfloat型の値として表わせない時には範囲エラーが発生します。この時、errnoにはERANGEの値が設定されます。また、計算結果がオーバフローの時は、正しく計算が行われた時と同様の符号のHUGE_VALFの値をリターン値として返します。逆に計算結果がアンダフローの時は、0をリターン値として返します。

【注】1. <mathf.h>の関数の呼び出しによって定義域エラーが発生する可能性がある場合は、結果の値をそのまま用いるのは危険です。必ず errno をチェックしてから用いてください。
例：

```
        .
        .
        .
1   x=asinf(a);
2   if (errno==EDOM)
3      printf("error¥n");
4   else
5      printf("result is : %f¥n",x);
        .
        .
        .
```

1 行目で、asinf 関数を使って逆正弦値を求めます。このとき、引数 a の値が、asinf 関数の定義域[-1.0,1.0]の範囲を超えていると、errno に値 EDOM が設定されます。2 行目で定義域エラーが生じたかどうかの判定をします。定義域エラーが生じれば、3 行目で、error を出力します。定義域エラーが生じなければ 5 行目で、逆正弦値を出力します。

2. 範囲エラーが発生するかどうかは、コンパイラによって定まる、浮動小数点型の内部表現形式によって異なります。例えば無限大を値として表現できる内部表現形式を採用している場合、範囲エラーの生じないように<mathf.h>のライブラリ関数を実現することができます。

3. cpu=sh2afpu|sh4|sh4a を指定し、かつ fabsf 関数および sqrtf 関数を使用した場合、もしくは cpu=sh2e を指定し、かつ fabsf 関数を使用した場合は、エラーが発生しても errno は設定されません。

処理系定義仕様

	項目	コンパイラの仕様
1	数学関数の入力引数値が範囲を超えたときの数学関数が返す値	非数を返します。非数の形式は「10.1.3 浮動小数点型の仕様」を参照してください
2	数学関数でアンダフローエラーが発生したときマクロ「ERANGE」の値が「errno」に設定されるかどうか	設定しません。
3	fmodf 関数で第 2 実引数の値が 0 の場合、範囲エラーとなるかどうか	範囲エラーとなります。

逆余弦

float acosf(float f)

説　明　浮動小数点値の逆余弦を計算します。

ヘッダ　<mathf.h>

リターン値　正常：f の逆余弦値
異常：定義域エラーの時は、非数を返します

引　数　f　逆余弦を求める浮動小数点値

例
```
#include <mathf.h>
float f, ret;
   ret=acosf(f);
```

エラー条件　f の値が[-1.0,1.0]の範囲を超えている時、定義域エラーになります。

備　考　acosf 関数のリターン値の範囲は[0,π]です。

逆正弦

float asinf(float f)

説　明　浮動小数点値の逆正弦を計算します。

ヘッダ　<mathf.h>

リターン値　正常：f の逆正弦値
異常：定義域エラーの時は、非数を返します

引　数　f　逆正弦を求める浮動小数点値

例
```
#include <mathf.h>
float f, ret;
   ret=asinf(f);
```

エラー条件　f の値が[-1.0,1.0]の範囲を超えている時、定義域エラーになります。

備　考　asinf 関数のリターン値の範囲は[-π/2,π/2]です。

逆正接

float atanf(float f)

説　明　　浮動小数点値の逆正接を計算します。

ヘッダ　　<mathf.h>

リターン値　　f の逆正接値

引　数　　f　　逆正接を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=atanf(f);
```

備　考　　atanf 関数のリターン値の範囲は(-π/2,π/2)です。

除算後の逆正接

float atan2f(float y, float x)

説　明　　浮動小数点値どうしを除算した結果の値の逆正接を計算します。

ヘッダ　　<mathf.h>

リターン値　　正常：y を x で除算したときの逆正接値
異常：定義域エラーの時は、非数を返します

引　数

x	除数
y	被除数

例

```
#include <mathf.h>
float x, y, ret;
   ret=atan2f(y,x);
```

エラー条件　　x, y の値がともに 0.0 の時、定義域エラーになります。

備　考　　atan2f 関数のリターン値の範囲は(-π,π]です。atan2f 関数の示す意味を図 10.7に示します。図に示すように、atan2f 関数の結果は、点(x,y)と原点を通る直線と x 軸をなす角を求めます。y=0.0 で x が負の時、結果はπとなります。
x=0.0 の時、y の値の正負に従って結果は±π/2 となります。但し、マイコンの設定によりゼロ除算例外が発生します。

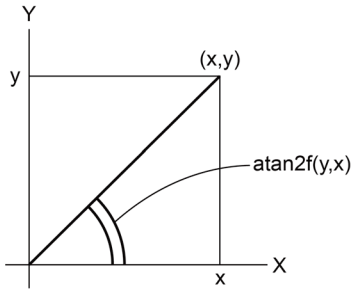


図 10.7　atan2f 関数の意味

余弦

float cosf(float f)

説　明　　浮動小数点値のラジアン値の余弦を計算します。

ヘッダ　　<mathf.h>

リターン値　　f の余弦値

引　数　　f　　余弦を求めるラジアン値

例

```
#include <mathf.h>
float f, ret;
   ret=cosf(f);
```

正弦

float sinf(float f)

説　明　　浮動小数点値のラジアン値の正弦を計算します。

ヘッダ　　<mathf.h>

リターン値　　f の正弦値

引　数　　f　　正弦を求めるラジアン値

例

```
#include <mathf.h>
float f, ret;
   ret=sinf(f);
```

正接

float tanf(float f)

説　明　浮動小数点値のラジアン値の正接を計算します。

ヘッダ　<mathf.h>

リターン値　f の正接値

引　数　f　正接を求めるラジアン値

例
```
#include <mathf.h>
float f, ret;
   ret=tanf(f);
```

双曲線余弦

float coshf(float f)

説　明　浮動小数点値の双曲線余弦を計算します。

ヘッダ　<mathf.h>

リターン値　f の双曲線余弦値

引　数　f　双曲線余弦を求める浮動小数点値

例
```
#include <mathf.h>
float f, ret;
   ret=coshf(f);
```

双曲線正弦

float sinhf(float f)

説　明　　浮動小数点値の双曲線正弦を計算します。

ヘッダ　　<mathf.h>

リターン値　　f の双曲線正弦値

引　数　　f　　双曲線正弦を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=sinhf(f);
```

双曲線正接

float tanhf(float f)

説　明　　浮動小数点値の双曲線正接を計算します。

ヘッダ　　<mathf.h>

リターン値　　f の双曲線正接値

引　数　　f　　双曲線正接を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=tanhf(f);
```

指数関数

float expf(float f)

説　明　浮動小数点値の指数関数を計算します。

ヘッダ　<mathf.h>

リターン値　f の指数関数値

引　数　f　指数関数を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=expf(f);
```

浮動小数点値を仮数、指数に分解

*float frexpf(float value, int *exp)*

説　明　浮動小数点値を[0.5,1.0)の値と 2 のべき乗の積に分解します。

ヘッダ　<mathf.h>

リターン値　value が 0.0 の時　：0.0
value が 0.0 でない時　：ret*2^(exp の指している領域の値)=value で定義される ret の値

引　数　value　[0.5,1.0)の値と 2 のべき乗の積に分解する浮動小数点値
exp　2 のべき乗値を格納する記憶域へのポインタ

例

```
#include <mathf.h>
float ret, value;
int *exp;
   ret=frexpf(value,exp);
```

備　考　frexpf 関数は、value を[0.5,1.0)の値と 2 のべき乗の積に分解します。exp の指す領域には、分解した結果の 2 のべき乗値を設定します。
リターン値 ret の値の範囲は[0.5,1.0)または 0.0 になります。
value が 0.0 ならば、exp の指す int 型の記憶域の内容と ret の値は 0.0 になります。

仮数、指数を浮動小数点値に変換

float ldexpf(float e, int f)

説　明　　浮動小数点値と 2 のべき乗の積を計算します。

ヘッダ　　<mathf.h>

リターン値　　$e*2^f$ の演算結果の値

引　数　　e　　2 のべき乗値を求める浮動小数点値
　　f　　2 のべき乗値

例

```
#include <mathf.h>
float ret, e;
int  f;
   ret=ldexpf(e, f);
```

自然対数

float logf(float f)

説　明　　浮動小数点値の自然対数を計算します。

ヘッダ　　<mathf.h>

リターン値　　正常：f の自然対数の値
　　異常：定義域エラーの時は、非数を返します

引　数　　f　　自然対数を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=logf(f);
```

エラー条件　　f の値が負の時、定義域エラーになります。
　　f の値が 0.0f の時、範囲エラーになります。

常用対数

float log10f(float f)

説　明　浮動小数点値の 10 を底とする対数を計算します。

ヘッダ　<mathf.h>

リターン値　正常：f は 10 を底とする対数値
異常：定義域エラーの時は、非数を返します

引　数　f　10 を底とする対数を求める浮動小数点値

例
```
#include <mathf.h>
float f, ret;
   ret=log10f(f);
```

エラー条件　f の値が負の値の時、定義域エラーになります。
f の値が 0.0 の時、範囲エラーになります。

浮動小数点値を整数部、小数部に分解

*float modff(float a, float *b)*

説　明　浮動小数点値を整数部分と小数部分に分解します。

ヘッダ　<mathf.h>

リターン値　a の小数部分

引　数　a　整数部分と小数部分に分解する浮動小数点値
b　整数部分を格納する記憶域を指すポインタ

例
```
#include <mathf.h>
float a, *b, ret;
   ret=modff(a,b);
```

べき乗

float powf(float x, float y)

説　明　　浮動小数点値のべき乗を計算します。

ヘッダ　　<mathf.h>

リターン値　　正常：x の y 乗の値
異常：定義域エラーの時は、非数を返します

引　数　　x　　べき乗される値
y　　べき乗する値

例

```
#include <mathf.h>
float x, y, ret;
   ret=powf(x, y);
```

エラー条件　　x の値が 0.0 で、かつ y の値が 0.0 以下の時、あるいは x の値が負で y の値が整数値でない時、定義域エラーになります。

平方根

float sqrtf(float f)

説　明　　浮動小数点値の正の平方根を計算します。

ヘッダ　　<mathf.h>

リターン値　　正常：f の正の平方根の値
異常：定義域エラーの時は、非数を返します

引　数　　f　　正の平方根を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=sqrtf(f);
```

エラー条件　　f の値が負の値の時、定義域エラーになります。

切り上げ

float ceilf(float f)

説　明　浮動小数点値の小数点以下を切り上げた整数値を求めます。

ヘッダ　<mathf.h>

リターン値　f の小数点以下を切り上げた整数値

引　数　f　小数点以下を切り上げる浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=ceilf(f);
```

備　考　ceilf 関数は、f の値より大きい、または等しい最小の整数値を float 型の値として返す関数です。したがって f の値が負の値の時は小数点以下を切り捨てた時の値を返します。

絶対値

float fabsf(float f)

説　明　浮動小数点値の絶対値を計算します。

ヘッダ　<mathf.h>

リターン値　f の絶対値

引　数　f　絶対値を求める浮動小数点値

例

```
#include <mathf.h>
float f, ret;
   ret=fabsf(f);
```

切り捨て

float floorf(float f)

説　明　　浮動小数点値の小数点以下を切り捨てた整数値を求めます。

ヘッダ　　<mathf.h>

リターン値　　f の小数点以下を切り捨てた整数値

引　数　　f　　小数点以下を切り捨てる浮動小数点値

例

```
#include <mathf.h>
float f, ret;
    ret=floorf(f);
```

備　考　　floorf 関数は、f の値を超えない範囲の整数の最大値を、float 型の値として返す関数です。
したがって f の値が負の値の時は小数点以下を切り上げた時の値を返します。

余り

float fmodf(float x, float y)

説　明　　浮動小数点値どうしを除算した結果の余りを計算します。

ヘッダ　　<mathf.h>

リターン値　　y の値が 0.0 の時：x
y の値が 0.0 でない時：x を y で除算した結果の余り

引　数　　x　　被除数
y　　除数

例

```
#include <mathf.h>
float x, y, ret;
    ret=fmodf(x, y);
```

備　考　　fmodf 関数では、引数 x、y、リターン値 ret の間には、次に示す関係が成立します。
x=y*i+ret　(ただし i は整数値)
また、リターン値 ret の符号は x の符号と同じ符号になります。
x/y の商を表現できない場合、結果の値は、保証しません。
y の値が 0.0 の時、マイコンの設定によりゼロ除算例外が発生します。

(11) <setjmp.h>

関数間の制御の移動をサポートします。

以下のマクロは、処理系定義です。

種別	定義名	説明
型 (マクロ)	jmp_buf	関数間の制御の移動を可能とする情報を保存しておくための記憶域に対応する型名です。
関数	setjmp	現在実行中の関数の jmp_buf で定義した実行環境を指定した記憶域に退避します。
	longjmp	setjmp 関数で退避していた関数の実行環境を回復し、setjmp 関数を呼び出したプログラムの位置に制御を移動します。

setjmp 関数は現在の関数の実行環境を退避します。その後 longjmp 関数を呼び出すことにより、setjmp 関数を呼び出したプログラム上の位置に戻ることができます。

以下に setjmp、longjmp 関数を使用して関数間の制御の移動をサポートした例を示します。

```
1     #include <stdio.h>
2     #include <setjmp.h>
3     jmp_buf env;
4     void sub();
5     void main()
6     {
7
8        if (setjmp(env)!=0){
9           printf("return from longjmp¥n");
10          exit(0);
11       }
12       sub();
13    }
14
15    void sub()
16    {
17       printf("subroutine is running ¥n");
18       longjmp(env,1);
19    }
```

【説明】

8 行目で setjmp 関数を呼んでいます。この時、setjmp 関数の呼び出された環境を、jmp_buf 型の変数 env に退避します。この時のリターン値は 0 なので、次に関数 sub が呼び出されます。

関数 sub の中で呼び出される longjmp 関数により、変数 env に退避した環境を回復します。その結果、プログラムは、あたかも 8 行目の setjmp 関数からリターンしたかのようにふるまいます。ただし、この時のリターン値は longjmp 関数の第 2 引数で指定した値(1)になります。
その結果、9 行目以降が実行されます。

大域 goto 飛び先設定

int setjmp(jmp_buf env)

説　明　　現在実行中の関数の実行環境を、指定した記憶域に退避します。

ヘッダ　　<setjmp.h>

リターン値　　setjmp 関数を呼び出した時　　：0
longjmp 関数からのリターン時　：0 以外

引　数　　env　　実行環境を退避する記憶域へのポインタ

例

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

備　考　　setjmp 関数により退避された実行環境は、longjmp 関数において使用されます。
setjmp 関数として呼び出された時のリターン値は 0 ですが、longjmp 関数からリターンしてきた時のリターン値は、longjmp 関数で指定した第 2 引数の値となります。
setjmp 関数を複雑な式から呼び出す場合、式の評価の途中結果等の現在の実行環境の一部が失われる可能性があります。setjmp 関数は setjmp 関数の結果と定数式の比較という形だけで使用し、複雑な式の中では呼び出さないようにしてください。
setjmp 関数へのポインタを使った間接呼び出しはしないでください。

大域 goto

void longjmp(jmp_buf env, int ret)

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

ヘッダ　　<setjmp.h>

引　数　　env　　実行環境を退避した記憶域へのポインタ
ret　　setjmp 関数へのリターンコード

例

```
#include <setjmp.h>
int ret;
jmp_buf env;
   longjmp(env,ret);
```

備　考　　longjmp 関数は、同じプログラム中で最後に呼び出された setjmp 関数によって退避された関数の実行環境を第 1 引数 env で指定された記憶域から回復し、その setjmp 関数を呼び出したプログラムの位置に制御を移します。この時 longjmp 関数の第 2 引数 ret が setjmp 関数のリターン値として返ります。ただし、ret が 0 の時は setjmp 関数へのリターン値としては 1 が返ります。
setjmp 関数が呼び出されていない時、あるいは setjmp 関数を呼び出した関数がすでに return 文を実行している時は、longjmp 関数の動作は保証しません。

(12) <stdarg.h>

可変個の引数を持つ関数に対し、その引数の参照を可能にします。

以下はすべて処理系定義です。

種別	定義名	説明
型(マクロ)	va_list	可変個の引数を参照するために、va_start, va_arg, va_end マクロで共通に使用される変数の型です。
関数(マクロ)	va_start	可変個の引数の参照を行うため、初期設定処理を行います。
	va_arg	可変個の引数を持つ関数に対して、現在参照中引数の次の引数への参照を可能とします。
	va_end	可変個の引数を持つ関数の引数への参照を終了させます。

本標準インクルードファイルで定義しているマクロを使用したプログラムの例を以下に示します。

```
1     #include <stdio.h>
2     #include <stdarg.h>
3
4     extern void prlist(int count,...);
5
6     void main()
7     {
8        prlist(1, 1);
9        prlist(3, 4, 5, 6);
10       prlist(5, 1, 2, 3, 4, 5);
11    }
12
13    void prlist(int count,...)
14    {
15       va_list ap;
16       int i;
17
18       va_start(ap,count);
19       for(i=0;i<count;i++)
20          printf("%d",va_arg(ap,int));
21       putchar('¥n');
22       va_end(ap);
23    }
```

【説明】

この例では、第 1 引数に出力するデータの数を指定し、以下の引数をその数だけ出力する関数 prlist を実現しています。

18 行目で、可変個の引数への参照を va_start で初期化します。その後引数を一つ出力するたびに、va_arg マクロによって次の引数を参照します(20 行目)。va_arg マクロでは、引数の型名(この場合は int 型)を第 2 引数に指定します。

引数の参照が終了したら、va_end マクロを呼び出します(22 行目)。

可変個引数取り出し開始

void va_start(va_list ap, parmN)

説　明　　可変個の引数への参照を行うため、初期設定処理を行います。

ヘッダ　　<stdarg.h>

引　数　　ap　　可変個の引数にアクセスするための変数
parmN　　最右端の引数の識別子

例

```
#include <stdarg.h>
void func(int count,...)
{
   va_list ap;
      va_start(ap,count);
}
```

備　考　　va_start マクロは、va_arg、va_end マクロによって使用される ap の初期化を行います。
また、parmN には、外部関数定義における引数の並びの最右端の引数の識別子、すなわち「,...」の直前の識別子を指定します。
可変個の名前のない引数を参照するためには、va_start マクロ呼び出しを一番初めに実行する必要があります。

可変個引数取り出し

type va_arg(va_list ap, type)

説　明　　可変個の引数を持つ関数に対して、現在参照中の引数の次の引数への参照を可能とします。

ヘッダ　　<stdarg.h>

リターン値　　引数の値

引　数　　ap　　可変個の引数にアクセスするための変数
type　　アクセスする引数の型

例

```
#include <stdarg.h>
va_list ap;
int ret;
   ret=va_arg(ap,int);
```

備　考　　va_start マクロで初期化した va_list 型の変数を第 1 引数に指定します。
ap の値は va_arg を使用する度に更新され、結果として可変個の引数が順次本マクロのリターン値として返されます。
第 2 引数 type は、参照する型を指定してください。
ap は va_start によって初期化された ap と同じでなければなりません。
type に char 型、unsigned char 型、short 型、unsigned short 型、float 型のように関数の引数に指定した時に型変換によってサイズが変わる型を指定した場合、正しく引数を参照することができなくなります。このような型を指定すると動作は保証しません。

可変個引数取り出し終了

void va_end(va_list ap)

説　明　可変個の引数を持つ関数の引数への参照を終了させます。

ヘッダ　`<stdarg.h>`

引　数　`ap`　可変個の引数を参照するための変数

例

```
#include <stdarg.h>
va_list ap;
    va_end(ap);
```

備　考　ap は va_start によって初期化された ap と同じでなければなりません。
また、関数からの return 前に va_end マクロが呼び出されない時は、その関数の動作は保証しません。

(13) <stdio.h>

ストリーム入出力用ファイルの入出力に関する処理を行います。

以下の定数(マクロ)はすべて処理系定義です。

種別	定義名	説明
定数(マクロ)	FILE	ストリーム入出力処理で必要とするバッファへのポインタやエラー指示子、終了指示子などの各種制御情報を保存しておく構造体の型です。
	_IOFBF	バッファ領域の使用方法として、入出力処理はすべてバッファ領域を使用することを示しています。
	_IOLBF	バッファ領域の使用方法として、入出力処理は行単位でバッファ領域を使用することを示しています。
	_IONBF	バッファ領域の使用方法として、入出力処理はバッファ領域を使用しないことを示しています。
	BUFSIZ	入出力処理において必要とするバッファの大きさです。
	EOF	ファイルの終わり(End Of File)すなわちファイルからそれ以上の入力が無いことを示しています。
	L_tmpnam*	tmpnam 関数によって生成される一時ファイル名の文字列を格納するのに十分な大きさの配列のサイズです。
	SEEK_CUR	ファイルの現在の読み書き位置を現在の位置からのオフセットに移すことを示しています。
	SEEK_END	ファイルの現在の読み書き位置をファイルの終了位置からのオフセットに移すことを示しています。
	SEEK_SET	ファイルの現在の読み書き位置をファイルの先頭位置からのオフセットに移すことを示しています。
	SYS_OPEN*	処理系が同時にオープンすることができることを保証するファイルの数です。
	TMP_MAX*	tmpnam 関数によって生成される一意なファイル名の個数の最大値です。
	stderr	標準エラーファイルに対するファイルポインタです。
	stdin	標準入力ファイルに対するファイルポインタです。
	stdout	標準出力ファイルに対するファイルポインタです。
関数	fclose	ストリーム入出力用ファイルをクローズします。
	fflush	ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。
	fopen	ストリーム入出力用ファイルを指定したファイル名によってオープンします。
	freopen	現在オープンされているストリーム入力出用ファイルをクローズし、新しいファイルを指定したファイル名で再オープンします。
	setbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。
	setvbuf	ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。
	fprintf	書式に従ってストリーム入出力用ファイルへデータを出力します。
	fscanf	ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。

【注】* 本処理系では、定義されません。

種別	定義名	説明
関数	printf	データを書式に従って変換し、標準出力ファイル(stdout)へ出力します。
	scanf	標準入力ファイル(stdin)からデータを入力し、書式に従って変換します。
	sprintf	データを書式に従って変換し、指定した領域へ出力します。
	sscanf	指定した記憶域からデータを入力し、書式に従って変換します。
	vfprintf	可変個の引数リストを書式に従って指定したストリーム入出力用ファイルに出力します。
	vprintf	可変個の引数リストを書式に従って標準出力ファイル(stdout)に出力します。
	vsprintf	可変個の引数リストを書式に従って指定した領域に出力します。
	fgetc	ストリーム入出力用ファイルから 1 文字入力します。
	fgets	ストリーム入出力用ファイルから文字列を入力します。
	fputc	ストリーム入出力用ファイルへ 1 文字出力します。
	fputs	ストリーム入出力用ファイルへ文字列を出力します。
	getc	(マクロ) ストリーム入出力用ファイルから 1 文字入力します。
	getchar	(マクロ) 標準入力ファイルから 1 文字入力します。
	gets	標準入力ファイルから文字列を入力します。
	putc	(マクロ) ストリーム入出力用ファイルへ 1 文字出力します。
	putchar	(マクロ) 標準出力ファイルへ 1 文字出力します。
	puts	標準出力ファイルへ文字列を出力します。
	ungetc	ストリーム入出力用ファイルへ 1 文字をもどします。
	fread	ストリーム入出力用ファイルから指定した記憶域にデータを入力します。
	fwrite	記憶域からストリーム入出力用ファイルにデータを出力します。
	fseek	ストリーム入出力用ファイルの現在の読み書き位置を移動させます。
	ftell	ストリーム入出力用ファイルの現在の読み書き位置を求めます。
	rewind	ストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
	clearerr	ストリーム入出力用ファイルのエラー状態をクリアします。
	feof	ストリーム入出力用ファイルが終わりであるかどうかを判定します。
	ferror	ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
	perror	標準エラーファイル(stderr)に、エラー番号に対応したエラーメッセージを出力します。

処理系定義仕様

	項目	コンパイラの仕様
1	入力テキストの最終の行が終了を示す改行文字を必要とするかどうか	規定しません。低水準インタフェースルーチンの仕様によります。
2	改行文字の直前にかき出された空白文字は、読み込み時に読み込まれるかどうか	
3	バイナリファイルに書かれたデータに付加されるヌル文字の数	
4	追加モード時のファイル位置指定子の初期値	
5	テキストファイルへの出力によってそれ以降のファイルのデータが失われるかどうか	
6	ファイルバッファリングの仕様	
7	長さ 0 のファイルが存在するかどうか	
8	正当なファイル名の構成規則	
9	同時に同じファイルをオープンできるかどうか	
10	fprintf 関数における%p 書式変換の出力形式	16 進数出力となります
11	fscanf 関数における%p 書式変換の入力形式 fscanf 関数での変換文字「－」の意味	16 進数入力となります。 先頭、最後あるいは「^」の直後でない場合、直前の文字と直後の範囲を示します。
12	fgetpos,ftell 関数で設定される errno の値	fgetpos 関数はサポートしていません。 ftell 関数については規定しません。 低水準インタフェースルーチンの仕様によります。
13	perror 関数が生成するメッセージ出力形式	メッセージの出力形式を(a)に示します。
14	calloc,malloc,realloc 関数でサイズが 0 のときの動作	0 バイトの領域を割り付けます。

(a) perror 関数の出力形式は、
<文字列>：<error に設定したエラー番号に対応するエラーメッセージ>
となります。

(b) printf 関数、fprintf 関数で、浮動小数点の無限大および非数を表示するときの形式を表 10.41 に示します。

表 10.41 無限大および非数の表示形式

	値	表示形式
1	正の無限大	＋＋＋＋＋＋
2	負の無限大	－－－－－－
3	非数	＊＊＊＊＊＊

ストリーム入出力用ファイルに対する一連の入出力処理を行ったプログラムの例を以下に示します。

```
1     #include <stdio.h>
2
3     void main()
4     {
5        int c;
6        FILE *ifp, *ofp;
7
8        if ((ifp=fopen("INPUT.DAT","r"))==NULL){
9           fprintf(stderr,"cannot open input file¥n");
10          exit(1);
11       }
12       if ((ofp=fopen("OUTPUT.DAT","w"))==NULL){
13          fprintf(stderr,"cannot open output file¥n");
14          exit(1);
15       }
16       while ((c=getc(ifp))!=EOF)
17          putc(c, ofp);
18       fclose(ifp);
19       fclose(ofp);
20    }
```

【説明】

ファイル INPUT.DAT の内容をファイル OUTPUT.DAT へコピーするプログラムです。

8 行目の fopen 関数で入力ファイル INPUT.DAT を、12 行目の fopen 関数で出力ファイル OUTPUT.DAT をオープンします。オープンに失敗した場合、fopen 関数のリターン値として NULL が返されますので、エラーメッセージを出力してプログラムを終了させます。

fopen 関数が正常に終了した時、オープンしたファイルの情報を格納するデータ(FILE 型)へのポインタが返されますので、これらを変数 ifp、ofp に設定します。

オープンが成功した後は、これらの FILE 型のデータを用いて入出力を行います。

ファイルの処理が終了したら、fclose 関数でファイルをクローズします。

ファイルクローズ

*int fclose(FILE *fp)*

説　明　　ストリーム入出力用ファイルをクローズします。

ヘッダ　　<stdio.h>

リターン値　　正常：0
異常：0 以外

引　数　　fp　　ファイルポインタ

例

```
#include <stdio.h>
FILE *fp;
int ret;
   ret=fclose(fp);
```

備　考　　fclose 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルをクローズします。fclose 関数は、ストリーム入出力用ファイルの出力ファイルがオープンされており、まだ出力されていないデータがバッファに残っている時は、それをファイルに出力してからクローズします。
また、入出力用のバッファがシステムによって自動的に割り付けられていた場合は、それを解放します。

バッファフラッシュ

*int fflush(FILE *fp)*

説　明　　ストリーム入出力用ファイルのバッファの内容をファイルへ出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：0
異常：0 以外

引　数　　fp　　ファイルポインタ

例

```
#include  <stdio.h>
FILE *fp;
int ret;
   ret=fflush(fp);
```

備　考　　fflush 関数は、ストリーム入出力用ファイルの出力ファイルがオープンされている時、ファイルポインタ fp で指定されたストリーム入出力用ファイルのバッファの未出力内容をファイルに出力します。また、入力ファイルがオープンされている時、ungetc 関数の指定を無効にします。

ファイルオープン

*FILE *fopen(const char *fname, const char *mode)*

説　明　ストリーム入出力用ファイルを、指定したファイル名によってオープンします。

ヘッダ　<stdio.h>

リターン値　正常：オープンしたファイルのファイル情報を指すファイルポインタ
異常：NULL

引　数

fname	ファイル名を示す文字列へのポインタ
mode	ファイルアクセスモードを示す文字列へのポインタ

例

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

備　考　fopen 関数は、fname が指す文字列をファイル名とするストリーム入出力用ファイルをオープンします。書き出しモードあるいは追加モードで存在しないファイルをオープンしようとした時は、可能な限り新しいファイルを作成します。また既存のファイルに対して書き出しモードでオープンした時は、ファイルの先頭から書き込みが行われ、以前に書き込まれていたファイルの内容は消去されます。
追加モードでオープンしたファイルは、そのファイルの終わりの位置から書き出しの処理が行われます。更新モードでオープンしたファイルは、このファイルに対して入力と出力の両方の処理を行うことができます。
ただし、出力処理は後に fflush、fseek、rewind 関数が実行されることなしに入力処理を続けることはできません。
また同様に入力処理の後に fflush、fseek、rewind 関数が実行されることなしに出力処理を続けることはできません。
また、ファイルアクセスモードを示す文字列の後にオープンの方法を指示する文字が付くこともあります。
同一のファイルを複数のタスクで同時にオープンすることはできません。

ファイル再オープン

*FILE *freopen(const char *fname, const char *mode, FILE *fp)*

説　明　現在オープンされているストリーム入出力用ファイルをクローズし、新しいファイルを指定したファイル名で再オープンします。

ヘッダ　`<stdio.h>`

リターン値　正常：fp
異常：NULL

引　数

fname	新しいファイル名を示す文字列へのポインタ
mode	ファイルアクセスモードを示す文字列へのポインタ
fp	現在オープンされているストリーム入出力用ファイルのファイルポインタ

例

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

備　考　freopen 関数は、まず、ファイルポインタ fp の示すストリーム入出力用ファイルをクローズします(このクローズ処理が正しく行われない時でも以下の処理は続けます)。
次に、その fp の指す FILE 構造体を再使用して、ファイル名 fname で示すファイルを、ストリーム入出力用にオープンします。
freopen 関数は一時にオープンするファイル数が限られているときなどに有効です。
freopen 関数は通常、fp と同じ値を返しますが、エラーが発生した時は、NULL を返します。

バッファ領域設定

*void setbuf(FILE *fp, char buf[BUFSIZ])*

説　明　ストリーム入出力用のバッファ領域をユーザプログラム側で定義して設定します。

ヘッダ　`<stdio.h>`

引　数

fp	ファイルポインタ
buf	バッファ領域へのポインタ

例

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

備　考　setbuf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対して buf の指す記憶域を入出力用のバッファ領域として使用するように定義します。この結果、大きさが BUFSIZ のバッファ領域を使用した入出力処理が行われます。

バッファリング制御

*int setvbuf(FILE *fp, char *buf, int type, size_t size)*

説　明　　ストリーム入出力用のバッファ領域をユーザプログラムの側で定義して設定します。

ヘッダ　　<stdio.h>

リターン値　　正常：0
異常：0 以外

引　数

fp	ファイルポインタ
buf	バッファ領域へのポインタ
type	バッファの管理方式
size	バッファ領域の大きさ

例

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

備　考　　setvbuf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対して buf の指す記憶域を入出力用のバッファ領域として使用するように定義します。
このバッファ領域の使用方法としては、以下の三通りの方法があります。

(a) type に_IOFBF を指定した時
入出力処理はすべてバッファ領域を使用して行います。

(b) type に_IOLBF を指定した時
入出力処理は行単位でバッファ領域を使用して行います。すなわち、入出力データは、改行文字が書かれた時、バッファ領域が一杯になった時、入力が要求された時にバッファ領域から取り出されることになります。

(c) type に_IONBF を指定した時
入出力処理はバッファ領域を使用せず行います。
setvbuf 関数は通常 0 を返しますが、type あるいは size に不正な値が与えられた時、あるいはバッファ領域の使用方法等の要求が受け入れられなかった時には 0 以外の値を返します。

バッファ領域は、オープンされているストリーム入出力用ファイルがクローズされる前に解放してはいけません。また、setvbuf 関数は、ストリーム入出力用ファイルがオープンされてから入出力用処理が行われるまでの間で使用してください。

書式付きファイル出力

*int fprintf(FILE *fp, const char *control [, arg]　…)*

説　明　　書式に従って、ストリーム入出力用ファイルへデータを出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：変換し出力した文字数
異常：負の値

引　数

fp	ファイルポインタ
control	書式を示す文字列へのポインタ
arg,…	書式に従って出力されるデータの並び

例

```
#include <stdio.h>
FILE *fp;
const char *control="%s";
int ret;
char buffer[]="Hello World¥n";
   ret=fprintf(fp,control,buffer);
```

備　考　　fprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、ファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。
fprintf 関数は、通常は変換し出力したデータの個数を返しますが、エラー発生時には負の値を返します。
書式の仕様は以下のとおりです。

【書式の概要】
書式を表す文字列は、2 種類の文字列から構成されます。
・通常の文字
次の変換仕様を示す文字列以外の文字はそのまま出力されます。
・変換仕様
変換仕様は、%で始まる文字列で、後に続く引数の変換方法を指定します。変換仕様の形式は次の規則に従います。

%[フラグ…] { [*] / [フィールド幅] } (. { [*] / [精度] }) [パラメータのサイズ指定] 変換文

この変換仕様に対して、実際に出力する引数が無い時は、その動作は保証しません。また、変換仕様よりも実際に出力する引数の個数が多い時は、余分な引数はすべて無視されます。

【変換仕様の説明】
(a) フラグ
符号を付けるなどの出力するデータに対する修飾を指定します。指定できるフラグの種類と意味を**表 10.42** に示します。

表 10.42 フラグの種類と意味

	種類	意味
1	-	変換したデータの文字数が指定したフィールド幅より少ない時、そのデータをフィールド内で左詰めにして出力します。
2	+	符号付きのデータに変換する時、そのデータの符号に従って、変換したデータの先頭にプラスあるいはマイナス符号を付けます。
3	空白	符号付きのデータの変換において、変換したデータの先頭に符号が付かない時、そのデータの先頭に空白を付けます。 「＋」と共に使用した時、本フラグは無視されます。
4	#	表 10.44 で説明する変換の種類に従って、変換後のデータに修飾を行います。 1. c, d, i, s, u 変換の時 本フラグは無視されます。 2. o 変換の時 変換したデータの先頭に 0 を付けます。 3. x(あるいは X)変換の時 変換したデータの先頭に 0x(あるいは 0X)を付けます。 4. e, E, f, g, G 変換の時 変換したデータに小数点以下がない時でも、小数点を出力します。 また、g, G 変換の時は、変換後のデータの後に付く 0 は取り除きません。

(b) フィールド幅

変換したデータを出力する文字数を任意の 10 進数で指定します。
変換したデータの文字数がフィールド幅より少ない時、フィールド幅までそのデータの先頭に空白が付けられます(ただし、フラグとして'-'を指定した時は、データの後に空白が付けられます)。
もし、変換したデータの文字数がフィールド幅より大きい時は、フィールド幅は、変換結果を出力できる幅に拡張されます。
また、フィールド幅指定の先頭が 0 で始まっている時は、出力するデータの先頭には空白ではなく文字「0」が付けられます。

(c) 精度

表 10.44 で説明する変換の種類に従って変換したデータの精度を指定します。
精度は、ピリオド(.)の後に 10 進整数を続ける形式で指定します。10 進整数を省略した時は、0 を指定したものと仮定します。
精度を指定した結果、フィールド幅の指定との間に矛盾が生じれば、フィールド幅の指定を無効とします。
各変換の種類と精度指定の意味を以下に示します。

・d, i, o, u, x, X 変換の時
変換したデータの最小の桁数を示します。
・e, E, f 変換の時
変換したデータの小数点以下の桁数を示します。
・g, G 変換の時
変換したデータの最大有効桁数を示します。
・s 変換の時
印字される最大文字数を示します。

(d) パラメータのサイズ指定

d, i, o, u, x, X, e, E, f, g, G 変換の時(表 10.44 参照)
変換するデータのサイズ(short 型、long 型、long long 型、long double 型)を指定します。これ以外の変換の時は、本指定を無視します。表 10.43 にサイズ指定の種類とその意味を示します。

表 10.43　パラメータのサイズ指定の種類とその意味

	種類	意味
1	h	d, i, o, u, x, X 変換において、変換するデータが short 型あるいは unsigned short 型であることを指定します。
2	l	d, i, o, u, x, X 変換において、変換するデータが long 型、unsigned long 型あるいは、double 型であることを指定します。
3	L	e, E, f, g, G 変換において、変換するデータが long double 型であることを指定します。
4	ll	d, i, o, u, x, X 変換において、変換するデータが long long 型あるいは、unsigned long long 型であることを指定します。n 変換において、変換するデータが long long 型へのポインタ型であることを指定します。

(e) 変換文字

変換するデータをどのような形式に変換するかを指定します。

もし、変換するデータが構造体や配列型の時や、それらの型を指すポインタの時は、s 変換で文字の配列を変換する時、p 変換でポインタを変換する時を除いてその動作は保証しません。

表 10.44 に変換文字と変換方式を示します。この表に述べられていない英文字を変換文字として指定した時は、その動作は保証しません。また、それ以外の文字を指定した時の動作はコンパイラによって異なります。

表 10.44 変換文字と変換の方式

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
1	d	d 変換	int 型データを符号付き 10 進数の文字列に変換します。d 変換と i 変換は同じ仕様です。	int 型	精度指定は、最低で何文字出力されるかを示しています。もし、変換後の文字数が精度の値より少ない時は、文字列の先頭に 0 が付きます。また、精度を省略した時は、1 が仮定されます。さらに、0 の値を持つデータを精度に 0 を指定して変換し出力しようとした時は、何も出力されません。
2	i	i 変換		int 型	
3	o	o 変換	int 型データを符号無しの 8 進数の文字列に変換します。	int 型	
4	u	u 変換	int 型データを符号無しの 10 進数の文字列に変換します。	int 型	
5	x	x 変換	int 型データを符号無しの 16 進数に変換します。16 進文字には a，b，c，d，e，f を用います。	int 型	
6	X	X 変換	int 型データを符号無しの 16 進数に変換します。16 進文字には A，B，C，D，E，F を用います。	int 型	
7	f	f 変換	double 型データを「[-]ddd.ddd」の形式の 10 進数の文字列に変換します。	double 型	精度の指定は、小数点以降の桁数を表わします。小数点以降の文字が存在する時には、必ず小数点の前に 1 桁の数字が出力されます。精度を省略した時は、6 が仮定されます。また、精度に 0 を指定した時は、小数点と小数点以降の文字は出力しません。出力するデータは丸められます。
8	e	e 変換	double 型データを「[-]d.ddde±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	精度の指定は、小数点以降の桁数を表わします。変換した文字は小数点の前に 1 桁の数字が出力され、小数点以降に精度に等しい桁数の数字が出力される形式となります。精度を省略した時は6が仮定されます。また、精度に 0 を指定した時は、小数点以降の文字は出力しません。出力するデータは丸められます。
9	E	E 変換	double 型データを「[-]d.dddE±dd」の形式の 10 進数の文字列に変換します。指数は、少なくとも 2 桁出力されます。	double 型	
10	g	g 変換(あるいは G 変換)	変換する値と有効桁数を指定する精度の値から f 変換の形式で出力するか e 変換(あるいは E 変換)の形式で出力するかを決め double 型データを出力します。もし、変換されたデータの指数が-4 より小さいか、有効桁数を指定する精度より大きい時には e 変換(あるいは E 変換)の形式に変換します。	double 型	精度の指定は、変換されたデータの最大有効桁数を示します。
11	G			double 型	
12	c	c 変換	int 型のデータを unsigned char 型データとし、そのデータに対応する文字に変換します。	int 型	精度の指定は無効です。

	変換文字	変換の種類	変換の方式	変換の対象とするデータの型	精度に対する注意事項
13	s	s 変換	char 型へのポインタ型データが指す文字列を文字列の終了を示すヌル文字まで、あるいは、精度で指定された文字数分出力します(ただしヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません)。	char 型へのポインタ型	精度の指定は出力する文字数を示します。もし、精度が省略された時は、データが指す文字列のヌル文字までの文字が出力されます(ただし、ヌル文字は出力されません。また、空白、水平タブ、改行文字は変換文字列に含まれません)。
14	p	p 変換	データをポインタとして、コンパイラごとに定義された印字可能な文字列に変換します。	void 型へのポインタ	精度の指定は無効です。
15	n	データの変換は生じません。	データは int 型へのポインタ型とみなされ、このデータが指す記憶域にいままで、出力したデータの文字数を設定します。	int 型へのポインタ型	
16	%	データの変換は生じません。	%を出力します。	なし	

(f) フィールド幅あるいは精度に対する*指定

フィールド幅あるいは精度指定の値として*を指定することができます。この時は、この変換仕様に対応するパラメータの値がフィールド幅あるいは精度指定の値として使用されます。このパラメータが負のフィールド幅を持つ時は、正のフィールド幅にフラグ-が指定されたと解釈します。また、負の精度を持つ時は、精度が省略されたものと解釈します。

書式付きファイル入力

*int fscanf(FILE *fp, const char *control [, ptr] …)*

説　明　ストリーム入出力用ファイルからデータを入力し、書式に従って変換します。

ヘッダ　<stdio.h>

リターン値　正常：入力変換に成功したデータの個数
異常：入力データの変換を行う前に入力データが終了した時：EOF

引　数

fp	ファイルポインタ
control	書式を示す文字列へのポインタ
ptr,…	入力したデータを格納する記憶域へのポインタ

例

```
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret,buffer[10];
   ret=fscanf(fp,control,buffer);
```

備　考　fscanf 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルからデータを入力し、control が指す書式を文字列に従って変換、編集して、その結果を ptr の指す記憶域へ格納します。
データを入力するための書式の仕様を以下に示します。

【書式の概要】
書式を表す文字列は、以下の 3 種類の文字列から構成されます。
・空白文字
空白(' ')、水平タブ('¥t')あるいは改行文字('¥n')を指定すると、入力データを次の空白類文字でない文字まで読み飛ばす処理を行います。
・通常の文字
上の空白文字でも%でもない文字を指定すると、入力データを 1 文字入力します。ここで入力した文字は書式を表す文字列の中に指定した文字と一致していなければなりません。
・変換仕様
変換仕様は、%で始まる文字列で、書式を表す文字列の後に続く引数の指す領域に入力データを変換して格納する方法を指定します。変換仕様の形式は次の規則に従います。

%[*][フィールド幅][変換後のデータのサイズ]変換文字

書式中の変換仕様に対して入力したデータを格納する記憶域へのポインタがない時は、その動作は保証しません。また、書式が終了したにもかかわらず、入力データを格納する記憶域へのポインタが残っている時は、そのポインタは無視されます。

【変換仕様の説明】
・*指定
入力したデータを引数が指す記憶域に格納することを抑止します。
・フィールド幅
入力するデータの最大文字数を 10 進数字で指定します。
・変換後のデータのサイズ
d, i, o, u, x, X, e, E, f 変換の時(**表 10.46** 参照)、変換後のデータのサイズ(short 型、long 型、long long 型、long double 型)を指定します。これ以外の変換の時は、本指定を無視します。**表 10.45** にサイズ指定の種類とその意味を示します。

表 10.45　変換後のデータのサイズ指定の種類とその意味

	種類	意味
1	h	d，i，o，u，x，X 変換において、変換後のデータは short 型であることを指定します。
2	l	d，i，o，u，x，X 変換において、変換後のデータは long 型であることを指定します。 また、e，E，f 変換において、変換後のデータは double 型であることを指定します。
3	L	e，E，f 変換において、変換後のデータは、long double 型であることを指定します。
4	ll	d，i，o，u，x，X 変換において、変換後のデータは long long 型であることを指定します。

・変換文字

入力するデータは、各変換文字が指定する変換の種類に従って変換します。ただし、空白類文字を読み込んだ場合、変換の対象として許されていない文字を読み込んだ場合、あるいは指定されたフィールド幅を超えた場合は処理を終了します。

表 10.46 変換文字と変換の内容

	変換文字	変換の種類	変換の方式	対応するパラメータの型名
1	d	d 変換	10 進数字の文字列を整数型データに変換します。	整数型
2	i	i 変換	先頭に符号が付いている 10 進数字の文字列、あるいは最後に u(U)または l(L)が付いている 10 進数字の文字列を整数型データに変換します。また、先頭が 0x(あるいは 0X)で始まっている文字列は、16 進数字として解釈し、文字列を int 型データに変換します。さらに、先頭が 0 で始まっている文字列は、8 進数字として解釈し文字列を int 型データに変換します。	整数型
3	o	o 変換	8 進数字の文字列を整数型データに変換します。	整数型
4	u	u 変換	符号無しの 10 進数字の文字列を整数型データに変換します。	整数型
5	x	x 変換	16 進数字の文字列を整数型データに変換します。 x 変換と X 変換に意味の違いはありません。	整数型
6	X	X 変換		
7	s	s 変換	空白、水平タブ、改行文字を読み込むまでをひとつの文字列として変換します。文字列の最後にはヌル文字を付加します(変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です)。	文字型
8	c	c 変換	1 文字を入力します。この時、入力する文字が空白類文字であっても読み飛ばすことはしません。もし、空白類文字以外の文字だけを読み込む時は、%1s と指定してください。また、フィールド幅が指定されている時は、その指定分の文字が読み込まれます。したがって、この時、変換したデータを格納する記憶域は、指定分の大きさが必要です。	char 型
9	e	e 変換	浮動小数点型を示す文字列を浮動小数点型データに変換します。e 変換と E 変換、g 変換と G 変換にそれぞれ意味の違いはありません。入力形式は strtod 関数で表現できる浮動小数点型です。	浮動小数点型
10	E	E 変換		
11	f	f 変換		
12	g	g 変換		
13	G	G 変換		
14	p	p 変換	fprintf 関数において、p 変換で変換される形式の文字列をポインタ型データに変換します。	void 型へのポインタ型
15	n	データの変換は生じません。	データの入力は行わず、いままでに入力したデータの文字数が設定されます。	整数型
16	[	[変換	[の後に文字の集合、その後に]を指定します。この文字集合は、文字列を構成する文字の集合を定義しています。もし、文字集合の最初の文字が ^ でない時は、入力データはこの文字集合にない文字が最初に読み込まれるまでをひとつの文字列として入力します。もし、最初の文字が ^ の時は、 ^ を除いた文字集合の文字が最初に読み込まれるまでをひとつの文字列として入力します。入力した文字列の最後には自動的にヌル文字を付加します(変換したデータを設定する文字列は、ヌル文字を含めて格納できるサイズが必要です)。	文字型
17	%	データの変換は生じません。	%を読み込みます。	なし

変換文字が表 10.46 に示す文字以外の英文字の時は、その動作は保証しません。また、その他の文字の時は、その動作は処理系定義です。

書式付き出力

*int printf(const char *control [, arg] …)*

説　明　　データを書式に従って変換し、標準出力ファイル(stdout)へ出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：変換し出力した文字数
異常：負の値

引　数　　control　　書式を示す文字列へのポインタ
arg,…　　書式に従って出力されるデータ

例

```
#include <stdio.h>
const char *control="%s";
int ret;
char buffer[]="Hello World¥n";
   ret=printf(control,buffer);
```

備　考　　printf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、標準出力ファイル(stdout)へ出力します。
書式の仕様の詳細は fprintf 関数を参照してください。

書式付き入力

*int scanf(const char *control [, ptr] …)*

説　明　　標準入力ファイル(stdin)からデータを入力し、書式に従って変換します。

ヘッダ　　<stdio.h>

リターン値　　正常：入力変換に成功したデータの個数
異常：EOF

引　数　　control　　書式を示す文字列へのポインタ
ptr,…　　入力変換したデータを格納する記憶域へのポインタ

例

```
#include <stdio.h>
const char *control="%d";
int ret,buffer[10];
   ret=scanf(control, buffer);
```

備　考　　scanf 関数は、標準入力ファイル(stdin)からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。
scanf 関数は、入力変換に成功したデータの個数をリターン値として返します。最初の変換の前に標準入力ファイルが終了した時には EOF を返します。
書式の仕様の詳細は fscanf 関数を参照してください。
%e 変換では、double 型の場合は l、long double 型の場合は L で指定します。デフォルトの型は float 型です。

書式付き文字列出力

*int sprintf(char *s, const char *control [, arg] …)*

説　明　データを書式に従って変換し、指定した領域へ出力します。

ヘッダ　<stdio.h>

リターン値　変換した文字数

引　数

s	データを出力する記憶域へのポインタ
control	書式を示す文字列へのポインタ
arg,…	書式に従って出力されるデータ

例

```
#include <stdio.h>
char *s;
const char *control="%s";
int ret;
char buffer[]="Hello World¥n";
   ret=sprintf(s,control,buffer);
```

備　考　sprintf 関数は、control が指す書式を示す文字列に従って、引数 arg を変換、編集し、s の指す記憶域へ出力します。
変換して出力した文字列の最後には、ヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。
書式の仕様の詳細は fprintf 関数を参照してください。

書式付き文字列入力

*int sscanf(const char *s, const char *control [, ptr] …)*

説　明　指定した記憶域からデータを入力し、書式に従って変換します。

ヘッダ　<stdio.h>

リターン値　正常：入力変換に成功したデータの個数
異常：EOF

引　数

s	入力するデータがある記憶域
control	書式を示す文字列へのポインタ
ptr,…	入力変換したデータを格納する記憶域へのポインタ

例

```
#include <stdio.h>
const char *s, *control="%d";
int ret,buffer[10];
   ret=sscanf(s,control,buffer);
```

備　考　sscanf 関数は、s の指す記憶域からデータを入力し、control が指す書式を示す文字列に従って、そのデータを変換、編集して、その結果を ptr の指す記憶域へ格納します。
sscanf 関数は、入力変換に成功したデータの個数を返します。また、最初の変換の前に入力するデータが終了した時には EOF を返します。
書式の仕様の詳細は fscanf 関数を参照してください。

可変個引数ファイル出力

*int vfprintf(FILE *fp, const char *control, va_list arg)*

説　明　可変個の引数リストを書式に従って、指定したストリーム入出力用ファイルに出力します。

ヘッダ　<stdio.h>

リターン値　正常：変換し出力した文字数
異常：負の値

引　数

fp	ファイルポインタ
control	書式を示す文字列へのポインタ
arg	引数リスト

例

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int  ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vfprintf(fp, control, ap);
   va_end(ap);
}
```

備　考　vfprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、fp の示すストリーム入出力用ファイルへ出力します。
vfprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。
また、vfprintf 関数では va_end マクロは呼び出しません。
書式の仕様の詳細は fprintf 関数を参照してください。
引数リストを示す arg は、va_start(およびそれに続く va_arg マクロ)によって初期化されていなければなりません。

可変個引数出力

*int vprintf(const char *control, va_list arg)*

説　明　　可変個の引数リストを書式に従って標準出力ファイル(stdout)に出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：変換し出力した文字数
異常：負の値

引　数　　control　　書式を示す文字列へのポインタ
arg　　引数リスト

例

```
#include <stdarg.h>
#include <stdio.h>
FILE *fp;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++)
      ret=vprintf(control, ap);
   va_end(ap);
}
```

備　考　　vprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、標準出力ファイルへ出力します。
vprintf 関数は、変換し出力したデータの個数を返しますが出力エラーが発生した時は負の値を返します。
また、vprintf 関数では va_end マクロは呼び出しません。
書式の仕様の詳細は fprintf 関数を参照してください。
引数リストを示す arg は、va_start(およびそれに続く va_arg マクロ)によって初期化されていなければなりません。

可変個引数文字列出力

*int vsprintf(char *s, const char *control, va_list arg)*

説　明　可変個の引数リストを書式に従って、指定した記憶域に出力します。

ヘッダ　<stdio.h>

リターン値　正常：変換した文字数
異常：負の数

引　数

s	データを出力する記憶域へのポインタ
control	書式を示す文字列へのポインタ
arg	引数リスト

例

```
#include <stdarg.h>
#include <stdio.h>
char *s;
const char *control="%d";
int ret;

void prlist(int count ,...)
{
   va_list ap;
   int i;
   va_start(ap, count);
   for(i=0;i<count;i++) {
      ret=vsprintf(s,control,buffer);
      va_arg(ap,int);
      s += ret;
    }
}
```

備　考　vsprintf 関数は、control が指す書式を示す文字列に従って、可変個の引数リストを順に変換、編集し、s により指される記憶域へ出力します。
変換して出力した文字列の最後にヌル文字が付加されます。このヌル文字はリターン値である出力した文字数の中には含まれません。
書式の仕様の詳細は fprintf 関数を参照してください。
引数リストを示す arg は、va_start(およびそれに続く va_arg マクロ)によって初期化されていなければなりません。

ファイルから 1 文字入力

*int fgetc(FILE *fp)*

説　明　　ストリーム入出力用ファイルから 1 文字入力します。

ヘッダ　　<stdio.h>

リターン値　　正常： ファイルの終了の時　　：EOF
　　ファイルの終了でない時 ：入力した文字
　　異常： EOF

引　数　　fp　　ファイルポインタ

例

```
#include <stdio.h>
FILE *fp;
int ret;
   ret=fgetc(fp);
```

エラー条件　　読み込みエラーが発生した時、そのファイルに対してのエラー指示子が設定されます。

備　考　　fgetc 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから 1 文字入力します。
fgetc 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は、EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

ファイルから文字列入力

*char *fgets(char *s, int n, FILE *fp)*

説　明　ストリーム入出力用ファイルから文字列を入力します。

ヘッダ　<stdio.h>

リターン値　正常：ファイルの終了の時　：NULL
ファイルの終了でない時　：s
異常：NULL

引　数

s	文字列を入力する記憶域へのポインタ
n	文字列を入力する記憶域のバイト数
fp	ファイルポインタ

例

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

備　考　fgets 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから、ポインタ s の指す記憶域に文字列を入力します。
fgets 関数は、n-1 文字まであるいは改行文字を入力するまで、またはファイルの終わりになるまで文字を入力し、入力文字列の最後にヌル文字を付け加えます。
fgets 関数は通常、文字列を入力する記憶域へのポインタ s を返しますが、ファイルが終了した時やエラー発生の時は NULL を返します。
ファイルが終了した時は、s が指す記憶域の内容は変化しませんが、エラー発生の時は、s が指す記憶域の内容は保証しません。

ファイルに 1 文字出力

*int fputc(int c, FILE *fp)*

説　明　　ストリーム入出力用ファイルへ 1 文字出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：出力した文字
異常：EOF

引　数　　c　　出力する文字
fp　　ファイルポインタ

例

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

エラー条件　　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　　fputc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力ファイルへ出力します。
fputc 関数は、通常出力した文字 c を返しますが、エラー発生の時は、EOF を返します。

ファイルに文字列出力

*int fputs(const char *s, FILE *fp)*

説　明　ストリーム入出力用ファイルへ文字列を出力します。

ヘッダ　<stdio.h>

リターン値　正常：0
異常：0 以外

引　数

s	出力する文字列へのポインタ
fp	ファイルポインタ

例

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

備　考　fputs 関数は、s の指すヌル文字の直前までの文字列をファイルポインタ fp の示すストリーム入出力用ファイルへ出力します。この時、文字列の終了を示すヌル文字は出力されません。
fputs 関数は、通常 0 を返しますが、エラー発生の時は、0 以外の値を返します。

ファイルから 1 文字入力

*int getc(FILE *fp)*

説　明　ストリーム入出力用ファイルから 1 文字入力します。

ヘッダ　<stdio.h>

リターン値　正常：ファイルの終了の時　：EOF
ファイルの終了でない時　：入力した文字
異常：EOF

引　数　fp　ファイルポインタ

例
```
#include <stdio.h>
FILE *fp;
int ret;
    ret=getc(fp);
```

エラー条件　読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

備　考　getc 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから 1 文字入力します。
getc 関数は、通常入力した 1 文字を返しますがファイルの終了やエラー発生の時は、EOF を返します。またファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

1 文字入力

int getchar(void)

説　明　標準入力ファイル(stdin)から、1 文字入力します。

ヘッダ　<stdio.h>

リターン値　正常：ファイルの終了の時　：EOF
ファイルの終了でない時　：入力した文字
異常：EOF

例
```
#include <stdio.h>
int ret;
    ret=getchar();
```

エラー条件　読み込みエラーが発生した時、そのファイルに対してエラー指示子が設定されます。

備　考　getchar 関数は標準入力ファイル(stdin)から 1 文字入力します。
getchar 関数は、通常入力した 1 文字を返しますが、ファイルの終了やエラー発生の時は EOF を返します。また、ファイルの終了の時には、そのファイルに対するファイル終了指示子が設定されます。

文字列入力

*char *gets(char *s)*

説　明　　標準入力ファイル(stdin)から文字列を入力します。

ヘッダ　　<stdio.h>

リターン値　　正常： ファイルの終了の時　　：NULL
　　ファイルの終了でない時　：s
　　異常： NULL

引　数　　s　　文字列を入力する記憶域へのポインタ

例

```
#include <stdio.h>
char *ret, *s;
   ret=gets(s);
```

備　考　　gets 関数は、標準入力ファイル(stdin)から、s で始まる記憶域へ文字列を入力します。
gets 関数は、ファイルの終了か、改行文字を入力するまで文字を入力し、改行文字の代わりにヌル文字を付け加えます。
gets 関数は、通常文字列を入力する記憶域へのポインタ s を返しますが、標準入力ファイルの終了やエラー発生の時は、NULL を返します。
標準入力ファイルが終了した時は、s が指す記憶域の内容は変化しませんが、エラー発生の時は s が指す記憶域の内容は保証しません。

ファイルに 1 文字出力

*int putc(int c, FILE *fp)*

説　明　　ストリーム入出力用ファイルへ 1 文字出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：出力した文字
　　異常：EOF

引　数　　c　　出力する文字
　　fp　　ファイルポインタ

例

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

エラー条件　　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　　putc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力ファイルへ出力します。
putc 関数は、通常出力した文字 c を返しますが、エラー発生の時は EOF を返します。

1 文字出力

int putchar(int c)

説　明　　標準出力ファイル(stdout)へ1文字出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：出力した文字
異常：EOF

引　数　　c　　出力する文字

例

```
#include <stdio.h>
int c, ret;
    ret=putchar(c);
```

エラー条件　　書き出しエラーが発生した時は、そのファイルに対してエラー指示子が設定されます。

備　考　　putchar関数は、文字cを標準出力ファイル(stdout)へ出力します。putcharマクロは、通常出力した文字cを返しますが、エラー発生の時はEOFを返します。

文字列出力

*int puts(const char *s)*

説　明　　標準出力ファイル(stdout)へ文字列を出力します。

ヘッダ　　<stdio.h>

リターン値　　正常：0
異常：0以外

引　数　　s　　出力する文字列へのポインタ

例

```
#include <stdio.h>
const char *s;
int ret;
    ret=puts(s);
```

備　考　　puts関数は、sの指す文字列を標準出力ファイル(stdout)へ出力します。この時、文字列の終了を示す文字は出力されず、代わりに改行文字を出力します。
puts関数は、通常0を返しますが、エラー発生の時は0以外の値を返します。

ファイルに 1 文字返却

*int ungetc(int c, FILE *fp)*

説　明　ストリーム入出力用ファイルへ 1 文字を戻します。

ヘッダ　<stdio.h>

リターン値　正常：戻した文字
異常：EOF

引　数

c	戻す文字
fp	ファイルポインタ

例

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

備　考　ungetc 関数は、文字 c をファイルポインタ fp の示すストリーム入出力用ファイルへ戻します。
また、ここで戻された文字は、fflush，fseek，rewind 関数を呼び出さなければ次の入力データとなります。
ungetc 関数は、通常戻した文字 c を返しますが、エラー発生の時は EOF を返します。
ungetc 関数が fflush，fseek，rewind 関数を実行することなく 2 回以上呼び出された時の動作は保証しません。また、ungetc 関数が実行されるとファイルに対する現在の位置指示子が一つ戻されますが、この位置指示子がすでにファイルの先頭に位置している時は、位置指示子は保証しません。

ファイル読み込み

*size_t fread(void *ptr, size_t size, size_t n, FILE *fp)*

説　明　　ストリーム入出力用ファイルから、指定した記憶域にデータを入力します。

ヘッダ　　<stdio.h>

リターン値　　size もしくは n が 0 の時　：0
size,n がともに 0 でない時 ：入力に成功したメンバ数

引　数

ptr	データを入力する記憶域へのポインタ
size	1 メンバのバイト数
n	入力するメンバの数
fp	ファイルポインタ

例

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

備　考　　fread 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルから ptr が指す記憶域に size で指定したバイト数を 1 メンバとしたデータを n メンバ入力します。この時、ファイルに対する位置指示子は入力したバイト数分進められます。
fread 関数は、実際に入力に成功したメンバ数を返しますので、通常 n と同じ値になります。しかし、ファイルが終了した時やエラー発生の時は、それまで入力に成功したメンバ数を返しますので、n より小さな値となります。ファイルの終了かエラー発生かの区別は、ferror, feof 関数を用いて行ってください。
size もしくは n が 0 の時、リターン値として 0 を返し、ptr の指す記憶域の内容は変化しません。また、エラーが発生した時、または、メンバの途中までしか入力できなかった時は、そのファイルの位置指示子は保証しません。

ファイル書き込み

*size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fp)*

説　明　メモリ領域からストリーム入出力用ファイルにデータを出力します。

ヘッダ　<stdio.h>

リターン値　出力に成功したメンバ数

引　数

ptr	出力するデータを格納している記憶域へのポインタ
size	1 メンバのバイト数
n	出力するメンバの数
fp	ファイルポインタ

例

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

備　考　fwrite 関数は、ptr の指す記憶域から、ファイルポインタ fp の示すストリーム入出力用ファイルに、size で指定したバイト数を 1 メンバとしたデータを n メンバ出力します。
この時、ファイルに対する位置指示子は出力したバイト数進められます。
fwrite 関数は、実際に出力に成功したメンバ数を返しますので、通常 n と同じ値になります。しかし、エラー発生の時はそれまで出力に成功したメンバ数を返しますので、n より小さな値となります。
エラー発生の時、そのファイルの位置指示子は保証しません。

ファイル読み書き位置移動

*int fseek(FILE *fp, long offset, int type)*

説　明　　ストリーム入出力用ファイルの現在の読み書き位置を移動します。

ヘッダ　　<stdio.h>

リターン値　　正常：0
異常：0 以外

引　数

fp	ファイルポインタ
offset	オフセットの種類で指定された位置からのオフセット
type	オフセットの種類

例

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

備　考　　fseek 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をオフセットの種類 type で指定した場所から offset バイト先の位置に移動します。
オフセットの種類を表 10.47 に示します。
fseek 関数は、通常は 0 を返しますが、不適当な要求に対しては 0 以外の値を返します。

表 10.47　オフセットの種類

	オフセットの種類	意味
1	SEEK_SET	ファイルの先頭から offset バイト先の位置に移動します。この時、offset で指定する値は 0 か正でなければなりません。
2	SEEK_CUR	ファイルの現在位置から offset バイト先の位置に移動します。この時、offset で指定する値が正ならばファイルの後方に、負ならばファイルの先頭に向かって移動します。
3	SEEK_END	ファイルの終わりから offset バイト先の位置に移動します。この時 offset で指定する値は 0 か負でなければなりません。

テキストファイルの時は、オフセットの種類は SEEK_SET で、かつ offset は 0 かそのファイルに対する ftell 関数によって返された値でなければなりません。また、fseek 関数を呼び出すことによって ungetc 関数の効果はなくなりますので注意が必要です。

ファイル読み書き位置取得

*long ftell(FILE *fp)*

説　明　ストリーム入出力用ファイルの現在の読み書き位置を求めます。

ヘッダ　<stdio.h>

リターン値　現在の位置指示子の位置 (テキストファイル)
ファイルの先頭から現在位置までのバイト数 (バイナリファイル)

引　数　fp　ファイルポインタ

例

```
#include <stdio.h>
FILE *fp;
long ret;
   ret=ftell(fp);
```

備　考　ftell 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置を求めます。
ftell 関数は、バイナリファイルの時、ファイルの先頭から現在位置までのバイト数を返しますが、テキストファイルの時は、ここで返した値が fseek 関数で使用できるように処理系定義の値を位置指示子の位置として返します。
ftell 関数を 2 回テキストファイルに適用した時、そのリターン値の差が実際のファイル上の隔たりを表すことにはなりません。

ファイル先頭に移動

*void rewind(FILE *fp)*

説　明　ストリーム入出力用ファイルの現在の読み書き位置を、ファイルの先頭に移動します。

ヘッダ　<stdio.h>

引　数　fp　ファイルポインタ

例

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

備　考　rewind 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルの現在の読み書き位置をファイルの先頭に移動します。
また、rewind 関数は、そのファイルに対する終了指示子とエラー指示子をクリアします。
rewind 関数を呼び出すことによって、ungetc 関数の効果はなくなりますので、注意が必要です。

エラー状態クリア

*void clearerr(FILE *fp)*

説　明　　ストリーム入出力用ファイルのエラー状態をクリアします。

ヘッダ　　<stdio.h>

引　数　　fp　　ファイルポインタ

例

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

備　考　　clearerr 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルに対するエラー指示子と終了指示子をクリアします。

ファイル終了判定

*int feof(FILE *fp)*

説　明　　ストリーム入出力用ファイルが終わりであるかどうかを判定します。

ヘッダ　　<stdio.h>

リターン値　　ファイルが終わりの時　　：0 以外
ファイルが終わりでない時　：0

引　数　　fp　　ファイルポインタ

例

```
#include <stdio.h>
FILE *fp;
int ret;
   ret=feof(fp);
```

備　考　　feof 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルが終了したかどうかを判定します。
feof 関数は、指定したストリーム入出力用ファイルに対するファイル終了指示子を調べ、設定されていればファイルが終わりであるとして、0 以外の値を返します。設定されていなければ、ファイルはまだ終わりではないとして 0 を返します。

ファイルエラー状態判定

*int ferror(FILE *fp)*

説　明　　ストリーム入出力用ファイルがエラー状態であるかどうかを判定します。

ヘッダ　　<stdio.h>

リターン値　　ファイルがエラー状態の時　　：0 以外
ファイルがエラー状態でない時：0

引　数　　fp　　ファイルポインタ

例

```
#include <stdio.h>
FILE *fp;
int ret;
   ret=ferror(fp);
```

備　考　　ferror 関数は、ファイルポインタ fp の示すストリーム入出力用ファイルがエラー状態であるかどうかを判定します。
ferror 関数は、指定したストリーム入出力用ファイルに対するエラー指示子を調べ、設定されていれば、エラー状態にあるとして 0 以外の値を返します。設定されていなければ、エラー状態ではないとして 0 を返します。

エラーメッセージ出力

*void perror(const char *s)*

説　明　　標準エラーファイル(stderr)に、エラー番号に対応したエラーメッセージを出力します。

ヘッダ　　<stdio.h>

引　数　　s　　エラーメッセージへのポインタ

例

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

備　考　　perror 関数は標準エラーファイル(stderr)へ s で示されるエラーメッセージと errno とを対応させ出力します。
出力するメッセージは、もし、s が NULL でなく、s の指す文字列がヌル文字でなければ、s の指す文字列にコロンと空白とその後に処理系定義のエラーメッセージを続け、最後に改行文字を付けた形式で出力されます。

(14) <stdlib.h>

C プログラムでの標準的処理を行う関数を定義しています。

以下のマクロは、処理系定義です。

種別	定義名	説明
型(マクロ)	div_t	div 関数のリターン値の構造体の型です。
	ldiv_t	ldiv 関数のリターン値の構造体の型です。
	lldiv_t	lldiv 関数のリターン値の構造体の型です。
定数(マクロ)	RAND_MAX	rand 関数において生成する擬似乱数整数の最大値です。
関数	atof	数を表現する文字列を double 型の浮動小数点値に変換します。
	atoi	10 進数を表現する文字列を int 型の整数値に変換します。
	atol	10 進数を表現する文字列を long 型の整数値に変換します。
	atoll	10 進数を表現する文字列を long long 型の整数値に変換します。
	atolfixed	数を表現する文字列を long _ _fixed 型の固定小数点数値に変換します。
	atolaccum	数を表現する文字列を long _ _accum 型の固定小数点数値に変換します。
	strtod	数を表現する文字列を double 型の浮動小数点値に変換します。
	strtol	数を表現する文字列を long 型の整数値に変換します。
	strtoul	数を表現する文字列を unsigned long 型の整数値に変換します。
	strtoll	数を表現する文字列を long long 型の整数値に変換します。
	strtoull	数を表現する文字列を unsigned long long 型の整数値に変換します。
	strtolfixed	数を表現する文字列を long _ _fixed 型の固定小数点数値に変換します。
	strtolaccum	数を表現する文字列を long _ _accum 型の固定小数点数値に変換します。
	rand	0 から RAND_MAX の間の擬似乱数整数を生成します。
	srand	rand 関数で生成する擬似乱数列の初期値を設定します。
	calloc	記憶域を割り当てて、すべての割り当てられた記憶域を 0 で初期化します。
	free	指定された記憶域を解放します。
	malloc	記憶域を割り当てます。
	realloc	記憶域の大きさを指定された大きさに変更します。
	calloc_ _X	X 記憶域を割り当てて、すべての割り当てられた X 記憶域を 0 で初期化します。
	free_ _X	指定された X 記憶域を解放します。
	malloc_ _X	X 記憶域を割り当てます。
	realloc_ _X	X 記憶域の大きさを指定された大きさに変更します。
	calloc_ _Y	Y 記憶域を割り当てて、すべての割り当てられた Y 記憶域を 0 で初期化します。
	free_ _Y	指定された Y 記憶域を解放します。
	malloc_ _Y	Y 記憶域を割り当てます。
	realloc_ _Y	Y 記憶域の大きさを指定された大きさに変更します。
	bsearch	2 分割検索を行います。
	qsort	ソートを行います。
	abs	int 型整数の絶対値を計算します。
	div	int 型整数の除算の商と余りを計算します。
	labs	long 型整数の絶対値を計算します。
	ldiv	long 型整数の除算の商と余りを計算します。
	llabs	long long 型整数の絶対値を計算します。
	lldiv	long long 型整数の除算の商と余りを計算します。

文字列を double 型に変換

*double atof(const char *nptr)*

説　明　数を表現する文字列を、double 型の浮動小数点値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された double 型の浮動小数点値

引　数　nptr　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
double ret;
    ret=atof(nptr);
```

エラー条件　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　変換は、浮動小数点型の形式に合わない最初の文字までに対して行います。
atof 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtod 関数を使用してください。

文字列を int 型に変換

*int atoi(const char *nptr)*

説　明　10 進数を表現する文字列を、int 型の整数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された int 型の整数値

引　数　nptr　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
int ret;
    ret=atoi(nptr);
```

エラー条件　変換後の値がオーバフローをおこした時は errno を設定します。

備　考　変換は、10 進数の形式に合わない最初の文字までに対して行います。
atoi 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtol 関数を使用してください。

文字列を long 型に変換

*long atol(const char *nptr)*

説　明　　10 進数を表現する文字列を、long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　変換された long 型の整数値

引　数　　nptr　　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
long ret;
   ret=atol(nptr);
```

エラー条件　　変換後の値がオーバフローをおこした時は errno を設定します。

備　考　　変換は、10 進数の形式に合わない最初の文字までに対して行います。
atol 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtol 関数を使用してください。

文字列を long long 型に変換

*long long atoll (const char *nptr)*

説　明　　10 進数を表現する文字列を、long long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　変換された long long 型の整数値

引　数　　nptr　　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
long long ret;
   ret=atoll(nptr);
```

エラー条件　　変換後の値がオーバフローをおこした時は errno を設定します。

備　考　　変換は、10 進数の形式に合わない最初の文字までに対して行います。
atoll 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。エラー時に保証された値を得たい場合は、strtoll 関数を使用してください。

文字列を long _ _fixed 型に変換

*long _ _fixed atolfixed (const char *nptr)*

説　明　数を表現する文字列を、long _ _fixed 型の固定小数点数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された long _ _fixed 型の固定小数点数値

引　数　nptr　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
long _ _fixed ret;
   ret=atolfixed(nptr);
```

エラー条件　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
変換は、固定小数点の形式に合わない最初の文字までに対して行います。
atolfixed 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。
エラー時に保証された値を得たい場合は、strtolfixed 関数を使用してください。

文字列を long _ _accum 型に変換

*long _ _accum atolaccum (const char *nptr)*

説　明　数を表現する文字列を、long _ _accum 型の固定小数点数値に変換します。

ヘッダ　<stdlib.h>

リターン値　変換された long _ _accum 型の固定小数点数値

引　数　nptr　変換する数を表現する文字列のポインタ

例

```
#include <stdlib.h>
const char *nptr;
long _ _accum ret;
   ret=atolaccum(nptr);
```

エラー条件　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
変換は、固定小数点の形式に合わない最初の文字までに対して行います。
atolaccum 関数は、オーバフロー等のエラーが生じた場合、結果の値を保証しません。
エラー時に保証された値を得たい場合は、strtolaccum 関数を使用してください。

文字列を double 型に変換

*double strtod(const char *nptr, char **endptr)*

説　明　　数を表現する文字列を double 型の浮動小数点値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常：nptr が指している文字列が浮動小数点型を構成しない文字で始まっている時：0
nptr が指している文字列が浮動小数点型を構成する文字で始まっている時
：変換された double 型の浮動小数点値
異常：変換後の値がオーバフローの時：変換する文字列の符号と同符号をもつ HUGE_VAL
変換後の値がアンダフローの時：0

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	浮動小数点値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
double ret;
   ret=strtod(nptr,endptr);
```

エラー条件　　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　　strtod 関数は、最初の数字もしくは小数点から浮動小数点値を構成しない文字の直前までを「10.1.3(4) **浮動小数点演算の仕様**」の規則に従って double 型の浮動小数点値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、浮動小数点型を構成しない最初の文字へのポインタを設定します。数字を読み込む前に浮動小数点型を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

文字列を long 型に変換

*long strtol(const char *nptr, char **endptr, int base)*

説　明　　数を表現する文字列を long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常： nptr が指している文字列が整数を構成しない文字で始まっている時：0
nptr が指している文字列が整数を構成する文字で始まっている時
：変換された long 型の整数値
異常： 変換後の値がオーバフローの時：変換する文字列の符号に従って LONG_MAX
あるいは LONG_MIN

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ
base	変換の基数(0 又は 2～36)

例

```
#include <stdlib.h>
long ret;
const char *nptr;
char **endptr;
int base;
   ret=strtol(nptr,endptr,base);
```

エラー条件　　変換後の値がオーバフローをおこした時は、errno を設定します。

備　考　　strtol 関数は、最初の数字から整数を構成しない最初の文字の前までを long 型の整数値に変換します。
endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。
base の値が 0 の時は、「10.1.1(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A)から z(もしくは Z)までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X)も無視されます。

文字列を unsigned long 型に変換

*unsigned long strtoul (const char *nptr, char **endptr, int base)*

説　明　　数を表現する文字列を unsigned long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常： nptr が指している文字列が整数を構成しない文字で始まっている時：0
nptr が指している文字列が整数を構成する文字で始まっている時
：変換された unsigned long 型の整数値
異常： 変換後の値がオーバフローの時：ULONG_MAX

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ
base	変換の基数(0 又は 2～36)

例

```
#include <stdlib.h>
unsigned long ret;
const char *nptr;
char **endptr;
int base;
    ret=strtoul(nptr,endptr,base);
```

エラー条件　　変換後の値がオーバフローをおこした時は、errno を設定します。

備　考　　strtoul 関数は、最初の数字から整数を構成しない最初の文字の前までを unsigned long 型の整数値に変換します。
endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。
base の値が 0 の時は、「10.1.1(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A)から z(もしくは Z)までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X)も無視されます。

文字列を long long 型に変換

*long long strtoll (const char *nptr, char **endptr, int base)*

説　明　　数を表現する文字列を long long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常： nptr が指している文字列が整数を構成しない文字で始まっている時：0
nptr が指している文字列が整数を構成する文字で始まっている時
：変換された long long 型の整数値
異常： 変換後の値がオーバフローの時：変換する文字列の符号に従って LLONG_MAX
あるいは LLONG_MIN

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ
base	変換の基数(0 又は 2～36)

例

```
#include <stdlib.h>
long long ret;
const char *nptr;
char **endptr;
int base;
   ret=strtoll(nptr,endptr,base);
```

エラー条件　　変換後の値がオーバフローをおこした時は、errno を設定します。

備　考　　strtoll 関数は、最初の数字から整数を構成しない最初の文字の前までを long long 型の整数値に変換します。
endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。
base の値が 0 の時は、「10.1.1(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A)から z(もしくは Z)までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X)も無視されます。

文字列を unsigned long long 型に変換

*unsigned long long strtoull (const char *nptr, char **endptr, int base)*

説　明　　数を表現する文字列を unsigned long long 型の整数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常： nptr が指している文字列が整数を構成しない文字で始まっている時：0
nptr が指している文字列が整数を構成する文字で始まっている時
：変換された unsigned long long 型の整数値
異常： 変換後の値がオーバフローの時：ULLONG_MAX

引　数

nptr	変換する数を表現する文字列へのポインタ
endptr	整数を構成しない最初の文字へのポインタを格納する記憶域へのポインタ
base	変換の基数(0 又は 2～36)

例

```
#include <stdlib.h>
unsigned long long ret;
const char *nptr;
char **endptr;
int base;
   ret=strtoull(nptr,endptr,base);
```

エラー条件　　変換後の値がオーバフローをおこした時は、errno を設定します。

備　考　　strtoull 関数は、最初の数字から整数を構成しない最初の文字の前までを unsigned long long 型の整数値に変換します。
endptr の指す記憶域に、整数を構成しない最初の文字へのポインタを設定します。最初の数字を読み込む前に整数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL 場合、この設定は行われません。
base の値が 0 の時は、「10.1.1(4) 整数」の規則に従って変換されます。base の値が 2 から 36 の間の時は、変換する時の基数を示しています。ここで変換する文字列中の a(もしくは A)から z(もしくは Z)までの文字は、10 から 35 の値に対応付けられます。base の値より大きいか等しい文字が、変換する文字列の中にある時は、そこで変換処理を終了します。また、符号の後にある 0 は、変換の時は無視され、また、base が 16 の時の 0x(もしくは 0X)も無視されます。

文字列を long _ _fixed 型に変換

*long _ _fixed strtolfixed (const char *nptr, char **endptr)*

説　明　数を表現する文字列を long _ _fixed 型の固定小数点数値に変換します。

ヘッダ　<stdlib.h>

リターン値　正常： nptr が指している文字列が固定小数点数を構成しない文字で始まっている時：0
nptr が指している文字列が固定小数点数を構成する文字で始まっている時
：変換された long _ _fixed 型の固定小数点数値
異常： 変換後の値がオーバフローの時：変換する文字列の符号に従って LFIXED_MAX
あるいは LFIXED_MIN
変換後の値がアンダフローの時：0

引　数　nptr　変換する数を表現する文字列へのポインタ
endptr　固定小数点数値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
long _ _fixed ret;
   ret=strtolfixed(nptr,endptr);
```

エラー条件　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。strtolfixed 関数は、最初の数字もしくは小数点から固定小数点数値を構成しない文字の直前までを long _ _fixed 型の固定小数点数値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、固定小数点数を構成しない最初の文字へのポインタを設定します。数字を読み込む前に固定小数点数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

文字列を long _ _accum 型に変換

*long _ _accum strtolaccum (const char *nptr, char **endptr)*

説　明　　数を表現する文字列を long _ _accum 型の固定小数点数値に変換します。

ヘッダ　　<stdlib.h>

リターン値　　正常： nptr が指している文字列が固定小数点数を構成しない文字で始まっている時：0
nptr が指している文字列が固定小数点数を構成する文字で始まっている時
：変換された long _ _accum 型の固定小数点数値
異常： 変換後の値がオーバフローの時：変換する文字列の符号に従って LACCUM_MAX
あるいは LACCUM_MIN
変換後の値がアンダフローの時：0

引　数　　nptr　　変換する数を表現する文字列へのポインタ
endptr　　固定小数点数値を構成していない最初の文字へのポインタを格納する記憶域へのポインタ

例

```
#include <stdlib.h>
const char *nptr;
char **endptr;
long _ _accum ret;
   ret=strtolaccum(nptr,endptr);
```

エラー条件　　変換後の値がオーバフロー/アンダフローをおこした時は errno を設定します。

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。strtolaccum 関数は、最初の数字もしくは小数点から固定小数点数値を構成しない文字の直前までを long _ _accum 型の固定小数点数値に変換します。ただし、指数部も小数点も現われなかった時は、小数点は文字列の最後の数字の後に続くと仮定されます。endptr の指す領域には、固定小数点数を構成しない最初の文字へのポインタを設定します。数字を読み込む前に固定小数点数を構成しない文字がある場合は nptr の値を設定します。endptr が NULL の場合、この設定は行われません。

擬似乱数生成

int rand(void)

説　明　0 から RAND_MAX の間の擬似乱数整数を生成します。

ヘッダ　<stdlib.h>

リターン値　擬似乱数整数値

例

```
#include <stdlib.h>
int ret;
    ret=rand();
```

擬似乱数列初期設定

void srand(unsigned int seed)

説　明　rand 関数で生成する擬似乱数列の初期値を設定します。

ヘッダ　<stdlib.h>

引　数　seed　擬似乱数列生成の初期値

例

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

備　考　srand 関数は、rand 関数が擬似乱数列を生成するための初期値を設定します。したがって、rand 関数で擬似乱数値を生成している時に、再度 srand 関数で、同じ値の初期値を設定すると、擬似乱数列はくり返し生成されることになります。
rand 関数が srand 関数より先に呼ばれた時は、擬似乱数列の生成の初期値として 1 が設定されます。

初期化付き記憶域割り当て

*void *calloc(size_t nelem, size_t elsize)*

説　明　　記憶域を割り当てて、すべての割り当てられた記憶域を 0 で初期化します。

ヘッダ　　<stdlib.h>

リターン値　　正常：割り当てられた記憶域の先頭のアドレス
異常：記憶域の割り当てができなかった時、または引数のいずれかが 0 の時：NULL

引　数　　nelem　　要素の数
elsize　　一つの要素の占めるバイト数

例

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

備　考　　elsize バイト単位の記憶域を nelem 個記憶域に割り当てます。また、その割り当てられた記憶域のすべてのビットは 0 で初期化されます。

記憶域解放

*void free(void *ptr)*

説　明　　指定された記憶域を解放します。

ヘッダ　　<stdlib.h>

引　数　　ptr　　解放する記憶域のアドレス

例

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

備　考　　ptr が指す記憶域を解放し、再度割り当てて使用することを可能とします。ptr が NULL であれば何もしません。
解放しようとした記憶域が、calloc、malloc、realloc 関数で割り当てられた記憶域でない時、または、すでに free、realloc 関数によって解放されていた時の動作は保証しません。また、解放された後の記憶域を参照した時の動作も保証しません。

記憶域割り当て

*void *malloc(size_t size)*

説　明　　記憶域を割り当てます。

ヘッダ　　<stdlib.h>

リターン値　　正常：割り当てられた記憶域の先頭アドレス
異常：記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　　size　　割り当てる記憶域のバイト数

例

```
#include <stdlib.h>
size_t size;
void *ret;
   ret=malloc(size);
```

備　考　　size で示されるバイトの分だけ記憶域を割り当てます。

記憶域割り当てサイズ変更

*void *realloc(void *ptr, size_t size)*

説　明　　記憶域の大きさを指定された大きさに変更します。

ヘッダ　　<stdlib.h>

リターン値　　正常：変更した記憶域の先頭アドレス
異常：記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　　ptr　　変更する記憶域の先頭アドレス
size　　変更後の記憶域のバイト数

例

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

備　考　　ptr の指す記憶域の大きさを size で示されるバイト分の大きさの記憶域に変更します。もし、新しく割り当てられた記憶域の大きさが、変更前の記憶域の大きさより小さい時は、新しく割り当てられた記憶域の大きさまでの内容は変化しません。
ptr が calloc、malloc、realloc 関数で割り当てられた記憶域へのポインタでない時、またはすでに free、realloc 関数によって解放されている記憶域へのポインタの時、動作はされません。

初期化付き X 記憶域割り当て

*void _ _X *calloc_ _X (size_t nelem, size_t elsize)*

説　明　　X 記憶域を割り当てて、すべての割り当てられた X 記憶域を 0 で初期化します。

ヘッダ　　<stdlib.h>

リターン値　　正常：割り当てられた X 記憶域の先頭のアドレス
異常：X 記憶域の割り当てができなかった時、または引数のいずれかが 0 の時：NULL

引　数　　nelem　　要素の数
elsize　　一つの要素の占めるバイト数

例

```
#include <stdlib.h>
size_t nelem, elsize;
void _ _X *ret;
   ret=calloc_ _X(nelem,elsize);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。elsize バイト単位の X 記憶域を nelem 個 X 記憶域に割り当てます。また、その割り当てられた X 記憶域のすべてのビットは 0 で初期化されます。

X 記憶域解放

*void free_ _X (void _ _X *ptr)*

説　明　　指定された X 記憶域を解放します。

ヘッダ　　<stdlib.h>

引　数　　ptr　　解放する X 記憶域のアドレス

例

```
#include <stdlib.h>
void _ _X *ptr;
   free_ _X(ptr);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。ptr が指す X 記憶域を解放し、再度割り当てて使用することを可能とします。ptr が NULL であれば何もしません。
解放しようとした X 記憶域が、calloc_ _X、malloc_ _X、realloc_ _X 関数で割り当てられた X 記憶域でない時、または、すでに free_ _X、realloc_ _X 関数によって解放されていた時の動作は保証しません。また、解放された後の X 記憶域を参照した時の動作も保証しません。

X 記憶域割り当て

*void _ _X *malloc_ _X (size_t size)*

説　明　　X 記憶域を割り当てます。

ヘッダ　　<stdlib.h>

リターン値　　正常：割り当てられた X 記憶域の先頭アドレス
異常：X 記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　　size　　割り当てる X 記憶域のバイト数

例

```
#include <stdlib.h>
size_t size;
void _ _X *ret;
   ret=malloc_ _X(size);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
size で示されるバイトの分だけ X 記憶域を割り当てます。

X 記憶域割り当てサイズ変更

*void _ _X *realloc_ _X (void _ _X *ptr, size_t size)*

説　明　　X 記憶域の大きさを指定された大きさに変更します。

ヘッダ　　<stdlib.h>

リターン値　　正常：変更した X 記憶域の先頭アドレス
異常：X 記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　　ptr　　変更する X 記憶域の先頭アドレス
size　　変更後の X 記憶域のバイト数

例

```
#include <stdlib.h>
size_t size;
void _ _X *ptr, *ret;
   ret=realloc_ _X(ptr,size);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
ptr の指す X 記憶域の大きさを size で示されるバイト分の大きさの X 記憶域に変更します。
もし、新しく割り当てられた X 記憶域の大きさが、変更前の X 記憶域の大きさより小さい時は、新しく割り当てられた X 記憶域の大きさまでの内容は変化しません。
ptr が calloc_ _X、malloc_ _X、realloc_ _X 関数で割り当てられた X 記憶域へのポインタでない時、またはすでに free、realloc 関数によって解放されている X 記憶域へのポインタの時、動作は保証しません。

初期化付き Y 記憶域割り当て

*void _ _Y *calloc_ _Y (size_t nelem, size_t elsize)*

説　明　　Y 記憶域を割り当てて、すべての割り当てられた Y 記憶域を 0 で初期化します。

ヘッダ　　<stdlib.h>

リターン値　　正常：割り当てられた Y 記憶域の先頭のアドレス
異常：Y 記憶域の割り当てができなかった時、または引数のいずれかが 0 の時：NULL

引　数　　nelem　　要素の数
elsize　　一つの要素の占めるバイト数

例

```
#include <stdlib.h>
size_t nelem, elsize;
void _ _Y *ret;
   ret=calloc_ _Y(nelem,elsize);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。elsize バイト単位の Y 記憶域を nelem 個 Y 記憶域に割り当てます。また、その割り当てられた Y 記憶域のすべてのビットは 0 で初期化されます。

Y 記憶域解放

*void free_ _Y (void _ _Y *ptr)*

説　明　　指定された Y 記憶域を解放します。

ヘッダ　　<stdlib.h>

引　数　　ptr　　解放する Y 記憶域のアドレス

例

```
#include <stdlib.h>
void _ _Y *ptr;
   free_ _Y(ptr);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。ptr が指す Y 記憶域を解放し、再度割り当てて使用することを可能とします。ptr が NULL であれば何もしません。
解放しようとした Y 記憶域が、calloc_ _Y、malloc_ _Y、realloc_ _Y 関数で割り当てられた Y 記憶域でない時、または、すでに free_ _Y、realloc_ _Y 関数によって解放されていた時の動作は保証しません。また、解放された後の Y 記憶域を参照した時の動作も保証しません。

Y 記憶域割り当て

*void __Y *malloc__Y (size_t size)*

説　明　Y 記憶域を割り当てます。

ヘッダ　<stdlib.h>

リターン値　正常：割り当てられた Y 記憶域の先頭アドレス
異常：Y 記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　size　割り当てる Y 記憶域のバイト数

例
```
#include <stdlib.h>
size_t size;
void *ret;
   ret=malloc_ _Y(size);
```

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
size で示されるバイトの分だけ Y 記憶域を割り当てます。

Y 記憶域割り当てサイズ変更

*void __Y *realloc__Y (void __Y *ptr, size_t size)*

説　明　Y 記憶域の大きさを指定された大きさに変更します。

ヘッダ　<stdlib.h>

リターン値　正常：変更した Y 記憶域の先頭アドレス
異常：Y 記憶域の割り当てができなかった時、または size が 0 の時：NULL

引　数　ptr　変更する Y 記憶域の先頭アドレス
size　変更後の Y 記憶域のバイト数

例
```
#include <stdlib.h>
size_t size;
void *ptr, *ret;
   ret=realloc_ _Y(ptr,size);
```

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc を指定した場合のみ使用可能です。
ptr の指す Y 記憶域の大きさを size で示されるバイト分の大きさの Y 記憶域に変更します。
もし、新しく割り当てられた Y 記憶域の大きさが、変更前の Y 記憶域の大きさより小さい時は、新しく割り当てられた Y 記憶域の大きさまでの内容は変化しません。
ptr が calloc_ _Y、malloc_ _Y、realloc_ _Y 関数で割り当てられた Y 記憶域へのポインタでない時、またはすでに free_ _Y、realloc_ _Y 関数によって解放されている Y 記憶域へのポインタの時、動作は保証しません。

二分割検索

*void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))*

説　明　　二分割検索を行います。

ヘッダ　　<stdlib.h>

リターン値　　一致するメンバが検索できた時　　：一致したメンバへのポインタ
一致するメンバが検索できなかった時　：NULL

引　数

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

例

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

備　考　　key の指すデータと一致するメンバを、base の指すテーブルの中で二分割検索法によって検索します。比較を行う関数は、比較する二つのデータへのポインタ p1(第 1 引数)、p2(第 2 引数)を受け取り、以下の仕様に従って結果を返してください。

*p1<*p2 の時、負の値を返します。
*p1==*p2 の時、0 を返します。
*p1>*p2 の時、正の値を返します。

検索対象となる各メンバは、昇順に並んでいる必要があります。

ソート

*void qsort (const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))*

説　明　　ソートを行います。

ヘッダ　　<stdlib.h>

引　数

base	ソート対象となるテーブルへのポインタ
nmemb	ソート対象のメンバの数
size	ソート対象のメンバのバイト数
compar	比較を行う関数へのポインタ

例

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

備　考　　base の指すテーブルのデータをソートします。データの並べる順序は、比較を行う関数へのポインタによって指定します。この関数は、比較する二つのデータへのポインタ p1(第 1 引数)、p2(第 2 引数)を受け取り、以下の仕様に従って結果を返してください。

*p1<*p2 の時、負の値を返します。
*p1==*p2 の時、0 を返します。
*p1>*p2 の時、正の値を返します。

絶対値

int abs(int i)

説　明　　int 型整数の絶対値を求めます。

ヘッダ　　<stdlib.h>

リターン値　　i の絶対値

引　数

i	絶対値を求める整数

例

```
#include <stdlib.h>
int i, ret;
   ret=abs(i);
```

備　考　　i の絶対値を求めた結果、int 型整数値として表現できない時の動作は保証しません。

商と余り

div_t div(int numer, int denom)

説　明　　int 型整数の除算の商と余りを計算します。

ヘッダ　　<stdlib.h>

リターン値　　numer を denom で除算した結果の商と余り

引　数　　numer　　被除数
denom　　除数

例
```
#include <stdlib.h>
int numer, denom;
div_t ret;
   ret=div(numer,denom);
```

絶対値

long labs(long j)

説　明　　long 型整数の絶対値を計算します。

ヘッダ　　<stdlib.h>

リターン値　　j の絶対値

引　数　　j　　絶対値を求める整数

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

備　考　　j の絶対値を求めた結果、long 型の整数値として表現できない時の動作は保証しません。

商と余り

ldiv_t ldiv(long numer, long denom)

説　明　long 型整数の除算の商と余りを計算します。

ヘッダ　<stdlib.h>

リターン値　numer を denom で除算した結果の商と余り

引　数

numer	被除数
denom	除数

例

```
#include <stdlib.h>
long numer, denom;
ldiv_t ret;
   ret=ldiv(numer,denom);
```

絶対値

long long llabs(long long j)

説　明　long long 型整数の絶対値を計算します。

ヘッダ　<stdlib.h>

リターン値　j の絶対値

引　数

j	絶対値を求める整数

例

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

備　考　j の絶対値を求めた結果、long long 型の整数値として表現できない時の動作は保証しません。

商と余り

lldiv_t lldiv(long long numer,long long denom)

説　明　　long long 型整数の除算の商と余りを計算します。

ヘッダ　　<stdlib.h>

リターン値　　numer を denom で除算した結果の商と余り

引　数　　numer　　被除数
denom　　除数

例

```
#include <stdlib.h>
long long numer, denom;
lldiv_t ret;
   ret=lldiv(numer,denom);
```

(15) <string.h>

文字配列の操作に必要な種々の関数を定義します。

種別	定義名	説明
関数	memcpy	複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
	memcpy_ _X_ _X	複写元の X 記憶域の内容を指定した大きさ分、複写先の X 記憶域に複写します。
	memcpy_ _X_ _Y	複写元の Y 記憶域の内容を指定した大きさ分、複写先の X 記憶域に複写します。
	memcpy_ _Y_ _X	複写元の X 記憶域の内容を指定した大きさ分、複写先の Y 記憶域に複写します。
	memcpy_ _Y_ _Y	複写元の Y 記憶域の内容を指定した大きさ分、複写先の Y 記憶域に複写します。
	strcpy	複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。
	strncpy	複写元の文字列を指定された文字数分、複写先の記憶域に複写します。
	strcat	文字列の後に、文字列を連結します。
	strncat	文字列に文字列を指定した文字数分、連結します。
	memcmp	指定された二つの記憶域の比較を行います。
	strcmp	指定された二つの文字列を比較します。
	strncmp	指定された二つの文字列を指定された文字数分まで比較します。
	memchr	指定された記憶域において、指定された文字が最初に現われる位置を検索します。
	strchr	指定された文字列において、指定された文字が最初に現われる位置を検索します。
	strcspn	指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。
	strpbrk	指定された文字列において、別に指定された文字列中の文字が最初に現われる位置を検索します。
	strrchr	指定された文字列において指定された文字が最後に現われる位置を検索します。
	strspn	指定された文字列を先頭から調べ別に指定した文字列中の文字が先頭から何文字続くかを求めます。
	strstr	指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。
	strtok	指定した文字列をいくつかの字句に切り分けます。
	memset	指定された記憶域の先頭から指定された文字を指定された文字数分設定します。
	strerror	エラーメッセージを設定します。
	strlen	文字列の文字数を計算します。
	memmove	複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。複写元と複写先の記憶域が重なっていても、正しく複写されます。

処理系定義仕様

	項目	コンパイラの仕様
1	strerror 関数が返すエラーメッセージの内容	「12.3 C 標準ライブラリ関数のエラーメッセージ」を参照してください。

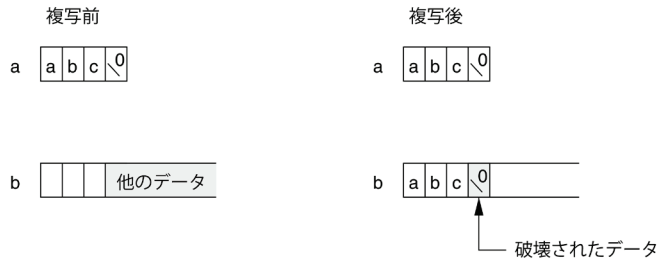
本標準インクルードファイル内で定義されている関数を使用する時は、以下の二つの事項に注意する必要があります。

(1) 文字列の複写を行う時、複写先の領域が複写元の領域よりも小さい場合、動作は保証しませんので注意が必要です。

例：

```
char a[]="abc";
char b[3];
     :
     :
strcpy(b,a);
```

この場合、配列 a のサイズは(ヌル文字を含めて)4 バイトです。したがって、strcpy 関数によって複写を行うと、配列 b の領域以外のデータを書き換えることになります。

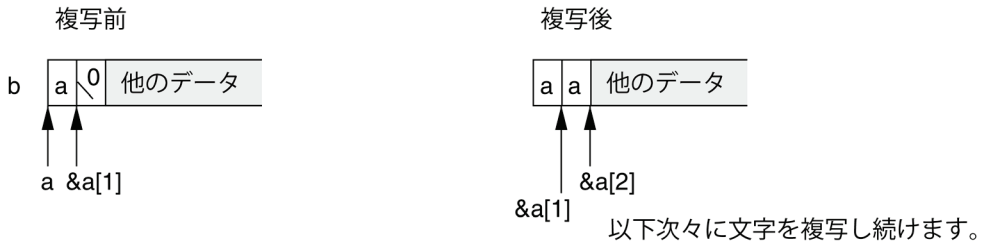


(2) 文字列の複写を行う時、複写元の領域と複写先の領域が重なっていると正しい動作は保証しませんので注意が必要です。

例：

```
int a[]="a";
     :
     :
strcpy(&a[1], a);
     :
     :
```

この場合、複写元の文字列がヌル文字に達する以前に、ヌル文字の上に文字'a'を書き込むことになります。したがって、複写元の文字列のデータに続くデータを書き換えることになります。



記憶域複写

*void *memcpy(void *s1, const void *s2, size_t n)*

説　明　複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の記憶域へのポインタ
n	複写する文字数

例

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

X 記憶域複写

*void _ _X *memcpy_ _X_ _X(void _ _X *s1, const void _ _X *s2, size_t n)*

説　明　複写元の X 記憶域の内容を、指定した大きさ分、複写先の X 記憶域に複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の X 記憶域へのポインタ
s2	複写元の X 記憶域へのポインタ
n	複写する文字数

例

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

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ使用可能です。

X 記憶域複写

*void _ _X *memcpy_ _X_ _Y(void _ _X *s1, const void _ _Y *s2, size_t n)*

説　明　　複写元の Y 記憶域の内容を、指定した大きさ分、複写先の X 記憶域に複写します。

ヘッダ　　<string.h>

リターン値　　s1 の値

引　数

s1	複写先の X 記憶域へのポインタ
s2	複写元の Y 記憶域へのポインタ
n	複写する文字数

例

```
#include <string.h>
void _ _X *ret, *s1;
const void _ _Y *s2;
size_t n;
   ret=memcpy_ _X_ _Y(s1,s2,n);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ使用可能です。

Y 記憶域複写

*void _ _Y *memcpy_ _Y_ _X(void _ _Y *s1, const void _ _X *s2, size_t n)*

説　明　　複写元の X 記憶域の内容を、指定した大きさ分、複写先の Y 記憶域に複写します。

ヘッダ　　<string.h>

リターン値　　s1 の値

引　数

s1	複写先の Y 記憶域へのポインタ
s2	複写元の X 記憶域へのポインタ
n	複写する文字数

例

```
#include <string.h>
void _ _Y *ret, *s1;
const void _ _X *s2;
size_t n;
   ret=memcpy_ _Y_ _X(s1,s2,n);
```

備　考　　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ使用可能です。

Y 記憶域複写

*void _ _Y *memcpy_ _Y_ _Y(void _ _Y *s1, const void _ _Y *s2, size_t n)*

説　明　複写元の Y 記憶域の内容を、指定した大きさ分、複写先の Y 記憶域に複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の Y 記憶域へのポインタ
s2	複写元の Y 記憶域へのポインタ
n	複写する文字数

例

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

備　考　本関数は cpu=sh2dsp|sh3dsp|sh4aldsp および dspc オプションを指定した場合のみ使用可能です。

文字列複写

*char *strcpy(char *s1, const char *s2)*

説　明　複写元の文字列の内容を、複写先の記憶域にヌル文字も含めて複写します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の文字列へのポインタ

例

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

文字列複写

*char *strncpy(char *s1, const char *s2, size_t n)*

説　明　　複写元の文字列を指定された文字数分、複写先の記憶域に複写します。

ヘッダ　　<string.h>

リターン値　　s1 の値

引　数

s1	複写先の記憶域へのポインタ
s2	複写元の文字列へのポインタ
n	複写する文字数

例

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

備　考　　s2 で指された文字列の先頭から最高 n 文字を s1 で指される記憶域に複写します。s2 で指定された文字列の文字数が n 文字より短い時は、n 文字になるまでヌル文字が付加されます。s2 で指された文字列の文字数が n 文字より長い時は、s1 に複写された文字列はヌル文字で終了しないことになります。

文字列連結

*char *strcat(char *s1, const char *s2)*

説　明　　文字列の後に、文字列を連結します。

ヘッダ　　<string.h>

リターン値　　s1 の値

引　数

s1	連結される文字列へのポインタ
s2	連結する文字列へのポインタ

例

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

備　考　　s1 で指された文字列の最後に、s2 で指された文字列を連結します。この時、s2 の指す文字列の最後を示すヌル文字も複写します。また、s1 で指された文字列の最後のヌル文字は削除されます。

文字列連結

*char *strncat(char *s1, const char *s2, size_t n)*

説　明　文字列に文字列を指定した文字数分連結します。

ヘッダ　<string.h>

リターン値　s1 の値

引　数

s1	連結される文字列へのポインタ
s2	連結する文字列へのポインタ
n	連結する文字数

例

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

備　考　s2 で指された文字列の先頭から最高 n 文字を s1 で指された文字列の最後に付加します。s1 で指された文字列の最後のヌル文字は s2 の先頭文字で置き換えられます。
また、連結された後の文字列の最後には、必ずヌル文字が付加されます。

記憶域比較

*int memcmp(const void *s1, const void *s2, size_t n)*

説　明　指定された二つの記憶域の内容を比較します。

ヘッダ　<string.h>

リターン値

s1 で指された記憶域	>	s2 で指された記憶域の時：正の値
s1 で指された記憶域	==	s2 で指された記憶域の時：0
s1 で指された記憶域	<	s2 で指された記憶域の時：負の値

引　数

s1	比較される記憶域へのポインタ
s2	比較する記憶域へのポインタ
n	比較する記憶域の文字数

例

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

備　考　s1 で指された記憶域と s2 で指された記憶域の、最初の n 文字分の内容を比較します。
この比較は処理系定義です。

文字列比較

*int strcmp(const char *s1, const char *s2)*

説　明　　指定された二つの文字列の内容を比較します。

ヘッダ　　<string.h>

リターン値　　s1 で指された文字列 > s2 で指された文字列の時：正の値
s1 で指された文字列 == s2 で指された文字列の時：0
s1 で指された文字列 < s2 で指された文字列の時：負の値

引　数　　s1　比較される文字列へのポインタ
s2　比較する文字列へのポインタ

例

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

備　考　　s1 で指された文字列と、s2 で指された文字列の内容を比較し、その結果をリターン値として設定します。
この比較は処理系定義です。

文字列比較

*int strncmp(const char *s1, const char *s2, size_t n)*

説　明　　指定された二つの文字列を指定された文字分まで比較します。

ヘッダ　　<string.h>

リターン値　　s1 で指された文字列 > s2 で指された文字列の時：正の値
s1 で指された文字列 == s2 で指された文字列の時：0
s1 で指された文字列 < s2 で指された文字列の時：負の値

引　数　　s1　比較される文字列へのポインタ
s2　比較する文字列へのポインタ
n　比較する文字数の最大値

例

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

備　考　　s1 で指された文字列と、s2 で指された文字列を最初の n 文字以内の範囲で、その内容を比較します。
この比較は処理系定義です。

記憶域内文字検索

*void *memchr(const void *s, int c, size_t n)*

説　明　　指定された記憶域において、指定された文字が最初に現われる位置を検索します。

ヘッダ　　<string.h>

リターン値　　検索の結果見つかった時　　：見つけられた文字へのポインタ
検索の結果見つからなかった時：NULL

引　数　　s　　検索を行う記憶域へのポインタ
c　　検索する文字
n　　検索を行う文字数

例

```
#include <string.h>
const void *s;
int c;
size_t n;
void *ret;
   ret=memchr(s,c,n);
```

備　考　　s で指定された記憶域の先頭から n 文字の中で最初に現われた c の文字と同一文字の位置へのポインタをリターン値として返します。

最初の文字位置

*char *strchr(const char *s, int c)*

説　明　　指定された文字列において、指定された文字が最初に現われる位置を検索します。

ヘッダ　　<string.h>

リターン値　　検索の結果見つかった時　　：見つけられた文字へのポインタ
検索の結果見つからなかった時：NULL

引　数　　s　　検索を行う文字列へのポインタ
c　　検索する文字

例

```
#include <string.h>
const char *s;
int c;
char *ret;
   ret=strchr(s,c);
```

備　考　　s で指定された文字列中で最初に現われた c の文字と同一文字へのポインタをリターン値として返します。
s によって指される文字列の終了を現わすヌル文字も検索の対象として含まれます。

指定文字群が最初に現れるまでの文字数

*size_t strcspn(const char *s1, const char *s2)*

説　明　　指定された文字列を先頭から調べ、別に指定した文字列中の文字以外の文字が先頭から何文字続くか求めます。

ヘッダ　　<string.h>

リターン値　　s2 が指す文字列を構成する文字以外の文字が構成される文字列 s1 の先頭からの長さ

引　数　　s1　　調べられる文字列へのポインタ
s2　　s1 を調べるための文字列へのポインタ

例

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

備　考　　s2 が指す文字列を構成する文字以外の文字が、文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の文字数をリターン値として返します。
s2 によって指される文字列の終了を表すヌル文字は、s2 で指された文字列の一部とはみなされません。

指定文字群が最初に現れる位置

*char *strpbrk(const char *s1, const char *s2)*

説　明　　指定された文字列内において、別に指定された文字列中の文字が最初に現われる位置を検索します。

ヘッダ　　<string.h>

リターン値　　検索の結果見つかった時　　：見つかった文字へのポインタ
検索の結果見つからなかった時：NULL

引　数　　s1　　検索を行う文字列へのポインタ
s2　　s1 内で検索する文字を示す文字列へのポインタ

例

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

備　考　　s1 で指された文字列において、s2 で指された文字列中の文字の一つが最初に現われる所を検索し、そのポインタをリターン値として返します。

最後の文字位置

*char *strrchr(const char *s, int c)*

説　明　　指定された文字列において、指定された文字が最後に現われる位置を検索します。

ヘッダ　　<string.h>

リターン値　　検索の結果見つかった時　　：見つかった文字へのポインタ
検索の結果見つからなかった時：NULL

引　数　　s　　検索を行う文字列へのポインタ
c　　検索する文字

例

```
#include <string.h>
const char *s;
int c;
char *ret;
   ret=strrchr(s,c);
```

備　考　　s で指された文字列の中で c で指定する文字と同一の文字が最後に現われた位置へのポインタをリターン値として返します。
s によって指される文字列の終了を表すヌル文字も検索の対象として含まれます。

指定文字群が連続する部分の長さ

*size_t strspn(const char *s1, const char *s2)*

説　明　　指定された文字列を先頭から調べ、別に指定した文字列中の文字が先頭から何文字続くかを求めます。

ヘッダ　　<string.h>

リターン値　　s1 の先頭から、s2 で指定した文字が続いている文字数

引　数　　s1　　調べられる文字列へのポインタ
s2　　s1 を調べるための文字列へのポインタ

例

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

備　考　　s2 が指す文字列を構成する文字が文字列として何文字続くかを s1 で指された文字列の先頭から調べ、その文字列の文字数をリターン値として返します。

最初の文字列位置

*char *strstr(const char *s1, const char *s2)*

説　明　　指定された文字列において、別に指定した文字列が最初に現われる位置を検索します。

ヘッダ　　<string.h>

リターン値　　検索の結果見つかったとき　　：見つけられた文字へのポインタ
検索の結果見つからなかったとき：NULL

引　数　　s1　　検索を行う文字列へのポインタ
s2　　検索する文字列へのポインタ

例

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

備　考　　s1 で指された文字列において、s2 で指された文字列が最初に現われる所を検索し、そのポインタをリターン値として返します。

字句切り分け

*char *strtok(char *s1, const char *s2)*

説　明　　指定した文字列をいくつかの字句に切り分けます。

ヘッダ　　<string.h>

リターン値　　字句に切り分けられた時　　：切り分けた字句の先頭へのポインタ
字句に切り分けられなかった時：NULL

引　数　　s1　　いくつかの字句に切り分ける文字列へのポインタ
s2　　文字列を切り分けるための文字からなる文字列へのポインタ

例

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

備　考　　strtok 関数は文字列を切り分けるために連続的に呼び出されます。
(a) 最初の呼び出し時
s1 で指された文字列を先頭から s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。
(b) 2 回目以降の呼び出し時
以前に切り分けられた字句の次の文字から、s2 で指された文字列中の文字によって字句に切り分けます。その結果字句に切り分けられれば、その字句の先頭へのポインタを、分けられなければ NULL をリターン値として返します。
2 回目以降の呼び出しの時は、第 1 引数には NULL を指定します。また、s2 で指された文字列は呼び出しのたびに異なっていてもかまいません。切り出された字句の最後にはヌル文字が付きます。
strtok 関数の使用例を以下に示します。

例：

```
1    #include <string.h>
2    static char s1[]="a@b,@c/@d";
3    char *ret;
4
5    ret=strtok(s1,"@");
6    ret=strtok(NULL,",@");
7    ret=strtok(NULL,"/@");
8    ret=strtok(NULL,"@");
```

【説明】
この例は、文字列「"a@b,@c/@d"」を strtok 関数を用いて a,b,c,d という字句に切り分けるプログラムを示しています。
2 行目で文字列 s1 に初期値として、文字列"a@b,@c/@d"を設定しています。
5 行目では、「@」を区切り文字として字句を切り分けるため、strtok 関数を呼び出します。この結果、文字'a'へのポインタがリターン値として得られ、文字'a'の次の最初の区切り文字である「@」にヌル文字を埋め込みます。この結果、文字列"a"が切り出されます。
以下、同一の文字列から次々に字句を切り出すために第 1 引数に NULL を指定して strtok 関数を呼び出します。
この結果、文字列"b"、"c"、"d"が次々に切り出されます。

文字繰り返し

*void *memset(void *s, int c, size_t n)*

説　明　　指定された記憶域の先頭から、指定された文字を指定された文字数分設定します。

ヘッダ　　<string.h>

リターン値　　s の値

引　数　　s　　文字が設定される記憶域へのポインタ
c　　設定する文字
n　　設定する文字数

例

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

備　考　　s で指された記憶域に n 文字分、文字 c を設定します。

エラーメッセージ文字列

*char *strerror(int s)*

説　明　　エラー番号を指定して、それに対するエラーメッセージを返します。

ヘッダ　　<string.h>

リターン値　　エラー番号に対応するエラーメッセージ (文字列) へのポインタ

引　数　　s　　エラー番号

例

```
#include <string.h>
char *ret;
int s;
   ret=strerror(s);
```

備　考　　エラー番号 s に対応するエラーメッセージへのポインタをリターン値として返します。
エラーメッセージの内容に関しては処理系定義です。
リターン値として返されたエラーメッセージを修正した時、動作は保証しません。

文字列の文字数

*size_t strlen(const char *s)*

説　明　文字列の文字数を計算します。

ヘッダ　<string.h>

リターン値　文字列の文字数

引　数　s　長さを求める文字列へのポインタ

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

備　考　s が指す文字列の終了を表すヌル文字は、文字列の文字数としては計算に入れません。

記憶域移動

*void *memmove(void *s1, const void *s2, size_t n)*

説　明　複写元の記憶域の内容を指定した大きさ分、複写先の記憶域に複写します。
また、複写元と複写先の記憶域が、重なっている部分があっても、複写元の重なっている部分を上書きする前に複写するので正しく複写されます。

ヘッダ　<string.h>

リターン値　s1 の値

引　数　s1　複写先の記憶域へのポインタ
s2　複写元の記憶域へのポインタ
n　複写する文字数

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

10.4.2 EC++クラスライブラリ

(1) ライブラリの概要

C++プログラムから標準的に利用できる EC++クラスライブラリの仕様について説明します。ここでは、クラスライブラリの種類と対応する標準インクルードファイルについて説明します。以降では、ライブラリの構成に従って各クラスライブラリの仕様について説明します。

- ライブラリの種類

表 10.48 にクラスライブラリの種類と対応する標準インクルードファイルを示します。

表 10.48 クラスライブラリの種類と標準インクルードファイルの対応

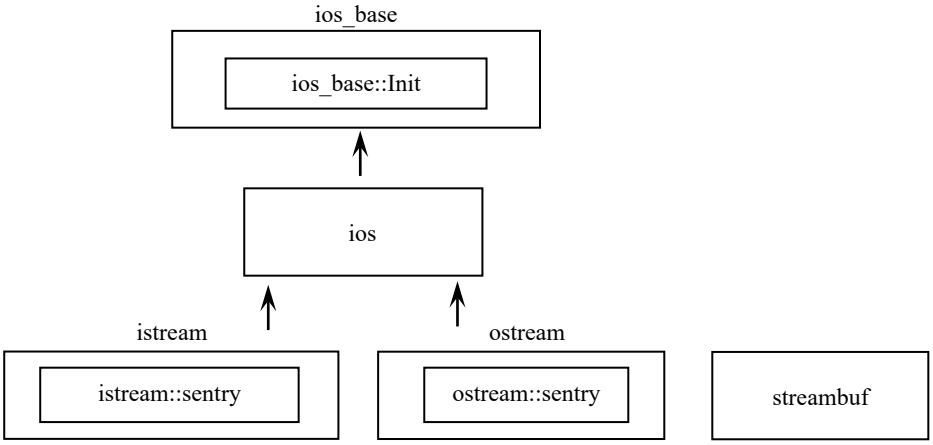
	ライブラリの種類	内容	標準インクルードファイル
1	ストリーム入出力用クラスライブラリ	入出力操作を行うライブラリです。	<ios>,<streambuf>,<istream>,<ostream>,<iostream>,<iomanip>
2	メモリ操作用ライブラリ	メモリの確保・解放を行うライブラリです。	<new>
3	複素数計算用クラスライブラリ	複素数データ演算を行うライブラリです。	<complex>
4	文字列操作用クラスライブラリ	文字列操作を行うライブラリです。	<string>

(2) ストリーム入出力用クラスライブラリ

ストリーム入出力用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <ios>
  入出力用書式設定、入出力状態管理を行うデータメンバおよび関数メンバを定義します。
  ios クラスの他に、Init クラス、ios_base クラスを定義します。
- <streambuf>
  ストリームバッファに対する関数を定義します。
- <istream>
  入力ストリームからの入力関数を定義します。
- <ostream>
  出力ストリームへの出力関数を定義します。
- <iostream>
  入出力関数を定義します。
- <iomanip>
  引数を持つマニピュレータを定義します。

これらのクラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。なお、streambuf クラスには派生関係はありません。



ストリーム入出力用クラスライブラリで共通に使用される型名を示します。

種別	定義名	説明
型	streamoff	long 型で定義された型です。
	streamsize	size_t 型で定義された型です。
	int_type	int 型で定義された型です。
	pos_type	long 型で定義された型です。
	off_type	long 型で定義された型です。

(a) ios_base::Init クラス

種別	定義名	説明
変数	init_cnt	ストリーム入出力オブジェクト数をカウントする静的データメンバです。 低水準インタフェースで 0 に初期化する必要があります。
関数	Init()	コンストラクタです
	~Init()	デストラクタです。

ios_base::Init::Init()

クラス Init のコンストラクタです。
init_cnt をインクリメントします。

ios_base::Init::~Init()

クラス Init のデストラクタです。
init_cnt をデクリメントします。

(b)　ios_base クラス

種別	定義名	説明
型	fmtflags	フォーマット制御情報を表す型です。
	iostate	ストリームバッファの入出力状態を表す型です。
	openmode	ファイルのオープンモードを表す型です。
	seekdir	ストリームバッファのシーク状態を表す型です。
変数	fmtfl	書式フラグです。
	wide	フィールド幅です。
	prec	出力時の精度(小数点以下の桁数)です。
	fillch	詰め文字です。
関数	void _ec2p_init_base()	初期化します。
	void _ec2p_copy_base(ios_base&ios_base_dt)	ios_base_dt をコピーします。
	ios_base()	コンストラクタです。
	~ios_base()	デストラクタです。
	fmtflags flags() const	書式フラグ(fmtfl)を参照します。
	fmtflags flags(fmtflags fmtflg)	fmtflg&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
	fmtflags setf(fmtflags fmtflg)	fmtflg を書式フラグ(fmtfl)に設定します。
	fmtflags setf(fmtflags fmtflg, fmtflags mask)	mask&fmtflg を書式フラグ(fmtfl)に設定します。
	void unsetf(fmtflags mask)	~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
	char fill() const	詰め文字(fillch)を参照します。
	char fill(char ch)	ch を詰め文字(fillch)に設定します。
	int precision() const	精度(prec)を参照します。
	streamsize precision(streamsize preci)	preci を精度(prec)に設定します。
	streamsize width() const	フィールド幅(wide)を参照します。
	streamsize width(streamsize wd)	wd をフィールド幅(wide)に設定します。

ios_base::fmtflags

入出力に関するフォーマット制御情報を定義します。

fmtflags の各ビットマスクの定義は以下のようになります。

```
const ios_base::fmtflags ios_base::boolalpha      = 0x0000;
const ios_base::fmtflags ios_base::skipws         = 0x0001;
const ios_base::fmtflags ios_base::unitbuf        = 0x0002;
const ios_base::fmtflags ios_base::uppercase      = 0x0004;
const ios_base::fmtflags ios_base::showbase       = 0x0008;
const ios_base::fmtflags ios_base::showpoint      = 0x0010;
const ios_base::fmtflags ios_base::showpos        = 0x0020;
const ios_base::fmtflags ios_base::left           = 0x0040;
const ios_base::fmtflags ios_base::right          = 0x0080;
const ios_base::fmtflags ios_base::internal       = 0x0100;
const ios_base::fmtflags ios_base::adjustfield    = 0x01c0;
const ios_base::fmtflags ios_base::dec            = 0x0200;
const ios_base::fmtflags ios_base::oct            = 0x0400;
const ios_base::fmtflags ios_base::hex            = 0x0800;
const ios_base::fmtflags ios_base::basefield      = 0x0e00;
const ios_base::fmtflags ios_base::scientific     = 0x1000;
const ios_base::fmtflags ios_base::fixed          = 0x2000;
const ios_base::fmtflags ios_base::floatfield     = 0x3000;
const ios_base::fmtflags ios_base::_fmtmask       = 0x3fff;
```

ios_base::iostate

ストリームバッファの入出力状態を定義します。

iostate の各ビットマスクの定義は以下のようになります。

```
const ios_base::iostate ios_base::goodbit      = 0x0;
const ios_base::iostate ios_base::eofbit       = 0x1;
const ios_base::iostate ios_base::failbit      = 0x2;
const ios_base::iostate ios_base::badbit       = 0x4;
const ios_base::iostate ios_base::_statemask   = 0x7;
```

ios_base::openmode

ファイルのオープンモードを定義します。

openmode の各ビットマスクの定義は以下のようになります。

```
const ios_base::openmode ios_base::in      = 0x01;  入力用のファイルを open します。
const ios_base::openmode ios_base::out     = 0x02;  出力用のファイルを open します。
const ios_base::openmode ios_base::ate     = 0x04;  オープン後一度だけ eof に seek します。
const ios_base::openmode ios_base::app     = 0x08;  書き込む度に eof に seek します。
const ios_base::openmode ios_base::trunc   = 0x10;  ファイルを上書きモードで open します。
const ios_base::openmode ios_base::binary  = 0x20;  ファイルをバイナリモードで open します。
```

ios_base::seekdir
ストリームバッファのシーク状態を定義します。
引き続き入力または出力を行うためのストリーム内の位置を決定します。
seekdir の各ビットマスクの定義は以下のようになります。

```
const ios_base::seekdir ios_base::beg    = 0x0;
const ios_base::seekdir ios_base::cur    = 0x1;
const ios_base::seekdir ios_base::end    = 0x2;
```

void ios_base::_ec2p_init_base()
以下の値で初期設定します。

```
fmtfl   = skipws | dec;
wide    = 0;
prec    = 6;
fillch  = ' ';
```

void ios_base::_ec2p_copy_base(ios_base& ios_base_dt)
ios_base_dt をコピーします。

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

ios_base::~ios_base()
クラス ios_base のデストラクタです。

ios_base::fmtflags ios_base::flags() const
書式フラグ(fmtfl)を参照します。
リターン値は、書式フラグ(fmtfl)です。

ios_base::fmtflags ios_base::flags(fmtflags fmtflg)
fmtflg&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl)です。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg)
fmtflg を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl)です。

ios_base::fmtflags ios_base::setf(fmtflags fmtflg, fmtflags mask)
mask&fmtflg の値を書式フラグ(fmtfl)に設定します。
リターン値は、設定前の書式フラグ(fmtfl)です。

void ios_base::unsetf(fmtflags mask)
~mask&書式フラグ(fmtfl)を書式フラグ(fmtfl)に設定します。

char ios_base::fill() const
詰め文字(fillch)を参照します。
リターン値は、詰め文字(fillch)です。
char ios_base::fill(char ch)
ch を詰め文字として設定します。
リターン値は、設定前の詰め文字(fillch)です。

int ios_base::precision() const
精度(prec)を参照します。
リターン値は、精度(prec)です。

streamsize ios_base::precision(streamsize preci)
preci を精度(prec)に設定します。
リターン値は、設定前の精度(prec)です。

streamsize ios_base::width() const
フィールド幅(wide)を参照します。
リターン値は、フィールド幅(wide)です。

streamsize ios_base::width(streamsize wd)
wd をフィールド幅(wide)に設定します。
リターン値は、設定前のフィールド幅(wide)です。

(c)　ios クラス

種別	定義名	説明
変数	sb	streambuf オブジェクトへのポインタです。
	tiestr	ostream オブジェクトへのポインタです。
	state	streambuf への状態フラグです。
関数	ios()	コンストラクタです。
	ios(streambuf* sbptr)	
	void init(streambuf* sbptr)	初期設定を行います。
	virtual ~ios()	デストラクタです。
	operator void*() const	エラー有無(!state&(badbit | failbit))を判定します。
	bool operator!() const	エラー有無(state&(badbit | failbit))を判定します。
	iostate rdstate() const	状態フラグ(state)を参照します。
	void clear(iostate st = goodbit)	指定された状態(st)を除いて状態フラグ(state)をクリアします。
	void setstate(iostate st)	st を状態フラグ(state)に設定します。
	bool good() const	エラー有無(state==goodbit)を判定します。
	bool eof() const	入力ストリームの最後かどうか(state&eofbit)を判定します。
	bool bad() const	エラー有無(state&badbit)を判定します。
	bool fail() const	入力テキストが要求パターンと不一致であるかどうか(state&(badbit | failbit))判定します。
	ostream* tie() const	ostream オブジェクトへのポインタ(tiestr)を参照します。
	ostream* tie(ostream* tstrptr)	tstrptr を ostream オブジェクトへのポインタ(tiestr)に設定します。
	streambuf* rdbuf() const	streambuf オブジェクトへのポインタ(sb)を参照します。
	streambuf* rdbuf(streambuf* sbptr)	sbptr を streambuf オブジェクトへのポインタ(sb)に設定します。
	ios& copyfmt(const ios& rhs)	rhs の状態フラグ(state)をコピーします。

ios::ios()

クラス ios のコンストラクタです。
init(0)を呼び出し、初期値をそのメンバオブジェクトに設定します。

ios::ios(streambuf* sbptr)

クラス ios のコンストラクタです。
init(sbptr)を呼び出し、初期値をそのメンバオブジェクトに設定します。

void ios::init(streambuf* sbptr)

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

virtual ios::~ios()

クラス ios のデストラクタです。

ios::operator void*() const

エラー有無(!state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
エラー有の場合：false
エラー無の場合：true

bool ios::operator!() const

エラー有無(state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
エラー有の場合：true
エラー無の場合：false

iostate ios::rdstate() const

状態フラグ(state) を参照します。
リターン値は、状態フラグ(state)です。

void ios::clear(iostate st = goodbit)

指定された状態(st)を除いて状態フラグ(state)をクリアします。
streambuf オブジェクトへのポインタ(sb)が 0 のときは、状態フラグ(state)に badbit を設定します。

void ios::setstate(iostate st)

st を状態フラグ(state)に設定します。

bool ios::good() const

エラー有無(state==goodbit)を判定します。
リターン値は次のとおりです。
エラー有の場合：false
エラー無の場合：true

bool ios::eof() const

入力ストリームの最後かどうか(state&eofbit)を判定します。
リターン値は次のとおりです。
入力ストリームの最後の場合　　：true
入力ストリームの最後以外の場合：false

bool ios::bad() const

エラー有無(state&badbit)を判定します。
リターン値は次のとおりです。
エラー有の場合：true
エラー無の場合：false

bool ios::fail() const

入力テキストが要求パターンと不一致であるかどうか(state&(badbit | failbit))を判定します。
リターン値は次のとおりです。
不一致の場合：true
一致の場合　：false

ostream* ios::tie() const
ostream オブジェクトへのポインタ(tiestr)を参照します。
リターン値は、ostream オブジェクトへのポインタ(tiestr)です。

ostream* ios::tie(ostream* tstrptr)
tstrptr を ostream オブジェクトへのポインタ(tiestr)に設定します。
リターン値は、設定前の ostream オブジェクトへのポインタ(tiestr)です。

streambuf* ios::rdbuf() const
streambuf オブジェクトへのポインタ(sb)を参照します。
リターン値は、streambuf オブジェクトへのポインタ(sb)です。

streambuf* ios::rdbuf(streambuf* sbptr)
sbptr を streambuf オブジェクトへのポインタ(sb)に設定します。
リターン値は、設定前の streambuf オブジェクトへのポインタ(sb)です。

ios& ios::copyfmt(const ios& rhs)
rhs の状態フラグ(state)をコピーします。
リターン値は*this です。

(d) ios クラスマニピュレータ

種別	定義名	説明
関数	ios_base& boolalpha(ios_base& str)	bool 型の書式に設定します。
	ios_base& noboolalpha(ios_base& str)	bool 型の書式をクリアします。
	ios_base& showbase(ios_base& str)	基数表示接頭辞モードに設定します。
	ios_base& noshowbase(ios_base& str)	基数表示接頭辞モードをクリアします。
	ios_base& showpoint(ios_base& str)	小数点生成モードに設定します。
	ios_base& noshowpoint(ios_base& str)	小数点生成モードをクリアします。
	ios_base& showpos(ios_base& str)	+記号生成モードに設定します。
	ios_base& noshowpos(ios_base& str)	+記号生成モードをクリアします。
	ios_base& skipws(ios_base& str)	空白読み飛ばしモードに設定します。
	ios_base& noskipws(ios_base& str)	空白読み飛ばしモードをクリアします。
	ios_base& uppercase(ios_base& str)	大文字変換モードに設定します。
	ios_base& nouppercase(ios_base& str)	大文字変換モードをクリアします。
	ios_base& internal(ios_base& str)	内部補充モードに設定します。
	ios_base& left(ios_base& str)	左側補充モードに設定します。
	ios_base& right(ios_base& str)	右側補充モードに設定します。
	ios_base& dec(ios_base& str)	10 進モードに設定します。
	ios_base& hex(ios_base& str)	16 進モードに設定します。
	ios_base& oct(ios_base& str)	8 進モードに設定します。
	ios_base& fixed(ios_base& str)	固定小数点モードに設定します。
	ios_base& scientific(ios_base& str)	科学表記法モードに設定します。

ios_base& boolalpha(ios_base& str)

bool 型の書式に設定します。
リターン値は str です。

ios_base& noboolalpha(ios_base& str)

bool 型の書式をクリアします。
リターン値は str です。

ios_base& showbase(ios_base& str)

データのはじめに基数を表示させるモードに設定します。
16 進数のときは、0x を行の先頭に付加します。10 進数のときは、そのまま出力します。
8 進数のときは、0 を行の先頭に付加します。
リターン値は str です。

ios_base& noshowbase(ios_base& str)

データのはじめに基数を表示させるモードをクリアします。
リターン値は str です。

ios_base& showpoint(ios_base& str)
小数点を出力するモードに設定します。
精度の指定がない場合、小数点以下 6 桁で表示します。
リターン値は str です。

ios_base& noshowpoint(ios_base& str)
小数点を出力するモードをクリアします。
リターン値は str です。

ios_base& showpos(ios_base& str)
+記号生成出力モード(正の数に対して+の符号を付加)に設定します。
リターン値は str です。

ios_base& noshowpos(ios_base& str)
+記号生成出力モードをクリアします。
リターン値は str です。

ios_base& skipws(ios_base& str)
空白読み飛ばし入力モード(連続する空白をスキップ)に設定します。
リターン値は str です。

ios_base& noskipws(ios_base& str)
空白読み飛ばし入力モードをクリアします。
リターン値は str です。

ios_base& uppercase(ios_base& str)
大文字変換出力モードに設定します。
16 進の基数表現が大文字の 0X になり、数値自体も大文字になります。
浮動小数点の指数表現も大文字の E になります。
リターン値は str です。

ios_base& nouppercase(ios_base& str)
大文字変換出力モードをクリアします。
リターン値は str です。

ios_base& internal(ios_base& str)
フィールド幅(wide)の範囲で出力時に
　符号、基数
　詰め文字(fill)
　数値
の順で出力します。
リターン値は str です。

ios_base& left(ios_base& str)
フィールド幅(wide)の範囲で出力時に左詰めします。
リターン値は str です。

ios_base& right(ios_base& str)
フィールド幅(wide)の範囲で出力時に右詰めします。
リターン値は str です。

ios_base& dec(ios_base& str)
変換基数を 10 進モードに設定します。
リターン値は str です。

ios_base& hex(ios_base& str)
変換基数を 16 進モードに設定します。
リターン値は str です。

ios_base& oct(ios_base& str)
変換基数を 8 進モードに設定します。
リターン値は str です。

ios_base& fixed(ios_base& str)
固定小数点出力モードに設定します。
リターン値は str です。

ios_base& scientific(ios_base& str)
科学表記法出力モード(指数表記)に設定します。
リターン値は str です。

(e)　streambuf クラス

種別	定義名	説明
定数	eof	ファイル終了を示します。
変数	_B_cnt_ptr	バッファの有効データ長へのポインタです。
	B_beg_ptr	バッファのベースポインタへのポインタです。
	_B_len_ptr	バッファの長さへのポインタです。
	B_next_ptr	バッファの次の読み出し位置へのポインタです。
	B_end_ptr	バッファの終端位置へのポインタです。
	B_beg_pptr	制御バッファの先頭位置へのポインタです。
	B_next_pptr	バッファの次の読み出し位置へのポインタです。
	C_flg_ptr	ファイルの入出力制御フラグへのポインタです。
関数	char* _ec2p_getflag() const	ファイル入出力制御フラグのポインタを参照します。
	char*& _ec2p_gnptr()	バッファの次の読み出し位置へのポインタを参照します。
	char*& _ec2p_pnptr()	バッファの次の書き込み位置へのポインタを参照します。
	void _ec2p_bcntplus()	バッファの有効データ長をインクリメントします。
	void _ec2p_bcntminus()	バッファの有効データ長をデクリメントします。
	void _ec2p_setbPtr(char** begptr, char** curptr, long* cntptr, long* lenptr, char* flgptr)	streambuf のポインタを設定します。
	streambuf()	コンストラクタです。
	virtual ~streambuf()	デストラクタです。
	streambuf* pubsetbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。この関数では setbuf(s,n)*1 を呼び出します。
	pos_type pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	way で指定された方法で入出力ストリームの読み書き位置を移動させます。この関数では seekoff(off,way,which)*1 を呼び出します。
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	ストリームの先頭から現在の位置までのオフセットを求めます。この関数では seekpos(sp,which)*1 を呼び出します。
	int pubsync()	出力ストリームをフラッシュします。この関数では sync()*1 を呼び出します。
	streamsize in_avail()	入力ストリームの最後尾から現在位置までのオフセットを求めます。
	int_type snextc()	次の一文字を読み込みます。
	int_type sbumpc()	一文字読み込みポインタを次に設定します。
	int_type sgetc()	一文字読み込みます。

種別	定義名	説明
関数	int sgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	int_type sputbackc(char c)	読み込み位置をプットバックします。
	int sungetc()	読み込み位置をプットバックします。
	int sputc(char c)	文字 c を挿入します。
	int_type sputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	char* eback() const	入力ストリームの先頭ポインタを求めます。
	char* gptr() const	入力ストリームの次ポインタを求めます。
	char* egptr() const	入力ストリームの最後尾ポインタを求めます。
	void gbump(int n)	入力ストリームの次ポインタを n 進めます。
	void setg(char* gbeg, char* gnext, char* gend)	入力ストリームの各ポインタを代入します。
	char* pbase() const	出力ストリームの先頭ポインタを求めます。
	char* pptr() const	出力ストリームの次ポインタを求めます。
	char* epptr() const	出力ストリームの最後尾ポインタを求めます。
	void pbump(int n)	出力ストリームの次ポインタを n 進めます。
	void setp(char* pbeg, char* pend)	出力ストリームの各ポインタを設定します。
	virtual streambuf* setbuf(char* s, streamsize n)*1	派生する各クラスごとに、個別に定義する演算を実行します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))*1	ストリーム位置を変更します。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out)) *1	ストリーム位置を変更します。
	virtual int sync()*1	出力ストリームをフラッシュします。
	virtual int showmanyc()*1	入力ストリームの有効な文字数を求めます。
	virtual streamsize xsgetn(char* s, streamsize n)	s の指す記憶領域に n 個の文字を設定します。
	virtual int_type underflow()*1	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type uflow()*1	次ポインタの一文字を読み込みます。
	virtual int_type pbackfail(int_type c = eof)*1	c によって示される文字をプットバックします。
	virtual streamsize xsputn(const char* s, streamsize n)	s の指す n 個の文字を挿入します。
	virtual int_type overflow(int_type c = eof)*1	c を出力ストリームに挿入します。

【注】*1 このクラスでは処理を定義していません。

streambuf::streambuf()
コンストラクタです。
以下の値で初期化します。
_B_cnt_ptr = B_beg_ptr = B_next_ptr = B_end_ptr = C_flg_ptr = _B_len_ptr = 0
B_beg_pptr = &B_beg_ptr
B_next_pptr = &B_next_ptr

virtual streambuf::~streambuf()
デストラクタです。

streambuf* streambuf::pubsetbuf(char* s, streamsize n)
ストリーム入出力用のバッファを確保します。
この関数では setbuf(s,n)を呼び出します。
リターン値は、*this です。

pos_type streambuf::pubseekoff(off_type off, ios _base::seekdir way,
ios_base::openmode which = (ios_base::openmode)(ios_base::in | ios_base::out))
way で指定された方法で入出力ストリームの読み書き位置を移動させます。
この関数では seekoff(off,way,which)を呼び出します。
リターン値は、新たに設定されたストリームの位置です。

pos_type streambuf::pubseekpos(pos_type sp, ios_base::openmode which =
(ios_base::openmode)(ios_base::in | ios_base::out))
ストリームの先頭から現在の位置までのオフセットを求めます。
現在のストリームポインタから sp だけ移動します。
この関数では seekpos(sp,which)を呼び出します。
リターン値は、先頭からのオフセットです。

int streambuf::pubsync()
出力ストリームをフラッシュします。
この関数では sync()を呼び出します。
リターン値は 0 です。

streamsize streambuf::in_avail()
入力ストリームの最後尾から現在位置までのオフセットを求めます。
リターン値は次のとおりです。
読み込み位置が有効の場合　：最後尾から現在位置までのオフセット
読み込み位置が無効の場合　：0(showmanyc()を呼び出します)

int_type streambuf::snextc()
一文字読み込みます。読み込んだ文字が eof でなければ、次の一文字を読み込みます。
リターン値は次のとおりです。
eof でない場合　：読み込んだ文字
eof の場合　：eof

int_type streambuf::sbumpc()

一文字読み込みポインタを次に設定します。

リターン値は次のとおりです。

読み込み位置が無効でない場合 ：読み込んだ文字
読み込み位置が無効の場合 ：eof

int_type streambuf::sgetc()

一文字読み込みます。

リターン値は次のとおりです。

読み込み位置が無効でない場合 ：読み込んだ文字
読み込み位置が無効の場合 ：eof

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

s の指す記憶領域に n 個の文字を設定します。

文字列中に eof を検出した場合、設定を終了します。

リターン値は、設定した文字数です。

int_type streambuf::sputbackc(char c)

読み込み位置が正常で読み込み位置のプットバックデータが c と同一の場合、読み込み位置をプットバックします。

リターン値は次のとおりです。

プットバックできた場合 ：c の値
プットバックできなかった場合 ：eof

int streambuf::sungetc()

読み込み位置が正常である場合、読み込み位置をプットバックします。

リターン値は次のとおりです。

プットバックできた場合 ：プットバックした値
プットバックできなかった場合 ：eof

int streambuf::sputc(char c)

文字 c を挿入します。

リターン値は次のとおりです。

書き込み位置が正しい場合 ：c の値
書き込み位置が不正な場合 ：eof

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

s の指す n 個の文字を挿入します。

バッファが n より小さい場合は、バッファサイズ分だけ挿入します。

リターン値は、挿入された文字数です。

char* streambuf::eback() const

入力ストリームの先頭ポインタを求めます。

リターン値は、先頭ポインタです。

char* streambuf::gptr() const
入力ストリームの次ポインタを求めます。
リターン値は、次ポインタです。

char* streambuf::egptr() const
入力ストリームの最後尾ポインタを求めます。
リターン値は、最後尾ポインタです。

void streambuf::gbump(int n)
入力ストリームの次ポインタを n 進めます。

void streambuf::setg(char* gbeg, char* gnext, char* gend)
入力ストリームの各ポインタに、以下の設定を行います。

```
*B_beg_pptr  = gbeg;
*B_next_pptr = gnext;
B_end_ptr    = gend;
*_B_cnt_ptr  = gend-gnext;
*_B_len_ptr  = gend-gbeg;
```

char* streambuf::pbase() const
出力ストリームの先頭ポインタを求めます。
リターン値は、先頭ポインタです。

char* streambuf::pptr() const
出力ストリームの次ポインタを求めます。
リターン値は、次ポインタです。

char* streambuf::epptr() const
出力ストリームの最後尾ポインタを求めます。
リターン値は、最後尾ポインタです。

void streambuf::pbump(int n)
出力ストリームの次ポインタを n 進めます。

void streambuf::setp(char* pbeg, char* pend)
出力ストリームの各ポインタに、以下の設定を行います。

```
*B_beg_pptr  = pbeg;
*B_next_pptr = pbeg;
B_end_ptr    = pend;
*_B_cnt_ptr  = pend-pbeg;
*_B_len_ptr  = pend-pbeg;
```

virtual streambuf* streambuf::setbuf(char* s, streamsize n)
streambuf から派生する各クラスごとに、個別に定義する演算を実行します。
リターン値は*this です。このクラスでは処理を定義していません。

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

ストリーム位置を変更します。
リターン値は-1 です。このクラスでは処理を定義していません。

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

ストリーム位置を変更します。
リターン値は-1 です。このクラスでは処理を定義していません。

virtual int streambuf::sync()

出力ストリームをフラッシュします。
リターン値は 0 です。このクラスでは処理を定義していません。

virtual int streambuf::showmanyc()

入力ストリームの有効な文字数を求めます。
リターン値は 0 です。このクラスでは処理を定義していません。

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

s の指す記憶領域に n 個の文字を設定します。
バッファが n より小さい場合は、バッファサイズ分だけ設定します。
リターン値は、入力された文字数です。

virtual int_type streambuf::underflow()

ストリーム位置を動かさずに一文字読み込みます。
リターン値は eof です。このクラスでは処理を定義していません。

virtual int_type streambuf::uflow()

次ポインタの一文字を読み込みます。
リターン値は eof です。このクラスでは処理を定義していません。

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

c によって示される文字をプットバックします。
リターン値は eof です。このクラスでは処理を定義していません。

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

s の指す n 個の文字を挿入します。
バッファが n より小さい場合は、バッファサイズ分だけ挿入します。
リターン値は、挿入された文字数です。

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

c を出力ストリームに挿入します。
リターン値は eof です。このクラスでは処理を定義していません。

(f)　istream::sentry クラス

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

istream::sentry::sentry(istream& is, bool noskipws = _false)

内部クラス sentry のコンストラクタです。
good()が非 0 の場合、フォーマット付きまたはフォーマットなし入力を可能にします。
tie()が非 0 の場合、関連する出力ストリームをフラッシュします。

istream::sentry::~sentry()

内部クラス sentry のデストラクタです。

istream::sentry::operator bool()

ok_を参照します。
リターン値は ok_です。

(g) istream クラス

種別	定義名	説明
変数	chcount	最後にコールされた入力関数が抽出した文字数です。
関数	int _ec2p_getistr(char* str, unsigned int dig, int mode)	str を dig が示す基数で変換します。
	istream(streambuf* sb)	コンストラクタです。
	virtual ~istream()	デストラクタです。
	istream& operator>>(bool& n)	抽出した文字を n に格納します。
	istream& operator>>(short& n)	
	istream& operator>>(unsigned short& n)	
	istream& operator>>(int& n)	
	istream& operator>>(unsigned int& n)	
	istream& operator>>(long& n)	
	istream& operator>>(unsigned long& n)	
	istream& operator>>(long long& n)	
	istream& operator>>(unsigned long long& n)	
	istream& operator>>(float& n)	
	istream& operator>>(double& n)	
	istream& operator>>(long double& n)	
	istream& operator>>(void*& p)	void を指すポインタに変換して p に格納します。
	istream& operator>>(streambuf* sb)	文字を抽出し、sb の指す記憶領域へ格納します。
	streamsize gcount() const	chcount(抽出文字数)を求めます。
	int_type get()	文字を抽出します。
	istream& get(char& c)	文字を抽出し c に格納します。
	istream& get(signed char& c)	
	istream& get(unsigned char& c)	
	istream& get(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& get(signed char* s, streamsize n)	
	istream& get(unsigned char* s, streamsize n)	
	istream& get(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。文字列内に 'delim'を検出したら、入力を終了します。
	istream& get(signed char* s, streamsize n, char delim)	
	istream& get(unsigned char* s, streamsize n, char delim)	
	istream& get(streambuf& sb)	文字列を抽出し、sb の指す記憶領域に格納します。
	istream& get(streambuf& sb, char delim)	文字列を抽出し、sb の指す記憶領域に格納します。途中で文字'delim'を検出したら、入力を終了します。

種別	定義名	説明
関数	istream& getline(char* s, streamsize n)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
	istream& getline(signed char* s, streamsize n)	
	istream& getline(unsigned char* s, streamsize n)	
	istream& getline(char* s, streamsize n, char delim)	サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。途中で文字 'delim'を検出したら、入力を終了します。
	istream& getline(signed char* s, streamsize n, char delim)	
	istream& getline(unsigned char* s, streamsize n, char delim)	
	istream& ignore(streamsize n = 1, int_type delim = streambuf::eof)	n 個の文字を読み飛ばします。途中で文字'delim'を検出したら、読み飛ばし処理を中止します。
	int_type peek()	次の入手可能な入力文字を求めます。
	istream& read(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	istream& read(signed char* s, streamsize n)	
	istream& read(unsigned char* s, streamsize n)	
	streamsize readsome(char* s, streamsize n)	サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
	streamsize readsome(signed char* s, streamsize n)	
	streamsize readsome(unsigned char* s, streamsize n)	
	istream& putback(char c)	文字を入力ストリームに戻します。
	istream& unget()	入力ストリームの位置を戻します。
	int sync()	入力ストリームがあるかどうかを調べます。この関数は streambuf::pubsync()を呼び出します。
	pos_type tellg()	入力ストリームの位置を調べます。この関数は streambuf::pubseekoff(0,cur,in)を呼び出します。
	istream& seekg(pos_type pos)	現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos)を呼び出します。
	istream& seekg(off_type off, ios_base::seekdir dir)	dir で指定された方法で入力ストリームの読み込み位置を移動します。この関数は streambuf::pubseekoff(off,dir)を呼び出します。

int istream::_ec2p_getistr(char* str, unsigned int dig, int mode)
str を dig が示す基数で変換します。
リターン値は、変換した基数です。

istream::istream(streambuf* sb)
クラス istream のコンストラクタです。
ios::init(sb)を呼び出します。
chcount=0 の設定を行います。

virtual istream::~istream()
クラス istream のデストラクタです。

istream& istream::operator>>(bool& n)
istream& istream::operator>>(short& n)
istream& istream::operator>>(unsigned short& n)
istream& istream::operator>>(int& n)
istream& istream::operator>>(unsigned int& n)
istream& istream::operator>>(long& n)
istream& istream::operator>>(unsigned long& n)
istream& istream::operator>>(long long& n)
istream& istream::operator>>(unsigned long long& n)
istream& istream::operator>>(float& n)
istream& istream::operator>>(double& n)
istream& istream::operator>>(long double& n)
抽出した文字を n に格納します。
リターン値は*this です。

istream& istream::operator>>(void*& p)
抽出した文字を void*型に変換し、p の指す記憶領域に格納します。
リターン値は*this です。

istream& istream::operator>>(streambuf* sb)
文字を抽出し、sb の指す記憶領域に格納します。
抽出文字がない場合は、setstate(failbit)を呼び出します。
リターン値は*this です。

streamsize istream::gcount() const
chcount(抽出文字数)を参照します。
リターン値は chcount です。

int_type istream::get()
文字を抽出します。
リターン値は次のとおりです。
抽出可能の場合：抽出した文字
抽出不可の場合：setstate(failbat)を呼び出して、streambuf::eof

istream& istream::get(char& c)
istream& istream::get(signed char& c)
istream& istream::get(unsigned char& c)
　文字を抽出し c に格納します。抽出した文字が streambuf::eof の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::get(char* s, streamsize n)
istream& istream::get(signed char* s, streamsize n)
istream& istream::get(unsigned char* s, streamsize n)
　サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::get(char* s, streamsize n, char delim)
istream& istream::get(signed char* s, streamsize n, char delim)
istream& istream::get(unsigned char* s, streamsize n, char delim)
　サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
　文字列内に'delim'を検出したら、終了します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::get(streambuf& sb)
　文字列を抽出し、sb の指す記憶領域に格納します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::get(streambuf& sb, char delim)
　文字列を抽出し、sb の指す記憶領域に格納します。
　途中で文字'delim'を検出したら、終了します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::getline(char* s, streamsize n)
istream& istream::getline(signed char* s, streamsize n)
istream& istream::getline(unsigned char* s, streamsize n)
　サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

istream& istream::getline(char* s, streamsize n, char delim)
istream& istream::getline(signed char* s, streamsize n, char delim)
istream& istream::getline(unsigned char* s, streamsize n, char delim)
　サイズ n-1 の文字列を抽出し、s の指す記憶領域に格納します。
　途中で文字'delim'を検出したら、終了します。
　ok_==false または抽出した文字数が 0 の場合は、failbit を設定します。
　リターン値は*this です。

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

n 個の文字を読み飛ばします。
途中で文字'delim'を検出したら、読み飛ばし処理を中止します。
リターン値は*this です。

int_type istream::peek()

次の入力可能な入力文字を求めます。
リターン値は次のとおりです。

ok_==false の場合　: streambuf::eof
ok_!=false の場合　: rdbuf()->sgetc()

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

ok_!=false の場合、サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
抽出した文字数が n と異なる場合、eofbit を設定します。
リターン値は*this です。

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

サイズ n の文字列を抽出し、s の指す記憶領域に格納します。
文字数がストリームサイズより大きければ、ストリームサイズ分格納します。
リターン値は、抽出した文字数です。

istream& istream::putback(char c)

文字 c を入力ストリームに戻します。プットバックした文字が streambuf::eof の場合は、badbit を設定します。
リターン値は*this です。

istream& istream::unget()

入力ストリームのポインタをひとつ戻します。
抽出した文字が streambuf::eof の場合、badbit を設定します。
リターン値は*this です。

int istream::sync()

入力ストリームがあるかどうかを調べます。
この関数は streambuf::pubsync()を呼び出します。
リターン値は次のとおりです。

入力ストリームがない場合　: streambuf::eof
入力ストリームがある場合　: 0

pos_type istream::tellg()

入力ストリームの位置を調べます。
この関数は streambuf::pubseekoff(0,cur,in)を呼び出します。
リターン値は次のとおりです。

ストリームの先頭からのオフセット
ただし、入力処理にエラーが発生した場合は-1

istream& istream::seekg(pos_type pos)

現在のストリームポインタから pos だけ移動します。
この関数は streambuf::pubseekpos(pos)を呼び出します。
リターン値は*this です。

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

dir で指定された方法で入力ストリームの読み込み位置を移動します。
この関数は streambuf::pubseekoff(off,dir)を呼び出します。
入力処理にエラーがある場合は処理は行いません。
リターン値は*this です。

(h) istream クラスマニピュレータ

種別	定義名	説明
関数	istream& ws(istream& is)	空白類を読み飛ばします。

istream& ws(istream& is)

空白類を読み飛ばします。

リターン値は is です。

(i)　istream メンバ外関数

種別	定義名	説明
関数	istream& operator>>(istream& in, char* s)	文字列を抽出し、s の指す記憶領域に格納します。
	istream& operator>>(istream& in, signed char* s)	
	istream& operator>>(istream& in, unsigned char* s)	
	istream& operator>>(istream& in, char& c)	文字を抽出し、c に格納します。
	istream& operator>>(istream& in, singed char& c)	
	istream& operator>>(istream& in, unsigned char& c)	

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

文字列を抽出し、s の指す記憶領域に格納します。
(フィールド幅-1)個の文字を格納したか、または入力ストリームに streambuf::eof が現れたか、または次の入力可能な文字 c が isspace(c)==1 の場合、処理は終了します。格納文字数が 0 の場合は failbit を設定します。
リターン値は in です。

istream& operator>>(istream& in, char& c)
istream& operator>>(istream& in, singed char& c)
istream& operator>>(istream& in, unsigned char& c)

文字を抽出し、c に格納します。
抽出入力がない場合、failbit を設定します。
リターン値は in です。

(j) ostream::sentry クラス

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

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

内部クラス sentry のコンストラクタです。

good()が非 0 かつ tie()が非 0 なら flush()を呼び出します。__ec2p_os に os を設定します。

ostream::sentry::~sentry()

内部クラス sentry のデストラクタです。

__ec2p_os->flags() & ios_base::unitbuf が真なら、flush()を呼び出します。

ostream::sentry::operator bool()

ok_を参照します。

リターン値は ok_です。

(k)　ostream クラス

種別	定義名	説明
関数	ostream(streambuf* sbptr)	コンストラクタです。
	virtual ~ostream()	デストラクタです。
	ostream& operator<<(bool n)	n を出力ストリームに挿入します。
	ostream& operator<<(short n)	
	ostream& operator<<(unsigned short n)	
	ostream& operator<<(int n)	
	ostream& operator<<(unsigned int n)	
	ostream& operator<<(long n)	
	ostream& operator<<(unsigned long n)	
	ostream& operator<<(long long n)	
	ostream& operator<<(unsigned long long n)	
	ostream& operator<<(float n)	
	ostream& operator<<(double n)	
	ostream& operator<<(long double n)	
	ostream& operator<<(void* n)	
	ostream& operator<<(streambuf* sbptr)	sbptr の出力列を出力ストリームに挿入します。
	ostream& put(char c)	文字 c を出力ストリームに挿入します。
	ostream& write(const char* s, streamsize n)	s の n 個の文字を出力ストリームに挿入します。
	ostream& write(const signed char* s, streamsize n)	
	ostream& write(const unsigned char* s, streamsize n)	
	ostream& flush()	出力ストリームをフラッシュします。この関数は streambuf::pubsync()を呼び出します。
	pos_type tellp()	現在の書き込み位置を求めます。この関数は streambuf::pubseekoff(0,cur,out)を呼び出します。
	ostream& seekp(pos_type pos)	ストリームの先頭から現在の位置までのオフセットを求めます。現在のストリームポインタから pos だけ移動します。この関数は streambuf::pubseekpos(pos)を呼び出します。
	ostream& seekp(off_type off, seekdir dir)	dir を基準として、ストリームの書き込み位置を off 分だけ移動します。この関数は streambuf::pubseekoff(off,dir)を呼び出します。

ostream::ostream(streambuf* sbptr)
コンストラクタです。
ios(sbptr)を呼び出します。

virtual ostream::~ostream()
デストラクタです。

ostream& ostream::operator<<(bool n)
ostream& ostream::operator<<(short n)
ostream& ostream::operator<<(unsigned short n)
ostream& ostream::operator<<(int n)
ostream& ostream::operator<<(unsigned int n)
ostream& ostream::operator<<(long n)
ostream& ostream::operator<<(unsigned long n)
ostream& ostream::operator<<(long long n)
ostream& ostream::operator<<(unsigned long long n)
ostream& ostream::operator<<(float n)
ostream& ostream::operator<<(double n)
ostream& ostream::operator<<(long double n)
ostream& ostream::operator<<(void* n)
sentry::ok_==true のとき、n を出力ストリームに挿入します。
sentry::ok_==false のとき、failbit を設定します。
リターン値は*this です。

ostream& ostream::operator<<(streambuf* sbptr)
sentry::ok_==true のとき、sbptr の出力列を出力ストリームに挿入します。
sentry::ok_==false のとき、failbit を設定します。
リターン値は*this です。

ostream& ostream::put(char c)
sentry::ok_==true かつ rdbuf()->sputc(c)!=streambuf::eof のとき、c を出力ストリームに挿入します。
上記以外のとき、badbit を設定します。
リターン値は*this です。

ostream& ostream::write(const char* s, streamsize n)
ostream& ostream::write(const signed char* s, streamsize n)
ostream& ostream::write(const unsigned char* s, streamsize n)
sentry::ok_==true かつ rdbuf()->sputn(s, n)==n のとき、s の n 個の文字を出力ストリームに挿入します。
上記以外のとき、badbit を設定します。
リターン値は*this です。

ostream& ostream::flush()
出力ストリームをフラッシュします。
この関数は streambuf::pubsync()を呼び出します。
リターン値は*this です。

pos_type ostream::tellp()

現在の書き込み位置を求めます。
この関数は streambuf::pubseekoff(0,cur,out)を呼び出します。
リターン値は次のとおりです。

現在のストリームの位置
ただし、処理中にエラーが発生した場合は-1

ostream& ostream::seekp(pos_type pos)

エラーがないとき、ストリームの先頭から現在の位置までのオフセットを求めます。
また、現在のストリームポインタから pos だけ移動します。
この関数は streambuf::pubseekpos(pos)を呼び出します。
リターン値は*this です。

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

エラーがないとき、dir を基準として off 分ストリームの位置を移動します。
この関数は streambuf::pubseekoff(off,dir)を呼び出します。
リターン値は*this です。

(l) ostream クラスマニピュレータ

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

ostream& endl(ostream& os)

ストリームに改行文字を挿入します。
出力ストリームをフラッシュします。この関数は flush()を呼び出します。
リターン値は os です。

ostream& ends(ostream& os)

出力ストリームにヌルコードを挿入します。
リターン値は os です。

ostream& flush(ostream& os)

出力ストリームをフラッシュします。この関数は streambuf::sync()を呼び出します。
リターン値は os です。

(m)　ostream メンバ外関数

種別	定義名	説明
関数	ostream& operator<<(ostream& os, char s)	s を出力ストリームに挿入します。
	ostream& operator<<(ostream& os, signed char s)	
	ostream& operator<<(ostream& os, unsigned char s)	
	ostream& operator<<(ostream& os, const char* s)	
	ostream& operator<<(ostream& os, const singed char* s)	
	ostream& operator<<(ostream& os, const unsigned char* s)	

ostream& operator<<(ostream& os, char s)
ostream& operator<<(ostream& os, signed char s)
ostream& operator<<(ostream& os, unsigned char s)
ostream& operator<<(ostream& os, const char* s)
ostream& operator<<(ostream& os, const singed char* s)
ostream& operator<<(ostream& os, const unsigned char* s)

sentry::ok_==true かつエラーがないとき、s を出力ストリームに挿入します。
上記以外のとき、failbit を設定します。
リターン値は os です。

(n) smanip クラスマニピュレータ

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

smanip resetiosflags(ios_base::fmtflags mask)
mask 値で指定されたフラグをクリアします。
リターン値は、入出力対象のオブジェクトです。

smanip setiosflags(ios_base::fmtflags mask)
書式フラグ(fmtfl)を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setbase(int base)
出力時に用いる基数を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setfill(char c)
詰め文字(fillch)を設定します。
リターン値は、入出力対象のオブジェクトです。

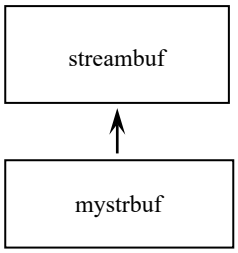
smanip setprecision(int n)
精度(prec)を設定します。
リターン値は、入出力対象のオブジェクトです。

smanip setw(int n)
フィールド幅(wide)を設定します。
リターン値は、入出力対象のオブジェクトです。

(o)　EC++入出力ライブラリの使用例

istream, ostream のオブジェクトの初期化時に streambuf のかわりに mystrbuf クラスのオブジェクトへのポインタを使うことにより入出力ストリームが使用可能になります。

クラスの派生関係は次のようになります。矢印は、派生クラスから基底クラスを参照していることを示します。



種別	定義名	説明
変数	_file_Ptr	ファイルポインタです。
関数	mystrbuf() mystrbuf(void* ptr)	コンストラクタです。streambuf バッファの初期化を行います。
	virtual ~mystrbuf()	デストラクタです。
	void* myfptr() const	FILE 型構造体へのポインタを返します。
	mystrbuf* open(const char* filename, int mode)	ファイル名とモードを指定して、ファイルをオープンします。
	mystrbuf* close()	ファイルのクローズを行います。
	virtual streambuf* setbuf(char* s, streamsize n)	ストリーム入出力用のバッファを確保します。
	virtual pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual pos_type seekpos(pos_type sp, ios_base::openmode = (ios_base::openmode) (ios_base::in | ios_base::out))	ストリームポインタの位置を変えます。
	virtual int sync()	ストリームをフラッシュします。
	virtual int showmanyc()	入力ストリームの有効な文字数を返します。
	virtual int_type underflow()	ストリーム位置を動かさずに一文字読み込みます。
	virtual int_type pbackfail(int_type c = streambuf::eof)	c によって示される文字をプットバックします。
	virtual int_type overflow(int_type c = streambuf::eof)	c によって示される文字を挿入します。
	void _Init(_f_type* fp)	初期処理です。

例：

```
#include <istream>
#include <ostream>
#include <mystrbuf>
#include <string>
#include <new>
#include <stdio.h>
void main(void)
{
   mystrbuf myfin(stdin);
   mystrbuf myfout(stdout);
   istream mycin(&myfin);
   ostream mycout(&myfout);

   int i;
   short s;
   long l;
   char c;
   string str;

   mycin >> i >> s >> l >> c >> str;
   mycout << "This is EC++ Library." << endl
          << i << s << l << c << str << endl;
   return;
}
```

(3) メモリ管理用ライブラリ

メモリの管理用ライブラリに対応するヘッダファイルは以下の通りです。

- <new>

  メモリの確保・解放を行う関数を定義します。

_ec2p_new_handler 変数に例外処理関数のアドレスを設定することにより、メモリ確保に失敗した場合、例外処理を実行することができます。

_ec2p_new_handler は static 変数で、初期値は NULL です。このハンドラを使用することにより、リエントラント性は失われます。

例外処理関数に要求される動作：

- 割り当て可能な領域を作成して返します。
- 作成できない場合の動作は規定されていません。

種別	定義名	説明
型	new_handler	void 型を返す関数へのポインタ型です。
変数	_ec2p_new_handler	例外処理関数へのポインタです。
関数	void* operator new(size_t size)	size 分の領域を確保します。
	void* operator new[ ](size_t size)	size 分の配列領域を確保します。
	void* operator new(size_t size, void* ptr)	ptr の指している領域を記憶領域として割り当てます。
	void* operator new[ ](size_t size, void* ptr)	ptr の指している領域を配列領域として割り当てます。
	void operator delete(void* ptr)	領域を解放します。
	void operator delete[ ](void* ptr)	配列領域を解放します。
	new_handler set_new_handler(new_handler new_P)	_ec2p_new_handler に例外処理関数アドレス(new_P)を設定します。

void* operator new(size_t size)

size バイト分の領域を割り当てます。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

void* operator new[ ](size_t size)

size 分の配列領域を確保します。

領域割り当てに失敗し、かつ new_handler が設定されていれば、new_handler を呼び出します。

リターン値は次のとおりです。

領域確保に成功した場合：void 型へのポインタ

領域確保に失敗した場合：NULL

void* operator new(size_t size, void* ptr)
ptr の指している領域を記憶領域として割り当てます。
リターン値は ptr です。
void* operator new[ ](size_t size, void* ptr)
ptr の指している領域を配列領域として割り当てます。
リターン値は ptr です。

void operator delete(void* ptr)
ptr が指す記憶領域を解放します。ptr が NULL のときは何もしません。

void operator delete[ ](void* ptr)
ptr が指す配列領域を解放します。ptr が NULL のときは何もしません。

new_handler set_new_handler(new_handler new_P)
_ec2p_new_handler に new_P を設定します。
リターン値は_ec2p_new_handler です。

(4) 複素数計算用クラスライブラリ

複素数計算用クラスライブラリに対応するヘッダファイルは以下のとおりです。

- <complex>
  float_complex クラス、double_complex クラスを定義します。

これらのクラスには派生関係はありません。

(a) float_complex クラス

種別	定義名	説明
型	value_type	float 型です。
変数	_re	float 精度の実数部を定義します。
	_im	float 精度の虚数部を定義します。
関数	float_complex(float re = 0.0f, float im = 0.0f)	コンストラクタです。
	float_complex(const double_complex& rhs)	
	float real() const	実数部(_re)を求めます。
	float imag() const	虚数部(_im)を求めます。
	float_complex& operator=(float rhs)	rhs を実数部にコピーします。虚数部は 0.0f を設定します。
	float_complex& operator+=(float rhs)	rhs を実数部に加算し、和を*this に格納します。
	float_complex& operator-=(float rhs)	rhs を実数部から減算し、差を*this に格納します。
	float_complex& operator*=(float rhs)	rhs を乗算し、積を*this に格納します。
	float_complex& operator/=(float rhs)	rhs で除算し、商を*this に格納します。
	float_complex& operator=(const float_complex& rhs)	rhs をコピーします。
	float_complex& operator+=(const float_complex& rhs)	rhs を加算し、和を*this に格納します。
	float_complex& operator-=(const float_complex& rhs)	rhs を減算し、差を*this に格納します。
	float_complex& operator*=(const float_complex& rhs)	rhs を乗算し、積を*this に格納します。
	float_complex& operator/=(const float_complex& rhs)	rhs で除算し、商を*this に格納します。

float_complex::float_complex(float re = 0.0f, float im = 0.0f)

クラス float_complex のコンストラクタです。
以下の値で初期化します。
_re = re;
_im= im;

float_complex::float_complex(const double_complex& rhs)

クラス float_complex のコンストラクタです。
以下の値で初期化します。
_re = (float)rhs.real();
_im= (float)rhs.imag();

float float_complex::real() const

実数部を求めます。

リターン値は、this->_re です。

float float_complex::imag() const
虚数部を求めます。
リターン値は、this->_im です。

float_complex& float_complex::operator=(float rhs)
rhs を実数部(_re)にコピーします。虚数部(_im)は 0.0f を設定します。
リターン値は*this です。

float_complex& float_complex::operator+=(float rhs)
rhs を実数部(_re)に加算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

float_complex& float_complex::operator-=(float rhs)
rhs を実数部(_re)から減算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
リターン値は*this です。

float_complex& float_complex::operator*=(float rhs)
rhs と乗算し、結果を*this に格納します。
(_re=_re*rhs, _im=_im*rhs)
リターン値は*this です。

float_complex& float_complex::operator/=(float rhs)
rhs で除算し、結果を*this に格納します。
(_re=_re/rhs, _im=_im/rhs)
リターン値は*this です。

float_complex& float_complex::operator=(const float_complex& rhs)
rhs をコピーします。
リターン値は*this です。

float_complex& float_complex::operator+=(const float_complex& rhs)
rhs を加算し、結果を*this に格納します。
リターン値は*this です。

float_complex& float_complex::operator-=(const float_complex& rhs)
rhs を減算し、結果を*this に格納します。
リターン値は*this です。

float_complex& float_complex::operator*=(const float_complex& rhs)
rhs と乗算し、結果を*this に格納します。
リターン値は*this です。
float_complex& float_complex::operator/=(const float_complex& rhs)
rhs で除算し、結果を*this に格納します。
リターン値は*this です。

(b)　float_complex メンバ外関数

種別	定義名	説明
関数	float_complex operator+(const float_complex& lhs)	lhs の単項＋演算を行います。
	float_complex operator+(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を加算し、和を lhs に格納します。
	float_complex operator+(const float_complex& lhs, const float& rhs)	
	float_complex operator+(const float& lhs, const float_complex& rhs)	
	float_complex operator-(const float_complex& lhs)	lhs の単項－演算を行います。
	float_complex operator-(const float_complex& lhs, const float_complex& rhs)	lhs から rhs を減算し、差を lhs に格納します。
	float_complex operator-(const float_complex& lhs, const float& rhs)	
	float_complex operator-(const float& lhs, const float_complex& rhs)	
	float_complex operator*(const float_complex& lhs, const float_complex& rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	float_complex operator*(const float_complex& lhs, const float& rhs)	
	float_complex operator*(const float& lhs, const float_complex& rhs)	
	float_complex operator/(const float_complex& lhs, const float_complex& rhs)	lhs を rhs で除算し、商を lhs に格納します。
	float_complex operator/(const float_complex& lhs, const float& rhs)	
	float_complex operator/(const float& lhs, const float_complex& rhs)	
	bool operator==(const float_complex& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const float_complex& lhs, const float& rhs)	

種別	定義名	説明
関数	bool operator==(const float& lhs, const float_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator!=(const float_complex& lhs, const float_complex& rhs)	
	bool operator!=(const float_complex& lhs, const float& rhs)	
	bool operator!=(const float& lhs, const float_complex& rhs)	
	istream& operator>>(istream& is, float_complex& x)	u,(u),または(u,v) (u:実数部、v:虚数部)形式の x を入力します。
	ostream& operator<<(ostream& os, float_complex& x)	x を u,(u)または (u,v) (u:実数部、v:虚数部)形式で出力します。
	float real(const float_complex& x)	実数部を求めます。
	float imag(const float_complex& x)	虚数部を求めます。
	float abs(const float_comlex& x)	絶対値を求めます。
	float arg(const float_complex& x)	位相角度を求めます。
	float norm(const float_complex& x)	２乗の絶対値を求めます。
	float_complex conj(const float_complex& x)	共役複素数を求めます。
	float_complex polar(const float& rho, const float& theta)	大きさが rho で位相角度が theta の複素数に対応する float_complex 値を求めます。
	float_complex cos(const float_complex& x)	複素余弦を求めます。
	float_complex cosh(const float_complex& x)	複素双曲余弦を求めます。
	float_complex exp(const float_complex& x)	指数関数を求めます。
	float_complex log(const float_complex& x)	自然対数を求めます。
	float_complex log10(const float_complex& x)	常用対数を求めます。
	float_complex pow(const float_complex& x, int y)	x の y 乗を求めます。
	float_complex pow(const float_complex& x, const float& y)	
	float_complex pow(const float_complex& x, const float_complex& y)	
	float_complex pow(const float& x, const float_complex& y)	
	float_complex sin(const float_complex& x)	複素正弦を求めます。
	float_complex sinh(const float_complex& x)	複素双曲正弦を求めます。

種別	定義名	説明
関数	float_complex sqrt(const float_complex& x)	右半空間における範囲での平方根を求めます。
	float_complex tan(const float_complex& x)	複素正接を求めます。
	float_complex tanh(const float_complex& x)	複素双曲正接を求めます。

float_complex operator+(const float_complex& lhs)
lhs の単項＋演算を行います。
リターン値は lhs です。

float_complex operator+(const float_complex& lhs, const float_complex& rhs)
float_complex operator+(const float_complex& lhs, const float& rhs)
float_complex operator+(const float& lhs, const float_complex& rhs)
lhs と rhs を加算し、結果を lhs に格納します。
リターン値は、float_complex(lhs)+=rhs です。

float_complex operator-(const float_complex& lhs)
lhs の単項－演算を行います。
リターン値は、float_complex(-lhs.real(),-lhs.imag())です。

float_complex operator-(const float_complex& lhs, const float_complex& rhs)
float_complex operator-(const float_complex& lhs, const float& rhs)
float_complex operator-(const float& lhs, const float_complex& rhs)
lhs から rhs を減算し、結果を lhs に格納します。
リターン値は、float_complex(lhs)-=rhs です。

float_complex operator*(const float_complex& lhs, const float_complex& rhs)
float_complex operator*(const float_complex& lhs, const float& rhs)
float_complex operator*(const float& lhs, const float_complex& rhs)
lhs と rhs を乗算し、結果を lhs に格納します。
リターン値は、float_complex(lhs)*=rhs です。

float_complex operator/(const float_complex& lhs, const float_complex& rhs)
float_complex operator/(const float_complex& lhs, const float& rhs)
float_complex operator/(const float& lhs, const float_complex& rhs)
lhs を rhs で除算し、結果を lhs に格納します。
リターン値は、float_complex(lhs)/=rhs です。

bool operator==(const float_complex& lhs, const float_complex& rhs)
bool operator==(const float_complex& lhs, const float& rhs)
bool operator==(const float& lhs, const float_complex& rhs)
lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。
リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag()です。

bool operator!=(const float_complex& lhs, const float_complex& rhs)
bool operator!=(const float_complex& lhs, const float& rhs)
bool operator!=(const float& lhs, const float_complex& rhs)

lhs と rhs の実数部どうし、虚数部どうしを比較します。float 型引数の場合、虚数部は float 型の 0.0f と仮定されます。
リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()です。

istream& operator>>(istream& is, float_complex& x)

u,(u), または(u,v) (u は実数部、v は虚数部)の形式の x を入力します。入力値は float_complex に変換されます。
u,(u),(u,v)形式以外が入力された場合は、is.setstate(ios_base::failbit)を呼びます。
リターン値は is です。

ostream& operator<<(ostream& os, const float_complex& x)

x を os に出力します。
出力形式は u,(u)または(u,v) (u は実数部、v は虚数部)です。
リターン値は os です。

float real(const float_complex& x)

実数部を求めます。
リターン値は x.real()です。

float imag(const float_complex& x)

虚数部を求めます。
リターン値は x.imag()です。

float abs(const float_complex& x)

絶対値を求めます。
リターン値は、$(|x.real()|^2 + |x.imag()|^2)^{1/2}$ です。

float arg(const float_complex& x)

位相角度を求めます。
リターン値は、atan2f(x.imag() , x.real())です。

float norm(const float_complex& x)

2 乗の絶対値を求めます。
リターン値は、$|x.real()|^2 + |x.imag()|^2$ です。

float_complex conj(const float_complex& x)

共役複素数を求めます。
リターン値は、float_complex(x.real(), (-1)*x.imag())です。

float_complex polar(const float& rho, const float& theta)

大きさが rho で位相角度(偏角)が theta の複素数に対応する float_complex 値を求めます。
リターン値は、float_complex(rho*cosf(theta), rho*sinf(theta))です。

float_complex cos(const float_complex& x)
複素余弦を求めます。
リターン値は、float_complex(cosf(x.real())*coshf(x.imag()), (-1)*sinf(x.real())*sinhf(x.imag()))です。

float_complex cosh(const float_complex& x)
複素双曲余弦を求めます。
リターン値は、cos(float_complex((-1)*x.imag(), x.real()))です。

float_complex exp(const float_complex& x)
指数関数を求めます。
リターン値は、expf(x.real())*cosf(x.imag()),expf(x.real())*sinf(x.imag())です。

float_complex log(const float_complex& x)
(e を底とする)自然対数を求めます。
リターン値は、float_complex(logf(abs(x)), arg(x))です。

float_complex log10(const float_complex& x)
(10 を底とする)常用対数を求めます。
リターン値は、float_complex(log10f(abs(x)), arg(x)/logf(10))です。

float_complex pow(const float_complex& x, int y)
float_complex pow(const float_complex& x, const float& y)
float_complex pow(const float_complex& x, const float_complex& y)
float_complex pow(const float& x, const float_complex& y)
x の y 乗を求めます。
pow(0,0)のとき、定義域エラーになります。
リターン値は次のとおりです。

float_complex pow(const float_complex& x,const float_complex& y)の場合	: exp(y*logf(x))
上記以外	: exp(y*log(x))

float_complex sin(const float_complex& x)
複素正弦を求めます。
リターン値は、float_complex(sinf(x.real())*coshf(x.imag()), cosf(x.real())*sinhf(x.imag()))です。

float_complex sinh(const float_complex& x)
複素双曲正弦を求めます。
リターン値は、float_complex(0,-1)*sin(float_complex((-1)*x.imag(),x.real()))　です。

float_complex sqrt(const float_complex& x)
右半空間における範囲での平方根を求めます。
リターン値は、float_complex(sqrtf(abs(x))*cosf(arg(x)/2), sqrtf(abs(x))*sinf(arg(x)/2))です。

float_complex tan(const float_complex& x)
複素正接を求めます。
リターン値は、sin(x)/cos(x)です。

float_complex tanh(const float_complex& x)
複素双曲正接を求めます。
リターン値は、sinh(x)/cosh(x)です。

(c) double_complex クラス

種別	定義名	説明
型	value_type	double 型です。
変数	_re	double 精度の実数部を定義します。
	_im	double 精度の虚数部を定義します。
関数	double_complex(double re = 0.0, double im = 0.0)	コンストラクタです。
	double_complex(const float_complex&)	
	double real() const	実数部を求めます。
	double imag() const	虚数部を求めます。
	double_complex& operator=(double rhs)	rhs を実数部にコピーします。虚数部は 0.0 を設定します。
	double_complex& operator+=(double rhs)	rhs を実数部に加算し、和を*this に格納します。
	double_complex& operator-=(double rhs)	rhs を実数部から減算し、差を*this に格納します。
	double_complex& operator*=(double rhs)	rhs を乗算し、積を*this に格納します。
	double_complex& operator/=(double rhs)	rhs で除算し、商を*this に格納します。
	double_complex& operator=(const double_complex& rhs)	rhs をコピーします。
	double_complex& operator+=(const double_complex& rhs)	rhs を加算し、和を*this に格納します。
	double_complex& operator-=(const double_complex& rhs)	rhs を減算し、差を*this に格納します。
	double_complex& operator*=(const double_complex& rhs)	rhs を乗算し、積を*this に格納します。
	double_complex& operator/=(const double_complex& rhs)	rhs で除算し、商を*this に格納します。

double_complex::double_complex(double re = 0.0, double im = 0.0)
クラス double_complex のコンストラクタです。
以下の値で初期化します。
_re = re;
_im= im;

double_complex::double_complex(const float_complex&)
クラス double_complex のコンストラクタです。
以下の値で初期化します。
_re = (double)rhs.real();
_im= (double)rhs.imag();

double double_complex::real() const
　実数部を求めます。
　リターン値は、this->_re です。

double double_complex::imag() const
　虚数部を求めます。
　リターン値は、this->_im です。

double_complex& double_complex::operator=(double rhs)
　rhs を実数部(_re)にコピーします。虚数部(_im)は 0.0 を設定します。
　リターン値は*this です。

double_complex& double_complex::operator+=(double rhs)
　rhs を実数部(_re)に加算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
　リターン値は*this です。

double_complex& double_complex::operator-=(double rhs)
　rhs を実数部(_re)から減算し、結果を実数部(_re)に格納します。虚数部(_im)の値は変わりません。
　リターン値は*this です。

double_complex& double_complex::operator*=(double rhs)
　rhs と乗算し、結果を*this に格納します。
　(_re=_re*rhs, _im=_im*rhs)
　リターン値は*this です。

double_complex& double_complex::operator/=(double rhs)
　rhs で除算し、結果を*this に格納します。
　(_re=_re/rhs, _im=_im/rhs)
　リターン値は*this です。

double_complex& double_complex::operator=(const double_complex& rhs)
　rhs をコピーします。
　リターン値は*this です。

double_complex& double_complex::operator+=(const double_complex& rhs)
　rhs を加算し、結果を*this に格納します。
　リターン値は*this です。

double_complex& double_complex::operator-=(const double_complex& rhs)
　rhs を減算し、結果を*this に格納します。
　リターン値は*this です。

double_complex& double_complex::operator*=(const double_complex& rhs)
　rhs と乗算し、結果を*this に格納します。
　リターン値は*this です。

double_complex& double_complex::operator/=(const double_complex& rhs)

rhs で除算し、結果を*this に格納します。

リターン値は*this です。

(d) double_complex メンバ外関数

種別	定義名	説明
関数	double_complex operator+(const double_complex& lhs)	lhs の単項＋演算を行います。
	double_complex operator+(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を加算し、和を lhs に格納します。
	double_complex operator+(const double_complex& lhs, const double& rhs)	
	double_complex operator+(const double& lhs, const double_complex& rhs)	
	double_complex operator-(const double_complex& lhs)	lhs の単項－演算を行います。
	double_complex operator-(const double_complex& lhs, const double_complex& rhs)	lhs から rhs を減算し、差を lhs に格納します。
	double_complex operator-(const double_complex& lhs, const double& rhs)	
	double_complex operator-(const double& lhs, const double_complex& rhs)	
	double_complex operator*(const double_complex& lhs, const double_complex& rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	double_complex operator*(const double_complex& lhs, const double& rhs)	
	double_complex operator*(const double& lhs, const double_complex& rhs)	
	double_complex operator/(const double_complex& lhs, const double_complex& rhs)	lhs を rhs で除算し、商を lhs に格納します。
	double_complex operator/(const double_complex& lhs, const double& rhs)	
	double_complex operator/(const double& lhs, const double_complex& rhs)	
	bool operator==(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。

種別	定義名	説明
関数	bool operator!=(const double_complex& lhs, const double_complex& rhs)	lhs と rhs の実数部どうし、虚数部どうしを比較します。
	bool operator==(const double_complex& lhs, const double& rhs)	
	bool operator==(const double& lhs, const double_complex& rhs)	
	bool operator!=(const double_complex& lhs, const double& rhs)	
	bool operator!=(const double& lhs, const double_complex& rhs)	
	istream& operator>>(istream& is, double_complex& x)	u,(u)または(u,v) (u:実数部、v:虚数部)形式の x を入力します。
	ostream& operator<<(ostream& os, const double_complex& x)	x を u,(u)または (u,v) (u:実数部、v:虚数部)形式で出力します。
	double real(const double_complex& x)	実数部を求めます。
	double imag(const double_complex& x)	虚数部を求めます。
	double abs(const double_comlex& x)	絶対値を求めます。
	double arg(const double_complex& x)	位相角度を求めます。
	double norm(const double_complex& x)	２乗の絶対値を求めます。
	double_complex conj(const double_complex& x)	共役複素数を求めます。
	double_complex polar(const double& rho, const double& theta)	大きさが rho で位相角度が theta の複素数に対応する double_complex 値を求めます。
	double_complex cos(const double_complex& x)	複素余弦を求めます。
	double_complex cosh(const double_complex& x)	複素双曲余弦を求めます。
	double_complex exp(const double_complex&)	指数関数を求めます。
	double_complex log(const double_complex& x)	自然対数を求めます。
	double_complex log10(const double_complex& x)	常用対数を求めます。
	double_complex pow(const double _complex& x, int y)	x の y 乗を求めます。
	double_complex pow(const double _complex& x, const double & y)	

種別	定義名	説明
関数	double_complex pow(const double _complex& x, const double _complex& y)	x の y 乗を求めます。
	double_complex pow(const double & x, const double _complex& y)	
	double_complex sin(const double_complex& x)	複素正弦を求めます。
	double_complex sinh(const double_complex& x)	複素双曲正弦を求めます。
	double_complex sqrt(const double_complex& x)	右半空間における範囲での平方根を求めます。
	double_complex tan(const double_complex& x)	複素正接を求めます。
	double_complex tanh(const double_complex& x)	複素双曲正接を求めます。

double_complex operator+(const double_complex& lhs)
lhs の単項＋演算を行います。
リターン値は lhs です。

double_complex operator+(const double_complex& lhs, const double_complex& rhs)
double_complex operator+(const double_complex& lhs, const double& rhs)
double_complex operator+(const double& lhs, const double_complex& rhs)
lhs と rhs を加算し、結果を lhs に格納します。
リターン値は、double_complex(lhs)+=rhs です。

double_complex operator-(const double_complex& lhs)
lhs の単項－演算を行います。
リターン値は、double_complex(-lhs.real(), -lhs.imag())です。

double_complex operator-(const double_complex& lhs, const double_complex& rhs)
double_complex operator-(const double_complex& lhs, const double& rhs)
double_complex operator-(const double& lhs, const double_complex& rhs)
lhs から rhs を減算し、結果を lhs に格納します。
リターン値は、double_complex(lhs)-=rhs です。

double_complex operator*(const double_complex& lhs, const double_complex& rhs)
double_complex operator*(const double_complex& lhs, const double& rhs)
double_complex operator*(const double& lhs, const double_complex& rhs)
lhs と rhs を乗算し、結果を lhs に格納します。
リターン値は、double_complex(lhs)*=rhs です。

double_complex operator/(const double_complex& lhs, const double_complex& rhs)
double_complex operator/(const double_complex& lhs, const double& rhs)
double_complex operator/(const double& lhs, const double_complex& rhs)
lhs を rhs で除算し、結果を lhs に格納します。

リターン値は、double_complex(lhs)/=rhs です。

bool operator==(const double_complex& lhs, const double_complex& rhs)
bool operator==(const double_complex& lhs, const double& rhs)
bool operator==(const double& lhs, const double_complex& rhs)
lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。
リターン値は、lhs.real()==rhs.real() && lhs.imag()==rhs.imag()です。

bool operator!=(const double_complex& lhs, const double_complex& rhs)
bool operator!=(const double_complex& lhs, const double& rhs)
bool operator!=(const double& lhs, const double_complex& rhs)
lhs と rhs の実数部どうし、虚数部どうしを比較します。double 型引数の場合、虚数部は double 型の 0.0 と仮定されます。
リターン値は、lhs.real()!=rhs.real() || lhs.imag()!=rhs.imag()です。

istream& operator>>(istream& is, double_complex& x)
u,(u)または(u,v) (u は実数部、v は虚数部)の形式の複素数 x を入力します。入力値は double_complex に変換されます。
u,(u),(u,v)形式以外が入力された場合は、is.setstate(ios_base::failbit)を呼びます。
リターン値は is です。

ostream& operator<<(ostream& os, const double_complex& x)
x を os に出力します。
出力形式は u,(u)または(u,v) (u は実数部、v は虚数部)です。
リターン値は os です。

double real(const double_complex& x)
実数部を求めます。
リターン値は x.real()です。

double imag(const double_complex& x)
虚数部を求めます。
リターン値は x.imag()です。

double abs(const double_complex& x)
絶対値を求めます。
リターン値は、$(|x.real()|^2 + |x.imag()|^2)^{1/2}$ です。

double arg(const double_complex& x)
位相角度を求めます。
リターン値は、atan2(x.imag() , x.real())です。
double norm(const double_complex& x)
2 乗の絶対値を求めます。
リターン値は、$|x.real()|^2 + |x.imag()|^2$ です。

double_complex conj(const double_complex& x)
共役複素数を求めます。
リターン値は、double_complex(x.real(), (-1)*x.imag())です。

double_complex polar(const double& rho, const double& theta)
大きさが rho で位相角度(偏角)が theta の複素数に対応する double_complex 値を求めます。
リターン値は、double_complex(rho*cos(theta), rho*sin(theta))です。

double_complex cos(const double_complex& x)
複素余弦を求めます。
リターン値は、double_complex(cos(x.real())*cosh(x.imag()), (-1)*sin(x.real())*sinh(x.imag()))です。

double_complex cosh(const double_complex& x)
複素双曲余弦を求めます。
リターン値は、cos(double_complex((-1)*x.imag(), x.real()))です。

double_complex exp(const double_complex& x)
指数関数を求めます。
リターン値は、exp(x.real())*cos(x.imag()),exp(x.real())*sin(x.imag())です。

double_complex log(const double_complex& x)
(e を底とする)自然対数を求めます。
リターン値は、double_complex(log(abs(x)), arg(x))です。

double_complex log10(const double_complex& x)
(10 を底とする)常用対数を求めます。
リターン値は、double_complex(log10(abs(x)), arg(x)/log(10))です。

double_complex pow(const double_complex& x, int y)
double_complex pow(const double_complex& x, const double& y)
double_complex pow(const double_complex& x, const double_complex& y)
double_complex pow(const double& x, const double_complex& y)
x の y 乗を求めます。
pow(0,0)のとき、定義域エラーになります。
リターン値は、exp(y*log(x))です。

double_complex sin(const double_complex& x)
複素正弦を求めます。
リターン値は、double_complex(sin(x.real())*cosh(x.imag()), cos(x.real())*sinh(x.imag()))です。

double_complex sinh(const double_complex& x)
　複素双曲正弦を求めます。
　リターン値は、double_complex(0,-1)*sin(double_complex((-1)*x.imag(),x.real()))です。

double_complex sqrt(const double_complex& x)
　右半空間における範囲での平方根を求めます。
　リターン値は、double_complex(sqrt(abs(x))*cos(arg(x)/2), sqrt(abs(x))*sin(arg(x)/2))です。
double_complex tan(const double_complex& x)
　複素正接を求めます。
　リターン値は、sin(x)/cos(x)です。

double_complex tanh(const double_complex& x)
　複素双曲正接を求めます。
　リターン値は、sinh(x)/cosh(x)です。

(5) 文字列操作用クラスライブラリ

文字列操作用クラスライブラリに対応するヘッダファイルは以下の通りです。

- <string>
  string クラスを定義します。

本クラスには派生関係はありません。

(a) string クラス

種別	定義名	説明
型	iterator	char*型です。
	const_iterator	const char*型です。
定数	npos	文字列の最大長(UINT_MAX 文字)です。
変数	s_ptr	オブジェクトが文字列を格納している領域へのポインタです。
	s_len	オブジェクトが格納している文字列長です。
	s_res	オブジェクトが文字列を格納するために確保している領域のサイズです。
関数	string(void)	コンストラクタです。
	string::string(const string& str, size_t pos = 0, size_t n = npos)	
	string::string(const char* str, size_t n)	
	string::string(const char* str)	
	string::string(size_t n, char c)	
	~string()	デストラクタです。
	string& operator=(const string& str)	str を代入します。
	string& operator=(const char* str)	
	string& operator=(char c)	c を代入します。
	iterator begin()	文字列の先頭ポインタを求めます。
	const_iterator begin() const	
	iterator end()	文字列の最後尾ポインタを求めます。
	const_iterator end() const	
	size_t size() const	格納されている文字列の文字列長を求めます。
	size_t length() const	
	size_t max_size() const	確保している領域のサイズを求めます。
	void resize(size_t n, char c)	格納可能な文字列の文字数を n に変更します。
	void resize(size_t n)	格納可能な文字列の文字数を n に変更します。
	size_t capacity() const	確保している領域のサイズを求めます。
	void reserve(size_t res_arg = 0)	領域の再割り当てを行います。
	void clear()	格納されている文字列を clear します。
	bool empty() const	格納している文字列の文字数が 0 かチェックします。

種別	定義名	説明
関数	const char& operator[](size_t pos) const	s_ptr[pos]を参照します。
	char& operator[](size_t pos)	
	const char& at(size_t pos) const	
	char& at(size_t pos)	
	string& operator+=(const string& str)	str の文字列を追加します。
	string& operator+=(const char* str)	
	string& operator+=(char c)	c の文字を追加します。
	string& append(const string& str)	str の文字列を追加します。
	string& append(const char* str)	
	string& append(const string& str, size_t pos, size_t n)	オブジェクトの位置 pos に str の文字列を n 文字分追加します。
	string& append(const char* str, size_t n)	文字列 str の n 文字分を追加します。
	string& append(size_t n, char c)	n 個の文字 c を追加します。
	string& assign(const string& str)	str の文字列を代入します。
	string& assign(const char* str)	
	string& assign(const string& str, size_t pos, size_t n)	位置 pos に文字列 str の n 文字分を代入します。
	string& assign(const char* str, size_t n)	文字列 str の n 文字分を代入します。
	string& assign(size_t n, char c)	n 個の文字 c を代入します。
	string& insert(size_t pos1, const string& str)	位置 pos1 に str の文字列を挿入します。
	string& insert(size_t pos1, const string& str, size_t pos2, size_t n)	位置 pos1 に str の文字列の位置 pos2 から n 文字分を挿入します。
	string& insert(size_t pos, const char* str, size_t n)	pos の位置に文字列 str を n 文字分挿入します。
	string& insert(size_t pos, const char* str)	pos の位置に文字列 str を挿入します。
	string& insert(size_t pos, size_t n, char c)	位置 pos に n 個の文字 c の文字列を挿入します。
	iterator insert(iterator p, char c = char())	p が指す文字列の前に文字 c を挿入します。
	void insert(iterator p, size_t n, char c)	p が指す文字の前に、n 個の文字 c を挿入します。
	string& erase(size_t pos = 0, size_t n = npos)	位置 pos から n 個分取り除きます。
	iterator erase(iterator position)	position により参照された文字を取り除きます。
	iterator erase(iterator first, iterator last)	範囲[first, last]において文字を取り除きます。
	string& replace(size_t pos1, size_t n1, const string& str)	位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。

種別	定義名	説明
関数	string& replace(size_t pos1, size_t n1, const char* str)	位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。
	string& replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)	位置 pos1 から n1 文字分の文字列を、str の位置 pos2 から n2 文字分の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, const char* str, size_t n2)	位置 pos から n1 文字分の文字列を、n2 個の str の文字列で置き換えます。
	string& replace(size_t pos, size_t n1, size_t n2, char c)	位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。
	string& replace(iterator i1, iterator i2, const string& str)	位置 i1 から i2 までの文字列を str の文字列で置き換えます。
	string& replace(iterator i1, iterator i2, const char* str)	
	string& replace(iterator i1, iterator i2, const char* str, size_t n)	位置 i1 から i2 までの文字列を str の文字列の n 文字分で置き換えます。
	string& replace(iterator i1, iterator i2, size_t n, char c)	位置 i1 から i2 までの文字列を n 個の文字 c で置き換えます。
	size_t copy(char* str, size_t n, size_t pos = 0) const	位置 pos に文字列 str の n 文字分の文字列をコピーします。
	void swap(string& str)	str の文字列と交換します。
	const char* c_str() const	文字列を格納している領域へのポインタを参照します。
	const char* data() const	
	size_t find(const string& str, size_t pos = 0) const	位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。

種別	定義名	説明
関数	size_t find(const char* str, size_t pos = 0) const	位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。
	size_t find(const char* str, size_t pos, size_t n) const	位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。
	size_t find(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t rfind(const string& str, size_t pos = npos) const	位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。
	size_t rfind(const char* str, size_t pos = npos) const	
	size_t rfind(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の n 文字分と同じ文字列が最後に現れる位置を検索します。
	size_t rfind(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。
	size_t find_first_of(const string& str, size_t pos = 0) const	位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(const char* str, size_t pos = 0) const	
	size_t find_first_of(const char* str, size_t pos, size_t n) const	位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。
	size_t find_first_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c が最初に現れる位置を検索します。
	size_t find_last_of(const string& str, size_t pos = npos) const	位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(const char* str, size_t pos = npos) const	
	size_t find_last_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。
	size_t find_last_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c が最後に現れる位置を検索します。
	size_t find_first_not_of(const string& str, size_t pos = 0) const	位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(const char* str, size_t pos = 0) const	

種別	定義名	説明
関数	size_t find_first_not_of(const char* str, size_t pos, size_t n)	位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。
	size_t find_first_not_of(char c, size_t pos = 0) const	位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。
	size_t find_last_not_of(const string& str, size_t pos = npos) const	位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(const char* str, size_t pos = npos) const	
	size_t find_last_not_of(const char* str, size_t pos, size_t n) const	位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。
	size_t find_last_not_of(char c, size_t pos = npos) const	位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。
	string substr(size_t pos = 0, size_t n = npos) const	格納された文字列に対し、範囲[pos,n]の文字列を持つオブジェクトを生成します。
	int compare(const string& str) const	文字列と str の文字列を比較します。
	int compare(size_t pos1, size_t n1, const string& str) const	位置 pos1 から n1 文字分の文字列と str を比較します。
	int compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const	位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。
	int compare(const char* str) const	str と比較します。
	int compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const	位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。

string::string(void)
以下のように設定します。
s_ptr = 0;
s_len = 0;
s_res = 1;

string::string(const string& str, size_t pos = 0, size_t n = npos)
str をコピーします。ただし、s_len は、n と s_len の小さい方の値になります。

string::string(const char* str, size_t n)
以下に設定します。
s_ptr = str;
s_len = n;
s_res = n+1;

string::string(const char* str)
以下に設定します。
s_ptr = str;
s_len = str の文字列長;
s_res = str の文字列長+1;

string::string(size_t n, char c)
以下に設定します。
s_ptr = 文字数 n で文字 c の文字列;
s_len = n;
s_res = n+1;

string::~string()
クラス string のデストラクタです。
文字列を格納している領域を解放します。

string& string::operator=(const string& str)
str のデータを代入します。
リターン値は*this です。

string& string::operator=(const char* str)
str から string オブジェクトを生成し、そのデータを代入します。
リターン値は*this です。

string& string::operator=(char c)
c から string オブジェクトを生成し、そのデータを代入します。
リターン値は*this です。

string::iterator string::begin()
string::const_iterator string::begin() const
文字列の先頭ポインタを求めます。
リターン値は、文字列の先頭ポインタです。

string::iterator string::end()
string::const_iterator string::end() const
文字列の最後尾ポインタを求めます。
リターン値は、文字列の最後尾ポインタです。

size_t string::size() const
size_t string::length() const
格納されている文字列の文字列長を求めます。
リターン値は、格納されている文字列の文字列長です。

size_t string::max_size() const
確保している領域のサイズを求めます。
リターン値は、確保している領域のサイズです。

void string::resize(size_t n, char c)
オブジェクトが格納可能な文字列の文字数を n に変更します。
n<=size()のとき、長さを n にした元の文字列と置き換えます。
n>size()のとき、元の文字列の後ろに長さ n になるまで c をつめた文字列と置き換えます。
n<=max_size()である必要があります。
n>max_size()の場合、n=max_size()として計算します。

void string::resize(size_t n)
オブジェクトが格納可能な文字列の文字数を n に変更します。
n<=size()のとき、長さを n にしたもとの文字列と置き換えます。
n<=max_size()である必要があります。

size_t string::capacity() const
確保している領域のサイズを求めます。
リターン値は、確保している領域のサイズです。

void string::reserve(size_t res_arg = 0)
記憶領域の再割り当てを行います。
reserve()後、capacity()は reserve()の引数より大きいかまたは等しくなります。
再割り当てを行うと、すべての参照、ポインタ、この数列の中の要素の参照する iterator を無効にします。

void string::clear()
格納されている文字列をクリアします。

bool string::empty() const
格納している文字列の文字数が 0 かチェックします。
リターン値は次のとおりです。
格納している文字列長が 0 の場合　：true
格納している文字列長が 0 以外の場合　：false

const char& string::operator[](size_t pos) const
char& string::operator[](size_t pos)
const char& string::at(size_t pos) const
char& string::at(size_t pos)
s_ptr[pos]を参照します。
リターン値は次のとおりです。
n<s_len の場合　：s_ptr [pos]
n>=s_len の場合　：'¥0'

string& string::operator+=(const string& str)
str が格納している文字列を追加します。
リターン値は*this です。

string& string::operator+=(const char* str)
str から string オブジェクトを生成し、その文字列を追加します。
リターン値は*this です。

string& string::operator+=(char c)
c から string オブジェクトを生成し、その文字列を追加します。
リターン値は*this です。

string& string::append(const string& str)
string& string::append(const char* str)
str の文字列をオブジェクトに追加します。
リターン値は*this です。

string& string::append(const string& str, size_t pos, size_t n)
オブジェクトの位置 pos に str の文字列を n 文字分追加します。
リターン値は*this です。

string& string::append(const char* str, size_t n)
文字列 str の n 文字分を追加します。
リターン値は*this です。

string& string::append(size_t n, char c)
n 個の文字 c を追加します。
リターン値は*this です。

string& string::assign(const string& str)
string& string::assign(const char* str)
str の文字列を代入します。
リターン値は*this です。

string& string::assign(const string& str, size_t pos, size_t n)
位置 pos に文字列 str の n 文字分を代入します。
リターン値は*this です。

string& string::assign(const char* str, size_t n)
文字列 str の n 文字分を代入します。
リターン値は*this です。

string& string::assign(size_t n, char c)
n 個の文字 c を代入します。
リターン値は*this です。

string& string::insert(size_t pos1, const string& str)
位置 pos1 に str の文字列を挿入します。
リターン値は*this です。

string& string::insert(size_t pos1, const string& str, size_t pos2, size_t n)
位置 pos1 に str の文字列の位置 pos2 から n 文字分を挿入します。
リターン値は*this です。

string& string::insert(size_t pos, const char* str, size_t n)
pos の位置に文字列 str を n 文字分挿入します。
リターン値は*this です。

string& string::insert(size_t pos, const char* str)
pos の位置に文字列 str を挿入します。
リターン値は*this です。

string& string::insert(size_t pos, size_t n, char c)
位置 pos に n 個の文字 c の文字列を挿入します。
リターン値は*this です。

string::iterator string::insert(iterator p, char c = char())
p が指す文字列の前に、文字 c を挿入します。
リターン値は、挿入された文字です。

void string::insert(iterator p, size_t n, char c)
p が指す文字の前に、n 個の文字 c を挿入します。

string& string::erase(size_t pos = 0, size_t n = npos)
位置 pos から n 個分取り除きます。
リターン値は*this です。

iterator string::erase(iterator position)
position により参照された文字を取り除きます。
リターン値は次のとおりです。
削除要素の次の iterator がある場合　：削除要素の次の iterator
削除要素の次の iterator がない場合　：end()

iterator string::erase(iterator first, iterator last)
範囲[first, last]において文字を取り除きます。
リターン値は次のとおりです。
last の次の iterator がある場合　：last の次の iterator
last の次の iterator がない場合　：end()

string& string::replace(size_t pos1, size_t n1, const string& str)
string& string::replace(size_t pos1, size_t n1, const char* str)
位置 pos1 から n1 文字分の文字列を、str の文字列で置き換えます。
リターン値は*this です。

string& string::replace(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2)
位置 pos1 から n1 文字分の文字列を、str の位置 pos2 から n2 文字分の文字列で置き換えます。
リターン値は*this です。

string& string::replace(size_t pos, size_t n1, const char* str, size_t n2)
位置 pos から n1 文字分の文字列を、n2 個の str の文字列で置き換えます。
リターン値は*this です。

string& string::replace(size_t pos, size_t n1, size_t n2, char c)
位置 pos から n1 文字分の文字列を、n2 個の文字 c で置き換えます。
リターン値は*this です。

string& string::replace(iterator i1, iterator i2, const string& str)
string& string::replace(iterator i1, iterator i2, const char* str)
位置 i1 から i2 までの文字列を str の文字列で置き換えます。
リターン値は*this です。

string& string::replace(iterator i1, iterator i2, const char* str, size_t n)
位置 i1 から i2 までの文字列を、str の n 文字分の文字列で置き換えます。
リターン値は*this です。

string& string::replace(iterator i1, iterator i2, size_t n, char c)
位置 i1 から i2 までの文字列を、n 個の文字 c で置き換えます。
リターン値は*this です。

size_t string::copy(char* str, size_t n, size_t pos = 0) const
位置 pos に文字列 str の n 文字分の文字列をコピーします。
リターン値は rlen です。

void string::swap(string& str)
str の文字列と交換します。

const char* string::c_str() const
const char* string::data() const
文字列を格納している領域へのポインタを参照します。
リターン値は s_ptr です。
size_t string::find(const string& str, size_t pos = 0) const
size_t string::find(const char* str, size_t pos = 0) const
位置 pos 以降で str の文字列と同じ文字列が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find(const char* str, size_t pos, size_t n) const
位置 pos 以降で str の n 文字分と同じ文字列が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find(char c, size_t pos = 0) const
位置 pos 以降で文字 c が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(const string& str, size_t pos = npos) const
size_t string::rfind(const char* str, size_t pos = npos) const
位置 pos 以前で str の文字列と同じ文字列が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(const char* str, size_t pos, size_t n) const
位置 pos 以前で文字列 str の n 文字分と同じ文字列が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::rfind(char c, size_t pos = npos) const
位置 pos 以前で文字 c が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(const string& str, size_t pos = 0) const
size_t string::find_first_of(const char* str, size_t pos = 0) const
位置 pos 以降で文字列 str に含まれる任意の文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(const char* str, size_t pos, size_t n) const
位置 pos 以降で文字列 str の n 文字分に含まれる任意の文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_of(char c, size_t pos = 0) const
位置 pos 以降で文字 c が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(const string& str, size_t pos = npos) const
size_t string::find_last_of(const char* str, size_t pos = npos) const
位置 pos 以前で文字列 str に含まれる任意の文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(const char* str, size_t pos, size_t n) const
位置 pos 以前で文字列 str の n 文字分に含まれる任意の文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_of(char c, size_t pos = npos) const
位置 pos 以前で文字 c が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(const string& str, size_t pos = 0) const
size_t string::find_first_not_of(const char* str, size_t pos = 0) const
位置 pos 以降で str 中の任意の文字と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(const char* str, size_t pos, size_t n) const
位置 pos 以降で str の先頭から n 文字までの任意の文字と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_first_not_of(char c, size_t pos = 0) const
位置 pos 以降で文字 c と異なった文字が最初に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(const string& str, size_t pos = npos) const
size_t string::find_last_not_of(const char* str, size_t pos = npos) const
位置 pos 以前で str 中の任意の文字と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(const char* str, size_t pos, size_t n) const
位置 pos 以前で str の先頭から n 文字までの任意の文字と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

size_t string::find_last_not_of(char c, size_t pos = npos) const
位置 pos 以前で文字 c と異なった文字が最後に現れる位置を検索します。
リターン値は、文字列のオフセットです。

string string::substr(size_t pos = 0, size_t n = npos) const

格納された文字列に対し、範囲[pos,n]の文字列を持つオブジェクトを生成します。
リターン値は、範囲[pos,n]の文字列を持つオブジェクトです。

int string::compare(const string& str) const

文字列と str の文字列を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：0
文字列が異なる場合 ：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const string& str) const

位置 pos1 から n1 文字分の文字列と str を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：0
文字列が異なる場合 ：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const string& str, size_t pos2, size_t n2) const

位置 pos1 から n1 文字分の文字列と str の位置 pos2 から n2 文字分の文字列を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：0
文字列が異なる場合 ：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(const char* str) const

str と比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：0
文字列が異なる場合 ：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

int string::compare(size_t pos1, size_t n1, const char* str, size_t n2 = npos) const

位置 pos1 から n1 文字分の文字列と str の n2 文字分の文字列を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：0
文字列が異なる場合 ：this->s_len>str.s_len のとき 1
this->s_len < str.s_len のとき-1

(b)　string クラスマニピュレータ

種別	定義名	説明
関数	string operator+(const string& lhs, const string& rhs)	lhs の文字列(または文字)に rhs の文字列(または文字)を追加し、オブジェクトを生成してその文字列を格納します。
	string operator+(const char* lhs, const string& rhs)	
	string operator+(char lhs, const string& rhs)	
	string operator+(const string& lhs, const char* rhs)	
	string operator+(const string& lhs, char rhs)	
	bool operator==(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator==(const char* lhs, const string& rhs)	
	bool operator==(const string& lhs, const char* rhs)	
	bool operator!=(const string& lhs, const string& rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator!=(const char* lhs, const string& rhs)	
	bool operator!=(const string& lhs, const char* rhs)	
	bool operator<(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<(const char* lhs, const string& rhs)	
	bool operator<(const string& lhs, const char* rhs)	
	bool operator>(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>(const char* lhs, const string& rhs)	
	bool operator>(const string& lhs, const char* rhs)	
	bool operator<=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<=(const char* lhs, const string& rhs)	
	bool operator<=(const string& lhs, const char* rhs)	
	bool operator>=(const string& lhs, const string& rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>=(const char* lhs, const string& rhs)	
	bool operator>=(const string& lhs, const char* rhs)	
	void swap(string& lhs, string& rhs)	lhs の文字列と rhs の文字列を交換します。
	istream& operator>>(istream& is, string& str)	文字列を str に抽出します。
	ostream& operator<<(ostream& os, const string& str)	文字列を挿入します。
	istream& getline(istream& is, string& str, char delim)	is から文字列を抽出し，str に付加します。途中で文字'delim'を検出したら、入力を終了します。
	istream& getline(istream& is, string& str)	is から文字列を抽出し，str に付加します。途中で改行文字を検出したら、入力を終了します。

string operator+(const string& lhs, const string& rhs)
string operator+(const char* lhs, const string& rhs)
string operator+(char lhs, const string& rhs)
string operator+(const string& lhs, const char* rhs)
string operator+(const string& lhs, char rhs)

lhs の文字列(または文字)に rhs の文字列(または文字)を追加し、オブジェクトを生成してその文字列を格納します。
リターン値は、結合した文字列を格納するオブジェクトです。

bool operator==(const string& lhs, const string& rhs)
bool operator==(const char* lhs, const string& rhs)
bool operator==(const string& lhs, const char* rhs)

lhs の文字列と rhs の文字列を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：true
文字列が異なる場合 ：false

bool operator!=(const string& lhs, const string& rhs)
bool operator!=(const char* lhs, const string& rhs)
bool operator!=(const string& lhs, const char* rhs)

lhs の文字列と rhs の文字列を比較します。
リターン値は次のとおりです。

文字列が同一の場合 ：false
文字列が異なる場合 ：true

bool operator<(const string& lhs, const string& rhs)
bool operator<(const char* lhs, const string& rhs)
bool operator<(const string& lhs, const char* rhs)

lhs の文字列長と rhs の文字列長を比較します。
リターン値は次のとおりです。

lhs.s_len < rhs.s_len の場合 ：true
lhs.s_len >= rhs.s_len の場合 ：false

bool operator>(const string& lhs, const string& rhs)
bool operator>(const char* lhs, const string& rhs)
bool operator>(const string& lhs, const char* rhs)

lhs の文字列長と rhs の文字列長を比較します。
リターン値は次のとおりです。

lhs.s_len > rhs.s_len の場合 ：true
lhs.s_len <= rhs.s_len の場合 ：false

bool operator<=(const string& lhs, const string& rhs)
bool operator<=(const char* lhs, const string& rhs)
bool operator<=(const string& lhs, const char* rhs)
lhs の文字列長と rhs の文字列長を比較します。
リターン値は次のとおりです。
lhs.s_len <= rhs.s_len の場合　: true
lhs.s_len > rhs.s_len の場合　: false

bool operator>=(const string& lhs, const string& rhs)
bool operator>=(const char* lhs, const string& rhs)
bool operator>=(const string& lhs, const char* rhs)
lhs の文字列長と rhs の文字列長を比較します。
リターン値は次のとおりです。
lhs.s_len >= rhs.s_len の場合　: true
lhs.s_len < rhs.s_len の場合　: false

void swap(string& lhs, string& rhs)
lhs の文字列と rhs の文字列を交換します。

istream& operator>>(istream& is, string& str)
文字列を str に抽出します。
リターン値は is です。

ostream& operator<<(ostream& os, const string& str)
文字列を挿入します。
リターン値は os です。

istream& getline(istream& is, string& str, char delim)
is から文字列を抽出し、str に付加します。
途中で文字'delim'を検出したら、入力を終了します。
リターン値は is です。

istream& getline(istream& is, string& str)
is から文字列を抽出し、str に付加します。
途中で改行文字を検出したら、入力を終了します。
リターン値は is です。

10.4.3 リエントラントライブラリ

標準ライブラリ構築ツールで reent オプションを指定して作成したライブラリは、rand、srand 関数を除いてすべてリエントラントに実行できます。

reent オプションを指定していない場合について、**表 10.49** にリエントラントライブラリ一覧を示します。表中、△で示した関数は、errno 変数を設定しますので、プログラム中で errno を参照していなければリエントラントに実行できます。

リエントラント欄　○：リエントラント　×：ノンリエントラント　△：errno を設定

表 10.49 リエントラントライブラリ一覧

標準インクルードファイル	関数名	リエントラント
stddef.h	offsetof	○
assert.h	assert	×
ctype.h	isalnum	○
	isalpha	○
	iscntrl	○
	isdigit	○
	isgraph	○
	islower	○
	isprint	○
	ispunct	○
	isspace	○
	isupper	○
	isxdigit	○
	tolower	○
	toupper	○
math.h	acos	△
	asin	△
	atan	△
	atan2	△
	cos	△
	sin	△
	tan	△
	cosh	△
	sinh	△
	tanh	△
	exp	△
	frexp	△
	ldexp	△
	log	△
	log10	△
	modf	△
	pow	△
	sqrt	△
	ceil	△
	fabs	△
	floor	△
	fmod	△
mathf.h	acosf	△
	asinf	△
	atanf	△
	atan2f	△
	cosf	△
	sinf	△
	tanf	△
	coshf	△
	sinhf	△
	tanhf	△
	expf	△
	frexpf	△
	ldexpf	△
	logf	△
	log10f	△
	modff	△
	powf	△
	sqrtf	△
	ceilf	△
	fabsf	△
	floorf	△
	fmodf	△
setjmp.h	setjmp	○
	longjmp	○
stdarg.h	va_start	○
	va_arg	○
	va_end	○
stdio.h	fclose	×
	fflush	×
	fopen	×
	freopen	×
	setbuf	×
	setvbuf	×
	fprintf	×
	fscanf	×
	printf	×
	scanf	×

標準インクルードファイル	関数名	リエントラント
stdio.h	sprintf	△
	sscanf	△
	vfprintf	×
	vprintf	×
	vsprintf	△
	fgetc	×
	fgets	×
	fputc	×
	fputs	×
	getc	×
	getchar	×
	gets	×
	putc	×
	putchar	×
	puts	×
	ungetc	×
	fread	×
	fwrite	×
	fseek	×
	ftell	×
	rewind	×
	clearerr	×
	feof	×
	ferror	×
	perror	×
stdlib.h	atof	△
	atoi	△
	atol	△
	atoll	△
	atolfixed	△
	atolaccum	△
	strtod	△
	strtol	△
	strtoul	△
	strtoll	△
	strtoull	△
	strtolfixed	△
	strtolaccum	△
	rand	×
	srand	×
	calloc	×
	free	×

標準インクルードファイル	関数名	リエントラント
stdlib.h	malloc	×
	realloc	×
	free__X	×
	malloc__X	×
	realloc__X	×
	calloc__X	×
	free__Y	×
	malloc__Y	×
	realloc__Y	×
	calloc__Y	×
	bsearch	○
	qsort	○
	abs	○
	div	△
	labs	○
	llabs	○
	ldiv	△
	lldiv	△
string.h	memcpy	○
	memcpy__X__X	○
	memcpy__X__Y	○
	memcpy__Y__X	○
	memcpy__Y__Y	○
	strcpy	○
	strncpy	○
	strcat	○
	strncat	○
	memcmp	○
	strcmp	○
	strncmp	○
	memchr	○
	strchr	○
	strcspn	○
	strpbrk	○
	strrchr	○
	strspn	○
	strstr	○
	strtok	×
	memset	○
	strerror	○
	strlen	○
	memmove	○

10.4.4 未サポートライブラリ

C 言語仕様で定義しているライブラリのうち、本コンパイラでサポートしていないライブラリを表 10.50 に示します。

表 10.50 サポートしていないライブラリ

	ヘッダファイル	ライブラリ名
1	locale.h*1	setlocale、localeconv
2	signal.h*1	signal、raise
3	stdio.h	remove、rename、tmpfile、tmpnam、fgetpos、fsetpos
4	stdlib.h	abort、atexit、exit、getenv、system、 mblen、mbtowc、wctomb、mbstowcs、wcstombs
5	string.h	strcoll、strxfrm
6	time.h*1	clock、difftime、mktime、time、asctime、ctime、gmtime、localtime、 strftime

【注】*1 ヘッダファイルをサポートしません。

10.4.5 DSP ライブラリ

(1) 概要

SH2-DSP、SH3-DSP、および SH4AL-DSP(以降、合わせて単に SH-DSP と呼ぶこととします)で使用できるデジタル信号処理(DSP)ライブラリについて説明します。本ライブラリは標準的な DSP 関数を含んでおり、単独または連続的に使用することによって、DSP 演算を行うことができます。

本ライブラリは以下の関数を含んでいます。

- 高速フーリエ変換
- 窓関数
- フィルタ
- 畳み込みと相関
- その他

本ライブラリ関数は高速フーリエ変換とフィルタを除いてリエントラントです。

本ライブラリを使用するときは、表 10.51 に示すファイルをインクルードしてください。また、表 10.52 に示すように、マイコンおよびコンパイルオプションに対応するライブラリをリンクしてください。

本ライブラリを呼び出した際、関数が正常に終了した場合は EDSP_OK を、異常があった場合は EDSP_BAD_ARG もしくは EDSP_NO_HEAP をリターン値として返します。リターン値の詳細については各関数の説明を参照してください。

表 10.51 DSP ライブラリ用のインクルードファイル

	ライブラリの種類	内容	インクルードファイル
1	DSP ライブラリ	DSP 演算を行うライブラリです。	<ensigdsp.h>
			<filt_ws.h>*1

【注】*1 フィルタ関数を使用する場合、ユーザプログラム内で 1 回のみインクルードしてください。

表 10.52 DSP ライブラリ一覧

マイコン	オプション		ライブラリ名
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP	-pic=0	-endian=big	sh3dspnb.lib
	-pic=1	-endian=big	sh3dsppb.lib
	-pic=0	-endian=little	sh3dspnl.lib
	-pic=1	-endian=little	sh3dsppl.lib
SH4AL-DSP	-pic=0	-endian=big	sh4aldspnb.lib
	-pic=1	-endian=big	sh4aldsppb.lib
	-pic=0	-endian=little	sh4aldspnl.lib
	-pic=1	-endian=little	sh4aldsppl.lib

(2) データフォーマット

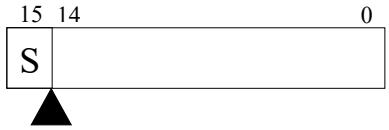
本ライブラリはデータを符号付き 16 ビット固定小数点数として扱います。符号付き 16 ビット固定小数点数は図 10.8(a)に示すように、小数点が最上位ビット(MSB)の右側に固定されたデータフォーマットとなっており、$-1 \sim 1-2^{-15}$ の範囲の値を表現できます。

本ライブラリでは、データの受け渡しは short 型のデータフォーマットを使用します。したがって、C/C++プログラムから本ライブラリを使用する場合、データを符号付き 16 ビット固定小数点数で表現する必要があります。

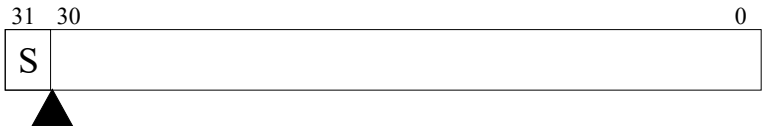
(例) +0.5 は符号付き 16 ビット固定小数点数で表現すると H'4000 です。したがってライブラリ関数に渡す short 型実引数は H'4000 となります。

本ライブラリ内部の演算では、符号付き 32 ビット固定小数点数と符号付き 40 ビット固定小数点数も使用します。符号付き 32 ビット固定小数点数は図 10.8(b)に示すデータフォーマットとなっており、$-1 \sim 1-2^{-31}$ の範囲の値を表現できます。符号付き 40 ビット固定小数点数は図 10.8(c)に示すように 8 ビットのガードビットが付加されたデータフォーマットとなっており、$-2^{8} \sim 2^{8}-2^{-31}$ の範囲の値を表現できます。

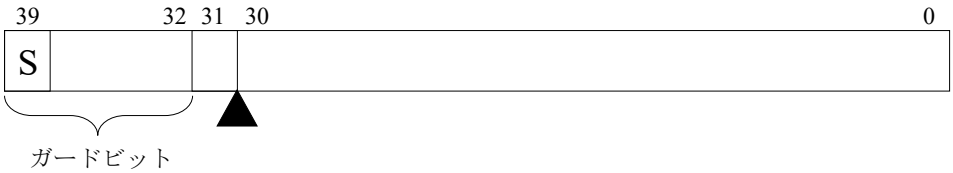
(a) 符号付き16ビット固定小数点数
($-1 \sim 1-2^{-15}$)



(b) 符号付き32ビット固定小数点数
($-1 \sim 1-2^{-31}$)



(c) 符号付き40ビット固定小数点数
($-2^{8} \sim 2^{8}-2^{-31}$)



S ：符号ビット
▲ ：小数点

図 10.8 データフォーマット

符号付き 16 ビット固定小数点数の乗算結果は符号付き 32 ビット固定小数点数で保持します。DSP 命令を用いた固定小数点乗算では、H'8000×H'8000 の場合だけオーバフローが発生することに注意してください。また乗算結果の最下位ビット(LSB)は常に 0 になります。乗算結果を次の演算に使用する場合、上位 16 ビットを取り出し、符号付き 16 ビット固定小数点数に変換します。このときアンダフローや精度低下が発生する可能性があります。

本ライブラリの積和演算では、加算結果を符号付き 40 ビット固定小数点数で保持します。加算のときにオーバフローが発生しないように注意してください。

演算の際、オーバフローが発生すると正しい結果が得られません。オーバフローを防ぐためには、係数や入力データをスケーリングする必要があります。本ライブラリには、スケーリングの機能が組み込まれています。スケーリングの詳細については各関数の説明を参照してください。

(3) 効率

本ライブラリ関数は、SH-DSP 上で高速に実行するように最適化しています。

ライブラリを効率よく活用するために、開発するシステムのメモリマップを決める際には、できるだけ以下の 2 つの推奨事項に従ってください。

- プログラムコードセグメントは、1 サイクルでの 32 ビットリードをサポートしているメモリに配置する。
- データセグメントは、1 サイクルでの 16(または 32)ビットリード・ライトをサポートしているメモリに配置する。

使用するマイコンが、ライブラリコードとデータを配置するのに十分な容量の 32 ビットメモリを内蔵している場合は、その 32 ビットメモリに配置するのが最適です。その他のメモリを使用しなければならない場合は、可能な限り上記の推奨事項に従ってください。

(4) DSP-C との併用について

DSP ライブラリと DSP-C を 1 つのプログラムで併用することができます。各関数の説明で、ライブラリ関数の型が 2 通り記述されている場合、上段が dspc オプション未指定時、下段が dspc オプション指定時に使用できます。1 通りの記述だけの関数は、dspc オプション指定有無に関わらず共通となります。

(5) 高速フーリエ変換

(a) 関数一覧

関数	説明
FftComplex	not-in-place 複素数 FFT を実行します。
FftReal	not-in-place 実数 FFT を実行します。
IfftComplex	not-in-place 複素数逆 FFT を実行します。
IfftReal	not-in-place 実数逆 FFT を実行します。
FftInComplex	in-place 複素数 FFT を実行します。
FftInReal	in-place 実数 FFT を実行します。
IfftInComplex	in-place 複素数逆 FFT を実行します。
IfftInReal	in-place 実数逆 FFT を実行します。
LogMagnitude	複素数データを対数絶対値に変換します。
InitFft	FFT 回転係数を生成します。
FreeFft	FFT 回転係数の格納に使用したメモリを解放します。

【注】※ not-in-place、in-place については「(5)(e) FFT 構造」を参照してください。

これらの関数は、ユーザが定義したスケーリングを使って、順方向高速フーリエ変換と逆方向高速フーリエ変換を実行します。

順方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{-2j\pi n/N} \cdot x_n$$

ここで、s はスケーリングが行われるステージの数、N はデータ数を示しています。

逆方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{2j\pi n/N} \cdot x_n$$

スケーリングについては「(5)(d) スケーリング」を参照してください。

(b) 複素数データ配列フォーマット

FFT および IFFT の複素数データ配列は、実数を X メモリに、虚数を Y メモリに配置します。ただし、実数 FFT の出力データと実数 IFFT の入力データの配置は異なっています。実数、虚数を格納する配列をそれぞれ x,y とすると、x[0]には DC 成分の実数成分が入り、y[0]には DC 成分の虚数成分ではなく Fs/2 成分の実数成分が入ります(DC 成分と Fs/2 成分はどちらも実数で、虚数成分は 0 です)。

(c) 実数データ配列フォーマット

FFT および IFFT の実数データ配列フォーマットには、以下の 3 種類があります。

- 単一の配列に格納し、任意のメモリブロックに配置。
- 単一の配列に格納し、X メモリに配置。
- 2 つの配列に分けて格納。それぞれの配列のサイズは N/2 で、配列の前半は X メモリに配置し、後半は Y メモリに配置。

FftReal は 1 番目の指定方法のみです。IfftReal、FftInReal および IfftInReal は 2 番目か 3 番目の方法をユーザが選択します。

(d) スケーリング

基数 2 の FFT は各ステージで信号強度が倍になり、ピーク信号振幅も倍になります。そのため、高強度信号を変換する際にオーバフローが発生することがありますが、各ステージで信号を 1/2 にすることにより(これをスケーリングといいます)オーバフローを防ぐことができます。しかし、スケーリングしすぎると不要な量子化雑音が発生する可能性があります。

オーバフローや量子化雑音とスケーリングの最適なバランスは入力信号の特性に大きく依存します。スペクトルが大きなピークを持つ信号はオーバフローを防ぐために最大スケーリングが必要になりますが、インパルス信号ではスケーリングの必要はほとんどありません。

すべてのステージでスケーリングするのが最も安全な方法です。入力データが強度 2^{30} 未満であれば、この方法でオーバフローを防ぐことができます。本ライブラリでは、各ステージごとにスケーリングを行うかどうかを指定できます。したがって、スケーリング指定を精密に行うことによって、オーバフローと量子化雑音の影響を最小限に抑えることができます。

スケーリングの方法を指定するために、各 FFT 関数の引数に scale が含まれています。scale は最下位ビットから 1 ビットずつが各ステージに対応しています。対応する scale のビットが 1 に設定されているすべてのステージで、2 の除算を実行します。

本ライブラリは実行速度を上げるために基数 4 の FFT を使用しています。scale は最下位ビットから 2 ビットずつが各ステージに対応しています。どちらか 1 ビットが 1 に設定されていれば、2 の除算を実行します。両方が 1 に設定されていれば 4 の除算を実行します。つまり、2 つの基数 2 の FFT ステージが 1 つの基数 4 の FFT ステージに置き換えられたのと同じことになります。しかし、基数 2 の FFT よりも基数 4 の FFT の方が量子化雑音の発生する可能性があります。

以下に scale の例を示します。

- scale = H’FFFFFFFF(または size-1)はすべての基数 2 の FFT ステージでスケーリングを行います。すべての入力データの強度が 2^{30} 未満であれば、オーバフローは発生しません。
- scale = H’55555555 は 1 つおきの基数 2 の FFT ステージでスケーリングを行います。
- scale = 0 はスケーリングを行いません。

これらの scale の値は、ensigdsp.h で EFFTALLSCALE(H’FFFFFFFF)、EFFTMIDSCALE(H’55555555)、EFFTNOSCALE(0)と定義されています。

(e) FFT 構造

本ライブラリの FFT 構造には not-in-place FFT と in-place FFT の 2 種類があります。

not-in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM のユーザが指定した別の場所に格納します。

一方 in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM の同じ場所に格納します。この方法を用いると FFT の実行時間は増加しますが、使用メモリスペースが削減できます。

入力データを FFT 関数の他にも使用する場合は、not-in-place FFT を使用してください。また、メモリスペースを節約したい場合は、in-place FFT を使用してください。

not-in-place 複素数 FFT

int FftComplex (short op_x[ ], short op_y[ ], const short ip_x[ ], const short ip_y[ ], long size, long scale)
int FftComplex (_ _X _ _fixed op_x[ ], _ _Y _ _fixed op_y[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long size, long scale)

説　明　　複素数高速フーリエ変換を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

引　数

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定

備　考　　本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) スケーリング」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

not-in-place 実数 FFT

int FftReal (short op_x[ ], short op_y[ ], const short ip[ ], long size, long scale)
int FftReal (_ _X _ _fixed op_x[ ], _ _Y _ _fixed op_y[ ], const _ _fixed ip[ ], long size, long scale)

説　明　実数高速フーリエ変換を実行します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size

引　数

op_x[]	正の出力データの実数成分
op_y[]	正の出力データの虚数成分
ip[]	実数入力データ
size	FFT のサイズ
scale	スケーリング指定

備　考

op_x と op_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_S/2$ での出力データの値は実数なので、$F_S/2$ での実数出力は op_y[0]に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」「(5)(c) 実数データ配列フォーマット」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) スケーリング」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

not-in-place 複素数逆 FFT

int IfftComplex (short op_x[ ], short op_y[ ], const short ip_x[ ], const short ip_y[ ], long size, long scale)
int IfftComplex (_ _X _ _fixed op_x[ ], _ _Y _ _fixed op_y[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long size, long scale)

説　明　　複素数逆高速フーリエ変換を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

引　数

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定

備　考　　本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(5)(b) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

not-in-place 実数逆 FFT

int IfftReal (short op_x[ ], short scratch_y[ ], const short ip_x[ ], const short ip_y[ ], long size, long scale, int op_all_x)
int IfftReal (_ _X _ _fixed op_x[ ], _ _Y _ _fixed scratch_y[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long size, long scale, int op_all_x)

説　明　　実数逆高速フーリエ変換を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

引　数

op_x[]	実数出力データ
scratch_y[]	スクラッチメモリまたは実数出力データ
ip_x[]	正の入力データの実数成分
ip_y[]	正の入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

備　考

ip_x と ip_y には size/2 の正の入力データを格納してください。負の入力データは正の入力データの共役複素数です。また、0 と $F_S/2$ での入力データの値は実数なので、$F_S/2$ での実数入力は ip_y[0]に格納してください。
出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは op_x に格納されます。op_all_x=0 の場合、最初の size/2 の出力データは op_x に格納され、残りの size/2 の出力データは scratch_y に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(5)(b) **複素数データ配列フォーマット**」「(5)(c) **実数データ配列フォーマット**」を参照してください。
ip_x、ip_y はそれぞれ size/2 のデータを格納してください。op_x は op_all_x の値によって、size または size/2 のデータが格納されます。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

in-place 複素数 FFT

int FftInComplex (short data_x[ ], short data_y[ ], long size, long scale)
int FftInComplex (_ _X_ _fixed data_x[ ], _ _Y_ _fixed data_y[ ], long size, long scale)

説　明　　in-place 複素数高速フーリエ変換を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

引　数

data_x[]	入出力データの実数成分
data_y[]	入出力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定

備　考　　複素数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) スケーリング」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

in-place 実数 FFT

int FftInReal (short data_x[ ], short data_y[ ], long size, long scale, int ip_all_x)
int FftInReal (_ _X _ _fixed data_x[ ], _ _Y _ _fixed data_y[ ], long size, long scale, int ip_all_x)

説　明　in-place 実数高速フーリエ変換を実行します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・ip_all_x ≠ 0 または 1

引　数

data_x[]	入力時は実数データ、出力時は正の出力データの実数成分
data_y[]	入力時は実数データまたは未使用、出力時は正の出力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定
ip_all_x	入力データの配置指定

備　考

入力データのフォーマットは、ip_all_x で指定します。ip_all_x=1 の場合、全入力データは data_x から取り出します。ip_all_x=0 の場合、前半の size/2 の入力データは data_x から、後半の size/2 の入力データは data_y から取り出します。
本関数実行後、data_x と data_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_S/2$ での出力データの値は実数なので、$F_S/2$ での実数出力は data_y[0]に格納されます。
複素数と実数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」「(5)(c) 実数データ配列フォーマット」を参照してください。
data_y は size/2 のデータを格納します。data_x は ip_all_x の値によって size または size/2 のデータを格納します。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) スケーリング」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

in-place 複素数逆 FFT

int IfftInComplex (short data_x[ ], short data_y[ ], long size, long scale)
int IfftInComplex (_ _X _ _fixed data_x[ ], _ _Y _ _fixed data_y[ ], long size, long scale)

説　明	in-place 複素数逆高速フーリエ変換を実行します。	
ヘッダ	<ensigdsp.h>	
リターン値	EDSP_OK	成功
	EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size
引　数	data_x[]	入出力データの実数成分
	data_y[]	入出力データの虚数成分
	size	逆 FFT のサイズ
	scale	スケーリング指定

備　考

複素数データ配列の配置については「(5)(b) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

in-place 実数逆 FFT

int IfftInReal (short data_x[ ], short data_y[ ], long size, long scale, int op_all_x)
int IfftInReal (_ _X _ _fixed data_x[ ], _ _Y _ _fixed data_y[ ], long size, long scale, int op_all_x)

説　明　in-place 実数逆高速フーリエ変換を実行します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

引　数

data_x[]	入力時は正の入力データの実数成分、出力時は実数データ
data_y[]	入力時は正の入力データの虚数成分、出力時は実数データまたは未使用
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

備　考

data_x と data_y には size/2 の正の入力データを格納してください。負の入力データは正の入データの共役複素数です。また、0 と $F_S/2$ での入力データの値は実数なので、$F_S/2$ での実数入力は data_y[0]に格納してください。
出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは data_x に格納されます。op_all_x=0 の場合、前半の size/2 の出力データは data_x に格納され、後半の size/2 の出力データは data_y に格納されます。
複素数と実数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」「(5)(c) 実数データ配列フォーマット」を参照してください。
data_y は size/2 のデータを格納します。data_x は、op_all_x の値によって size または size/2 のデータが格納されます。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(5)(d) スケーリング」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

対数絶対値

int LogMagnitude (short output[ ], const short ip_x[ ], const short ip_y[ ], long no_elements, float fscale)
int LogMagnitude (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long no_elements, float fscale)

説　明　　複素数入力データの対数絶対値をデシベル単位で計算し、スケーリング結果を出力配列に書き込みます。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・no_elements > 32767 ・|fscale| ≥ $2^{15}/(10\log_{10}2^{31})$

引　数

output[]	実数出力 z
ip_x[]	入力の実数成分 x
ip_y[]	入力の虚数成分 y
no_elements	出力データ数 N
fscale	出力スケーリング係数

備　考　　$z(n)=10fscale\cdot\log_{10}(x(n)^2+y(n)^2)$　　$0 \le n < N$
複素数データ配列の配置については「(5)(b) 複素数データ配列フォーマット」を参照してください。

回転係数生成

int InitFft (long max_size)

説　明　FFT 関数で使用する回転係数（1/4 サイズ）を生成します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_NO_HEAP	malloc で確保できるメモリスペースが不十分
EDSP_BAD_ARG	以下のいずれかの場合です ・max_size < 2 ・max_size が 2 の累乗ではありません ・max_size > 32768

引　数

max_size	必要になる FFT の最大サイズ

備　考　回転係数は malloc によって確保されるメモリに格納されます。
回転係数が生成されると max_fft_size 大域変数が更新されます。max_fft_size は FFT の最大許容サイズを示します。
本関数は最初の FFT 関数を呼び出す前に必ず一度呼び出してください。
max_size は 8 以上としてください。
回転係数は max_size で指定した変換サイズで生成されます。max_size より小さいサイズの FFT 関数を実行したときも同じ回転係数を使用します。
回転係数のアドレスは内部変数内に格納されています。ここはユーザプログラムでアクセスしないでください。
本関数はリエントラントではありません。

回転係数解放

void FreeFft (void)

説　明　回転係数の格納に使用したメモリを解放します。

ヘッダ　<ensigdsp.h>

備　考　max_fft_size 大域変数を 0 にします。FreeFft を実行した後再び FFT 関数を実行するときには、その前に必ず InitFft を実行してください。
本関数はリエントラントではありません。

(6) 窓関数

(a) 関数一覧

関数名	説明
GenBlackman	ブラックマン窓を生成します。
GenHamming	ハミング窓を生成します。
GenHanning	ハニング窓を生成します。
GenTriangle	三角窓を生成します。

ブラックマン窓

int GenBlackman (short output[ ], long win_size)
int GenBlackman (_ _fixed output[ ], long win_size)

説　明　　ブラックマン窓を生成し、output に出力します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

引　数

output[]	出力データ W(n)
win_size	窓サイズ N

備　考　　実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \qquad 0 \le n < N$$

ハミング窓

int GenHamming (short output[ ], long win_size)
int GenHamming (_ _fixed output[ ], long win_size)

説　明　　ハミング窓を生成し、output に出力します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

引　数

output[]	出力データ W(n)
win_size	窓サイズ N

備　考　　実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

ハニング窓

int GenHanning (short output[ ], long win_size)
int GenHanning (_ _fixed output[ ], long win_size)

説　明　　ハニング窓を生成し、output に出力します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

引　数

output[]	出力データ W(n)
win_size	窓サイズ N

備　考　　実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

三角窓

int GenTriangle (short output[ ], long win_size)
int GenTriangle (_ _fixed output[ ], long win_size)

説　明　　三角窓を生成し、output に出力します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	win_size ≤ 1

引　数

output[]	出力データ W(n)
win_size	窓サイズ N

備　考　　実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \qquad 0 \le n < N$$

(7) フィルタ

(a) 関数一覧

関数名	説明
Fir	有限インパルス応答フィルタ処理を実行します。
Fir1	単一データ用有限インパルス応答フィルタ処理を実行します。
Iir	無限インパルス応答フィルタ処理を実行します。
Iir1	単一データ用無限インパルス応答フィルタ処理を実行します。
DIir	倍精度無限インパルス応答フィルタ処理を実行します。
DIir1	単一データ用倍精度無限インパルス応答フィルタ処理を実行します。
Lms	適応 FIR フィルタ処理を実行します。
Lms1	単一データ用適応 FIR フィルタ処理を実行します。
InitFir	FIR フィルタ用に作業領域を割り付けます。
InitIir	IIR フィルタ用に作業領域を割り付けます。
InitDIir	DIIR フィルタ用に作業領域を割り付けます。
InitLms	LMS フィルタ用に作業領域を割り付けます。
FreeFir	InitFir で割り付けられた作業領域を解放します。
FreeIir	InitIir で割り付けられた作業領域を解放します。
FreeDIir	InitDIir で割り付けられた作業領域を解放します。
FreeLms	InitLms で割り付けられた作業領域を解放します。

【注】 本関数のいずれかを使用する場合、ユーザプログラム内で 1 回のみ filt_ws.h をインクルードしてください。

(b) 係数のスケーリング

フィルタ処理を行うと飽和または量子化雑音が発生する可能性があります。これらはフィルタ係数のスケーリングを行うことによって最小限に抑えることができます。しかし、飽和と量子化雑音の影響をよく考えてスケーリングを行わなければなりません。係数が大きすぎると飽和が、小さすぎると量子化雑音が発生する可能性があります。

FIR(有限インパルス応答)フィルタの場合、以下の式が成り立つようにフィルタ係数を設定すれば飽和は起こりません。

$coeff[i] \neq H'8000$　(すべての i について)

$\Sigma |coeff| < 2^{24}$

$res\_shift = 24$

coeff はフィルタ係数、res_shift は出力で行われる右シフトのビット数です。

しかし、多くの入力信号の場合、もっと小さい res_shift の値(またはもっと大きな coeff の値)を使用しても飽和する可能性は少なく、量子化雑音も大幅に削減できます。また入力値に H'8000 が含まれている可能性があれば、すべての coeff の値は H'8001～H'7FFF の範囲になるように設定してください。

IIR(無限インパルス応答)フィルタは再帰的な構造になっています。そのため上述したようなスケーリング方法は適していません。

LMS(最小 2 乗平均)適応フィルタは FIR フィルタと同様です。しかし、係数を適応するときに飽和する場合があります。その場合は、係数に H'8000 を含まないように設定してください。

(c) 作業領域

デジタルフィルタでは、ある処理から次の処理へ保持しておかなければならない情報があります。これらの情報は、最小オーバーヘッドでアクセスすることができるメモリに格納します。本ライブラリでは、Y-RAM 領域を作業領域として使用します。作業領域はフィルタ処理を実行する前に Init 関数を呼び出して初期化してください。

作業領域メモリはライブラリ関数によってアクセスされます。なお、ユーザプログラムから作業領域を直接アクセスしないでください。

(d) メモリの使用

SH-DSP を効率よく使うために、フィルタ係数は X メモリに配置してください。入出力データは任意のメモリセグメントに配置することができます。

フィルタ係数は#pragma section を用いて X メモリに配置してください。

dspc オプション指定時は_ _X メモリ型修飾子を使用してください。この場合、#pragma section 指定は必要ありません。

各フィルタは Init 関数を用いてグローバルバッファから作業領域を割り付けます。グローバルバッファは Y メモリに配置します。

FIR

*int Fir (short output[ ], const short input[ ], long no_samples, const short coeff[ ], long no_coeffs, int res_shift, short *workspace)*
*int Fir (_ _fixed output[ ], const _ _fixed input[ ], long no_samples, const _ _X _ _fixed coeff[ ], long no_coeffs, int res_shift, _ _Y _ _fixed *workspace)*

説　明　有限インパルス応答(FIR)フィルタ処理を実行します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 6 ・res_shift < 0 ・res_shift > 25

引　数

output[]	出力データ y
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数 h
no_coeffs	係数の数(フィルタの長さ)K
res_shift	各出力に適用される右シフト
workspace	作業領域へのポインタ

備　考　最新の入力データは作業領域に保持されます。input をフィルタ処理した結果は output に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
係数のスケーリングについては「(7)(b) 係数のスケーリング」を参照してください。
本関数を呼び出す前に InitFir を呼び出し、フィルタの作業領域を初期化してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

単一データ用 FIR

*int Fir1 (short *output, short input, const short coeff[ ], long no_coeffs, int res_shift, short *workspace)*

説　明　　単一データ用に有限インパルス応答(FIR)フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_coeffs ≤ 6 ・res_shift < 0 ・res_shift > 25

引　数

output	出力データ y(n) へのポインタ
input	入力データ x(n)
coeff[]	フィルタ係数 h
no_coeffs	係数の数(フィルタの長さ)K
res_shift	各出力に適用される右シフト
workspace	作業領域へのポインタ

備　考　　最新の入力データは作業領域に保持されます。input をフィルタ処理した結果は*output に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
係数のスケーリングについては「(7)(b) 係数のスケーリング」を参照してください。
関数を呼び出す前に InitFir を呼び出し、フィルタの作業領域を初期化してください。
本関数はリエントラントではありません。

IIR

*int Iir (short output[ ], const short input[ ], long no_samples, const short coeff[ ], long no_sections, short *workspace)*
*int Iir (_ _fixed output[ ], const _ _fixed input[ ], long no_samples, const _ _X _ _fixed coeff[ ], long no_sections, _ _Y _ _fixed *workspace)*

説　明　　無限インパルス応答(IIR)フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・a_{0k} < 0 ・a_{0k} > 16

引　数

output[]	出力データ y_{K-1}
input[]	入力データ x_0
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

備　考

フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。
各バイカッドの出力は以下の式で与えられます。

$D_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)]\cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。
coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}, ... , b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。
各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

単一データ用IIR

*int Iir1 (short *output, short input, const short coeff[ ], long no_sections, short *workspace)*

説　明　　単一データ用に無限インパルス応答(IIR)フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・a_{0k} < 0 ・a_{0k} > 16

引　数

output	出力データ $y_{K-1}(n)$ へのポインタ
input	入力データ $x_0(n)$
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

備　考

フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。
各バイカッドの出力は以下の式で与えられます。

$D_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)]\cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。
coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}, ... , b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。
各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。
本関数はリエントラントではありません。

倍精度 IIR

*int DIir (short output[ ], const short input[ ], long no_samples, const long coeff[ ], long no_sections, long *workspace)*
*int DIir (_ _fixed output[ ], const _ _fixed input[ ], long no_samples, const _ _X long _ _fixed coeff[ ], long no_sections, _ _Y long _ _fixed *workspace)*

説　明　　倍精度無限インパルス応答フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・a_{0k} < 3 ・k < K-1 で a_{0k} > 32 ・k = K-1 で a_{0k} > 48

引　数

output[]	出力データ y_{K-1}
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

備　考

フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。
各バイカッドの出力は、以下の方程式で与えられます。

$D_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)]\cdot 2^{-31}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}\cdot 2^{2}$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション (k=0) の入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション (k=K-1) の出力は output に書き込まれます。
coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}, ... , b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。
DIir は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは正または負の最大値となります。最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。
本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。
遅延ノード $d_k(n)$ は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。
DIir は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k < K-1 のときは 31、k=K-1 のときは 47 に設定してください。
DIir より Iir の方が実行速度は速いので、倍精度計算の必要がなければ Iir を使用してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

単一データ用倍精度 IIR

*int DIir1 (short *output, const short input, const long coeff[ ], long no_sections, long *workspace)*

説　明　単一データ用に倍精度無限インパルス応答フィルタ処理を実行します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・a_{0k} < 3 ・k < K-1 で a_{0k} > 32 ・k = K-1 で a_{0k} > 48

引　数

output	出力データ $y_{K-1}(n)$ へのポインタ
input	入力データ $x_0(n)$
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

備　考

フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。

各バイカッドの出力は、以下の方程式で与えられます。

$D_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)]\cdot 2^{-31}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a_{0k}}\cdot 2^2$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション (k=0) への入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション (k=K-1) からの出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01}, ... , b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

DIir1 は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir1 と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは正または負の最大値となります。最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。

遅延ノード $d_k(n)$ は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。

DIir1 は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k < K-1 のときは 31、k=K-1 のときは 47 に設定してください。

DIir1 より Iir1 の方が実行速度は速いので、倍精度計算の必要がなければ Iir1 を使用してください。

本関数はリエントラントではありません。

適応 FIR

*int Lms (short output[ ], const short input[ ], const short ref_output[ ], long no_samples, short coeff[ ], long no_coeffs, int res_shift, short conv_fact, short *workspace)*
*int Lms (_ _fixed output[ ], const _ _fixed input[ ], const _ _fixed ref_output[ ], long no_samples, _ _X _ _fixed coeff[ ], long no_coeffs, int res_shift, short conv_fact, _ _Y _ _fixed *workspace)*

説　明　　最小 2 乗平均アルゴリズム(LMS)を使って、実数適応 FIR フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 6 ・res_shift < 0 ・res_shift > 25

引　数

output[]	出力データ y
input[]	入力データ x
ref_output[]	所望の出力値 d
no_samples	入力データの数 N
coeff[]	適応フィルタ係数 h
no_coeffs	係数の数 K
res_shift	各出力に適用される右シフト
conv_fact	収束係数 2μ
workspace	作業領域へのポインタ

備　考

FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

フィルタ係数の更新は Widrow-Hoff アルゴリズムを使用します。

$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$

ここで e(n)は所望する信号と実際の出力の誤差です。

$e(n)=d(n)-y(n)$

2μe(n)x(n-k)の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算結果とも上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。

係数のスケーリングについては「(7)(b) 係数のスケーリング」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。

conv_fact は通常正に設定してください。また H'8000 には設定しないでください。

本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。

output に input または ref_output と同じ配列を指定した場合、input または ref_output は上書きされます。

本関数はリエントラントではありません。

単一データ用適応 FIR

*int Lms1 (short *output, short input, short ref_output, short coeff[ ], long no_coeffs, int res_shift, short conv_fact, short *workspace)*

説　明　　最小 2 乗平均アルゴリズム(LMS)を使って、単一データ用に実数適応 FIR フィルタ処理を実行します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_coeffs ≤ 6 ・res_shift < 0 ・res_shift > 25

引　数

output	出力データ y(n)へのポインタ
input	入力データ x(n)
ref_output	所望の出力値 d(n)
coeff[]	適応フィルタ係数 h
no_coeffs	係数の数 K
res_shift	各出力に適用される右シフト
conv_fact	収束係数 2μ
workspace	作業領域へのポインタ

備　考　　FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res\_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
フィルタ係数の更新は Widrow-Hoff アルゴリズムを使用します。
$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$
ここで e(n)は所望する信号と実際の出力の誤差です。
e(n)=d(n)-y(n)
2μe(n)x(n-k)の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算とも、上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。
係数のスケーリングについては「(7)(b) 係数のスケーリング」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。
conv_fact は通常正に設定してください。また H'8000 には設定しないでください。
本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。
本関数はリエントラントではありません。

FIR 作業領域割り付け

*int InitFir (short **workspace, long no_coeffs)*
*int InitFir (_ _Y _ _fixed **workspace, long no_coeffs)*

説　明　Fir と Fir1 で使用する作業領域を割り付けます。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_coeffs ≤ 2

引　数

workspace	作業領域へのポインタへのポインタ
no_coeffs	係数の数 K

備　考　すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitFir で割り付けられた作業領域を操作することができます。ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

IIR 作業領域割り付け

*int InitIir (short **workspace, long no_sections)*
*int InitIir (_ _Y _ _fixed **workspace, long no_sections)*

説　明　Iir と Iir1 で使用する作業領域を割り付けます。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_sections < 1

引　数

workspace	作業領域へのポインタへのポインタ
no_sections	2 次フィルタセクションの数 K

備　考　すでに入力されているデータは 0 に初期化されます。
Iir と Iir1 だけが InitIir で割り付けられた作業領域を操作することができます。ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

倍精度 IIR 作業領域割り付け

*int InitDIir (long **workspace, long no_sections)*
*int InitDIir (_ _Y long _ _fixed **workspace, long no_sections)*

説　明　DIir と DIir1 で使用する作業領域を割り付けます。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_sections < 1

引　数

workspace	作業領域へのポインタへのポインタ
no_sections	2 次フィルタセクションの数 K

備　考　すでに入力されているデータは 0 に初期化されます。
DIir と DIir1 だけが InitDIir で割り付けられた作業領域を操作することができます。
本関数はリエントラントではありません。

適応 FIR 作業領域割り付け

*int InitLms (short **workspace, long no_coeffs)*
*int InitLms (_ _Y _ _fixed **workspace, long no_coeffs)*

説　明　Lms と Lms1 で使用する作業領域を割り付けます。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_coeffs ≤ 2

引　数

workspace	作業領域へのポインタへのポインタ
no_coeffs	係数の数 K

備　考　すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitLms で割り付けられた作業領域を操作することができます。ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

FIR 作業領域解放

*int FreeFir (short **workspace, long no_coeffs)*
*int FreeFir (_ _Y _ _fixed **workspace, long no_coeffs)*

説　明　InitFir で割り付けられた作業領域を解放します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	no_coeffs ≤ 2

引　数

workspace	作業領域へのポインタへのポインタ
no_coeffs	係数の数 K

備　考　本関数はリエントラントではありません。

IIR 作業領域解放

*int FreeIir (short **workspace, long no_sections)*
*int FreeIir (_ _Y _ _fixed **workspace, long no_sections)*

説　明　InitIir で割り付けられた作業領域を解放します。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	no_sections < 1

引　数

workspace	作業領域へのポインタへのポインタ
no_sections	2 次フィルタセクションの数 K

備　考　本関数はリエントラントではありません。

倍精度 IIR 作業領域解放

*int FreeDIir (long **workspace, long no_sections)*
*int FreeDIir (_ _Y long _ _fixed **workspace, long no_sections)*

説　明　　InitDIir で割り付けられた作業領域メモリを解放します。

ヘッダ　　<ensigdsp.h>

リターン値　　EDSP_OK　　成功
EDSP_BAD_ARG　　no_section ≤ 2

引　数　　workspace　　作業領域へのポインタへのポインタ
no_sections　　2 次フィルタセクションの数 K

備　考　　本関数はリエントラントではありません。

適応 FIR 作業領域解放

*int FreeLms (short **workspace, long no_coeffs)*
*int FreeLms (_ _Y _ _fixed **workspace, long no_coeffs)*

説　明　　InitLms で割り付けられた作業領域メモリを解放します。

ヘッダ　　<ensigdsp.h>

リターン値　　EDSP_OK　　成功
EDSP_BAD_ARG　　no_coeffs < 1

引　数　　workspace　　作業領域へのポインタへのポインタ
no_coeffs　　係数の数 K

備　考　　本関数はリエントラントではありません。

(8)　畳み込みと相関

(a)　関数一覧

関数名	説明
ConvComplete	2 つの配列の完全な畳み込みを計算します。
ConvCyclic	2 つの配列の周期的な畳み込みを計算します。
ConvPartial	2 つの配列の部分的な畳み込みを計算します。
Correlate	2 つの配列の相関を計算します。
CorrCyclic	2 つの配列の周期的な相関を計算します。

これらの関数を使用する際は、2 つの入力配列のうち 1 つは X メモリに、もう 1 つは Y メモリに配置してください。出力配列はどのメモリに配置してもかまいません。

完全畳み込み

int ConvComplete (short output[ ], const short ip_x[ ], const short ip_y[ ], long x_size, long y_size, int res_shift)
int ConvComplete (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long x_size, long y_size, int res_shift)

説　明　　2 つの入力配列 x,y を完全に畳み込み、結果を出力配列 z に書き出します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・res_shift < 0 ・res_shift > 25

引　数

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

備　考

$$z(m) = \left[\sum_{i=0}^{X-1} x(i)\ y(m-i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < X+Y-1$$

入力配列外のデータは 0 として読み込まれます。
出力配列サイズは X+Y-1 以上に設定してください。

周期的畳み込み

int ConvCyclic (short output[ ], const short ip_x[ ], const short ip_y[ ], long size, int res_shift)
int ConvCyclic (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long size, int res_shift)

説　明　　2 つの入力配列 x,y を周期的に畳み込み、結果を出力配列 z に書き出します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 6 ・res_shift < 0 ・res_shift > 25

引　数

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
res_shift	各出力に適用される右シフト

備　考

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\ y\left(\left|m - i + N\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$ は剰余 (i % N) を意味します。

部分的畳み込み

int ConvPartial (short output[ ], const short ip_x[ ], const short ip_y[ ], long x_size, long y_size, int res_shift)
int ConvPartial (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long x_size, long y_size, int res_shift)

説　明　　本関数は 2 つの入力配列 x,y を畳み込み、結果を出力配列 z に書き出します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 5 ・y_size < 1 ・res_shift < 0 ・res_shift > 25

引　数

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

備　考　　入力配列外のデータから引き出された出力は含まれていません。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m \le |A-B|$$

ただし、配列の個数は a < b で、A は a のサイズ、B は b のサイズです。
出力配列サイズは|X-Y|+1 以上に設定してください。
入力配列外のデータは 0 として読み込まれます。

相関

int Correlate (short output[ ], const short ip_x[ ], const short ip_y[ ], long x_size, long y_size, long no_corr, int x_is_larger, int res_shift)
int Correlate (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long x_size, long y_size, long no_corr, int x_is_larger, int res_shift)

説　明　　2 つの入力配列 x,y の相関を求め、結果を出力配列 z に書き出します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・no_corr < 1 ・res_shift < 0 ・res_shift > 25 ・x_is_larger ≠ 0 または 1

引　数

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
no_corr	計算する相関の数 M
x_is_larger	X=Y のときの配列指定
res_shift	各出力に適用される右シフト

備　考　　以下の式では配列の個数は a>b で、A は a のサイズとします。X=Y の場合は、x_is_larger=1 とすると x を a とし、x_is_larger=0 とすると x を b とします。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(i + m)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < M$$

A < X + M となっても差し支えありません。この場合、入力配列外のデータは 0 を使用します。

res_shift = 0 は通常の整数計算に、res_shift = 15 は小数計算に相当します。

周期的相関

int CorrCyclic (short output[ ], const short ip_x[ ], const short ip_y[ ], long size, int reverse, int res_shift)
int CorrCyclic (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long size, int reverse, int res_shift)

説　明　　周期的に配列 x,y の相関を求め、結果を出力配列 z に書き出します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 5 ・res_shift < 0 ・res_shift > 25 ・reverse ≠ 0 または 1

引　数

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
reverse	反転フラグ
res_shift	各出力に適用される右シフト

備　考

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$ は剰余 (i % N) を意味します。reverse=1 の場合、出力のデータは反転され、実際の計算は以下のようになります。

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\, x\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

(9) その他

(a) 関数一覧

関数名	説明
Limit	H'8000 のデータを H'8001 に置き換えます。
CopyXtoY	配列を X メモリから Y メモリにコピーします。
CopyYtoX	配列を Y メモリから X メモリにコピーします。
CopyToX	配列を指定した場所から X メモリにコピーします。
CopyToY	配列を指定した場所から Y メモリにコピーします。
CopyFromX	配列を X メモリから指定した場所にコピーします。
CopyFromY	配列を Y メモリから指定した場所にコピーします。
GenGWnoise	白色ガウス雑音を生成します。
MatrixMult	2 つのマトリックスの乗算をします。
VectorMult	2 つのデータの乗算をします。
MsPower	2 乗平均強度を求めます。
Mean	平均を求めます。
Variance	平均と偏差を求めます。
MaxI	整数配列の最大値を求めます。
MinI	整数配列の最小値を求めます。
PeakI	整数配列の最大絶対値を求めます。

H'8000→H'8001 置換

int Limit (short data_xy[ ], long no_elements, int data_is_x)
int Limit (__fixed data_xy[ ], long no_elements, int data_is_x)

説　明　　値が H'8000 の入力データを H'8001 に置き換えます。これにより、DSP 命令の固定小数点乗算の際にオーバフローが発生しないようにします。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・data_is_x ≠ 0 または 1

引　数

data_xy[]	データ配列
no_elements	データ数
data_is_x	データ配置指定

備　考　　この処理を行っても積和演算の加算でオーバフローが発生する可能性はあります。
data_is_x=1 のときはデータは X メモリに、data_is_x=0 のときはデータは Y メモリに配置してください。

X メモリ→Y メモリコピー

int CopyXtoY (short op_y[ ], const short ip_x[ ], long n)
int CopyXtoY (_ _Y _ _fixed op_y[ ], const _ _X _ _fixed ip_x[ ], long n)

説　明　　配列を ip_x から op_y へコピーします。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 6

引　数

op_y[]	出力配列
ip_x[]	入力配列
n	データ数

備　考　　ip_x は X メモリに、op_y は Y メモリに配置してください。

Y メモリ→X メモリコピー

int CopyYtoX (short op_x[ ], const short ip_y[ ], long n)
int CopyYtoX (_ _X _ _fixed op_x[ ], const _ _Y _ _fixed ip_y[ ], long n)

説　明　　配列を ip_y から op_x へコピーします。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

引　数

op_x[]	出力配列
ip_y[]	入力配列
n	データ数

備　考　　op_x は X メモリに、ip_y は Y メモリに配置してください。

X メモリへコピー

int CopyToX (short op_x[ ], const short input[ ], long n)
int CopyToX (_ _X _ _fixed op_x[ ], const _ _fixed input[ ], long n)

説　明　　配列 input を op_x へコピーします。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

引　数

op_x[]	出力配列
input[]	入力配列
n	データ数

備　考　　op_x は X メモリに、input は任意のメモリに配置してください。

Y メモリへコピー

int CopyToY (short op_y[ ], const short input[ ], long n)
int CopyToY (_ _Y _ _fixed op_y[ ], const _ _fixed input[ ], long n)

説　明　　配列 input を op_y へコピーします。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

引　数

op_y[]	出力配列
input[]	入力配列
n	データ数

備　考　　op_y は Y メモリに、input は任意のメモリに配置してください。

X メモリからコピー

int CopyFromX (short output[ ], const short ip_x[ ], long n)
int CopyFromX (_ _fixed output[ ], const _ _X_ _fixed ip_x[ ], long n)

説　明　配列 ip_x を output へコピーします。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

引　数

output[]	出力配列
ip_x[]	入力配列
n	データ数

備　考　ip_x は X メモリに、output は任意のメモリに配置してください。

Y メモリからコピー

int CopyFromY (short output[ ], const short ip_y[ ], long n)
int CopyFromY (_ _fixed output[ ], const _ _Y_ _fixed ip_y[ ], long n)

説　明　配列 ip_y を output へコピーします。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

引　数

output[]	出力配列
ip_y[]	入力配列
n	データ数

備　考　ip_y は Y メモリに、output は任意のメモリに配置してください。

白色ガウス雑音

int GenGWnoise (short output[ ], long no_samples, float variance)
int GenGWnoise (_ _fixed output[ ], long no_samples, float variance)

説　明　　平均が 0 で、ユーザが指定した偏差をもつ白色ガウス雑音を生成します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・variance ≤ 0.0

引　数

output[]	白色雑音データの出力
no_samples	出力データ数
variance	ノイズ分布の偏差 σ^2

備　考　　出力データは 2 つ 1 組で生成されます。1 組の出力データを生成するために rand 関数を使用し、x の 2 乗合計が 1 未満になる組が求まるまで-1～1 の間で 1 組の乱数 γ_1、γ_2 を生成します。そして、1 組の出力データ o_1、o_2 が以下の式で計算されます。

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

データ数を奇数に設定した場合、最後の組の 2 番目のデータは破棄されます。
本関数が呼び出している標準ライブラリの rand 関数はリエントラントではないので、生成される乱数 γ_1、γ_2 の順番が常に同じになるとは限りません。しかし、生成される白色雑音 o_1、o_2 の特性に影響を及ぼすことはありません。
本関数は浮動小数点演算を使用しています。浮動小数点演算は処理速度が遅くなるので、本関数は評価用として使うことをおすすめします。

マトリックス乗算

*int MatrixMult (void *op_matrix, const void *ip_x, const void *ip_y, long m, long n, long p, int x_first, int res_shift)*
*int MatrixMult (void *op_matrix, const _ _X void *ip_x, const _ _Y void *ip_y, long m, long n, long p, int x_first, int res_shift)*

説　明　　2 つのマトリックス x,y の乗算を行い、結果を op_matrix に配置します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・m,n,または p < 5 ・res_shift < 0 ・res_shift > 25 ・x_first ≠ 0 または 1

引　数

op_matrix	出力の第一データへのポインタ
ip_x	入力 x の第一データへのポインタ
ip_y	入力 y の第一データへのポインタ
m	マトリックス 1 の行数
n	マトリックス 1 の列数、マトリックス 2 の行数
p	マトリックス 2 の列数
x_first	マトリックス乗算の順番指定
res_shift	各出力に適用される右シフト

備　考

x_first=1 の場合、x･y を計算します。このとき、ip_x は m×n、ip_y は n×p、op_matrix は m×p となります。
x_first=0 の場合、y･x を計算します。このとき、ip_y は m×n、ip_x は n×p、op_matrix は m×p となります。
積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
各マトリックスは通常の C 様式(行優先順)で配置されます。

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

任意の配列サイズを指定できるようにするために、配列引数は void*で指定します。これらの引数は short 型変数を指すようにしてください。
入力配列 ip_x,ip_y と出力配列 op_matrix は別々に用意してください。

乗算

int VectorMult (short output[ ], const short ip_x[ ], const short ip_y[ ], long no_elements, int res_shift)
int VectorMult (_ _fixed output[ ], const _ _X _ _fixed ip_x[ ], const _ _Y _ _fixed ip_y[ ], long no_elements, int res_shift)

説　明　　ip_x,ip_y から 1 つずつデータを取り出して乗算を行い、結果を output に配置します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 4 ・res_shift < 0 ・res_shift > 16

引　数

output[]	出力
ip_x[]	入力 1
ip_y[]	入力 2
no_elements	データ数
res_shift	各出力に適用される右シフト

備　考　　出力は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。
なお、オーバフローしたときは正または負の最大値となります。
本関数はデータの乗算を行います。内積を計算する場合は m と p を 1 に設定して MatrixMult を使用してください。

平均 2 乗値

*int MsPower (long *output, const short input[ ], long no_elements, int src_is_x)*
*int MsPower (long _ _fixed *output, const _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　入力データの平均 2 乗値を求めます。

ヘッダ　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 6 ・src_is_x ≠ 0 または 1

引　数

output	出力へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

備　考

$$\text{平均2乗値} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

除算結果は最も近い整数値に丸められます。
演算の結果は 63 ビットで保持されます。no_elements が 2^{32} 以上の場合、オーバフローが発生することがあります。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

平均値

*int Mean (short *mean, const short input[ ], long no_elements, int src_is_x)*
*int Mean (_ _fixed *mean, const _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　　入力データの平均値を求めます。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 6 ・src_is_x ≠ 0 または 1

引　数

mean	input の平均 $\overline{x}$ へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

備　考

$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

除算結果は最も近い整数値に丸められます。
演算結果は 32 ビットで保持されます。no_elements が $2^{16}-1$ よりも大きい場合、オーバフローが発生することがあります。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

平均と偏差

*int Variance (long *variance, short *mean, const short input[ ], long no_elements, int src_is_x)*
*int Variance (long _ _fixed *variance, _ _fixed *mean, const _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　　input の平均と偏差を求めます。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 4 ・src_is_x ≠ 0 または 1

引　数

variance	入力の偏差 σ^2 へのポインタ
mean	データの平均 $\overline{x}$ へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

備　考

$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \overline{x}^2$$

除算結果は最も近い整数値に丸められます。
$\overline{x}$ は 32 ビットで保持されます。また、オーバフローのチェックはしません。
no_elements が $2^{16}-1$ よりも大きい場合、オーバフローが発生することがあります。
σ^2 は 63 ビットで保持されます。オーバフローのチェックはしません。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

最大値

*int MaxI (short **max_ptr, short input[ ], long no_elements, int src_is_x)*
*int MaxI (_ _fixed **max_ptr, _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　　配列 input の最大値を検索して、そのアドレスを max_ptr に返します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

引　数

max_ptr	最大データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

備　考　　複数のデータが同じ最大値をもつ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

最小値

*int MinI (short **min_ptr, short input[ ], long no_elements, int src_is_x)*
*int MinI (_ _fixed **min_ptr, _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　　配列 input の最小値を検索して、そのアドレスを min_ptr に返します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

引　数

min_ptr	最小データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

備　考　　複数のデータが同じ最小値をもつ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

最大絶対値

*int PeakI (short **peak_ptr, short input[ ], long no_elements, int src_is_x)*
*int PeakI (_ _fixed **peak_ptr, _ _fixed input[ ], long no_elements, int src_is_x)*

説　明　　配列 input の最大絶対値を検索して、そのアドレスを peak_ptr に返します。

ヘッダ　　<ensigdsp.h>

リターン値

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

引　数

peak_ptr	最大絶対値データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

備　考　　複数のデータが同じ最大絶対値をもつ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときはデータは X メモリに、src_is_x=0 のときはデータは Y メモリに配置してください。

11.　アセンブラ言語仕様

11.1　プログラムの要素

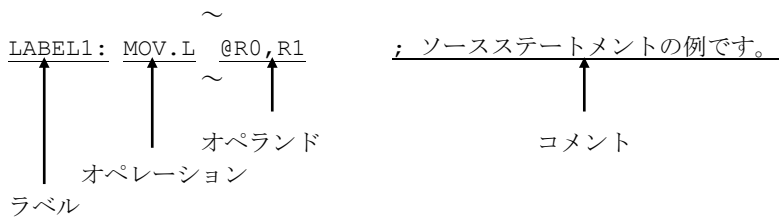
11.1.1　ソースステートメント

(1)　ソースステートメントの構成

ソースステートメントの構成を以下に示します。

[ラベル][△オペレーション[△オペランド]][コメント]

■コーディング例



(a)　ラベル

ソースステートメントにつける名札としてシンボルまたはローカルシンボルを書きます。シンボルとはプログラマが定義する名前です。

(b)　オペレーション

実行命令、DSP 命令、拡張命令、アセンブラ制御命令、各種制御文のニーモニック(名称)を書きます。実行命令、DSP 命令はマイコンの命令です。拡張命令はアセンブラが定義した命令で、実行命令と定数データ(リテラル)または複数の実行命令に展開される命令です。アセンブラ制御命令はアセンブラに指示を与える命令です。制御文には、ファイルインクルード機能、条件つきアセンブル機能、マクロ機能に関するものがあります。

(c)　オペランド

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

(d)　コメント

プログラムを分かりやすくするための注釈を書きます。

(2) ソースステートメントの書き方

ソースステートメントは ASCII 文字で記述します。ただし、文字列またはコメントの中にはかな・漢字(シフト JIS コード、EUC コード)、LATIN コード文字を記述できます。基本的に 1 つのソースステートメントは 1 行に納まるように書いてください。1 行の最大長は 8,192 文字です。

(a) ラベルの書き方

ラベルは次のように書きます。

- 1 カラム目から書き始める。
  または
- ラベル名の直後にコロン(:)をつける。

■良い例

```
LABEL1      ; 1 カラム目から書き始めています。
  LABEL2:   ; コロン(:)で終わっています。
```

■悪い例

```
  LABEL3    ; 1 カラム目から書き始めず、コロン(:)をつけてもいないので、
            ; アセンブラはラベルと見なしません。
```

(b) オペレーションの書き方

オペレーションは次のように書きます。

- ラベルを記述していない場合：2 カラム目以降から書き始める。
- ラベルを記述している場合:ラベルとの間に 1 つ以上の空白またはタブを置いて書き始める。

■コーディング例

```
            ADD         R0,R1       ; ラベルを記述していない場合です。
LABEL1:     ADD         R1,R2       ; ラベルを記述してある場合です。
```

(c) オペランドの書き方

オペランドはオペレーションとの間に 1 つ以上の空白またはタブを置いて書き始めます。

■コーディング例

```
ADD         R0,R1       ; ADD 命令のオペランドは 2 つです。
SHAL        R1          ; SHAL 命令のオペランドは 1 つです。
```

(d) コメントの書き方

セミコロン(;)の後に続けて書きます。アセンブラはセミコロンから行末までをコメントと見なします。

■コーディング例

```
ADD         R0,R1       ; R1 に R0 を加えます。
```

(3) 複数行にわたるソースステートメントの書き方

次のようなときにはプログラムを見やすくするため、1 つのソースステートメントを複数の行に分けて書くことができます。

- ソースステートメントが長くなる場合
- オペランドの 1 つ 1 つにコメントをつけたい場合

複数行にわたるソースステートメントは次の(a)～(c)の手順で記述してください。

(a) オペランドとオペランドを区切るカンマ(,)を切れ目として改行します。
(b) すぐ次の行の1カラム目にプラス(+)を書きます。
(c) そのプラスの後ろにソースステートメントの続きを書きます。

プラスの後ろに空白またはタブを入れても構いません。各行の最後にコメントを書くこともできます。

■コーディング例 1

```
            .DATA.L          H'FFFF0000,
+                            H'FF00FF00,
+                            H'FFFFFFFF
; 1 つのソースステートメントを 3 行にわたって書いた例です。
```

■コーディング例 2

```
            .DATA.L          H'FFFF0000,       ; 初期値 1
+                            H'FF00FF00,       ; 初期値 2
+                            H'FFFFFFFF        ; 初期値 3
; オペランド 1 つ 1 つに、コメントをつけた例です。
```

11.1.2 キーワード

キーワードは特別な意味を持つ語としてアセンブラが用意している名前です。キーワードにはレジスタ名、演算子、ロケーションカウンタがあります。レジスタ名はマイコン種別ごとに異なりますので各マイコンの「**プログラミングマニュアル**」を参照してください。キーワードはシンボルとしては使用できません。

- レジスタ名
  R0～R15、FR0～FR15、DR0～DR14(偶数番号のみ)、XD0～XD14(偶数番号のみ)、FV0～FV12(4の倍数の番号のみ)、R0_BANK～R7_BANK、SP*、SR、GBR、VBR、MACH、MACL、PR、PC、SSR、SPC、FPUL、FPSCR、MOD、RE、RS、DSR、A0、A0G、A1、A1G、M0、M1、X0、X1、Y0、Y1、XMTRX、DBR、SGR、TBR
- 演算子
  STARTOF、SIZEOF、HIGH、LOW、HWORD、LWORD、$EVEN、$ODD、$EVEN2、$ODD2
- ロケーションカウンタ
  $

【注】* R15 と SP は同じレジスタを表します。

11.1.3 シンボル

(1) シンボルの役割

シンボルはプログラマが定義する名前であり、次の役割を果たします。

- アドレスシンボル：データの格納場所、分岐先などのアドレスを表します。
- 定数シンボル　　：定数を表します。
- レジスタ別名　　：汎用レジスタおよび浮動小数点レジスタを表します。
- セクション名　　：セクションの名前を表します。

シンボルの使用例を以下に示します。

■コーディング例 1

```
        ～
      BRA    SUB1                ; BRA は分岐命令です。
        ～                       ; SUB1 は分岐先のアドレスシンボルを表します。
SUB1:
        ～
```

■コーディング例 2

```
        ～
MAX:  .EQU  100                  ; .EQU はシンボルに値を設定するアセンブラ制御命令です。
      MOV.B  #MAX,R0             ; MAX は値 100 を表します。
        ～
```

■コーディング例 3

```
        ～
MIN:  .REG  R0                   ; .REG はレジスタ別名を定義するアセンブラ制御命令です。
      MOV.B  #100,MIN            ; MIN は R0 の別名になります。
        ～
```

■コーディング例 4

```
        ～
      .SECTION  CD,CODE,ALIGN=4
                                 ; .SECTION はセクションを宣言するアセンブラ制御命令です。
                                 ; CD はこのセクションの名前となります。
        ～
```

(2) シンボルの名付け方

(a) シンボルに使用できる文字

次の ASCII 文字を使用できます。

- 英大文字、英小文字(A～Z、a～z)
- 数字(0～9)
- 下線(_)
- ドル($)

アセンブラはシンボル中の英大文字と英小文字を区別します。

(b) 先頭の文字

次のいずれかに限ります。

- 英大文字、英小文字(A～Z、a～z)
- 下線(_)
- ドル($)

【注】 ドル($)1 文字はロケーションカウンタを表すキーワードです。

(c) 最大文字数

とくに制限はありません。

(d) シンボルとして使用できない名前

キーワードはシンボルとして使用できません。また、以下に示すような名前はアセンブラが内部シンボル*として使用します。プログラマが同じ名前を使うことはできません。

_$Omm*mmm* (*m*は数字(0～F)です)
_$$*nnnnn* (*n*は数字(0～9)です)

【注】* 内部シンボルとはアセンブラの内部処理のため必要なシンボルです。内部シンボルはアセンブルリストやオブジェクトモジュールには出力されません。

(e) シンボルの定義と参照

シンボルはラベルとして記述することにより定義されます。参照するときはオペランドに記述します。例外として、.SECTION と.MACRO では定義するシンボルをオペランドに記述します。

また、.MACRO で定義したシンボル(マクロ名)はオペレーションに記述して参照します(マクロコール)。

シンボルを参照する場合、定義よりも参照が先に現れることがあります。このような参照を前方参照と呼びます。通常はこのような参照も問題なく可能ですが、一部で許されないことがありますので注意が必要です。

複数のソースファイルでプログラムを構成する場合、ファイルをまたがるシンボル参照が必要になります。定義したシンボルを他のソースファイルから参照できるようにすることを外部定義と呼びます。逆に、他のソースファイルで定義したシンボルを参照することを外部参照と呼びます。外部定義は.EXPORT、.GLOBAL で宣言します。外部参照は.IMPORT、.GLOBAL で宣言します。外部参照も前方参照と同様、許されないことがありますので注意が必要です。

11.1.4 定数

(1) 整数定数

整数定数は基数をつけて表現します。基数とはその整数定数が何進数であるかを示す記述です。

- 2 進数 ・・・・・・基数 B'と 2 進表現の数値で記述
- 8 進数 ・・・・・・基数 Q'と 8 進表現の数値で記述
- 10 進数・・・・・・基数 D'と 10 進表現の数値で記述
- 16 進数・・・・・・基数 H'と 16 進表現の数値で記述

アセンブラは基数の英大文字と英小文字を区別しません。基数と数値は間を空けずに続けて書いてください。基数は省略しても構いません。基数のない整数定数は(通常は)10 進数として扱われます(.RADIX によって何進数にするかを指定できます)。

■コーディング例

```
.DATA.B B'10001000 ;
.DATA.B Q'210      ; これらのソースステートメントは、全く同じ内容
.DATA.B D'136      ; を表しています。
.DATA.B H'88       ;
```

【補足】

B' ：BINARY(2 進)の意味です。
Q' ：OCTAL(8 進)の意味です。O は数字のゼロと紛らわしいので Q を使います。
D' ：DECIMAL(10 進)の意味です。
H' ：HEXADECIMAL(16 進)の意味です。

(2) 文字定数

文字定数は文字コードを値とする定数です。4 バイト以内の文字をダブルクォーテーション(")で囲んで記述してください。ASCII 文字、シフト JIS コードもしくは EUC コードのかな・漢字、LATIN コード文字を記述することができます。ASCII 文字は H'09(タブ)、H'20(空白)～H'7E(~)が使用できます。ダブルクォーテーション自身をデータとして使う場合は 2 つ続けて書いてください。かな・漢字を記述した場合、シフト JIS コードのときは sjis オプション、EUC コードのときは euc オプションを、LATIN コード文字を記述した場合は、latin1 オプションを指定してください。なお、シフト JIS コード、EUC コード、LATIN コードを混在して使うことはできません。

■コーディング例 1

```
.DATA.L "ABC"      ;.DATA.L H'00414243 と同じ意味です。
.DATA.W "AB"       ;.DATA.W H'4142     と同じ意味です。
.DATA.B "A"        ;.DATA.B H'41       と同じ意味です。
                                       ; A の ASCII コード・・・ H'41
                                       ; B の ASCII コード・・・ H'42
                                       ; C の ASCII コード・・・ H'43
```

■コーディング例 2

```
.DATA.B """"       ;ダブルクォーテーション 1 文字の文字定数です。
.DATA.L "漢字"     ;漢字
```

(3)　浮動小数点定数

浮動小数点定数は浮動小数点定数確保のアセンブラ制御命令で指定することができます。

(a)　浮動小数点定数の書き方

浮動小数点定数の表記には 10 進表記と 16 進表記の 2 種類があります。

- 10 進表記

F' [{±}] { n [. [m]] | .m } [t [[{±}] xx]]

F'　　: 10 進浮動小数点定数であることを示します。省略はできません。

[{±}] { n [. [m]] | .m }　　: n には整数部を 10 進で指定します。m には小数部を 10 進で指定します。整数部と小数部の値は、どちらか一方を省略できます。また、±を省略すると＋を仮定します。

t　　: 精度のコードを指定します。次の 2 種類があります。
・S　単精度
・D　倍精度
省略するとアセンブラ制御命令のオペレーションサイズに従います。

[{±}]xx　　: 指数部(10 のべき乗)の値を 10 進で指定します。
省略すると 0 乗を仮定します。また、±を省略すると＋を仮定します。

■例

F' 0.5S - 2= 0.5×10^{-2} = 0.005 = H' 3BA3D70A
F' .123D3 = 0.123×10^{3} = 123 = H' 405EC00000000000

- 16 進表記

H' xxxx [.t]

H'　　: 16 進であることを示します。省略はできません。

xxxx　　: 浮動小数点定数のビットパターンを 16 進で指定します。データ長より指定が短い場合は右づめに確保し、左側は必要分の 0 をつめます。長い場合は指定の右側の有効長分のデータを確保します。

t　　: 精度のコードを指定します。
次の 2 種類があります。
・S　単精度
・D　倍精度
省略するとアセンブラ制御命令のオペレーションサイズに従います。

この形式は、精度幅の 0 や無限大表示など 10 進表記の書き方では表記しにくいデータを記述するもので、確保すべき浮動小数点定数のビットパターンを直接指定します。

■例

H' 0123456789ABCDEF.S　→　H' 89ABCDEF
H' FFFF.D　→　H' 000000000000FFFF

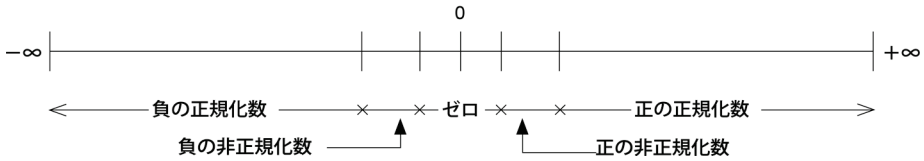
(b) 浮動小数点値の範囲

表 11.1に浮動小数点値のデータタイプを示します。

表 11.1 浮動小数点値のデータタイプ

データタイプ	内容
正規化数 (Normalized Number)	絶対値がアンダフロー境界以上でオーバフロー境界以下の値
非正規化数 (Denormalized Number)	絶対値が 0 より大きくアンダフロー境界未満の値
ゼロ	絶対値が 0 の値
無限大	絶対値がオーバフローより大きい値
非数 (Not a Number：NAN)	数値でない値 sNAN(signaling NAN)と qNAN(quiet NAN)があります

これらを数直線上に表すと次のようになります。ただし、非数は数値として扱わないため数直線上には表せません。



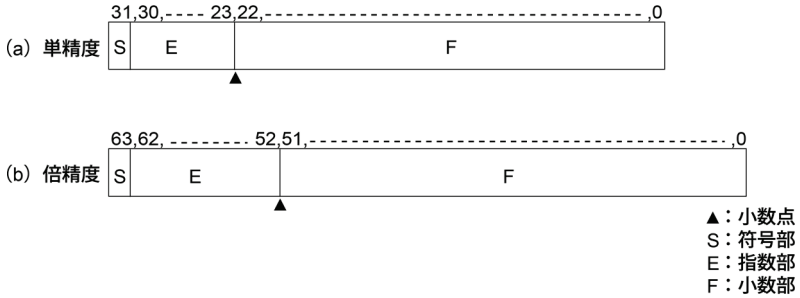
次にアセンブラで記述できる数値範囲を表 11.2に示します。

表 11.2 データ形式と数値範囲(絶対値)

データ形式		単精度	倍精度
正規化数	最大値	3.40×10^{38}	1.79×10^{308}
	最小値	1.18×10^{-38}	2.23×10^{-308}
非正規化数	最大値	1.17×10^{-38}	2.22×10^{-308}
	最小値	1.40×10^{-45}	4.94×10^{-324}

(c) 浮動小数点データ形式

FPU の浮動小数点データ形式を示します。



・符号部 (S)

数値の符号を表現します。0で正数、1で負数です。

・指数部 (E)

指数を表現します。データフォーマット中の指数部の値からある一定のバイアス値(bias)を引いた値が実際の指数になります。

・小数部 (F)

小数部は各ビットごとに重みをもっており、先頭ビットから2^{-1}，2^{-2}，・・・，2^{-n}
(nは小数部のビット長)に対応しています。

次にデータ形式のサイズを**表 11.3**に示します。

表 11.3　データ形式のサイズ

パラメータ	単精度	倍精度
ビット長	32 ビット	64 ビット
符号ビット(S)	1 ビット	1 ビット
指数部(E)	8 ビット	11 ビット
小数部(F)	23 ビット	52 ビット
指数のバイアス(bias)	127	1023

浮動小数点の数値は**表 11.3**の記号を用いると次のように表わせます。

$$2^{E-bias} \cdot (-1)^S \begin{cases} (1.F) & \text{：正規化数} \\ (0.F) & \text{：非正規化数} \end{cases}$$

$(1.F)= 1+b_0\times 2^{-1}+b_1\times 2^{-2}+\cdots+b_{n-1}\times 2^{-n}$　　b：小数部のビット位置
n：小数部のビット長

$(0.F)= b_0\times 2^{-1}+b_1\times 2^{-2}+\cdots+b_{n-1}\times 2^{-n}$

次にデータタイプごとの浮動小数点表現を示すと**表 11.4**のようになります。非数は数値でないため表わせません。

表 11.4　データタイプの浮動小数点表現

データタイプ	単精度	倍精度
正規化数	$(-1)^S \cdot 2^{E-127} \cdot (1.F)$	$(-1)^S \cdot 2^{E-1023} \cdot (1.F)$
非正規化数	$(-1)^S \cdot 2^{-126} \cdot (0.F)$	$(-1)^S \cdot 2^{-1022} \cdot (0.F)$
ゼロ	$(-1)^S \cdot 0$	
無限大	$(-1)^S \cdot \infty$	
非数	quiet NAN，signaling NAN	

(d) 浮動小数点定数の有効数字

浮動小数点定数確保の制御命令において、本アセンブラがオブジェクトコードを生成するとき、以下 2 通りの丸め方をサポートし、有効数字を設定します。

(a) Round to Nearest even(RN) : オブジェクトコード最下位ビットを最近値に丸めます。最近値が2つのとき、最下位ビットがゼロになるように丸めます。
(b) Round to Zero(RZ) : オブジェクトコード最下位ビットを切り捨てて0にします。

■例

.FDATA.S F' 1S-1 のオブジェクトコード
RN の場合：H' 3DCCCCCD
RZ の場合：H' 3DCCCCCC

(e) 非正規化数の扱い

非正規化数の扱いはマイコン種別により異なります。非正規化数を扱わないマイコンの場合、非正規化数の範囲の値はウォーニング 841 とし、0 のオブジェクトコードを出力します。非正規化数を扱うマイコンの場合、非正規化数の範囲の値はウォーニング 842 とし、非正規化数のオブジェクトコードを出力します。非正規化数の扱いはオプションで変更することができます。

■例

- 非正規化数を扱わないマイコンの場合
  .FDATA.S F' 1S-40 : ウォーニング841、オブジェクトコードH' 00000000
- 非正規化数を扱うマイコンの場合
  .FDATA.S F' 1S-40 : ウォーニング842、オブジェクトコードH' 000116C2

(4) 浮動小数点演算(四則演算)の仕様

浮動小数点定数確保の制御命令において、四則演算を記述することができます。

(a) 浮動小数点数 10 進表記方法

- 四則演算による 2 つ目以降の"F'"の省略が可能です。ただし、指数部と区別するため"F'"省略時は必ず小数部を記述してください。

  ■例

  F' 0.5+2.0 : F' 0.5+F' 2.0
  F' 0.5+2 : F' 0.5+2 2 は指数部と解釈
  F' 0.5*2 : エラー
- "F'"の後ろに"("を記述することはできません。

  ■例

  (F' 0.5+0.6)*0.7 : OK
  F' (0.5+0.6)*0.7 : NG
- .FDATA でカンマで区切って複数指定をする場合、カンマの後ろの"F'"は省略できません。

  ■例

  .FDATA.S F' 0.5, 0.6 : NG

【注】浮動小数点数の演算は演算対象が 10 進表記の数値同士に限り、四則演算を行うことが出来ま

す。10 進表記と 16 進表記の演算、および 16 進表記の演算はエラーとなります。
また、演算対象が 10 進表記同士でかつ、精度が異なる場合は、ウォーニングメッセージ(W)826 を出力し、オペレーションサイズで指定した精度を有効にします。

■例

F' 1.5S + F' 0.5S	OK
F' 1.5S + 0.5S	OK："F'"省略可能
F' 1.5S + H' 3BA3D70A	NG：(E)101　SYNTAX ERROR IN SOURCE STATEMENT
H' 3BA3D70A + H' 3BA3D70A	NG：(E)101　SYNTAX ERROR IN SOURCE STATEMENT
F' 1.5S + 0.5D	OK：(W)826　ILLEGAL PRECISION (F' 1.5S + F' 0.5S として演算)

(b)　結果の値の丸め方

浮動小数点の四則演算において、結果が内部表現の小数部の有効数字を超えた場合は、以下の規則に従って丸めを行います。

- 結果の値は、その値を近似する二つの浮動小数点定数の内部表現のうち、近い方に向かって丸めます。
- 結果の値が、その値を近似する二つの浮動小数点定数のちょうど中央になる場合は、小数部の最後の桁が 0 となる方向に丸めます。
- マイコンが SH-2E の場合、有効数字を超える部分を切り捨てます。
- マイコンが SH-4、SH-4A、SH2A-FPU の場合、round = nearest を指定した場合、有効数字を超える部分を四捨五入し、round = zero を指定した場合、有効数字を超える部分を切り捨てます。

(c) オーバフロー、アンダフロー、無効演算時の処理

実行時のオーバフロー、アンダフロー、無効演算に対しては、以下の処理を行います。

- オーバフローの場合は、結果の符号に従って正または負の無限大になります。
- アンダフローの場合は、結果の符号に従って正または負のゼロになります。
- 無効演算は、符号が逆の無限大を加算した場合、符号が同じ無限大を減算した場合に生じます。これらの場合、結果は非数になります。

【注】定数式に関しては、アセンブル時に演算を行います。
この時にオーバフロー、アンダフロー、無効演算を検出した場合は、ウォーニングレベルのエラーになります。

(5) 固定小数点定数

固定小数点定数は固定小数点定数確保のアセンブラ制御命令のオペランドで指定することができます。

(a) 固定小数点定数の書き方

固定小数点定数は-1.0～1.0 までの範囲の実数データが扱え、10 進数で記述します。固定小数点定数にはワードサイズとロングワードサイズの 2 種類があります。

- **ワードサイズ固定小数点定数**

2 バイトの符号付き整数で-1.0～1.0 の範囲の実数を表現します。2 バイトの符号付き整数の値 x (-32,768≦x≦32,767)が表現する実数値は x/32768 となります。

■例

固定小数点定数	ワードデータの表現
−1.0	H'8000
−0.5	H'C000
0.0	H'0000
0.5	H'4000
1.0	H'7FFF

- **ロングワードサイズ固定小数点定数**

4 バイトの符号付き整数で-1.0～1.0 の範囲の実数を表現します。4 バイトの符号付き整数の値 x (-2,147,483,648≦x≦2,147,483,647)が表現する実数値は x/2147483648 となります。

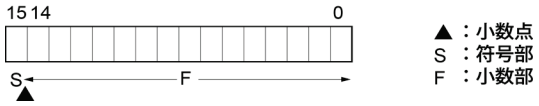
■例

固定小数点定数	ロングワードデータの表現
−1.0	H'80000000
−0.5	H'C0000000
0.0	H'00000000
0.5	H'40000000
1.0	H'7FFFFFFF

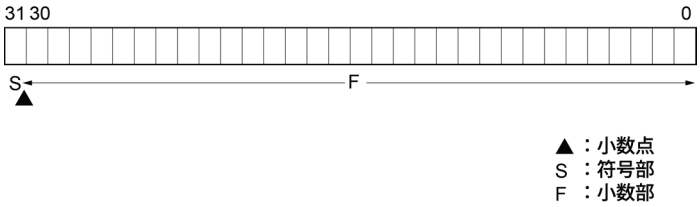
(b)　固定小数点データ形式

固定小数点定数のデータ形式はワードサイズの場合、符号 1 ビットと数値 15 ビット、ロングワードサイズの場合、符号 1 ビットと数値 31 ビットからなるもので、小数点は常に符号部の右側にあるものと考えます。

ワードサイズ



ロングワードサイズ



・符号部 (S)

数値の符号を表します。0で正数、1で負数です。

・小数部 (F)

小数部は各ビットに重みをもっており、先頭ビットから2^{-1},2^{-2},・・・・・,2^{-31}に対応します。

(c)　固定小数点定数の有効数字

ロングワードサイズの場合、31 ビットで表現できる値は 10 進で 9 桁となりますが、アセンブラは有効数字を 10 進 10 桁の小数で扱い、32 ビット目を RN(絶対値が 0 に近い方に丸める)で丸め、計算結果の上位 31 ビットを固定小数点データとします。

【注】　実際の固定小数点データの範囲は-1.0～0.9999999999 ですが、1.0 は 0.9999999999 と仮定し、H'7FFFFFFF とします。

11.1.5 ロケーションカウンタ

ロケーションカウンタはオブジェクトコード(実行命令やデータをコンピュータが理解できる形式に変換したコード)を配置するアドレス(ロケーション)を指し示します。ロケーションカウンタ値はオブジェクトコードの出力に応じて自動的に変化します。また、アセンブラ制御命令により意図的にロケーションカウンタ値を変えることもできます。

■コーディング例

```
                ～
.ORG            H'00001000    ;ロケーションカウンタに H'00001000 を設定しています。
.DATA.W         H'FF          ;このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                              ;ロケーションカウンタ値は H'00001002 に変わります。
.DATA.W         H'F0          ;このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                              ;ロケーションカウンタ値は H'00001004 に変わります。
.DATA.W         H'10          ;このアセンブラ制御命令のオブジェクトコードは長さ 2 バイトです。
                              ;ロケーションカウンタ値は H'00001006 に変わります。
.ALIGN          4             ;ロケーションカウンタ値を 4 の倍数に補正しています。
                              ;ロケーションカウンタ値は H'00001008 に変わります。
.DATA.L         H'FFFFFFFF    ;このアセンブラ制御命令のオブジェクトコードは長さ 4 バイトです。
                              ;ロケーションカウンタ値は H'0000100C に変わります。
                              ;.ORG はロケーションカウンタ値を設定するアセンブラ制御命令です。
                              ;.ALIGN はロケーションカウンタ値を補正するアセンブラ制御命令です。
                              ;.DATA はデータをメモリ上に確保するアセンブラ制御命令です。
                              ;.W はデータをワード (=2 バイト) 単位で扱うとの指定です。
                              ;.L はデータをロングワード (=4 バイト) 単位で扱うとの指定です。
                ～
```

ロケーションカウンタはドル($)で参照できます。

■コーディング例

```
LABEL1：      .EQU $          ;LABEL1 というシンボルにこの時点でのロケーション
                              ;カウンタ値を設定しています。
                              ;.EQU はシンボルに値を設定するアセンブラ制御命令です。
```

11.1.6 式

式は定数やシンボルと演算子を組み合わせて演算結果を求めるものであり、実行命令やアセンブラ制御命令のオペランドに使用します。

(1) 式の要素

式は項、演算子、カッコから構成されます。

(a) 項

項には次のものがあります。

- 定数
- ロケーションカウンタ($)
- シンボル(レジスタ別名を除く)
- 上記の項と演算子による演算結果

単独の項も式の一種です。

(b) 演算子

表 11.5に演算子の一覧を示します。

表 11.5 演算子一覧

演算区分	演算子	演算内容	書き方
算術演算	+	単項プラス	+ 項
	-	単項マイナス	- 項
	+	加算	項1 + 項2
	-	減算	項1 - 項2
	*	乗算	項1 * 項2
	/	除算	項1 / 項2
論理演算	~	単項否定	~ 項
	&	論理積	項1 & 項2
	|	論理和	項1 | 項2
	~	排他的論理和	項1 ~ 項2
シフト演算	<<	算術左シフト	項1 << 項2
	>>	算術右シフト	項1 >> 項2
セクション集合演算	STARTOF	セクション集合の先頭アドレスを求める	STARTOF セクション名
	SIZEOF	セクション集合のサイズをバイト単位で求める	SIZEOF セクション名
偶奇演算	$EVEN	2の倍数の時1、そうでない時0	$EVEN シンボル
	$ODD	2の倍数の時0、そうでない時1	$ODD シンボル
	$EVEN2	4の倍数の時1、そうでない時0	$EVEN2 シンボル
	$ODD2	4の倍数の時0、そうでない時1	$ODD2 シンボル
抽出演算	HIGH	上位バイト抽出	HIGH 項
	LOW	下位バイト抽出	LOW 項
	HWORD	上位ワード抽出	HWORD 項
	LWORD	下位ワード抽出	LWORD 項

(c) カッコ

丸カッコ( )によって演算の優先順位を変更できます。

(2) 演算の順序

1 つの式の中に複数の演算が含まれる場合、演算子の優先順位とカッコ指定によって演算を処理する順序が決まります。アセンブラは次の規則にしたがって演算を処理します。

- 規則 1
  カッコで括られた演算から処理する。
  カッコが多重になっているときはより内側のカッコで括られた演算を優先する。
- 規則 2
  演算子の優先順位が高いものから処理する。
- 規則 3
  演算子の優先順位が同じであるときは演算子の結合規則の向きに処理する。

表 11.6に演算子の優先順位と結合規則を示します。

表 11.6 演算子の優先順位と結合規則

優先順位		演算子	結合規則
1	(高)	+ - ~ STARTOF SIZEOF $EVEN $ODD $EVEN2 $ODD2 HIGH LOW HWORD LWORD *	右から左の順に演算を処理する。
2		* /	左から右の順に演算を処理する。
3		+ -	左から右の順に演算を処理する。
4		<< >>	左から右の順に演算を処理する。
5		&	左から右の順に演算を処理する。
6	(低)	| ~	左から右の順に演算を処理する。

【注】* 優先順位 1 の演算子は単項演算子です。

式の記述例を以下に示します。

■コーディング例 1

1 + (2 - (3 + (4 - 5)))

(a) (b) (c) (d)

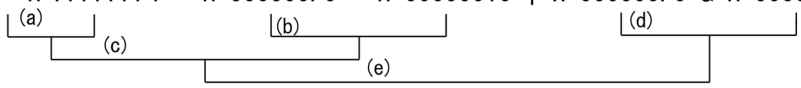
アセンブラは、(a)～(d)の順に計算します。

(a)の結果 …… -1
(b)の結果 …… 2
(c)の結果 …… 0
(d)の結果 …… 1

最終的な結果は、1になります。

■コーディング例 2

```
- H'FFFFFFF1 + H'000000F0 * H'00000010 | H'000000F0 & H'0000FFFF
```



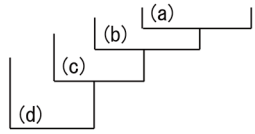
アセンブラは、(a)～(e)の順に計算します。

(a)の結果 …… H'0000000F
(b)の結果 …… H'00000F00
(c)の結果 …… H'00000F0F
(d)の結果 …… H'000000F0
(e)の結果 …… H'00000FFF

最終的な結果は、H'0000FFFになります。

■コーディング例 3

```
-  ~  -  ~  H'0000000F
```



アセンブラは、(a)～(d)の順に計算します。

(a)の結果 …… H'FFFFFFF0
(b)の結果 …… H'00000010
(c)の結果 …… H'FFFFFFEF
(d)の結果 …… H'00000011

最終的な結果は，H'00000011になります。

(3)　演算の詳細

(a)　STARTOF 演算

指定したセクションが最適化リンケージエディタで連結された後のセクション先頭アドレスを求めます。

(b)　SIZEOF 演算

指定したセクションが最適化リンケージエディタで連結された後のセクションサイズを求めます。

■コーディング例

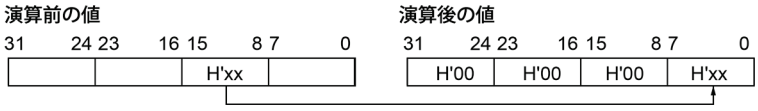
```
            .CPU            SH1
            .SECTION        INIT_RAM,DATA,ALIGN=4
            .RES.B          H'100
            .SECTION        INIT_DATA,DATA,ALIGN=4
INIT_BGN    .DATA.L         (STARTOF INIT_RAM)                      ; [1]
INIT_END    .DATA.L         (STARTOF INIT_RAM) + (SIZEOF INIT_RAM)  ; [2]
;
;
            .SECTION        MAIN,CODE,ALIGN=4
INITIAL:
            MOV.L           DATA1,R6
            MOV             #0,R5
            MOV.L           DATA1+4,R3
            BRA             LOOP2
            MOV.L           @R3,R4
LOOP1:
            MOV.L           R5,@R4
            ADD             #4,R4
LOOP2:
            MOV.L           @R6,R3
            CMP/HI          R3,R4
            BF              LOOP1
            RTS
            NOP
DATA1:
            .DATA.L         INIT_END
            .DATA.L         INIT_BGN
            .END
```

セクション(INIT_RAM)のデータ領域をゼロで初期化します。

[1]：セクション(INIT_RAM)の先頭アドレスを求めます。
[2]：セクション(INIT_RAM)の最終アドレスを求めます。

(c) HIGH 演算

4 バイト値の下位 2 バイトの上位バイトを抽出します。

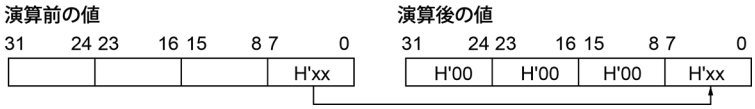


■コーディング例

```
LABEL       .EQU            H'00007FFF
            .DATA           HIGH LABEL      ; メモリ上に整数データ
                                            ; H'0000007F を確保します。
```

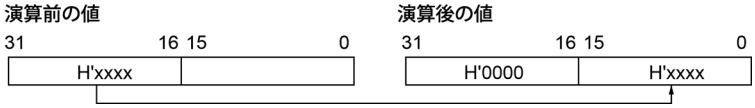
(d)　LOW 演算

4 バイト値の最下位バイトを抽出します。



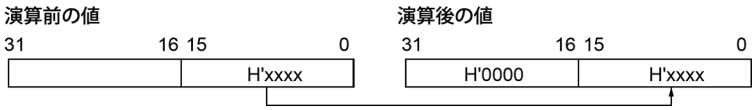
(e)　HWORD 演算

4 バイト値の上位 2 バイトを抽出します。



(f)　LWORD 演算

4 バイト値の下位 2 バイトを抽出します。



(g)　偶奇演算

アドレスシンボルの値が 2 の倍数または 4 の倍数かを判定します。

表 11.7に偶奇演算子の演算内容を示します。

表 11.7　偶奇演算子の演算内容

演算子	演算内容
$EVEN	2 の倍数の時 1、そうでない時 0
$ODD	2 の倍数の時 0、そうでない時 1
$EVEN2	4 の倍数の時 1、そうでない時 0
$ODD2	4 の倍数の時 0、そうでない時 1

■コーディング例

$ODD2 演算子を使用して現在のプログラムカウンタを求める例です。

```
LAB:
            MOVA        @(0,PC),R0
            ADD         #-4+2*$ODD2 LAB,R0      ; $ODD2 は LAB が
                                                ; 4 の倍数ならば 0 とし、
                                                ; 4 の倍数でないなら 1 とします。
```

(4) 式に関する注意事項

(a) 内部処理

アセンブラは式の値を 32 ビット符号付きで処理します。

■コーディング例

```
~H'F0
```

アセンブラは H'F0 を H'000000F0 と解釈します。したがって、~H'F0 は H'FFFFFF0F であり、H'0000000F ではありません。

(b) 算術演算

値が確定しなければならない箇所では乗算、除算で相対アドレスまたは外部参照シンボルを項とすることはできません。また、除算で 0 を除数とすることはできません。

■コーディング例

```
.IMPORT         SYM
.DATA           SYM/10          ; 正常となる
.ORG            SYM/10          ; エラーとなる
```

(c) 論理演算

論理演算で相対アドレスまたは外部参照シンボルを項とすることはできません。

11.1.7 文字列

文字列は一連の文字をデータとして考えます。文字列には次の ASCII 文字を使用できます。

ASCIIコード　H'09(タブ)
　　H'20(空白)～H'7E(~)

文字列中の 1 文字はその文字の ASCII コードを値としてもつバイトサイズのデータを表します。また、シフト JIS コードもしくは EUC コードのかな・漢字、LATIN コード文字も記述することができます。文字としてかな・漢字を記述した場合、シフト JIS コードのときは sjis オプション、EUC コードのときは euc オプションを指定してください。指定しない場合はホストマシンに依存する日本語コードとします。LATIN コード文字を記述した場合、latin1 オプションを指定してください。

文字列は文字の並びをダブルクォーテーション(")で囲んで記述してください。データを表す文字としてダブルクォーテーションを使う場合はダブルクォーテーションを 2 つ続けて書いてください。

■コーディング例

```
        .SDATA          "Hello!"        ; 文字列データ Hello!を確保しています。
        .SDATA          "アセンブラ"      ; 文字列データアセンブラを確保しています。
        .SDATA          """Hello!"""    ; 文字列データ"Hello!"を確保しています。

;       .SDATA は文字列データをメモリ上に確保するアセンブラ制御命令です。
```

【注】文字定数と文字列の違い
文字定数は数値です。データのサイズは 1 バイト、2 バイト、4 バイトのいずれかになります。
文字列は数値として扱えません。データのサイズは 1 バイト以上 255 バイト以下です。

11.1.8 ローカルラベル

(1) ローカルラベルの役割

ローカルラベルはアドレスシンボル間で局所的に有効なラベルです。ローカルラベルは有効範囲外の他のシンボルと衝突することがありませんので、他のシンボル名を意識せずに局所的な制御ができます。ローカルラベルは通常のアドレスシンボルと同様にラベルに記述することによって定義でき、オペランド内で参照できます。

なお、ローカルラベルはデバッグ時に参照できないほか、次の位置に指定できません。

- マクロ名
- セクション名
- オブジェクトモジュール名
- .ASSGINA、.ASSIGNC、.EQU、.ASSIGN、.REG、.FREG のラベル
- .EXPORT、.IMPORT、.GLOBAL のオペランド

ローカルラベルの使用例を以下に示します。

■コーディング例

```
LABEL1:                                     ; ローカルブロック 1 の開始
?0001:
            ～
            CMP/EQ          R1,R2
            BT              ?0002           ; ローカルブロック 1 の?0002 に分岐します。
            BRA             ?0001           ; ローカルブロック 1 の?0001 に分岐します。
?0002:
            ～
LABEL2:                                     ; ローカルブロック 2 の開始
?0001:
            ～
            CMP/GE          R1,R2
            BT              ?0002           ; ローカルブロック 2 の?0002 に分岐します。
            BRA             ?0001           ; ローカルブロック 2 の?0001 に分岐します。
?0002:
LABEL3:                                     ; ローカルブロック 3 の開始
```

(2) ローカルラベルの名付け方

(a) 文字の先頭
ローカルラベルは先頭が"?"で始まる文字列です。

(b) ローカルラベルに使用できる文字
ローカルラベルは先頭以外の文字が以下のASCII文字からなる文字列です。
・英大文字、英小文字(A～Z、a～z)
・数字(0～9)
・下線(_)
・ドル($)
アセンブラは英大文字と英小文字を区別しています。

(c) 最大文字数
2文字以上16文字以内です。17文字以上記述するとローカルシンボルとして扱われません。

(3) ローカルラベルの有効範囲

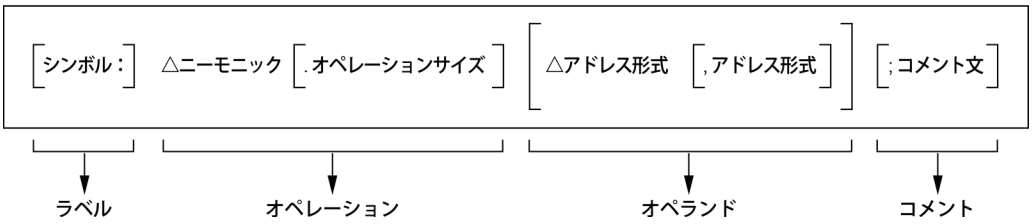
ローカルラベルの有効範囲をローカルブロックといいます。ローカルブロックの区切りはアドレスシンボルまたは.SECTION です。このローカルブロック内で定義されたローカルラベルは当該ローカルブロック内で参照できます。異なるローカルブロックに属するローカルラベルは、同じ名前であっても別のラベルと解釈し、エラーとはなりません。

【注】 .ASSIGNA、ASSIGNC、.EQU、.ASSIGN、.REG、.FREG で定義したアドレスシンボルは有効範囲の区切りとはみなしません。

11.2　実行命令

11.2.1　実行命令の概要

実行命令はマイコンの命令です。
マイコンはメモリ上にある実行命令のオブジェクトコードを解読して実行します。
実行命令のソースステートメントは基本的に次のように記述します。



本節ではニーモニック、オペレーションサイズ、アドレス形式について説明します。

(1)　ニーモニック

ニーモニックは実行命令を表します。処理内容を連想させる英文字の名前がニーモニックとして用意されています。
アセンブラはニーモニック中の英大文字と英小文字を区別しません。

(2)　オペレーションサイズ

オペレーションサイズはデータを操作する単位です。
指定できるオペレーションサイズは実行命令ごとに異なります。
アセンブラはオペレーションサイズの英大文字と英小文字を区別しません。

指定内容	データのサイズ
B	バイト
W	ワード(2 バイト)
L	ロングワード(4 バイト)
S	単精度(4 バイト)
D	倍精度(8 バイト)

(3)　アドレス形式

アクセスの対象となるデータ領域や分岐先アドレスなどを表します。
指定できるアドレス形式は実行命令ごとに異なります。
表 11.8にアドレス形式の一覧を示します。

表 11.8 アドレス形式一覧

アドレス形式	名 称	解 説
Rn	レジスタ直接	レジスタ上の領域です。
@Rn	レジスタ間接	メモリ上の領域です。Rn の値が領域の先頭アドレスを表します。
@Rn+	ポストインクリメントレジスタ間接	メモリ上の領域です。インクリメント*1する前の Rn の値が領域の先頭アドレスを表します。 マイコンは Rn を先に参照し、後からインクリメントします。
@-Rn	プリデクリメントレジスタ間接	メモリ上の領域です。デクリメント*2した後の Rn の値が、領域の先頭アドレスを表します。マイコンは Rn を先にデクリメントし、後から参照します。
@(disp,Rn)	ディスプレースメント付きレジスタ間接*3	メモリ上の領域です。領域の先頭アドレスは Rn の値+ディスプレースメント(disp)です。 Rn の内容は変わりません。
@(R0,Rn)	インデックス付きレジスタ間接	メモリ上の領域です。領域の先頭アドレスは Rn の値+R0 の値です。 Rn および R0 の内容は変わりません。
@(disp,GBR)	ディスプレースメント付き GBR 間接	メモリ上の領域です。領域の先頭アドレスは GBR の値+ディスプレースメント(disp)です。 GBR の内容は変わりません。
@(R0,GBR)	インデックス付き GBR 間接	メモリ上の領域です。領域の先頭アドレスは GBR の値+R0 の値です。 GBR および R0 の内容は変わりません。
@(disp,PC)	ディスプレースメント付き PC 相対	メモリ上の領域です。領域の先頭アドレスは PC の値+ディスプレースメント(disp)です。
@@(disp,TBR)	ディスプレースメント付き TBR 二重間接	メモリ上の領域です。領域の先頭アドレスは TBR の値+ディスプレースメント(disp)の内容です。 TBR の内容は変わりません。
symbol	シンボル指定による PC 相対	【分岐命令のオペランドである場合】 symbol は分岐先のアドレスを表します。アセンブラは symbol と PC の値からディスプレースメント(disp)を逆算します。 disp=symbol-PC です。
		【データ転送命令のオペランドである場合】 メモリ上の領域です。symbol は領域の先頭アドレスを表します。アセンブラは symbol と PC の値からディスプレースメント(disp)を逆算します。disp=symbol-PC です。
		【RS,RE レジスタを指定する命令(LDRS,LDRE)のオペランドである場合】 メモリ上の領域です。symbol は領域の先頭アドレスを表します。アセンブラは symbol と PC の値からディスプレースメント(disp)を逆算します。disp=symbol-PC です。
#imm	イミディエイト	定数を表します。

【注】 *1 インクリメント
オペレーションサイズがバイトのとき 1、ワード(2 バイト)のとき 2、ロングワード(4 バイト)のとき 4 を加えることです。

*2 デクリメント
オペレーションサイズがバイトのとき 1、ワード(2 バイト)のとき 2、ロングワード(4 バイト)のとき 4 を減じることです。

*3 ディスプレースメント
2 点間の距離です。本アセンブリ言語ではバイト単位で表現します。

ディスプレースメントとして許される値は、オペランドのアドレス形式やオペレーションサイズによって異なります。

表 11.9　ディスプレースメントとして許される値

アドレス形式	ディスプレースメント〔単位はバイト( )内は 10 進表現〕
@(disp,Rn)	オペレーションサイズがバイト(B)のとき H'00000000～H'0000000F(0～15) オペレーションサイズがワード(W)のとき H'00000000～H'0000001E(0～30) オペレーションサイズがロングワード(L)のとき H'00000000～H'0000003C(0～60)
@(disp:12,Rn)	オペレーションサイズがバイト(B)のとき H'00000000～H'00000FFF(0～4095) オペレーションサイズがワード(W)のとき H'00000000～H'00001FFE(0～8190) オペレーションサイズがロングワード(L)のとき H'00000000～H'00003FFC(0～16380)
@(disp,GBR)	オペレーションサイズがバイト(B)のとき H'00000000～H'000000FF(0～255) オペレーションサイズがワード(W)のとき H'00000000～H'000001FE(0～510) オペレーションサイズがロングワード(L)のとき H'00000000～H'000003FC(0～1020)
@(disp,PC)	【データ転送命令のオペランドである場合】 オペレーションサイズがワード(W)のとき H'00000000～H'000001FE(0～510) オペレーションサイズがロングワード(L)のとき H'00000000～H'000003FC(0～1020) 【RS、RE レジスタを設定する命令(LDRS,LDRE)のオペランドである場合】 H'FFFFFF00～H'000000FE(-256～254)
@@(disp,TBR)	H'00000000～H'000003FC(0～1020)
symbol	【分岐命令のオペランドである場合】 条件付きの分岐命令(BT,BF,BF/S,BT/S)のオペランドであるとき H'00000000～H'000000FF(0～255) H'FFFFFF00～H'FFFFFFFF(-256～-1) 無条件の分岐命令(BRA、BSR)のオペランドであるとき H'00000000～H'00000FFF(0～4095) H'FFFFF000～H'FFFFFFFF(-4096～-1) 【データ転送命令のオペランドである場合】 オペレーションサイズがワード(W)のとき H'00000000～H'000001FE(0～510) オペレーションサイズがロングワード(L)のとき H'00000000～H'000003FC(0～1020) 【RS、RE レジスタを設定する命令(LDRS,LDRE)のオペランドである場合】 H'FFFFFF00～H'000000FE(-256～254)

イミディエイトとして許される値は実行命令によって異なります。

表 11.10 イミディエイトとして許される値

実行命令	イミディエイト
TST,AND,OR,XOR	H'00000000～H'000000FF(0～255)
MOV	H'00000000～H'000000FF(0～255)
	H'FFFFFF80～H'FFFFFFFF(-128～-1)*1
ADD,CMP/EQ	H'00000000～H'000000FF(0～255)
	H'FFFFFF80～H'FFFFFFFF(-128～-1)*1
TRAPA	H'00000000～H'000000FF(0～255)
SETRC,LDRC	H'00000001～H'000000FF(1～255)*2
MOVI20	H'00000000～H'000FFFFF(0～1048575)
MOVI20S*3	H'00000000～H'0FFFFF00(0～268435200)
ビット操作命令	H'00000000～H'00000007(0～7)

【注】 *1 H'FFFFFF80～H'FFFFFFFF の範囲を正の 10 進数で記述しても構いません。

*2 SETRC,LDRC 命令のイミディエイトデータに 0 を設定した場合、ウォーニング 835 とし、オブジェクトコードには 0 を設定します。この場合、リピートする範囲は 1 回実行されます。
また、SETRC,LDRC 命令のイミディエイト値に外部参照シンボルを指定した場合、最適化リンケージエディタは H'00000000 ～ H'000000FF(0～255)の範囲のチェックとなります。

*3 イミディエイトデータの下位 8 ビットが 0 でない場合、ウォーニング 845 とし、イミディエイトデータの下位 8 ビットを切り捨て、0 に補正します。

【注】アセンブラは条件に応じてディスプレースメントを補正します。

条件		補正内容
オペレーションサイズがワードで ディスプレースメントが 2 の倍数でない	→ →	ディスプレースメントの下位 1 ビット を切り捨て、2 の倍数に補正
オペレーションサイズがロングワードで ディスプレースメントが 4 の倍数でない	→ →	ディスプレースメントの下位 2 ビット を切り捨て、4 の倍数に補正
分岐命令で ディスプレースメントが 2 の倍数でない	→ →	ディスプレースメントの下位 1 ビット を切り捨て、2 の倍数に補正

オペランドとして@(disp,Rn)、@(disp,GBR) 、@@ (disp,TBR)、@(disp,PC)を記述する場合にはアセンブラによるディスプレースメントの補正を考慮してください。

■コーディング例

```
MOV.L  @(63,PC),R0
```

アセンブラはディスプレースメントを 63 から 60 に補正して MOV.L @(60,PC), R0 と同じオブジェクトコードを生成します。また、ウォーニング 870 を通知します。

11.2.2 実行命令に関する注意事項

(1) オペレーションサイズに関する注意

ニーモニックとアドレス形式の組み合わせにより指定できるオペレーションサイズが異なります。

(a) SH-1 の実行命令とオペレーションサイズの組み合わせ

表 11.11 に SH-1 の実行命令とオペレーションサイズの組み合わせを示します。

表 11.11 実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ				
ニーモニック	アドレス形式	B	W	L	省略時	
MOV	#imm,Rn	○	△	△	B	*1
MOV	@(disp,PC),Rn	×	○	○	L	
MOV	symbol,Rn	×	○	○	L	
MOV	Rn,Rm	×	×	○	L	
MOV	Rn,@Rm	○	○	○	L	
MOV	@Rn,Rm	○	○	○	L	
MOV	Rn, @-Rm	○	○	○	L	
MOV	@Rn+, Rm	○	○	○	L	
MOV	R0, @(disp,Rn)	○	○	○	L	
MOV	Rn, @(disp,Rm)	×	×	○	L	*2
MOV	@(disp,Rn), R0	○	○	○	L	
MOV	@(disp,Rn), Rm	×	×	○	L	*3
MOV	Rn, @(R0,Rm)	○	○	○	L	
MOV	@(R0,Rn), Rm	○	○	○	L	
MOV	R0, @(disp,GBR)	○	○	○	L	
MOV	@(disp,GBR), R0	○	○	○	L	
MOVA	#imm, R0	×	×	△	L	
MOVA	@(disp,PC), R0	×	×	○	L	
MOVA	symbol,R0	×	×	○	L	
MOVT	Rn	×	×	○	L	
SWAP	Rn,Rm	○	○	×	W	
XTRCT	Rn,Rm	×	×	○	L	

【注】 *1 サイズ選択モードの場合、アセンブラが imm の値によりオペレーションサイズを決めます。
*2 この場合の Rn は R1～R15
*3 この場合の Rm は R1～R15

表 11.11 実行命令とオペレーションサイズの組み合わせ(その 2)

算術演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
ADD	Rn,Rm	×	×	○	L
ADD	#imm,Rn	×	×	○	L
ADDC	Rn,Rm	×	×	○	L
ADDV	Rn,Rm	×	×	○	L
CMP/EQ	#imm,R0	×	×	○	L
CMP/EQ	Rn,Rm	×	×	○	L
CMP/HS	Rn,Rm	×	×	○	L
CMP/GE	Rn,Rm	×	×	○	L
CMP/HI	Rn,Rm	×	×	○	L
CMP/GT	Rn,Rm	×	×	○	L
CMP/PZ	Rn	×	×	○	L
CMP/PL	Rn	×	×	○	L
CMP/STR	Rn,Rm	×	×	○	L
DIV1	Rn,Rm	×	×	○	L
DIV0S	Rn,Rm	×	×	○	L
DIV0U	(オペランドなし)	×	×	×	―
EXTS	Rn,Rm	○	○	×	W
EXTU	Rn,Rm	○	○	×	W
MAC	@Rn+,@Rm+	×	○	×	W
MULS	Rn,Rm	×	○	○	L *
MULU	Rn,Rm	×	○	○	L *
NEG	Rn,Rm	×	×	○	L
NEGC	Rn,Rm	×	×	○	L
SUB	Rn,Rm	×	×	○	L
SUBC	Rn,Rm	×	×	○	L
SUBV	Rn,Rm	×	×	○	L

【注】*W と L は同じオブジェクトコードとなります。

表 11.11　実行命令とオペレーションサイズの組み合わせ(その 3)

論理演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
AND	Rn,Rm	×	×	○	L
AND	#imm,R0	×	×	○	L
AND	#imm,@(R0,GBR)	○	×	×	B
NOT	Rn,Rm	×	×	○	L
OR	Rn,Rm	×	×	○	L
OR	#imm,R0	×	×	○	L
OR	#imm,@(R0,GBR)	○	×	×	B
TAS	@Rn	○	×	×	B
TST	Rn,Rm	×	×	○	L
TST	#imm,R0	×	×	○	L
TST	#imm,@(R0,GBR)	○	×	×	B
XOR	Rn,Rm	×	×	○	L
XOR	#imm,R0	×	×	○	L
XOR	#imm,@(R0,GBR)	○	×	×	B

表 11.11　実行命令とオペレーションサイズの組み合わせ(その 4)

シフト命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
ROTL	Rn	×	×	○	L
ROTR	Rn	×	×	○	L
ROTCL	Rn	×	×	○	L
ROTCR	Rn	×	×	○	L
SHAL	Rn	×	×	○	L
SHAR	Rn	×	×	○	L
SHLL	Rn	×	×	○	L
SHLR	Rn	×	×	○	L
SHLL2	Rn	×	×	○	L
SHLR2	Rn	×	×	○	L
SHLL8	Rn	×	×	○	L
SHLR8	Rn	×	×	○	L
SHLL16	Rn	×	×	○	L
SHLR16	Rn	×	×	○	L

表 11.11 実行命令とオペレーションサイズの組み合わせ(その 5)

分岐命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
BF	symbol	×	×	×	―
BT	symbol	×	×	×	―
BRA	symbol	×	×	×	―
BSR	symbol	×	×	×	―
JMP	@Rn	×	×	×	―
JSR	@Rn	×	×	×	―
RTS	(オペランドなし)	×	×	×	―

表 11.11 実行命令とオペレーションサイズの組み合わせ(その 6)

システム制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
CLRT	(オペランドなし)	×	×	×	―
CLRMAC	(オペランドなし)	×	×	×	―
LDC	Rn,SR	×	×	○	L
LDC	Rn,GBR	×	×	○	L
LDC	Rn,VBR	×	×	○	L
LDC	@Rn+,SR	×	×	○	L
LDC	@Rn+,GBR	×	×	○	L
LDC	@Rn+,VBR	×	×	○	L
LDS	Rn,MACH	×	×	○	L
LDS	Rn,MACL	×	×	○	L
LDS	Rn,PR	×	×	○	L
LDS	@Rn+,MACH	×	×	○	L
LDS	@Rn+,MACL	×	×	○	L
LDS	@Rn+,PR	×	×	○	L
NOP	(オペランドなし)	×	×	×	―
RTE	(オペランドなし)	×	×	×	―
SETT	(オペランドなし)	×	×	×	―
SLEEP	(オペランドなし)	×	×	×	―
STC	SR,Rn	×	×	○	L
STC	GBR,Rn	×	×	○	L
STC	VBR,Rn	×	×	○	L
STC	SR,@-Rn	×	×	○	L
STC	GBR,@-Rn	×	×	○	L
STC	VBR,@-Rn	×	×	○	L
STS	MACH,Rn	×	×	○	L
STS	MACL,Rn	×	×	○	L
STS	PR,Rn	×	×	○	L
STS	MACH,@-Rn	×	×	○	L
STS	MACL,@-Rn	×	×	○	L
STS	PR,@-Rn	×	×	○	L
TRAPA	#imm	×	×	○	L

【記号の意味】

Rm　：　汎用レジスタ(R0～R15)
Rn　：　汎用レジスタ(R0～R15)
R0　：　汎用レジスタ(R0 固定)
SR　：　ステータスレジスタ
GBR　：　グローバル・ベースレジスタ
VBR　：　ベクタ・ベースレジスタ
MACH　：　積和レジスタ(上位)
MACL　：　積和レジスタ(下位)
PR　：　プロシージャレジスタ
PC　：　プログラムカウンタ
imm　：　イミディエイト
disp　：　ディスプレースメント
symbol　：　シンボル
B　：　バイト
W　：　ワード(2 バイト)
L　：　ロングワード(4 バイト)
○　：　指定は有効
×　：　指定は無効、オペレーションサイズの指定を省略したのと同じ結果になる
△　：　アセンブラは拡張命令として解釈する

(b)　SH-2 の実行命令とオペレーションサイズの組み合わせ

表 11.12 に SH-1 に対して SH-2 で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.12　実行命令とオペレーションサイズの組み合わせ(その 1)

算術演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
MAC	@Rn+, @Rm+	×	○	○	W
MUL	Rn, Rm	×	×	○	L
DMULS	Rn, Rm	×	×	○	L
DMULU	Rn, Rm	×	×	○	L
DT	Rn	×	×	×	―

表 11.12　実行命令とオペレーションサイズの組み合わせ(その 2)

分岐命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
BF/S	symbol	×	×	×	―
BT/S	symbol	×	×	×	―
BRAF	Rn	×	×	×	―
BSRF	Rn	×	×	×	―

(c) SH-2E の実行命令とオペレーションサイズの組み合わせ

表 11.13 に SH-2 に対して SH-2E で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.13 実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ				
ニーモニック	アドレス形式	B	W	L	S	省略時
FLDI0	FRn	×	×	×	○	S
FLDI1	FRn	×	×	×	○	S
FMOV	@Rm,FRn	×	×	×	○	S
FMOV	FRn,@Rm	×	×	×	○	S
FMOV	@Rm+,FRn	×	×	×	○	S
FMOV	FRn,@-Rm	×	×	×	○	S
FMOV	@(R0,Rm),FRn	×	×	×	○	S
FMOV	FRm,@(R0,Rm)	×	×	×	○	S
FMOV	FRm,FRn	×	×	×	○	S

表 11.13 実行命令とオペレーションサイズの組み合わせ(その 2)

算術演算命令		オペレーションサイズ				
ニーモニック	アドレス形式	B	W	L	S	省略時
FABS	FRn	×	×	×	○	S
FADD	FRm,FRn	×	×	×	○	S
FCMP/EQ	FRm,FRn	×	×	×	○	S
FCMP/GT	FRm,FRn	×	×	×	○	S
FDIV	FRm,FRn	×	×	×	○	S
FMAC	FR0,FRm,FRn	×	×	×	○	S
FMUL	FRm,FRn	×	×	×	○	S
FNEG	FRn	×	×	×	○	S
FSUB	FRm,FRn	×	×	×	○	S

表 11.13　実行命令とオペレーションサイズの組み合わせ(その 3)

システム制御命令		オペレーションサイズ				
ニーモニック	アドレス形式	B	W	L	S	省略時
FLDS	FRm,FPUL	×	×	×	○	S
FLOAT	FPUL,FRn	×	×	×	○	S
FSTS	FPUL,FRn	×	×	×	○	S
FTRC	FRm,FPUL	×	×	×	○	S
LDS	Rm,FPUL	×	×	○	×	L
LDS	@Rm+,FPUL	×	×	○	×	L
LDS	Rm,FPSCR	×	×	○	×	L
LDS	@Rm+,FPSCR	×	×	○	×	L
STS	FPUL,Rn	×	×	○	×	L
STS	FPUL,@-Rn	×	×	○	×	L
STS	FPSCR,Rn	×	×	○	×	L
STS	FPSCR,@-Rn	×	×	○	×	L

【記号の意味】

FRm　：　浮動小数点レジスタ
FRn　：　浮動小数点レジスタ
FR0　：　浮動小数点レジスタ(FR0 固定)
FPUL　：　浮動小数点通信レジスタ
FPSCR　：　浮動小数点ステータス/コントロールレジスタ
S　：　単精度(4 バイト)

(d)　SH-3 の実行命令とオペレーションサイズの組み合わせ

表 11.14 に SH-2 に対して SH-3 で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.14　実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
PREF	@Rn	×	×	×	―

表 11.14　実行命令とオペレーションサイズの組み合わせ(その 2)

シフト演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
SHAD	Rn, Rm	×	×	○	L
SHLD	Rn, Rm	×	×	○	L

表 11.14 実行命令とオペレーションサイズの組み合わせ(その 3)

システム制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
CLRS	(オペランドなし)	×	×	×	―
SETS	(オペランドなし)	×	×	×	―
LDC	Rm, SSR	×	×	○	L
LDC	Rm, SPC	×	×	○	L
LDC	Rm, Rn_BANK	×	×	○	L
LDC	@Rm+, SSR	×	×	○	L
LDC	@Rm+, SPC	×	×	○	L
LDC	@Rm+, Rn_BANK	×	×	○	L
STC	SSR, Rn	×	×	○	L
STC	SPC, Rn	×	×	○	L
STC	Rm_BANK, Rn	×	×	○	L
STC	SSR, @-Rn	×	×	○	L
STC	SPC, @-Rm	×	×	○	L
STC	Rm_BANK, @-Rn	×	×	○	L
LDTLB	(オペランドなし)	×	×	×	―

【記号の意味】

Rn_BANK : バンク汎用レジスタ
SSR : 退避ステータスレジスタ
SPC : 退避プログラムカウンタ

(e)　SH-4 の実行命令とオペレーションサイズの組み合わせ

表 11.15 に SH-3 に対して SH-4 で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.15　実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FLDI0	FRn	×	×	×	○	×	S
FLDI1	FRn	×	×	×	○	×	S
FMOV	FRm,FRn	×	×	×	○	×	S
FMOV	FRn,@Rm	×	×	×	○	×	S
FMOV	FRn,@-Rm	×	×	×	○	×	S
FMOV	FRn,@(R0,Rm)	×	×	×	○	×	S
FMOV	@Rm,FRn	×	×	×	○	×	S
FMOV	@Rm+,FRn	×	×	×	○	×	S
FMOV	@(R0,Rm),FRn	×	×	×	○	×	S
FMOV	DRm,DRn	×	×	×	×	○	D
FMOV	DRm,@Rn	×	×	×	×	○	D
FMOV	DRm@-Rn	×	×	×	×	○	D
FMOV	DRm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,DRn	×	×	×	×	○	D
FMOV	@Rm+,DRn	×	×	×	×	○	D
FMOV	@(R0,Rm),DRn	×	×	×	×	○	D
FMOV	DRm,XDn	×	×	×	×	○	D
FMOV	XDm,DRn	×	×	×	×	○	D
FMOV	XDm,XDn	×	×	×	×	○	D
FMOV	XDm,@Rn	×	×	×	×	○	D
FMOV	XDm,@-Rn	×	×	×	×	○	D
FMOV	XDm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,XDn	×	×	×	×	○	D
FMOV	@Rm+,XDn	×	×	×	×	○	D
FMOV	@(R0,Rm),XDn	×	×	×	×	○	D

表 11.15 実行命令とオペレーションサイズの組み合わせ(その 2)

算術演算命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FABS	FRn	×	×	×	○	×	S
FABS	DRn	×	×	×	×	○	D
FADD	FRm,FRn	×	×	×	○	×	S
FADD	DRm,DRn	×	×	×	×	○	D
FCMP/EQ	FRm,FRn	×	×	×	○	×	S
FCMP/EQ	DRm,DRn	×	×	×	×	○	D
FCMP/GT	FRm,FRn	×	×	×	○	×	S
FCMP/GT	DRm,DRn	×	×	×	×	○	D
FDIV	FRm,FRn	×	×	×	○	×	S
FDIV	DRm,DRn	×	×	×	×	○	D
FIPR	FVm,FVn	×	×	×	○	×	S
FMAC	FR0,FRm,FRn	×	×	×	○	×	S
FMUL	FRm,FRn	×	×	×	○	×	S
FMUL	DRm,DRn	×	×	×	×	○	D
FNEG	FRn	×	×	×	○	×	S
FNEG	DRn	×	×	×	×	○	D
FSQRT	FRn	×	×	×	○	×	S
FSQRT	DRn	×	×	×	×	○	D
FSUB	FRm,FRn	×	×	×	○	×	S
FSUB	DRm,DRn	×	×	×	×	○	D
FTRV	XMTRX,FVn	×	×	×	○	×	S

表 11.15　実行命令とオペレーションサイズの組み合わせ(その 3)

システム制御命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FCNVDS	DRm,FPUL	×	×	×	×	○	D
FCNVSD	FPUL,DRn	×	×	×	×	○	D
FLDS	FRm,FPUL	×	×	×	○	×	S
FLOAT	FPUL,FRn	×	×	×	○	×	S
FLOAT	FPUL,DRn	×	×	×	×	○	D
FRCHG	(オペランドなし)	×	×	×	×	×	—
FSCHG	(オペランドなし)	×	×	×	×	×	—
FSTS	FPUL,FRn	×	×	×	○	×	S
FTRC	FRm,FPUL	×	×	×	○	×	S
FTRC	DRm,FPUL	×	×	×	×	○	D
LDC	Rm,DBR	×	×	○	×	×	L
LDC	@Rm+,DBR	×	×	○	×	×	L
LDS	Rm,FPUL	×	×	○	×	×	L
LDS	@Rm+,FPUL	×	×	○	×	×	L
LDS	Rm,FPSCR	×	×	○	×	×	L
LDS	@Rm+,FPSCR	×	×	○	×	×	L
OCBI	@Rn	×	×	×	×	×	—
OCBP	@Rn	×	×	×	×	×	—
OCBWB	@Rn	×	×	×	×	×	—
STC	DBR,Rn	×	×	○	×	×	L
STC	DBR,@-Rn	×	×	○	×	×	L
STC	SGR,Rn	×	×	○	×	×	L
STC	SGR,@-Rn	×	×	○	×	×	L
STS	FPUL,Rm	×	×	○	×	×	L
STS	FPUL,@-Rm	×	×	○	×	×	L
STS	FPSCR,Rm	×	×	○	×	×	L
STS	FPSCR, @-Rm	×	×	○	×	×	L

【記号の意味】

DRm　：　倍精度浮動小数点レジスタ
DRn　：　倍精度浮動小数点レジスタ
XDm　：　倍精度浮動小数点拡張レジスタ
XDn　：　倍精度浮動小数点拡張レジスタ
FVm　：　単精度浮動小数点ベクトルレジスタ
FVn　：　単精度浮動小数点ベクトルレジスタ
XMTRX　：　単精度浮動小数点拡張レジスタ行列
DBR　：　デバッグベクタベースレジスタ
SGR　：　退避ジェネラルレジスタ 15
D　：　倍精度(8 バイト)

(f) SH2-DSP、SH3-DSP の実行命令とオペレーションサイズの組み合わせ

表 11.16 に SH-2 に対して SH2-DSP、SH-3 に対して SH3-DSP で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.16 実行命令とオペレーションサイズの組み合わせ(その 1)

リピート制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
LDRS	@(disp, PC)	×	×	○	L
LDRS	symbol	×	×	○	L
LDRE	@(disp, PC)	×	×	○	L
LDRE	symbol	×	×	○	L
SETRC	Rn	×	×	×	―
SETRC	#imm	×	×	×	―

表 11.16 実行命令とオペレーションサイズの組み合わせ(その 2)

システム制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
LDC	Rn, MOD	×	×	○	L
LDC	Rn, RS	×	×	○	L
LDC	Rn, RE	×	×	○	L
LDC	@Rn+, MOD	×	×	○	L
LDC	@Rn+, RS	×	×	○	L
LDC	@Rn+, RE	×	×	○	L
LDS	Rn, DSR	×	×	○	L
LDS	Rn, A0	×	×	○	L
LDS	@Rn+, DSR	×	×	○	L
LDS	@Rn+, A0	×	×	○	L
STC	MOD, Rn	×	×	○	L
STC	RS, Rn	×	×	○	L
STC	RE, Rn	×	×	○	L
STC	MOD, @-Rn	×	×	○	L
STC	RS, @-Rn	×	×	○	L
STC	RE, @-Rn	×	×	○	L
STS	DSR, Rn	×	×	○	L
STS	A0, Rn	×	×	○	L
STS	DSR, @-Rn	×	×	○	L
STS	A0, @-Rn	×	×	○	L

【記号の意味】

MOD : モジュロレジスタ
RS : 繰り返し開始レジスタ
RE : 繰り返し終了レジスタ
DSR : DSP ステータスレジスタ
A0 : DSP データレジスタ(A0 以外に A1、X0、X1、Y0、Y1 が指定できます)

(g)　SH4AL-DSP の実行命令とオペレーションサイズの組み合わせ

表 11.17 に SH3-DSP に対して SH4AL-DSP で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.17　実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
CLRDMXY	(オペランドなし)	×	×	×	—
MOVCA	R0,@Rn	×	×	○	L
MOVCO	R0,@Rn	×	×	○	L
MOVLI	@Rn,R0	×	×	○	L
MOVUA	@Rn,R0	×	×	○	L
MOVUA	@Rn+,R0	×	×	○	L
PREFI	(オペランドなし)	×	×	×	ー
SETDMX	(オペランドなし)	×	×	×	—
SETDMY	(オペランドなし)	×	×	×	—

表 11.17　実行命令とオペレーションサイズの組み合わせ(その 2)

システム制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
ICBI	@Rn	×	×	○	L
LDC	Rn,DBR	×	×	○	L
LDC	@Rn+,DBR	×	×	○	L
LDC	Rn,SGR	×	×	○	L
LDC	@Rn+,SGR	×	×	○	L
OCBI	@Rn	×	×	×	—
OCBP	@Rn	×	×	×	—
OCBWB	@Rn	×	×	×	—
STC	DBR,Rn	×	×	○	L
STC	DBR,@-Rn	×	×	○	L
STC	SGR,Rn	×	×	○	L
STC	SGR,@-Rn	×	×	○	L
SYNCO	@Rn	×	×	○	L

表 11.17　実行命令とオペレーションサイズの組み合わせ(その 3)

リピート制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
LDRC	Rn	×	×	×	ー

(h) SH-4A の実行命令とオペレーションサイズの組み合わせ

表 11.18 に SH-4 に対して SH-4A で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.18 実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
MOVCO	R0,@Rn	×	×	○	L
MOVLI	@Rn,R0	×	×	○	L
MOVUA	@Rn,R0	×	×	○	L
MOVUA	@Rn+,R0	×	×	○	L
PREFI	(オペランドなし)	×	×	×	―

表 11.18 実行命令とオペレーションサイズの組み合わせ(その 2)

システム制御命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FPCHG	(オペランドなし)	×	×	×	×	×	—
ICBI	@Rn	×	×	×	×	×	—
LDC	Rn,SGR	×	×	○	×	×	L
LDC	@Rn+,SGR	×	×	○	×	×	L
FSCA	FPUL,DRn	×	×	×	○	×	S
FSRRA	FRn	×	×	×	○	×	S
SYNCO	@Rn	×	×	×	×	×	—

(i)　SH-2A の実行命令とオペレーションサイズの組み合わせ

表 11.19 に SH-2 に対して SH-2A で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.19　実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
MOV	R0,@Rn+	○	○	○	L
MOV	@-Rm,R0	○	○	○	L
MOV	Rm,@(disp:12,Rn)	○	○	○	L
MOV	@(disp:12,Rm),Rn	○	○	○	L
MOVI20	#imm20,Rn	×	×	○	L
MOVI20S	#imm20,Rn	×	×	○	L
MOVML	Rm,@-R15	×	×	○	L
MOVML	@R15+,Rn	×	×	○	L
MOVMU	Rm,@-R15	×	×	○	L
MOVMU	@R15+,Rn	×	×	○	L
MOVRT	Rn	×	×	○	L
MOVU	@(disp:12,Rm),Rn	○	○	×	W
NOTT	(オペランドなし)	×	×	×	—
PREF	@Rn	×	×	×	—

表 11.19　実行命令とオペレーションサイズの組み合わせ(その 2)

算術演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
CLIPS	Rn	○	○	×	W
CLIPU	Rn	○	○	×	W
DIVS	R0,Rn	×	×	○	L
DIVU	R0,Rn	×	×	○	L
MULR	R0,Rn	×	×	○	L

表 11.19　実行命令とオペレーションサイズの組み合わせ(その 3)

シフト演算命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
SHAD	Rn, Rm	×	×	○	L
SHLD	Rn, Rm	×	×	○	L

表 11.19　実行命令とオペレーションサイズの組み合わせ(その 4)

分岐命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
JSR/N	@Rn	×	×	×	—
JSR/N	@@(disp,TBR)	×	×	×	—
RTS/N	(オペランドなし)	×	×	×	—
RTV/N	Rn	×	×	×	—

表 11.19 実行命令とオペレーションサイズの組み合わせ(その 5)

システム制御命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
LDBANK	@Rm,R0	×	×	○	L
LDC	Rm,TBR	×	×	○	L
RESBANK	(オペランドなし)	×	×	×	—
STBANK	R0,@Rn	×	×	○	L
STC	TBR,Rn	×	×	○	L

表 11.19 実行命令とオペレーションサイズの組み合わせ(その 6)

ビット操作命令		オペレーションサイズ			
ニーモニック	アドレス形式	B	W	L	省略時
BAND	#imm,@(disp:12,Rn)	○	×	×	B
BANDNOT	#imm,@(disp:12,Rn)	○	×	×	B
BCLR	#imm,@(disp:12,Rn)	○	×	×	B
BCLR	#imm,Rn	×	×	○	L
BLD	#imm,@(disp:12,Rn)	○	×	×	B
BLD	#imm,Rn	×	×	○	L
BLDNOT	#imm,@(disp:12,Rn)	○	×	×	B
BOR	#imm,@(disp:12,Rn)	○	×	×	B
BORNOT	#imm,@(disp:12,Rn)	○	×	×	B
BSET	#imm,@(disp:12,Rn)	○	×	×	B
BSET	#imm,Rn	×	×	○	L
BST	#imm,@(disp:12,Rn)	○	×	×	B
BST	#imm,Rn	×	×	○	L
BXOR	#imm,@(disp:12,Rn)	○	×	×	B

【記号の意味】

TBR : ジャンプテーブルベースレジスタ

(j)　SH2A-FPU の実行命令とオペレーションサイズの組み合わせ

表 11.20 に SH-2E と SH-2A に対して SH2A-FPU で追加された実行命令とオペレーションサイズの組み合わせを示します。

表 11.20　実行命令とオペレーションサイズの組み合わせ(その 1)

データ転送命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FMOV	DRm,DRn	×	×	×	×	○	D
FMOV	DRm,@Rn	×	×	×	×	○	D
FMOV	DRm@-Rn	×	×	×	×	○	D
FMOV	DRm,@(R0,Rn)	×	×	×	×	○	D
FMOV	@Rm,DRn	×	×	×	×	○	D
FMOV	@Rm+,DRn	×	×	×	×	○	D
FMOV	@(R0,Rm),DRn	×	×	×	×	○	D
FMOV	@(disp:12,Rm),FRn	×	×	×	○	×	S
FMOV	@(disp:12,Rm),DRn	×	×	×	×	○	D
FMOV	FRm,@(disp:12,Rn)	×	×	×	○	×	S
FMOV	DRm,@(disp:12,Rn)	×	×	×	×	○	D

表 11.20　実行命令とオペレーションサイズの組み合わせ(その 2)

算術演算命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FABS	DRn	×	×	×	×	○	D
FADD	DRm,DRn	×	×	×	×	○	D
FCMP/EQ	DRm,DRn	×	×	×	×	○	D
FCMP/GT	DRm,DRn	×	×	×	×	○	D
FDIV	DRm,DRn	×	×	×	×	○	D
FMUL	DRm,DRn	×	×	×	×	○	D
FNEG	DRn	×	×	×	×	○	D
FSQRT	FRn	×	×	×	○	×	S
FSQRT	DRn	×	×	×	×	○	D
FSUB	DRm,DRn	×	×	×	×	○	D

表 11.20　実行命令とオペレーションサイズの組み合わせ(その 3)

システム制御命令		オペレーションサイズ					
ニーモニック	アドレス形式	B	W	L	S	D	省略時
FCNVDS	DRm,FPUL	×	×	×	×	○	D
FCNVSD	FPUL,DRn	×	×	×	×	○	D
FLOAT	FPUL,DRn	×	×	×	×	○	D
FSCHG	(オペランドなし)	×	×	×	×	×	—
FTRC	DRm,FPUL	×	×	×	×	○	D

(2) 遅延分岐命令に関する注意

無条件の分岐命令は遅延分岐命令です。

マイコンは遅延分岐命令の実行に先だってディレイスロット命令(メモリ上で遅延分岐命令の直後に位置する命令)を実行します。

ディレイスロット命令が不当なものである場合、アセンブラはエラー150 または 151 を通知します。

遅延分岐命令とディレイスロット命令との関係を表 11.21 に示します。

表 11.21 遅延分岐命令とディレイスロット命令の関係

ディレイスロット命令		遅延分岐命令									
		BF/S	BT/S	BRAF	BSRF	BRA	BSR	JMP	JSR	RTS	RTE
BF		×	×	×	×	×	×	×	×	×	×
BT		×	×	×	×	×	×	×	×	×	×
BF/S		×	×	×	×	×	×	×	×	×	×
BT/S		×	×	×	×	×	×	×	×	×	×
BRAF		×	×	×	×	×	×	×	×	×	×
BSRF		×	×	×	×	×	×	×	×	×	×
BRA		×	×	×	×	×	×	×	×	×	×
BSR		×	×	×	×	×	×	×	×	×	×
JMP		×	×	×	×	×	×	×	×	×	×
JSR		×	×	×	×	×	×	×	×	×	×
JSR/N		×	×	×	×	×	×	×	×	×	×
RTS		×	×	×	×	×	×	×	×	×	×
RTS/N		×	×	×	×	×	×	×	×	×	×
RTE		×	×	×	×	×	×	×	×	×	×
RTV/N		×	×	×	×	×	×	×	×	×	×
TRAPA		×	×	×	×	×	×	×	×	×	×
DIVS		×	×	×	×	×	×	×	×	×	×
DIVU		×	×	×	×	×	×	×	×	×	×
LDC	Rn,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
	@Rn+,SR	*1	*1	*1	*1	*1	*1	*1	*1	*1	*1
MOV	@(disp,PC),Rn	×	×	×	×	×	×	×	×	×	×
	symbol,Rn	×	×	×	×	×	×	×	×	×	×
MOVA	@(disp,PC),R0	×	×	×	×	×	×	×	×	×	×
	symbol,R0	×	×	×	×	×	×	×	×	×	×
LDRS	@(disp,PC)	×	×	×	×	×	×	×	×	×	×
	symbol	×	×	×	×	×	×	×	×	×	×
LDRE	@(disp,PC)	×	×	×	×	×	×	×	×	×	×
	symbol	×	×	×	×	×	×	×	×	×	×
拡張命令	MOV.L #imm,Rn	×	×	×	×	×	×	×	×	×	×
	MOV.W #imm,Rn	×	×	×	×	×	×	×	×	×	×
	MOVA #imm,R0	×	×	×	×	×	×	×	×	×	×
32 ビット命令*2		×	×	×	×	×	×	×	×	×	×
レジスタバンク関連命令*3		×	×	×	×	×	×	×	×	×	×
上記以外の命令		○	○	○	○	○	○	○	○	○	○

【記号の意味】

○　:　正常　アセンブラは指定どおりにオブジェクトコードを生成
×　:　エラー150 または 151
　　ディレイスロット命令が不当
　　アセンブラは NOP 命令のオブジェクトコード(H'0009)を生成
*1　:　マイコン種別が SH-1、SH-2、SH-2E、SH2-DSP、SH-2A、SH2A-FPU のとき正常
　　上記以外のときエラー150 または 151
*2　:　32 ビット命令
　　オペランドに 12 ビットディスプレースメント付きレジスタ間接または、20 ビットイミディエイトを指定する命令
　　MOVI20、MOVI20S 命令
*3　:　レジスタバンク関連命令
　　RESBANK、LDBANK、STBANK 命令

【注】遅延分岐命令とディレイスロット命令が別々のセクションに属する場合、アセンブラはディレイスロット命令の正当性をチェックしません。

(3)　アドレス計算に関する注意

オペランドのアドレス形式がディスプレースメント付き PC 相対@(disp,PC)である場合、PC の値を考慮してプログラミングする必要があります。

PC の値がどうなるかは状況によって異なります。

(a)　通常

PC の値は(実行中の命令の先頭アドレス+4)バイトです。

■例

絶対アドレス H'00001000 の MOV 命令を実行している最中と考えてください。

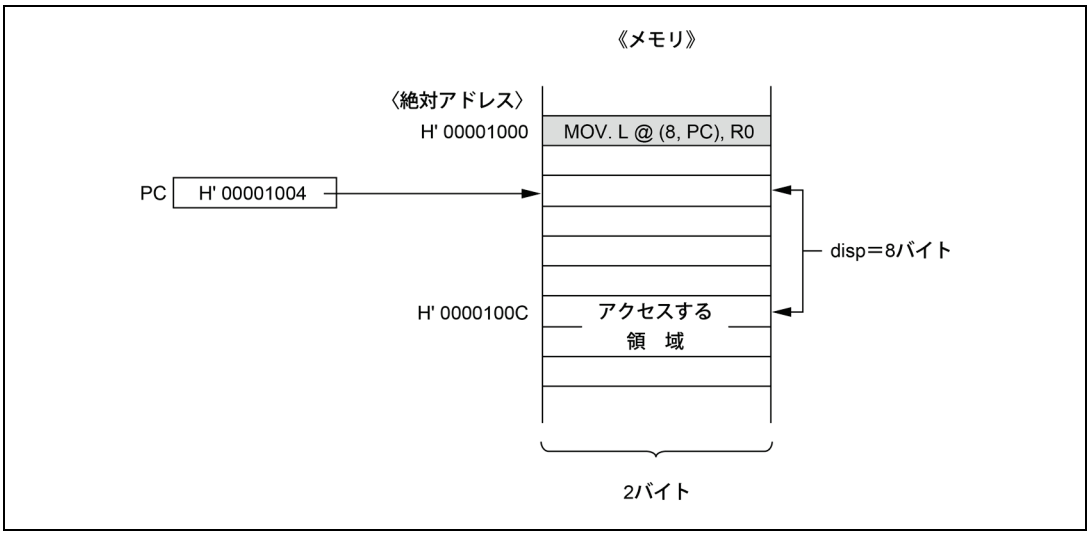


図 11.1　アドレス計算の例(通常)

(b) ディレイスロット命令を実行中

PC の値は(遅延分岐命令による分岐先アドレス+2)バイトとなります。

■例

絶対アドレス H'00001000 の MOV 命令を実行している最中と考えてください。

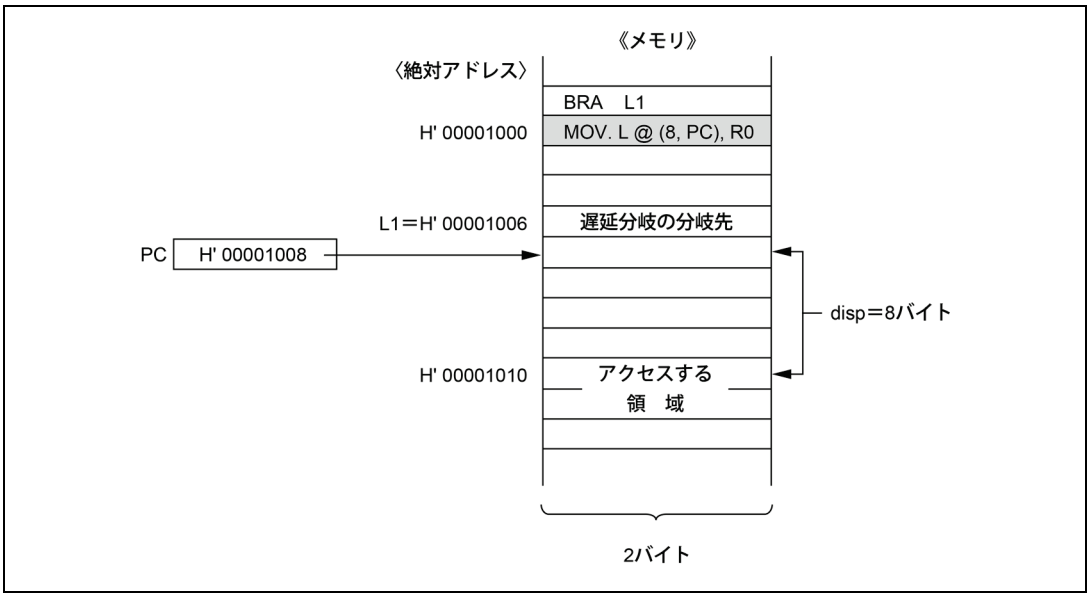


図 11.2 アドレス計算の例(分岐によって PC が変化する場合)

【補足】 オペランドがシンボル指定による PC 相対(symbol)である場合には、アセンブラは PC の値を考慮した上でディスプレースメントを逆算し、オブジェクトコードを生成します。

(c)　MOV.L @(disp,PC),Rn、および MOVA @(disp,PC),R0 のどちらかを実行中

マイコンは PC の値が 4 の倍数でないとき、下位 2 ビットを切り捨てて 4 の倍数に補正し、アドレスを計算します。

■例 1　マイコンが PC を補正する場合

H'00001002 番地の MOV 命令を実行している最中と考えてください。

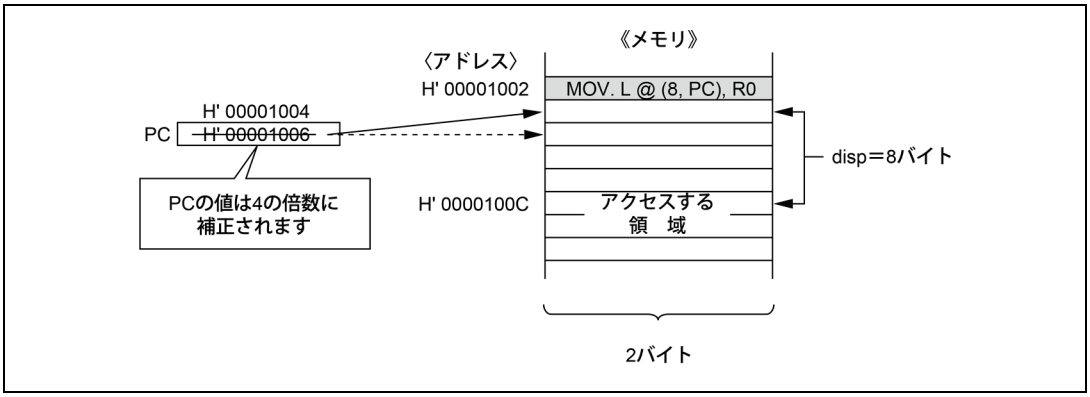


図 11.3　アドレス計算の例(マイコンが PC を補正する場合)

■例 2　マイコンが PC を補正しない場合

H'00001000 番地の MOV 命令を実行している最中と考えてください。

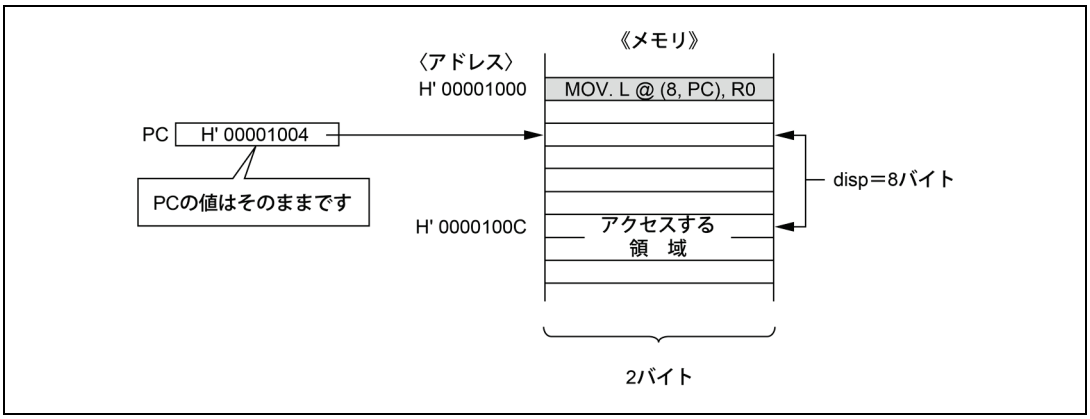


図 11.4　アドレス計算の例(マイコンが PC を補正しない場合)

【補足】　オペランドがシンボル指定による PC 相対(symbol)である場合、アセンブラは PC の値を考慮した上でディスプレースメントを逆算し、オブジェクトコードを生成します。

11.3 DSP 命令

11.3.1 プログラムの要素

(1) ソースステートメント

SH2-DSP、SH3-DSP、SH4AL-DSP の命令は実行命令と DSP 命令に分類できます。DSP 命令は DSP レジスタを操作する命令です。DSP 命令は実行命令とは異なる命令体系を持ち、記述方法も異なります。

DSP 命令では 1 ステートメント内に複数のオペレーションを記述することができます。

DSP 命令のオペレーションには以下の種類があります。

(a) DSP演算オペレーション
DSPレジスタ間の演算を指定するオペレーションで、以下のものがあります。
PABS、PADD、PADDC、PAND、PCLR、PCMP、PCOPY、PDEC、PDMSB、PINC、PLDS、PMULS、PNEG、POR、PRND、PSHA、PSHL、PSTS、PSUB、PSUBC、PSWAP、PXOR

(b) Xデータ転送オペレーション
DSPレジスタとXデータメモリ間のデータ転送を指定するオペレーションで、以下のものがあります。
MOVX、NOPX

(c) Yデータ転送オペレーション
DSPレジスタとYデータメモリ間のデータ転送を指定するオペレーションで、以下のものがあります。
MOVY、NOPY

(d) シングルデータ転送オペレーション
Xデータメモリ、Yデータメモリに限定されない一般のメモリとDSPレジスタ間のデータ転送を指定するオペレーションで、以下のものがあります。
MOVS

(2) 並列演算命令

並列演算命令は、DSP 演算と同時に DSP レジスタと X データメモリ、Y データメモリ間のデータ転送を行なうことを指定します。命令のサイズは 32 ビットになります。

並列演算命令の構成を以下に示します。

[<ラベル>][△<DSP 演算部>][△<データ転送部>] [<コメント>]

(a) DSP 演算部の記述方法

DSP 演算部の構成を以下に示します。

[<コンディション>△]<DSP 演算オペレーション>△<オペランド>[△…]

- **コンディション**

コンディションは並列動作命令を実行する条件を指定します。

コンディションには次のものがあります。

DCT、DCF

DCT は DC ビットが 1 の時に命令を実行することを指定します。
DCF は DC ビットが 0 の時に命令を実行することを指定します。

- DSP 演算オペレーション

DSP 演算を指定します。DSP 演算オペレーションを 2 個指定できるのは PADD と PMULS、PCLR と PMULS、PSUB と PMULS の組み合わせのみです。

(b)　データ転送部の記述方法

データ転送部の構成を以下に示します。

[<X データ転送オペレーション>[△<オペランド>]]
[△<Y データ転送オペレーション>[△<オペランド>]]

X データメモリの転送と Y データメモリの転送をこの順に指定します。
データ転送のオペレーションが NOPX または NOPY の場合は省略することができます。

■コーディング例

```
LABEL1:  PADD  A0,M0,A0  PMULS  X0,Y0,M0  MOVX.W  @R4+,X0  MOVY.W  @R6+,Y0 ; DSP 命令
ラベル    DSP   演算部                      データ転送部                        コメント

            DCT  PINC X1,A1   MOVX.W @R4,X0  MOVY.W @R6+,Y0
            DSP 演算部         データ転送部

            PCMP X1,M0   MOVX.W @R4,X0    ; Y メモリ転送を省略
            DSP 演算部    データ転送部          コメント
```

(3)　データ転送命令

データ転送命令は X データメモリの転送と Y データメモリの転送の組み合わせ、またはシングルデータ転送を指定します。

(a)　X データメモリの転送と Y データメモリの転送の組み合わせ

X データメモリの転送と Y データメモリの転送の組み合わせの構成を以下に示します。

[<ラベル>][△<X データ転送オペレーション>[△<オペランド>]]
[△<Y データ転送オペレーション>[△<オペランド>]] [<コメント>]

X データメモリの転送と Y データメモリの転送をこの順に指定します。データ転送のオペレーションが NOPX または NOPY の場合は省略することができます。ただし、並列演算命令の場合と異なり、X データメモリの転送と Y データメモリの転送の両方を省略することはできません。
X データメモリの転送命令と Y データメモリの転送命令の記述例を以下に示します。

■コーディング例

```
LABEL2:     MOVX.W @R4,X0                    ; データ転送命令(Y データメモリ転送を省略)
            MOVX.W @R4,X0  MOVY.W @R6+,Y0
```

(b) シングルデータ転送命令

シングルデータ転送命令の構成を以下に示します。

[<ラベル>][△<シングルデータ転送オペレーション>△<オペランド>] [<コメント>]

MOVS 命令を指定します。
シングルデータ転送命令の記述例を以下に示します。

■コーディング例

```
LABEL3:  MOVS.W @-R2,A0 ; シングルデータ転送
```

(4) 複数行にわたるソースステートメントの書き方

DSP 命令は 1 行に複数のオペレーションを記述することができるため、ソースステートメントが長くなり、プログラムが見にくくなります。そこで、DSP 命令ではオペランドを区切るカンマ(,)以外にもオペランドとオペレーションの間で区切ることができます。

複数行にわたるソースステートメントは次の(a)～(c)の手順で記述してください。

(a) オペランドとオペレーションを切れ目として改行します。
(b) すぐ次の行の1カラム目にプラス(+)を書きます。
(c) そのプラスの後ろにソースステートメントの続きを書きます。

プラスの後ろに空白またはタブを入れてもかまいません。

■コーディング例

```
            PADD            A0,M0,X0
+           PMULS           A1,Y1,M0
+           MOVX            @R4,X0
+           MOVY            @R6,Y1
```

; 1 つのソースステートメントを 4 行にわたって書いた例です。

11.3.2　DSP 命令

(1)　DSP 演算命令

表 11.22 に DSP 演算命令のニーモニックを示します。

表 11.22　DSP 演算命令ニーモニックの分類

分類	ニーモニック
DSP 算術演算命令	PADD, PSUB, PCOPY, PDMSB, PINC, PNEG, PMULS, PADDC, PSUBC, PCMP, PDEC, PABS, PRND, PCLR, PLDS, PSTS, PSWAP
DSP 論理演算命令	POR, PAND, PXOR
DSP シフト演算命令	PSHA, PSHL

(a)　オペレーションサイズ

DSP 演算命令にオペレーションサイズは指定できません。

(b)　アドレス形式

表 11.23 に DSP 演算命令のアドレス形式を示します。

表 11.23　DSP 演算命令のアドレス形式

アドレス形式	表記法
DSP レジスタ直接	Dp(DSP レジスタ名)
イミディエイトデータ	#imm

- DSP レジスタ直接

表 11.24 に DSP レジスタ直接に指定できるレジスタを示します。
表の中で Sx、Sy、Dz、Du、Se、Sf、Dg は「表 11.26　DSP 演算命令一覧」を参照してください。

表 11.24　DSP レジスタ直接に指定可能なレジスタ

		DSP レジスタ							
		A0	A1	M0	M1	X0	X1	Y0	Y1
Dp	Sx	○	○			○	○		
	Sy			○	○			○	○
	Dz	○	○	○	○	○	○	○	○
	Du	○	○			○		○	
	Se		○			○	○	○	
	Sf		○			○		○	○
	Dg	○	○	○	○				

- **イミディエイトデータ**

イミディエイトデータは PSHA、PSHL 命令の第 1 オペランドにのみ指定できます。指定できる値は以下の通りです。

- 値の種類
  定数、シンボル、または式を指定することができます。
- シンボルの種類
  イミディエイトデータには相対アドレスシンボルや外部参照シンボルを含めて、任意のシンボルが指定できます。
- 値の範囲
  表 11.25に指定できる値の範囲を示します。

表 11.25 イミディエイトデータの値の範囲

命令	数値の範囲
PSHA 命令	H'FFFFFFE0～H'00000020(-32～32)
PSHL 命令	H'FFFFFFF0～H'00000010(-16～16)

(c) 複数の DSP 演算命令の組み合わせ

PADD 命令と PMULS 命令、PCLR 命令と PMULS 命令、または PSUB 命令と PMULS 命令の組み合わせを指定することができます。この命令の組み合わせは本来一つの DSP 演算命令ですが、プログラムを読み易くするために PADD 命令、PCLR 命令または PSUB 命令のオペランドと PMULS 命令のオペランドを分割して記述できるようにしているものです。

複数の DSP 演算命令の組み合わせの例を以下に示します。

■コーディング例

```
PADD A0,M0,A0  PMULS X0,Y0,M0  NOPX          MOVY.W@R6+,Y0
PCLR A1        PMULS X0,Y0,M0  NOPX          MOVY.W@R6+,Y0
PSUB A1,M1,A1  PMULS X1,Y1,M1  MOVX @R4+,X0  NOPY
```

【注】 複数の DSP 演算命令の組み合わせでディスティネーションレジスタとして同一のレジスタを指定した場合、ウォーニング 701 になります。

```
PADD  A0,M0,A0  PMULS  X0,Y0,A0  →  ウォーニング 701
```

(d) コンディション付き DSP 演算命令

DSP 演算命令には DSR レジスタの DC ビットの状態により実行するかどうかを選択できるものがあります。コンディション DCT は DC ビットが 1 のときだけ命令を実行します。コンディション DCF は DC ビットが 0 のときだけ命令を実行します。

コンディションを付けられる DSP 演算命令には次のものがあります。

PABS、PADD、PAND、PCLR、PCOPY、PDEC、PDMSB、PINC、PLDS、PNEG、POR、PRND、PSHA、PSHL、PSTS、PSUB、PSWAP、PXOR

(e) DSP 演算命令一覧

表 11.26 に SH2-DSP、SH3-DSP の DSP 演算命令の一覧、また、表 11.27 に SH2-DSP、SH3-DSP に対して SH4AL-DSP で追加された DSP 演算命令の一覧を示します。表中の Sx、Sy、Dz、Du、Se、Sf、Dg のところに指定できるレジスタは表 11.24 を参照してください。

表 11.26 DSP 演算命令一覧

ニーモニック	アドレス形式	ニーモニック	アドレス形式
PABS	Sx,Dz	—	—
PABS	Sy,Dz	—	—
PADD	Sx,Sy,Dz	—	—
PADD	Sx,Sy,Du	PMULS	Se,Sf,Dg
PADDC	Sx,Sy,Dz	—	—
PAND	Sx,Sy,Dz	—	—
PCLR	Dz	—	—
PCMP	Sx,Sy	—	—
PCOPY	Sx,Dz	—	—
PCOPY	Sy,Dz	—	—
PDEC	Sx,Dz	—	—
PDEC	Sy,Dz	—	—
PDMSB	Sx,Dz	—	—
PDMSB	Sy,Dz	—	—
PINC	Sx,Dz	—	—
PINC	Sy,Dz	—	—
PLDS	Dz,MACH	—	—
PLDS	Dz,MACL	—	—
PMULS	Se,Sf,Dg	—	—
PNEG	Sx,Dz	—	—
PNEG	Sy,Dz	—	—
POR	Sx,Sy,Dz	—	—
PRND	Sx,Dz	—	—
PRND	Sy,Dz	—	—
PSHA	#imm,Dz	—	—
PSHA	Sx,Sy,Dz	—	—
PSHL	#imm,Dz	—	—
PSHL	Sx,Sy,Dz	—	—
PSTS	MACH,Dz	—	—
PSTS	MACL,Dz	—	—
PSUB	Sx,Sy,Dz	—	—
PSUB	Sx,Sy,Du	PMULS	Se,Sf,Dg
PSUBC	Sx,Sy,Dz	—	—
PXOR	Sx,Sy,Dz	—	—

表 11.27 DSP 演算命令一覧

ニーモニック	アドレス形式	ニーモニック	アドレス形式
PCLR	Du	PMULS	Se,Sf,Dg
PSUB	Sy,Sx,Dz	—	—
PSWAP	Sx,Dz	—	—
PSWAP	Sy,Dz	—	—

(2) データ転送命令

(a) ニーモニック

データ転送命令にはデュアルメモリ転送とシングルメモリ転送があります。デュアルメモリ転送は X メモリと DSP レジスタ間、Y メモリと DSP レジスタ間のデータ転送を同時に行います。シングルメモリ転送では任意のメモリと DSP レジスタ間のデータ転送を行います。

表 11.28 にデータ転送命令のニーモニック一覧を示します。

表 11.28 データ転送命令のニーモニック一覧

分類		ニーモニック
デュアルメモリ転送	X メモリ転送	NOPX
		MOVX
	Y メモリ転送	NOPY
		MOVY
シングルメモリ転送		MOVS

(b) オペレーションサイズ

NOPX、NOPY 命令はサイズ指定できません。

MOVX、MOVY 命令はワードサイズ(W) とロングワードサイズ(L)の指定ができます。サイズを指定しないとワードサイズと解釈します。

MOVS 命令はワードサイズ(W)とロングワードサイズ(L)の両方が指定できます。サイズを指定しないとロングワードサイズと解釈します。

(c) アドレス形式

表 11.29 にデータ転送命令で指定可能なアドレス形式を示します。

表 11.29 データ転送命令のアドレス形式

アドレス形式	表記法
DSP レジスタ直接	Dz
レジスタ間接	@Az
ポストインクリメントレジスタ間接	@Az+
インデックス付きレジスタ間接/ポストインクリメント	@Az+Iz
プリデクリメントレジスタ間接	@-Az

アドレス形式のうち、インデックス付きレジスタ間接/ポストインクリメントは DSP のデータ転送命令に特有なアドレス形式です。この形式はレジスタ Az の指す内容を参照した後、Az の内容を Iz の値だけインクリメントすることを示します。

(d)　アドレス形式に指定可能なレジスタ

表 11.30 に DSP レジスタ直接、レジスタ間接、ポストインクリメントレジスタ間接、インデックス付きレジスタ間接/ポストインクリメント、プリデクリメント間接に指定可能なレジスタを示します。

表の中で Dx、Dy、Dxy、Ds、Da、Ax、Ax2、Ay、Ay2、As、Ix、Iy、Is については表 11.31 を参照してください。

表 11.30　データ転送命令のアドレス形式に指定可能なレジスタ

		汎用レジスタ										DSP レジスタ									
		R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	A0	A1	M0	M1	X0	X1	Y0	Y1	A0G	A1G
Dz	Dx															○	○				
	Dy																	○	○		
	Dxy															○	○	○	○		
	Ds											○	○	○	○	○	○	○	○	○	○
	Da											○	○								
Az	Ax					○	○														
	Ax2	○	○			○	○														
	Ay							○	○												
	Ay2			○	○			○	○												
	As			○	○	○	○														
Iz	Ix									○											
	Iy										○										
	Is									○											

【注】同一命令内の DSP 演算命令とデータ転送命令の間で同一のレジスタをディスティネーションに指定した場合はウォーニング 703 になります。

```
PADD  A0,M0,Y0    NOPX    MOVY.W  @R6+,Y0  →ウォーニング 703
```

(e) データ転送命令一覧

表 11.31 に SH2-DSP、SH3-DSP のデータ転送命令の一覧、また、表 11.32 に SH2-DSP、SH3-DSP に対して SH4AL-DSP で追加されたデータ転送命令の一覧を示します。

表中の Dx、Dy、Dxy、Ds、Da、Ax、Ax2、Ay、Ay2、As、Ix、Iy、Is のところに指定できるレジスタは表 11.30 を参照してください。

表 11.31 データ転送命令一覧

種別	ニーモニック	アドレス形式
X データ転送命令	NOPX	―
	MOVX.W	@Ax,Dx
	MOVX.W	@Ax+,Dx
	MOVX.W	@Ax+Ix,Dx
	MOVX.W	Da,@Ax
	MOVX.W	Da,@Ax+
	MOVX.W	Da,@Ax+Ix
Y データ転送命令	NOPY	―
	MOVY.W	@Ay,Dy
	MOVY.W	@Ay+,Dy
	MOVY.W	@Ay+Iy,Dy
	MOVY.W	Da,@Ay
	MOVY.W	Da,@Ay+
	MOVY.W	Da,@Ay+Iy
シングルデータ転送命令	MOVS.W	@-As,Ds
	MOVS.W	@As,Ds
	MOVS.W	@As+,Ds
	MOVS.W	@As+Is,Ds
	MOVS.W	Ds,@-As
	MOVS.W	Ds,@As
	MOVS.W	Ds,@As+
	MOVS.W	Ds,@As+Is
	MOVS.L	@-As,Ds
	MOVS.L	@As,Ds
	MOVS.L	@As+,Ds
	MOVS.L	@As+Is,Ds
	MOVS.L	Ds,@-As
	MOVS.L	Ds,@As
	MOVS.L	Ds,@As+
	MOVS.L	Ds,@As+Is

表 11.32　データ転送命令一覧

種別	ニーモニック	アドレス形式
X データ転送命令	MOVX.W	@Ax2,Dxy
	MOVX.W	@Ax2+,Dxy
	MOVX.W	@Ax2+R8,Dxy
	MOVX. L	@Ax2,Dxy
	MOVX. L	@Ax2+,Dxy
	MOVX. L	@Ax2+R8,Dxy
Y データ転送命令	MOVY.W	@Ay2,Dxy
	MOVY.W	@Ay2+,Dxy
	MOVY.W	@Ay2+R9,Dxy
	MOVY. L	@Ay2,Dxy
	MOVY. L	@Ay2+,Dxy
	MOVY. L	@Ay2+R9,Dxy

11.4 アセンブラ制御命令

アセンブラ制御命令はアセンブラが解釈、実行する命令です。
書式の下線は、省略時の解釈を示します。
表 11.33 にアセンブラ制御命令の一覧を示します。

表 11.33 アセンブラ制御命令一覧

分類	ニーモニック	機能
マイコンに関するもの	.CPU	マイコン種別を指定する。
セクションまたはロケーションカウンタに関するもの	.SECTION	セクションを宣言する。
	.ORG	ロケーションカウンタ値を設定する。
	.ALIGN	ロケーションカウンタ値をアライメント数の倍数に補正する。
シンボルに関するもの	.EQU	シンボルに値を設定する。
	.ASSIGN	シンボルに値を設定または再設定する。
	.REG	レジスタ別名を定義する。
	.FREG	浮動小数点レジスタ名を定義する。
データまたはデータ領域を確保するもの	.DATA	整数データを確保する。
	.DATAB	整数データブロックを確保する。
	.SDATA	文字列データを確保する。
	.SDATAB	文字列データブロックを確保する。
	.SDATAC	計数付き文字列データを確保する。
	.SDATAZ	ゼロ終端文字列データを確保する。
	.FDATA	浮動小数点データを確保する。
	.FDATAB	浮動小数点データブロックを確保する。
	.XDATA	固定小数点データを確保する。
	.RES	データ領域を確保する。
	.SRES	文字列データ領域を確保する。
	.SRESC	計数付き文字列データ領域を確保する。
	.SRESZ	ゼロ終端文字列データ領域を確保する。
	.FRES	浮動小数点データ領域を確保する。
外部定義または外部参照に関するもの	.EXPORT	外部定義シンボルを宣言する。
	.IMPORT	外部参照シンボルを宣言する。
	.GLOBAL	外部参照シンボルまたは外部定義シンボルを宣言する。
オブジェクトモジュールに関するもの	.OUTPUT	オブジェクトモジュール、デバッグ情報の出力を制御する。
	.DEBUG	シンボルデバッグ情報の部分出力を制御する。
	.ENDIAN	エンディアン種別を指定する。
	.LINE	行番号を変更する。
アセンブルリストに関するもの	.PRINT	アセンブルリストの出力を制御する。
	.LIST	ソースプログラムリストの部分出力を制御する。
	.FORM	アセンブルリストの行数と桁数を設定する。
	.HEADING	ソースプログラムリストのヘッダを設定する。
	.PAGE	ソースプログラムリストを改ページする。
	.SPACE	ソースプログラムリストに空行を出力する。
その他	.PROGRAM	オブジェクトモジュール名を設定する。
	.RADIX	基数のない整数定数を何進数とするかを指定する。
	.END	ソースプログラムの終わりとエントリポイントを指定する。
	.STACK	指定したシンボルに対してスタック値を定義する。

【注】 コンパイラが出力するアセンブリソースには、.FILE が記述されます。.FILE はコンパイラ出力アセンブリソースにおいてのみ有効です。ユーザ作成アセンブリプログラムでは使用しないでください。

マイコン種別

.CPU

書　式　　△.CPU△<マイコン種別>

マイコン種別	対象マイコン
SH1	SH-1 用にアセンブルする。
SH2	SH-2 用にアセンブルする。
SH2E	SH-2E 用にアセンブルする。
SH2A	SH-2A 用にアセンブルする。
SH2AFPU	SH2A-FPU 用にアセンブルする。
SHDSP	SH2-DSP 用にアセンブルする。
SH3	SH-3 用にアセンブルする。
SH3DSP	SH3-DSP 用にアセンブルする。
SH4	SH-4 用にアセンブルする。
SH4A	SH-4A 用にアセンブルする。
SH4ALDSP	SH4AL-DSP 用にアセンブルする。

ラベルは記述できません。

説　明　　アセンブルするソースプログラムの対象とするマイコン種別を指定します。
.CPU はソースプログラムの最初に記述してください。アセンブルリストに関する制御命令を除いて、ソースプログラムの最初にない場合はエラーとなります。
.CPU の指定は 1 回限り有効です。
マイコン種別の優先順位は cpu オプション、.CPU、SHCPU 環境変数の順となります。
指定を省略した場合、SHCPU 環境変数で設定したマイコン種別が有効となります。

例

```
.CPU  SH2   ; SH-2 用にアセンブルします。
.SECTION  A,CODE,ALIGN=4
MOV.L      R0,R1
MOV.L      R0,R2
```

セクション宣言

.SECTION

書　式

```
△.SECTION△<セクション名>[,<セクション属性>[,<形式種別>]]
<セクション属性>  ={    CODE      |
                        DATA      |
                        STACK     |
                        DUMMY     }
<形式種別>        ={    LOCATE    = <先頭アドレス>      |
                        ALIGN     = <アライメント数>    }
```

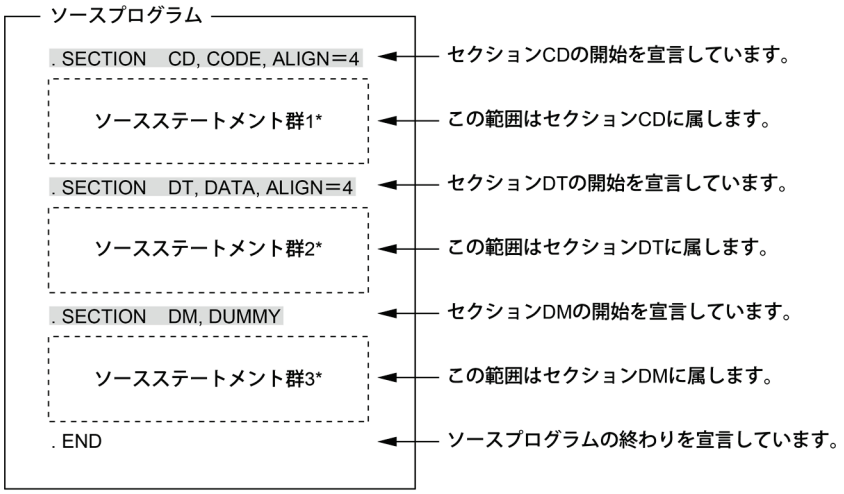
ラベルは記述できません。

説　明

セクションの宣言、再開を指定します。
セクションとはプログラムの１区切りであり、リンケージの処理単位です。
セクション属性は以下のようになります。

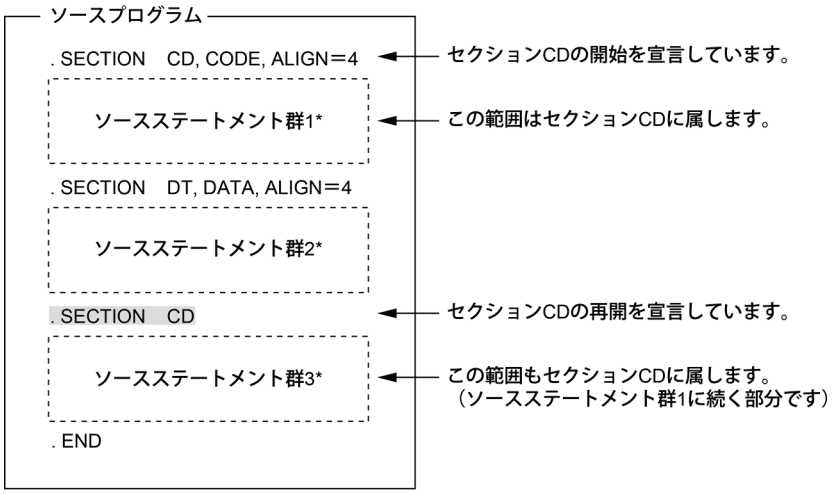
- CODE　…… コードセクション
- DATA　…… データセクション
- STACK　…… スタックセクション
- DUMMY　…… ダミーセクション

LOCATE=<先頭アドレス>を指定した場合、絶対アドレス形式でオブジェクトを出力します。
ALIGN=<アライメント数>を指定した場合、相対アドレス形式でオブジェクトを出力します。
最適化リンケージエディタは、そのセクションの先頭がアライメント数の倍数にあたる絶対アドレスにくるように調整します。
形式種別の指定がない場合、ALIGN=4 が設定されます。
セクションの宣言について簡単な例をあげて説明します。



【注】＊　この例では、「ソースステートメント群1～3」に
. SECTIONが現れないことを仮定しています。

すでに宣言してあるセクションを同じファイルの中で再び宣言し、再開できます。
セクションの再開について、簡単な例をあげて説明します。



【注】*　この例では、「ソースステートメント群1～3」に
. SECTIONが現れないことを仮定しています。

セクションを再開する場合、第 2 オペランドと第 3 オペランドの指定を省略してください(そのセクションを開始したときの指定がそのまま有効です)。
絶対アドレスセクションを開始するときには第 3 オペランドに「LOCATE=先頭アドレス」を指定してください。先頭アドレスはそのセクションが始まる絶対アドレスです。
先頭アドレスは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
先頭アドレスとして許される値は H'00000000～H'FFFFFFFF です。
(10 進表現では 0～4,294,967,295)
相対アドレスセクションを開始するときには第 3 オペランドに「ALIGN=アライメント数」を指定してください。最適化リンケージエディタはそのセクションの先頭がアライメント数の倍数にあたる絶対アドレスにくるように調整します。
アライメント数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
アライメント数として許される値は 2 のべき乗(2^0, 2^1, 2^2, … , 2^{31})です。
コードセクションの場合は 4 以上の値(2^2, 2^3, 2^4, … , 2^{31})です。
次のいずれかの場合にアセンブラはデフォルトセクションを用意します。
・セクションを宣言しないうちに実行命令、拡張命令、DSP 命令を記述している。
・セクションを宣言しないうちにデータを確保するアセンブラ制御命令を記述している。
・セクションを宣言しないうちに.ALIGN を記述している。
・セクションを宣言しないうちに.ORG を記述している。
・セクションを宣言しないうちにロケーションカウンタを参照している。
・セクションを宣言しないうちにラベルだけの行を記述している。

デフォルトセクションとは次のようなセクションです。
・セクション名：P
・セクションの種類：コードセクション、相対アドレスセクション(アライメント数=4)

例

```
        .ALIGN     4              ; この範囲は、デフォルトセクションPに属します。
        .DATA.L    H'11111111     ; デフォルトセクションPはコードセクションであり、
               ～                 ; アライメント数=4の相対アドレスセクションです。

    .SECTION CD,CODE,ALIGN=4
        MOV        R0,R1          ; この範囲は、セクションCDに属します。
        MOV        R0,R2          ; セクションCDはコードセクションであり、
               ～                 ; アライメント数=4の相対アドレスセクションです。

    .SECTION DT,DATA,LOCATE=H'00001000
 X1:    .DATA.L    H'22222222     ; この範囲は、セクションDTに属します。
        .DATA.L    H'33333333     ; セクションDTはデータセクションであり、
               ～                 ; 先頭アドレス=H'00001000の絶対アドレス
                                  ; セクションです。
```

この例では「～」の部分に.SECTION が現れないことを仮定しています。

ロケーションカウンタ値設定

.ORG

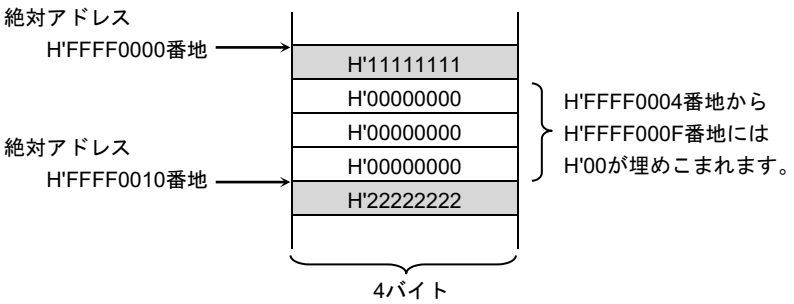
書　式　　△.ORG△<ロケーションカウンタ値>
ラベルは記述できません。

説　明　　セクション内のロケーションカウンタ値を指定した値に設定します。
.ORG によって実行命令やデータを特定のアドレスに配置できます。
ロケーションカウンタ値は次のように指定します。
・定数値またはセクション内のアドレスを指定する。
　かつ
・前方参照シンボルを使わずに指定する。
ロケーションカウンタ値として許される値は H'00000000～H'FFFFFFFF です。
(10 進表現では 0～4,294,967,295)
絶対アドレスセクションで指定する場合は、
　ロケーションカウンタ値 ≧ セクション先頭アドレス　(符号なしの値として比較)
となるようにしてください。
アセンブラはロケーションカウンタ値を次のように見なします。
・絶対アドレスセクション内では絶対アドレス
・相対アドレスセクション内では相対アドレス (セクション先頭からの相対的な距離)

例

```
.SECTION  DT,DATA,LOCATE=H'FFFF0000
.DATA.L    H'11111111
.ORG       H'FFFF0010        ;ロケーションカウンタ値を設定しています。
.DATA.L    H'22222222        ;整数データ H'22222222 を絶対アドレスの
～                           ;H'FFFF0010 番地に確保しています。
```

《メモリ空間》



ロケーションカウンタ値補正

.ALIGN

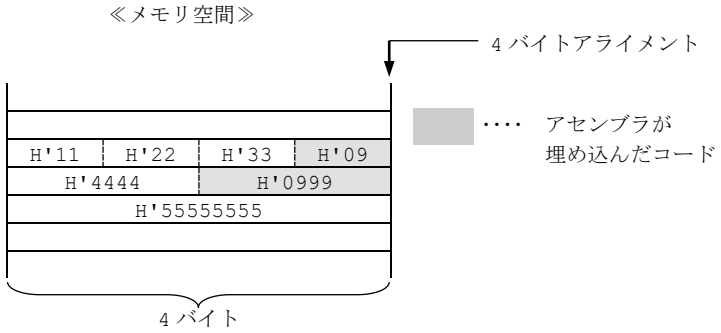
書　式　　△.ALIGN△<アライメント数>
ラベルは記述できません。

説　明　　セクション内のロケーションカウンタ値をアライメント数の倍数に補正します。
.ALIGN によって実行命令やデータを特定の境界(アドレスの区切り)に配置できます。
ロケーションカウンタ値は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
アライメント数として許される値は 2 のべき乗(2^0, 2^1, 2^2, … , 2^{31})です。
相対アドレスセクションで.ALIGN を使用する場合は
　.SECTION で指定するアライメント数 ≧ .ALIGN で指定するアライメント数
となるようにしてください。
コードセクション、データセクションに.ALIGN を記述すると、アセンブラは NOP 命令のオブジェクトコード*をメモリ上に埋めこみ、ロケーションカウンタ値を補正します。半端なバイトサイズの領域には H'09 を埋めこみます。
【注】* このようなオブジェクトコードはアセンブルリスト上に表れません。

例

```
.SECTION  P,CODE,ALIGN=4
.DATA.B    H'11
.DATA.B    H'22
.DATA.B    H'33
.ALIGN  2  ;ロケーションカウンタ値を 2 の倍数に補正しています。
.DATA.W    H'4444
.ALIGN  4  ;ロケーションカウンタ値を 4 の倍数に補正しています。
.DATA.L    H'55555555
～
```

バイトサイズの整数データ H'11 がもともと 4 バイトアライメントに位置するものと仮定します。
アセンブラは下図のようにオブジェクトコードを埋めこんでアライメント調整します。



シンボル値設定

.EQU

書　式　　<シンボル>[:]△.EQU△<シンボル値>

説　明　　シンボルに値を設定します。
.EQU で定義したシンボルは再定義できません。
シンボル値は次のように指定します。
・定数値、アドレス、外部参照シンボルの値*を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
シンボル値として許される値は H'00000000～H'FFFFFFFF です。
(10 進表現では-2,147,483,648～4,294,967,295)
【注】* 外部参照シンボル、外部参照シンボル+定数、外部参照シンボル-定数が記述できます。

例

```
        ～
X1:     .EQU        10              ;X1 の値は 10 になります。
X2:     .EQU        20              ;X2 の値は 20 になります。
        CMP/EQ      #X1,R0          ;CMP/EQ #10,R0 と同じです。
        BT          LABEL1
        CMP/EQ      #X2,R0          ;CMP/EQ #20,R0 と同じです。
        BT          LABEL2
        ～
```

シンボル値設定

.ASSIGN

書　式　　<シンボル>[:]△.ASSIGN△<シンボル値>

説　明　　シンボルに値を設定します。
.ASSIGN で定義したシンボルは.ASSIGN で再定義できます。
シンボル値は次のように指定します。
・定数値またはアドレスを指定する。
　かつ
・前方参照シンボルを使わずに指定する。
シンボル値として許される値は H'00000000～H'FFFFFFFF です。
(10 進表現では-2,147,483,648～4,294,967,295)
.ASSIGN による定義は定義した位置から有効です。
.ASSIGN で定義したシンボルには次の使用上の制限があります。
・外部参照または外部定義できない。
・デバッガで参照できない。

例

```
        ～
X1:     .ASSIGN     1
X2:     .ASSIGN     2
        CMP/EQ      #X1,R0          ;CMP/EQ #1,R0 と同じです。
        BT          LABEL1
        CMP/EQ      #X2,R0          ;CMP/EQ #2,R0 と同じです。
        BT          LABEL2
        ～
X1:     .ASSIGN     3
X2:     .ASSIGN     4
        CMP/EQ      #X1,R0          ;CMP/EQ #3,R0 と同じです。
        BT          LABEL3
        CMP/EQ      #X2,R0          ;CMP/EQ #4,R0 と同じです。
        BF          LABEL4
        ～
```

.REG

書　式　　<シンボル>[:]△.REG△<レジスタ名>
または
<シンボル>[:]△.REG△(<レジスタ名>)

説　明　　レジスタ別名を定義します。
.REG で定義したレジスタ別名は元のレジスタ名と全く同等に使用できます。
.REG で定義したレジスタ別名は再定義できません。
別名をつけることができるのは汎用レジスタ(R0～R15, SP)だけです。
.REG による定義は定義した位置から有効です。
.REG で定義したシンボルには次の使用上の制限があります。
・外部参照または外部定義できない。

例

```
        ～
MIN:    .REG        R10
MAX:    .REG        R11
        MOV         #0,MIN          ;MOV  #0,R10 と同じです。
        MOV         #99,MAX         ;MOV  #99,R11 と同じです。
        CMP/HS      MIN,R1
        BF          LABEL
        CMP/HS      R1,MAX
        BF          LABEL
        ～
```

浮動小数点レジスタ別名定義

.FREG

書　式　　<シンボル>[:]△.FREG△<浮動小数点レジスタ名>
または
<シンボル>[:]△.FREG△(<浮動小数点レジスタ名>)

説　明　　浮動小数点レジスタ別名を定義します。
.FREG で定義した浮動小数点レジスタ名は元のレジスタ名と全く同等に使用できます。
.FREG で定義したレジスタ別名は再定義できません。
別名をつけることができるのは以下の浮動小数点レジスタです。
・FRm(m＝0～15)
・DRn(n＝0,2,4,6,8,10,12,14)
・XDn(n＝0,2,4,6,8,10,12,14)
・Fvi(i＝0,4,8,12)
.FREG による定義は定義した位置から有効です。
.FREG で定義したシンボルには次の使用上の制限があります。
・外部参照または外部定義できない。
.FREG はマイコン種別が SH2E、SH4、SH4A、SH2AFPU のとき使用できます。

例

```
        ～
MAX:    .FREG       FR11
        FMOV        FR1,MAX          ;FMOV FR1,FR11 と同じです。
        FCMP/EQ     MAX,FR2          ;FCMP/EQ FR11,FR2 と同じです。
        BF          LABEL
        ～
```

整数データ確保

.DATA

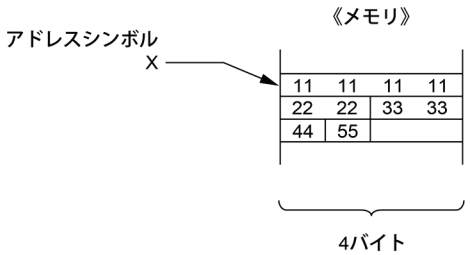
書　式　　[<シンボル>[:]]△.DATA[.オペレーションサイズ]△整数データ[,...]
　　<オペレーションサイズ> = { B | W | L }

説　明　　整数データをメモリ上に確保します。
オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略するとロングワードになります。
整数データには相対アドレス、外部参照シンボル、前方参照シンボルを含めて任意の値を指定できます。
オペレーションサイズおよび整数データの範囲は、次のようになります。

サイズ	データのサイズ		整数データの範囲(10 進表現)	
B	バイト	(1 バイト)	H'00000000～H'000000FF	(0～255)
			H'FFFFFF80～H'FFFFFFFF	(-128～-1)
W	ワード	(2 バイト)	H'00000000～H'0000FFFF	(0～65,535)
			H'FFFF8000～H'FFFFFFFF	(-32,768～-1)
L	ロングワード	(4 バイト)	H'00000000～H'FFFFFFFF	(0～4,294,967,295)
			H'80000000～H'FFFFFFFF	(-2,147,483,648～-1)

例

```
        ～
        .ALIGN  4
X:
        .DATA.L     H'11111111          ;
        .DATA.W     H'2222,H'3333       ;整数データを確保しています。
        .DATA.B     H'44,H'55           ;
        ～
```



【注】データは16進数です。

整数データブロック確保

.DATAB

書　式　　[<シンボル>[:]]△.DATAB[.<オペレーションサイズ>]△<ブロック数>,<整数データ>
　　<オペレーションサイズ> = { B | W | L }

説　明　　整数データを指定の数だけ連続してメモリ上に確保します。
オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略するとロングワードになります。
ブロック数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
整数データには相対アドレス、外部参照シンボル、前方参照シンボルを含めて任意の値を指定できます。
オペレーションサイズによって指定できるブロック数の範囲および整数データの範囲が異なります。
オペレーションサイズおよびブロック数の範囲は、次のようになります。

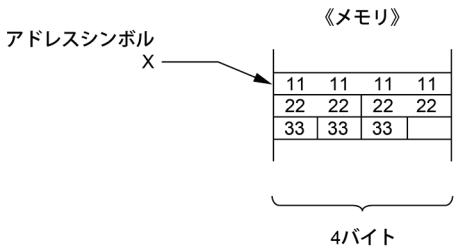
サイズ	データのサイズ		ブロック数の範囲(10 進表現)	
B	バイト	(1 バイト)	H'00000001～H'FFFFFFFF	(1～4,294,967,295)
W	ワード	(2 バイト)	H'00000001～H'7FFFFFFF	(1～2,147,483,647)
L	ロングワード	(4 バイト)	H'00000001～H'3FFFFFFF	(1～1,073,741,823)

整数データの範囲は、次のようになります。

サイズ	整数データの範囲(10 進表現)	
B	H'00000000～H'000000FF	(0～255)
	H'FFFFFF80～H'FFFFFFFF	(-128～-1)
W	H'00000000～H'0000FFFF	(0～65,535)
	H'FFFF8000～H'FFFFFFFF	(-32,768～-1)
L	H'00000000～H'FFFFFFFF	(0～4,294,967,295)
	H'80000000～H'FFFFFFFF	(-2,147,483,648～-1)

例

```
        ～
        .ALIGN  4
X:
        .DATAB.L   1,H'11111111      ;
        .DATAB.W   2,H'2222          ;整数データブロックを確保しています。
        .DATAB.B   3,H'33            ;
        ～
```



【注】データは16進数です。

文字列データ確保

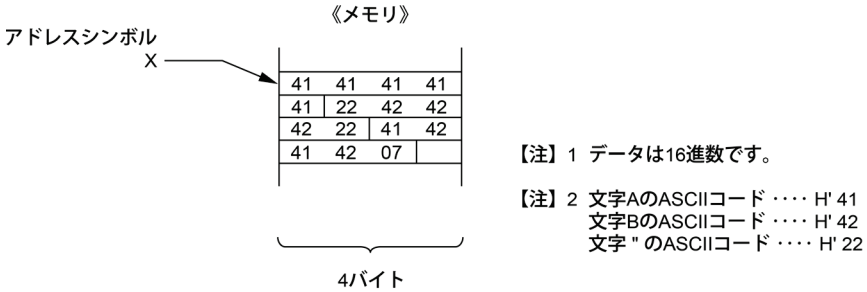
.SDATA

書　式　　[<シンボル>[:]]△.SDATA△"<文字列>"[,...]

説　明　　文字列データをメモリ上に確保します。
文字列に制御文字をつけ加えることができます。
記述形式は次のとおりです。
"文字列"<制御文字の ASCII コード>
制御文字の ASCII コードは次のように指定します。
・定数値を指定する。
かつ
・前方参照シンボルを使わずに指定する。

例

```
        ～
        .ALIGN  4
X:
        .SDATA      "AAAAA"          ;文字列データを確保しています。
        .SDATA      """BBB"""        ;ダブルクォーテーションを含む例です。
        .SDATA      "ABAB"<H'07>     ;制御文字をつけ加えた例です。
        ～
```



文字列データブロック確保

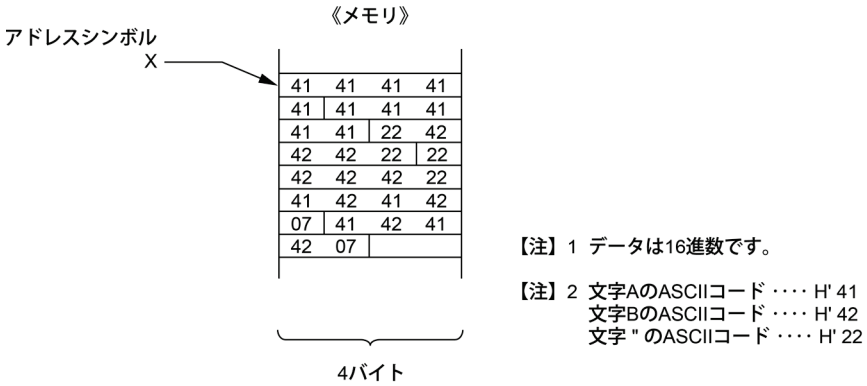
.SDATAB

書　式　　[<シンボル>[:]]△.SDATAB△<ブロック数>,"<文字列>"

説　明　　文字列データを指定の数だけ連続してメモリ上に確保します。
ブロック数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
ブロック数には１以上の値を指定してください。
ブロック数の上限値は文字列データの長さによって決まります。
(文字列データの長さ×ブロック数 ≦ H'FFFFFFFF　(4,294,967,295) バイト)
文字列に制御文字をつけ加えることができます。
記述形式は次のとおりです。
　"文字列"<制御文字の ASCII コード>
制御文字の ASCII コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        ～
        .ALIGN  4
X:
        .SDATAB     2,"AAAAA"          ;文字列データブロックを確保しています。
        .SDATAB     2,"""BBB"""        ;ダブルクォーテーションを含む例です。
        .SDATAB     2,"ABAB"<H'07>     ;制御文字をつけ加えた例です。
        ～
```



計数付き文字列データ確保

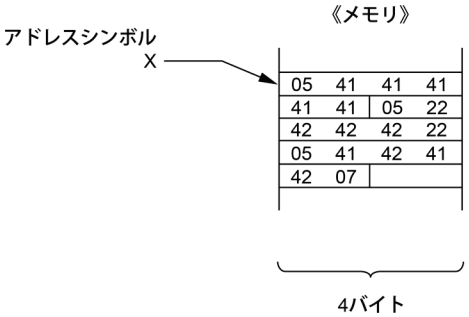
.SDATAC

書　式　　[<シンボル>[:]]△.SDATAC△"<文字列>" [,...]

説　明　　計数付き文字列データをメモリ上に確保します。
計数付き文字列とは文字列の先頭に 1 バイトの計数をつけ加えたものです。
計数は文字列データ (計数を含まないデータ) のサイズをバイト単位で表します。
文字列に制御文字をつけ加えることができます。
記述形式は次のとおりです。
　"文字列"<制御文字の ASCII コード>
制御文字の ASCII コードは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。

例

```
        ～
        .ALIGN  4
X:
        .SDATAC     "AAAAA"          ;計数付き文字列データを確保しています。
        .SDATAC     """BBB"""        ;ダブルクォーテーションを含む例です。
        .SDATAC     "ABAB"<H'07>     ;制御文字をつけ加えた例です。
        ～
```



【注】1 データは16進数です。

【注】2 文字AのASCIIコード ････ H' 41
文字BのASCIIコード ････ H' 42
文字 " のASCIIコード ････ H' 22

ゼロ終端文字列データ確保

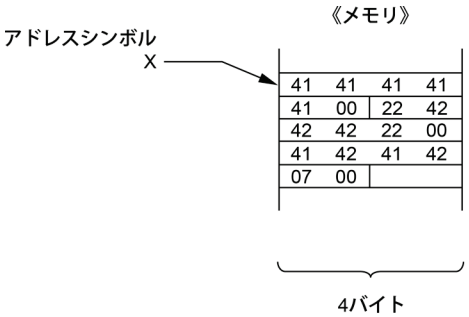
.SDATAZ

書　式　　[<シンボル>[:]]△.SDATAZ△"<文字列>"[,...]

説　明　　ゼロ終端文字列データをメモリ上に確保します。
ゼロ終端文字列とは文字列の最後に 1 バイトのゼロをつけ加えたものです。
文字列に制御文字をつけ加えることができます。
記述形式は次のとおりです。
"文字列"<制御文字の ASCII コード>
制御文字の ASCII コードは次のように指定します。
・定数値を指定する。
かつ
・前方参照シンボルを使わずに指定する。

例

```
        ～
        .ALIGN  4
X:
        .SDATAZ     "AAAAA"          ;ゼロ終端文字列データを確保しています。
        .SDATAZ     """BBB"""        ;ダブルクォーテーションを含む例です。
        .SDATAZ     "ABAB"<H'07>     ;制御文字をつけ加えた例です。
        ～
```



【注】1 データは16進数です。

【注】2 文字AのASCIIコード ････ H' 41
文字BのASCIIコード ････ H' 42
文字 " のASCIIコード ････ H' 22

浮動小数点データ確保

.FDATA

書　式

```
[<シンボル>[:]]△.FDATA[.<オペレーションサイズ>]△<浮動小数点定数>[,...]
        <オペレーションサイズ> = { S | D }
```

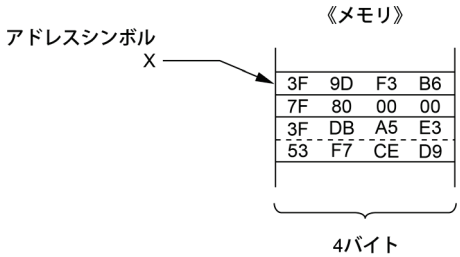
説　明

浮動小数点定数データをメモリ上に確保します。
オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略すると単精度になります。
オペレーションサイズは以下のとおりです。

指定内容	データのサイズ
S	単精度(4 バイト)
D	倍精度(8 バイト)

例

```
        ～
        .ALIGN  4
X:
        .FDATA.S    F'1.234          ;4 バイトの領域
                                     ;"3F9DF3B6"(F'1.234S)
                                     ;を確保します。
        .FDATA.S    H'7F800000.S     ;4 バイトの領域
                                     ;"7F800000"(H'7F800000.S)
                                     ;を確保します。
        .FDATA.D    F'4.32D-1        ;8 バイトの領域
                                     ;"3FDBA5E353F7CED9"(F'4.32D-1)
                                     ;を確保します。
        ～
```



備　考

オペランドとしては H'の形式(浮動小数点の 16 進表記)に対しての式は記述できません。
F'の形式(浮動小数点の 10 進表記)対しての式を記述できます。
浮動小数点の記法はアセンブラの浮動小数点定数と同じです。

```
  "F'"["+"|"-"](n["."[m]]|"."m)[("S"|"D")["+"|"-"]x] (n,m,x は 10 進字列)
```

指数記号(S または D)が無いときは、.FDATA のサイズ指定に従います。
.FDATA.S に倍数度定数が出てくると単精度に、.DATA.D に単精度定数が出てくると倍精度に変換します。
浮動小数点式の文法は以下の通りです。

```
  <式> ::= ["+"|"-"] <項>[("+"|"-") <項>]*
  <項> ::= <因子> [("*"|"/") <因子>]
  <因子> ::= <定数> | "(" <式> ")"
```

0.0 での除算はエラーになります。
浮動小数点定数以外の定数(整数、固定小数点)との混合演算はできません。

浮動小数点データブロック確保

.FDATAB

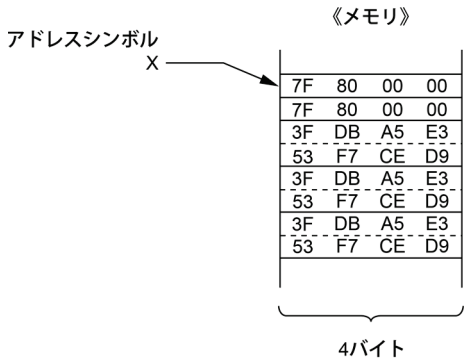
書　式　　[<シンボル>[:]]△.FDATAB[.<オペレーションサイズ>]△
<ブロック数>,<浮動小数点定数>
<オペレーションサイズ> = { S | D }

説　明　　浮動小数点定数データを指定の数だけ連続してメモリ上に確保します。
オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略すると単精度になります。
ブロック数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボル、外部参照シンボル、相対アドレスシンボルを使わずに指定する。
オペレーションサイズによって指定できるブロック数の範囲が異なります。
オペレーションサイズおよびブロック数の範囲は、次のようになります。

サイズ	データのサイズ		ブロック数の範囲(10 進表現)	
S	単精度	(4 バイト)	H'00000001～H'3FFFFFFF	(1～1,073,741,823)
D	倍精度	(8 バイト)	H'00000001～H'1FFFFFFF	(1～536,870,911)

例

```
        ～
        .ALIGN  4
X:
        .FDATAB.S  2,H'7F800000.S     ;4 バイトの領域
                                      ;"7F800000"(H'7F800000.S)
                                      ;を 2 個確保します。
        .FDATAB.D  3,F'4.32D-1        ;8 バイトの領域
                                      ;"3FDBA5E353F7CED9"(F'4.32D-1)
                                      ;を 3 個確保します。
        ～
```



固定小数点データ確保

.XDATA

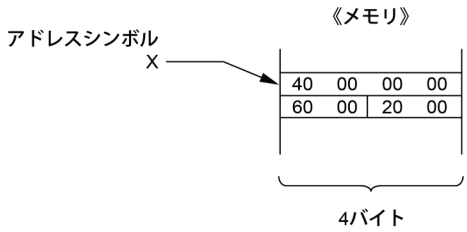
書　式　　[<シンボル>[:]]△.XDATA[.<オペレーションサイズ>]△<固定小数点定数>[,...]
　　<オペレーションサイズ> = { W | L }

説　明　　固定小数点定数データをメモリ上に確保します。
オペレーションサイズによって確保するデータのサイズが決まります。
オペレーションサイズを省略するとロングワードになります。
オペレーションサイズは以下のとおりです。

指定内容	データのサイズ
W	ワード(2 バイト)
L	ロングワード(4 バイト)

例

```
        ～
        .ALIGN  4
X:
        .XDATA.L    0.5             ;4 バイトの領域(H'40000000)
                                    ;を確保します。
        .XDATA.W    0.75,0.25       ;2 バイトの領域(H'6000)と(H'2000)
                                    ;を確保します。
        ～
```



データ領域確保

.RES

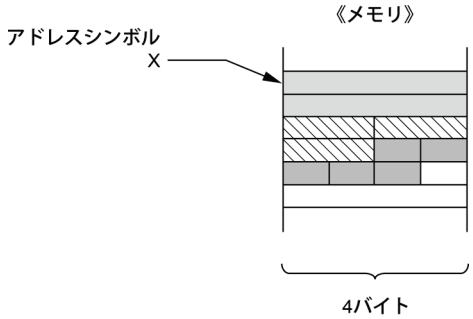
書　式　　[<シンボル>[:]]△.RES[.<オペレーションサイズ>]△<領域数>
　　<オペレーションサイズ> = { B | W | L }

説　明　　データ領域をメモリ上に確保します。
オペレーションサイズによって確保するデータ領域の単位サイズが決まります。
オペレーションサイズを省略するとロングワードになります。
領域数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
オペレーションサイズによって、指定できる領域数の範囲が異なります。
データのサイズと領域数の範囲は、次のようになります。

サイズ	データのサイズ		領域数の範囲(10 進表現)
B	バイト	(1 バイト)	H'00000001～H'FFFFFFFF (1～4,294,967,295)
W	ワード	(2 バイト)	H'00000001～H'7FFFFFFF (1～2,147,483,647)
L	ロングワード	(4 バイト)	H'00000001～H'3FFFFFFF (1～1,073,741,823)

例

```
        ～
        .ALIGN  4
X:
        .RES.L      2       ;ロングワードサイズの領域 2 つ分を確保しています。
        .RES.W      3       ;ワードサイズの領域 3 つ分を確保しています。
        .RES.B      5       ;バイトサイズの領域 5 つ分を確保しています。
        ～
```



文字列データ領域確保

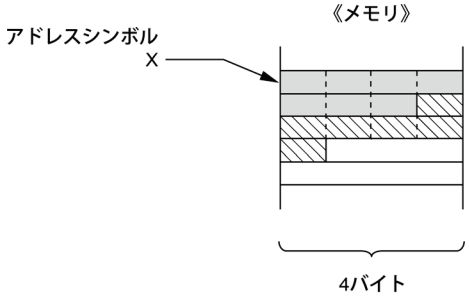
.SRES

書　式　　[<シンボル>[:]]△.SRES△<文字列領域サイズ>[,...]

説　明　　文字列用のデータ領域を確保します。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000001～H'FFFFFFFF です。
(10 進表現では 1～4,294,967,295)

例

```
        ～
        .ALIGN  4
X:
        .SRES       7       ;7 バイトの領域を確保しています。
        .SRES       6       ;6 バイトの領域を確保しています。
        ～
```



計数付き文字列データ領域確保

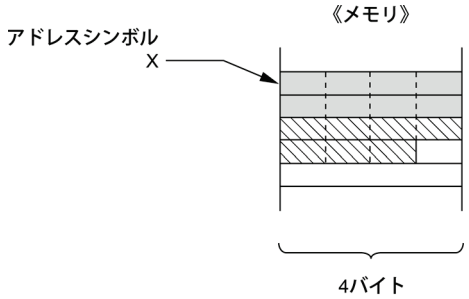
.SRESC

書　式　　[<シンボル>[:]]△.SRESC△<文字列領域サイズ>[,...]

説　明　　計数付き文字列用のデータ領域をメモリ上に確保します。
計数付き文字列とは文字列の先頭に 1 バイトの計数をつけ加えたものです。
計数は文字列用のデータ領域 (計数を含まない領域) のサイズをバイト単位で表します。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000000～H'000000FF です。
(10 進表現では 0～255)
メモリ上に確保される領域のサイズは文字列領域サイズ＋計数用の 1 バイトです。

例

```
        ～
       .ALIGN  4
X:
        .SRESC     7        ;7 バイト＋計数用の 1 バイトを確保しています。
        .SRESC     6        ;6 バイト＋計数用の 1 バイトを確保しています。
        ～
```



ゼロ終端文字列データ領域確保

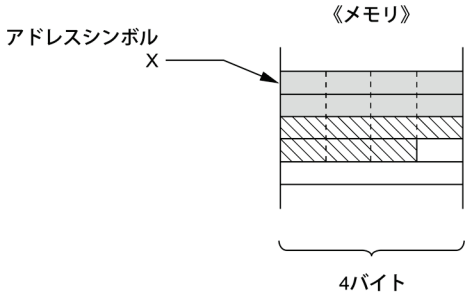
.SRESZ

書　式　　[<シンボル>[:]]△.SRESZ△<文字列領域サイズ>[,...]

説　明　　ゼロ終端文字列用のデータ領域をメモリ上に確保します。
ゼロ終端文字列とは文字列の終端に 1 バイトのゼロをつけ加えたものです。
文字列領域サイズは次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
文字列領域サイズとして許される値は H'00000000～H'000000FF です。
(10 進表現では 0～255)
メモリ上に確保される領域のサイズは文字列領域サイズ＋終端ゼロ用の 1 バイトです。

例

```
        ～
        .ALIGN  4
X:
        .SRESZ      7       ;7 バイト＋終端ゼロ用の 1 バイトを確保しています。
        .SRESZ      6       ;6 バイト＋終端ゼロ用の 1 バイトを確保しています。
        ～
```



浮動小数点データ領域確保

.FRES

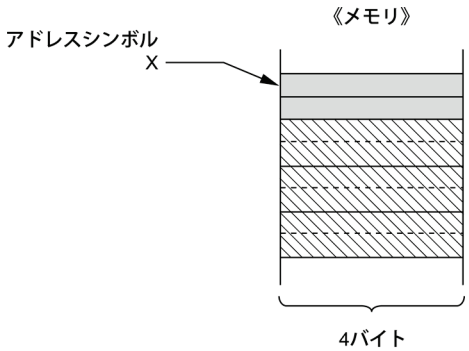
書　式　　[<シンボル>[:]]△.FRES[.<オペレーションサイズ>]△<領域確保数>
　　<オペレーションサイズ> = { S | D }

説　明　　浮動小数点データ領域をメモリ上に確保します。
オペレーションサイズによって確保するデータ領域の単位サイズが決まります。
オペレーションサイズを省略すると単精度になります。
領域数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボル、外部参照シンボル、相対アドレスシンボルを使わずに指定する。
オペレーションサイズは以下のとおりです。

指定内容	データのサイズ
S	単精度(４バイト)
D	倍精度(８バイト)

例

```
        ～
        .ALIGN  4
X:
        .FRES.S     2       ;領域 2 つ分を確保しています。
        .FRES.D     3       ;領域 3 つ分を確保しています。
        ～
```



外部定義シンボル宣言

.EXPORT

書　式　　△.EXPORT△<シンボル>[,...]
ラベルは記述できません。

説　明　　外部定義シンボルを宣言します。
外部定義シンボルの宣言はファイル内で定義しているシンボルを他のファイルで参照するために必要です。
外部定義シンボルとして宣言できるのは次のものです。
・定数シンボル(.ASSIGN で定義したものを除く)
・絶対アドレスシンボル(ダミーセクションのアドレスシンボルを除く)
・相対アドレスシンボル
シンボルを外部参照するには外部定義シンボルとして宣言するとともに外部参照シンボルとして宣言しなければなりません。
外部参照シンボルはシンボルを参照しているファイルで.IMPORT または.GLOBAL によって宣言します。

例　　ファイル A で定義しているシンボルをファイル B で参照する例です。
・ファイル A

```
        .EXPORT    X                ;X を外部定義シンボルとして宣言します。
        ～
X:      .EQU  H'10000000            ;X を定義します。
        ～
```

・ファイル B

```
        .IMPORT    X                ;X を外部参照シンボルとして宣言します。
        ～
        .ALIGN  4
        .DATA.L    X                ;X を参照します。
        ～
```

外部参照シンボル宣言

.IMPORT

書　式　　△.IMPORT△<シンボル>[,...]
ラベルは記述できません。

説　明　　外部参照シンボルを宣言します。
外部参照シンボルの宣言は他のファイルで定義しているシンボルを参照するために必要です。
ファイル内で定義しているシンボルは外部参照シンボルとして宣言できません。
シンボルを外部参照するには外部参照シンボルとして宣言するとともに外部定義シンボルとして宣言しなければなりません。
外部定義シンボルはシンボルを定義しているファイルで.EXPORT または.GLOBAL によって宣言します。

例　　ファイル A で定義しているシンボルをファイル B で参照する例です。

・ファイル A

```
        .EXPORT    X                ;X を外部定義シンボルとして宣言します。
        ～
X:      .EQU  H'10000000            ;X を定義します。
        ～
```

・ファイル B

```
        .IMPORT    X                ;X を外部参照シンボルとして宣言します。
        ～
        .ALIGN  4
        .DATA.L    X                ;X を参照します。
        ～
```

外部定義シンボル、外部参照シンボル宣言

.GLOBAL

書　式　　△.GLOBAL△<シンボル>[,...]
ラベルは記述できません。

説　明　　外部定義シンボルまたは外部参照シンボルを宣言します。
外部定義シンボルの宣言はファイル内で定義しているシンボルを他のファイルで参照するために必要です。外部参照シンボルの宣言は他のファイルで定義しているシンボルをファイル内で参照するために必要です。
ファイル内で定義しているシンボルを.GLOBAL で宣言するとそのシンボルは外部定義シンボルになります。
ファイル内で定義していないシンボルを.GLOBAL で宣言するとそのシンボルは外部参照シンボルになります。
外部定義シンボルとして宣言できるのは、次のものです。
・定数シンボル(.ASSIGN で定義したものを除く)
・絶対アドレスシンボル(ダミーセクションのアドレスシンボルを除く)
・相対アドレスシンボル
シンボルを外部参照するには外部定義シンボルとして宣言するとともに外部参照シンボルとして宣言しなければなりません。
外部定義シンボルはシンボルを定義しているファイルで.EXPORT または.GLOBAL によって宣言します。
外部参照シンボルはシンボルを参照しているファイルで.IMPORT または.GLOBAL によって宣言します。

例　　ファイル A で定義しているシンボルをファイル B で参照する例です。
・ファイル A

```
        .GLOBAL    X                ;X を外部定義シンボルとして宣言します。
        ～
X:      .EQU  H'10000000            ;X を定義します。
        ～
```

・ファイル B

```
        .GLOBAL    X                ;X を外部参照シンボルとして宣言します。
        ～
        .ALIGN  4
        .DATA.L    X                ;X を参照します。
        ～
```

オブジェクトモジュール、デバッグ情報出力制御

.OUTPUT

書　式　　△.OUTPUT△<出力指定>[,...]

```
<出力指定> = { OBJ | NOOBJ |
               DBG | NODBG }
```

ラベルは記述できません。

説　明　　オブジェクトモジュールまたはデバッグ情報の出力を制御します。

(1) オブジェクトモジュールの出力

オブジェクトモジュールの出力を制御します。

出力種別は次のようになります。

出力種別	出力制御
OBJ	出力
NOOBJ	出力抑止

(2) デバッグ情報の出力

デバッグ情報の出力を制御します。

出力種別は次のようになります。

出力種別	出力制御
DBG	出力
NODBG	出力抑止

.OUTPUT を 2 回以上使用して指定内容が矛盾するとエラーとなります。

【例】

```
    ～
.OUTPUT  OBJ
.OUTPUT  NODBG
    ～
```

← OK

```
    ～
.OUTPUT  OBJ
.OUTPUT  NOOBJ
    ～
```

← エラー

デバッグ情報の出力に関する指定はオブジェクトモジュールを出力する場合に限り有効です。オブジェクトモジュールとデバッグ情報の出力に関して、アセンブラはオプションによる指定を優先します。

例　　オブジェクトモジュールとデバッグ情報の出力に関して、オプションによる指定がないことを仮定して結果を説明しています。

・例 1

```
.OUTPUT    OBJ          ;オブジェクトモジュールを出力します。
                        ;デバッグ情報は出力しません。
～
```

・例 2

```
.OUTPUT    OBJ,DBG      ;オブジェクトモジュールとデバッグ情報を
                        ;出力します。
～
```

・例 3

```
.OUTPUT     OBJ,NODBG          ;オブジェクトモジュールを出力します。
                               ;デバッグ情報は出力しません。
～
```

備　考　　デバッグ情報はデバッガでプログラムをデバッグするときに必要な情報であり、オブジェクトモジュールの一部となります。
デバッグ情報はソースステートメントの行に関する情報、シンボルに関する情報などを含みます。

シンボルデバッグ情報部分出力制御

.DEBUG

書　式　　△.DEBUG△<出力指定>

<出力指定> = { ON | OFF }

ラベルは記述できません。

説　明　　シンボルデバッグ情報の部分出力を制御します。
デバッグに必要なシンボルに限定してシンボルデバッグ情報を出力するとアセンブル時間を短縮できるなどの利点があります。
.DEBUG による指定はオブジェクトモジュールを出力し、かつ、デバッグ情報を出力する場合に限り有効です。
出力種別は、以下のようになります。

出力種別	出力制御
ON	出力
OFF	出力抑止

例

```
～
.DEBUG      OFF                 ;アセンブラは次のソースステートメント
                                ;からシンボルデバッグ情報を出力しません。
～
.DEBUG      ON                  ;アセンブラは次のソースステートメント
                                ;からシンボルデバッグ情報を出力します。
～
```

備　考　　シンボルデバッグ情報とはデバッグ情報のうちのシンボルに関するものをいいます。

メモリのバイト並び順

.ENDIAN

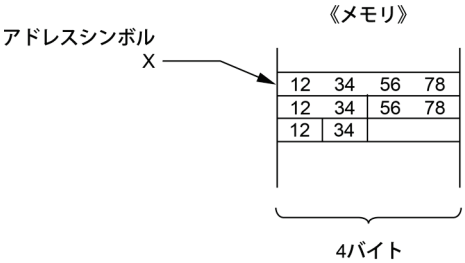
書　式　　△.ENDIAN△<エンディアン種別>
　　<エンディアン種別> = { BIG | LITTLE }
ラベルは記述できません。

説　明　　ターゲットマイコンのバイト並び順が Big Endian か Little Endian かを指定します。
.ENDIAN はソースプログラムの最初に記述してください。
エンディアン種別は、オプションによる指定を優先します。

指定内容	出力の制御
BIG	Big Endian でアセンブルする。
LITTLE	Little Endian でアセンブルする。

例　　・例 1(Big Endian を指定した場合)

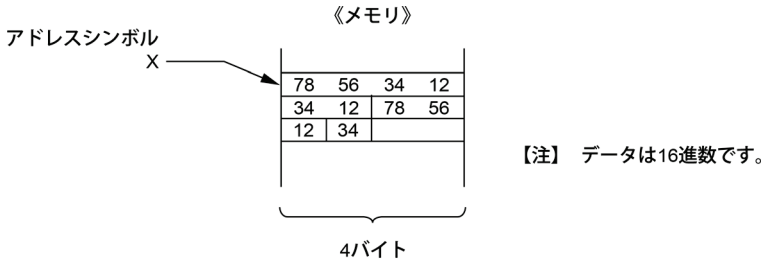
```
        .CPU        SH3
        .ENDIAN     BIG                 ;Big Endian を指定します。
        ～
X:
        .DATA.L     H'12345678          ;
        .DATA.W     H'1234,H'5678       ;整数データを確保しています。
        .DATA.B     H'12,H'34           ;
        ～
```



【注】 データは16進数です。

・例 2(Little Endian を指定した場合)

```
        .CPU        SH3
        .ENDIAN     LITTLE              ;Little Endian を指定します。
        ～
X:
        .DATA.L     H'12345678          ;
        .DATA.W     H'1234,H'5678       ;整数データを確保しています。
        .DATA.B     H'12,H'34           ;
        ～
```



行番号変更

.LINE

書　式　　△.LINE△["<ファイル名>",]<行番号>
ラベルは記述できません。

説　明　　アセンブラのエラーメッセージあるいはデバッグ時に参照する行番号とファイル名を変更します。
プログラム内の最初の.LINE 以降は次の.LINE まで行番号、ファイル名を更新しません。
コンパイラ(V3.0 以降)は、デバッグオプションを指定してアセンブリソースを出力する時にソースファイル行に対応する.LINE を生成します。
ファイル名を省略するとファイル名は変更されず、行番号だけが変更されます。

例　　shc –code=asmcode –debug test.c

ソースプログラム(test.c)

```
int func()
{
   int i,j;

   j=0;
   for (i=1;i<=10;i++){
      j+=i;
   }
   return(j);
}
```

アセンブリソースプログラム(test.src)

```
        .FILE       "C:¥asm¥test.c"
        .EXPORT     _func
        .SECTION    P,CODE,ALIGN=4
_func:                          ; function: func
                               ; frame size=0
        .LINE       "C:¥asm¥test.c",5
        MOV         #0,R5      ; H'00000000
        .LINE       "C:¥asm¥test.c",6
        MOV         #1,R2      ; H'00000001
        MOV         #10,R6     ; H'0000000A
L11:
        ADD         #-1,R6
        TST         R6,R6
        .LINE       "C:¥asm¥test.c",7
        ADD         R2,R5
        ADD         #1,R2
        BF          L11
        RTS
        MOV         R5,R0
        .END
```

アセンブルリスト出力制御

.PRINT

書　式　　△.PRINT△<出力指定> [,...]

```
<出力指定> = { LIST |NOLIST |SRC | NOSRC |
               CREF |NOCREF |SCT | NOSCT }
```

ラベルは記述できません。

説　明　　出力指定により、以下の各リストの出力/出力抑止を制御します。

(1) アセンブルリスト
(2) ソースプログラムリスト
(3) クロスリファレンスリスト
(4) セクション情報リスト

各出力指定により制御される内容は以下のとおりです。

項目	出力指定 *1		意味	制御内容
	出力	出力抑止		
(1)	LIST	NOLIST	アセンブルリストの出力制御 *2	アセンブルリストの出力/出力抑止
(2)	SRC	NOSRC	ソースプログラムリストの出力制御 *3 *4	ソースプログラムリストの出力/出力抑止
(3)	CREF	NOCREF	クロスリファレンスリストの出力制御 *3 *5	クロスリファレンスリストの出力/出力抑止
(4)	SCT	NOSCT	セクション情報リストの出力制御 *3 *6	セクション情報リストの出力/出力抑止

【注】
*1　本指定は 1 度限り有効です。
*2　list/nolist オプションの指定がない場合に有効です。
*3　アセンブルリスト出力時のみ有効です。
*4　source/nosource オプションの指定がない場合に有効です。
*5　cross_reference/nocross_reference オプションの指定がない場合に有効です。
*6　section/nosection オプションの指定がない場合に有効です。

.PRINT を 2 回以上使用して、指定内容が矛盾するとエラーとなります。

【例】

```
    ～
.PRINT LIST
.PRINT NOSRC
    ～
```

← OK

```
    ～
.PRINT LIST
.PRINT NOLIST
    ～
```

← エラー

例　　アセンブルリストの出力に関してオプションによる指定がないことを仮定して結果を説明しています。

・例 1

```
.PRINT     LIST                ;全種類のアセンブルリストを出力します。
～
```

・例 2

```
.PRINT     LIST,NOSRC,NOCREF ;セクション情報リストだけを出力します。
～
```

ソースプログラムリスト部分出力制御

.LIST

書　式

```
△.LIST△<出力指定> [,...]
        <出力指定> = { ON   | OFF    |
                       COND | NOCOND | DEF  | NODEF | CALL  | NOCALL |
                       EXP  | NOEXP  | CODE | NOCODE}
```

ラベルは記述できません。

説　明

出力指定により次のような働きをします。

(1) ソースステートメントの部分的な出力、出力抑止の制御
(2) 条件つきアセンブリ機能およびマクロ機能に関するソースステートメントの部分的な出力、出力抑止の制御
(3) オブジェクトコード表示行の部分的な出力、出力抑止の制御

各出力指定により制御される内容は以下のとおりです。

項目	出力指定		意味	制御内容
	出力	出力抑止		
(1)	ON	OFF	ソースステートメントの出力制御	本命令以降のソースステートメント
(2)	COND	NOCOND	条件つき不成立の出力制御*1	.AIF, .AIFDEF の不成立部分
	DEF	NODEF	定義の出力制御*1	マクロ定義部分 .AREPEAT, .AWHILE 定義部分 .INCLUDE .ASSIGNA, .ASSIGNC
	CALL	NOCALL	コールの出力制御*1	マクロコール文 .AIF, .AIFDEF, .AENDI
	EXP	NOEXP	展開の出力制御*1	マクロ展開部分 .AREPEAT, .AWHILE 展開部分
(3)	CODE	NOCODE	オブジェクトコード表示行の出力制御*1	制御命令のオブジェクトコード表示が、ソースステートメントの行数を超える部分

【注】*1　本指定は、show/noshow オプションの指定がない場合に有効です。

.LIST による指定はソースプログラムリストを出力する場合に限り有効です。
ソースプログラムリストの部分出力に関して、アセンブラはオプションによる指定を優先します。
.LIST 自体はソースプログラムリスト上に表示されません。

例

ソースプログラムリストの部分出力に関して、オプションによる指定がないことを仮定して結果を説明しています。

```
        .LIST       NOCOND,NODEF
        .MACRO      SHLRN COUNT,Rd

SHIFT   .ASSIGNA    ¥COUNT

        .AIF        ¥&SHIFT GE 16
        SHLR16      ¥Rd
SHIFT   .ASSIGNA    ¥&SHIFT-16
        .AENDI

        .AIF        ¥&SHIFT GE 8
        SHLR8       ¥Rd
SHIFT   .ASSIGNA    ¥&SHIFT-8
        .AENDI

        .AIF        ¥&SHIFT GE 4
        SHLR2       ¥Rd
        SHLR2       ¥Rd
SHIFT   .ASSIGNA    ¥&SHIFT-4
        .AENDI

        .AIF        ¥&SHIFT GE 2
        SHLR2       ¥Rd
SHIFT   .ASSIGNA    ¥&SHIFT-2
        .AENDI

        .AIF ¥&SHIFT GE 1
        SHLR        ¥Rd
        .AENDI
        .ENDM

        SHLRN       23,R0
```

ソースプログラムリストの部分出力を制御します。

汎用多ビットシフトをマクロ定義しています。

マクロコール

コーディング例のソースリスト出力結果
.LIST によってマクロ定義、.ASSIGNA、.ASSIGNC、.AIF または.AIFDEF の不成立部分の出力が抑止されています。

```
31                                  31
32                                  32        SHLRN   23,R0
33                                    M
35                                    M
36                                    M      .AIF 23 GE 16
37 00000000 4029                      C      SHLR16   R0
39                                    M      .AENDI
40                                    M
41                                    M      .AIF 7 GE 8
45                                    M
46                                    M      .AIF 7 GE 4
47 00000002 4009                      C      SHLR2    R0
48 00000004 4009                      C      SHLR2    R0
50                                    M      .AENDI
51                                    M
52                                    M      .AIF 3 GE 2
53 00000006 4009                      C      SHLR2    R0
55                                    M      .AENDI
56                                    M
57                                    M      .AIF 1 GE 1
58 00000008 4001                      C      SHLR     R0
59                                    M      .AENDI
```

アセンブルリスト行数/桁数/タブサイズ設定

.FORM

書　式　　△.FORM△<サイズ指定>[,...]
　　<サイズ指定> = { LIN = <行数> | COL = <桁数> | TAB = {4|8}}
ラベルは記述できません。

説　明　　アセンブルリストの 1 ページあたりの行数と 1 行あたりの桁数、およびタブサイズを設定します。
行数と桁数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
<行数>、<桁数>の許される値の範囲、および.FORM による指定もオプションによる指定もない場合の解釈は次の通りです。

指定内容	意味	許される値	未指定時
LIN=<行数>	1 ページあたりの行数 *1	20 ~ 255 *4	60
COL=<桁数>	1 行あたりの桁数 *2	79 ~ 255 *4	132
TAB={4|8}	タブサイズ *3	4,8 *5	8

【注】 *1　lines オプションの指定がない場合、有効になります。
*2　columns オプションの指定がない場合、有効になります。
*3　show オプションの tab サブオプションの指定が無い場合、有効になります。
*4　範囲外の値を指定した時は、下限値より小さい場合は下限値に、上限値より大きい場合は上限値が指定されます。この場合、エラーは表示されません。
*5　有効な値以外を指定した時は、8 が指定されます。この場合、エラーは表示されません。

アセンブルリストの行数と桁数に関して、アセンブラはオプションによる指定を優先します。
.FORM は 1 つのソースプログラムで何回でも使えます。

例　　アセンブルリストの行数と桁数、タブサイズの設定に関して、オプションによる指定がないことを仮定して結果を説明しています。

```
～
.FORM  LIN=60,COL=200,TAB=4   ;このページからアセンブルリスト 1 ページ
                              ;を 60 行にします。
                              ;また、この行からアセンブルリスト 1 行を
                              ;200 桁にします。
                              ;タブサイズを 4 としてアセンブルリストを
                              ;出力します。
～
.FORM  LIN=55,COL=150         ;このページからアセンブルリスト 1 ページ
                              ;を 55 行にします。
                              ;また、この行からアセンブルリスト 1 行を
                              ;150 桁にします。
～
```

ソースプログラムリストヘッダ設定

.HEADING

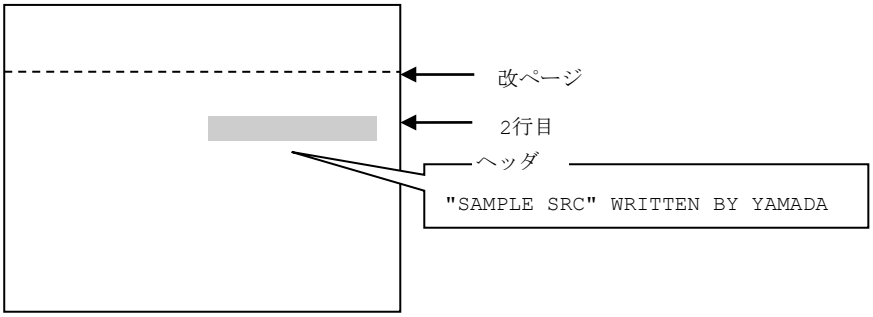
書　式　　△.HEADING△"<文字列>"
　　ラベルは記述できません。

説　明　　ソースプログラムリストのヘッダを設定します。
　　ヘッダとして設定できるのは 60 文字以内の文字列です。
　　.HEADING は 1 つのソースプログラムの中で何回でも使えます。
　　.HEADING による設定の有効範囲は次のようになります。
　　・ページの 1 行目で設定している場合、そのページから有効
　　・ページの 2 行目以降で設定している場合、次のページから有効

例

```
～
.HEADING   """SAMPLE.SRC""  WRITTEN BY YAMADA"
～
```

≪ソースプログラムリスト≫



ソースプログラムリスト改ページ

.PAGE

書　式　　△.PAGE
ラベルは記述できません。

説　明　　ソースプログラムリストを任意の位置で改ページします。
.PAGE がリストの１行目にある場合、その改ページ指定は無効になります。
.PAGE 自体はソースプログラムリスト上に表示されません。

例

```
～
MOV         R0,R1
RTS
MOV         R0,R2
.PAGE       ;セクションが切り替わるので改ページを指定しています。
.SECTION    DT,DATA,ALIGN=4
.DATA.L     H'11111111
.DATA.L     H'22222222
.DATA.L     H'33333333
～
```

≪ソースプログラムリスト≫

```
  18 00000022 6103                18          MOV       R0,R1
  19 00000024 000B                19          RTS
  20 00000026 6203                20          MOV       R0,R2
```

改ページ

```
*** SuperH RISC engine ASSEMBLER Ver. 5.1 ***   06/01/01 10:15:30
PROGRAM NAME =

  22 00000000                     22          .SECTION  DT,DATA,ALIGN
  23 00000000 11111111            23          .DATA.L   H'11111111
  24 00000004 22222222            24          .DATA.L   H'22222222
  25 00000008 33333333            25          .DATA.L   H'33333333
```

ソースプログラムリスト空行出力

.SPACE

書　式　　△.SPACE[△<行数>]
ラベルは記述できません。

説　明　　空行を指定の行数分ソースプログラムリストに出力します。
オペランドを省略すると空行を1行出力します。
.SPACE で出力する空行には行番号などの表示がありません。
行数は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボルを使わずに指定する。
行数として許される値は1～50 です。
.SPACE で空行を出力して改ページが生じる場合、アセンブラは改ページ以降の空行を出力しません。
.SPACE 自体はソースプログラムリスト上に表示されません。

例

```
.SECTION   DT1,DATA,ALIGN=4
.DATA.L    H'11111111
.DATA.L    H'22222222
.DATA.L    H'33333333
.DATA.L    H'44444444        ;セクションが切り替わる箇所で、
.SPACE  5                    ;5 行の空行を挿入しています。
.SECTION   DT2,DATA,ALIGN=4
～
```

≪ソースプログラムリスト≫

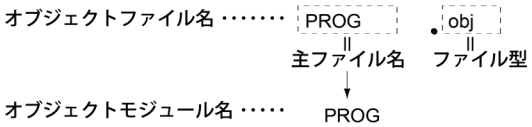
```
*** SuperH RISC engine ASSEMBLER Ver. 5.1 ***   06/01/01 10:15:30
PROGRAM NAME =
   1 00000000                    1          .SECTION  DT1,DATA,ALIGN=4
   2 00000000 11111111           2          .DATA.L   H'11111111
   3 00000004 22222222           3          .DATA.L   H'22222222
   4 00000008 33333333           4          .DATA.L   H'33333333
   5 0000000C 44444444           5          .DATA.L   H'44444444





   7 00000000                    7          .SECTION  DT2,DATA,ALIGN=4
                              ～
```

オブジェクトモジュール名設定

.PROGRAM

書　式　　△.PROGRAM△<オブジェクトモジュール名>
ラベルは記述できません。

説　明　　オブジェクトモジュール名を設定します。
オブジェクトモジュール名とは最適化リンケージエディタがオブジェクトモジュールを識別するために必要とする名前です。
オブジェクトモジュール名の付け方はシンボルの名付け方と同じです。
アセンブラはオブジェクトモジュール名の英大文字と英小文字を区別します。
.PROGRAM による設定は最初の 1 回だけが有効です。アセンブラは 2 回目以降の指定を無視します。
.PROGRAM による設定がない場合、デフォルト (暗黙) のオブジェクトモジュール名を設定します。デフォルトはオブジェクトファイル (オブジェクトモジュールの出力先) の主ファイル名です。



オブジェクトモジュール名はプログラムの中で使用しているシンボル名と重複しても構いません。

例

```
.PROGRAM    PROG1          ;オブジェクトモジュール名として PROG1 を
                           ;設定しています。
～
```

基数指定

.RADIX

書　式　　△.RADIX△<基数指定>
　　<基数指定> = { B | Q | D | H }
ラベルは記述できません。

説　明　　基数のない整数定数の基数を設定します。
基数指定の内容によって基数のない整数定数が何進数になるかが決まります。
.RADIX による指定を省略した場合、基数のない整数定数は 10 進数です。
基数のない整数定数が 16 進数になるよう指定した場合(基数指定 H)、整数定数の一番上位の桁が A～F であるときはその上に 0 をつけ加えてください(アセンブラは A～F で始まる記述をシンボルと見なします) 。
.RADIX による指定は指定した位置から有効です。

指定内容	基数のない整数定数
B	2 進数
Q	8 進数
D	10 進数
H	16 進数

例　　・例 1

```
        ～
        .RADIX  D
X:      .EQU    100             ;100 は 10 進数です。
        ～
        .RADIX  H
Y:      .EQU    64              ;64 は 16 進数です。
        ～
```

・例 2

```
        ～
        .RADIX  H
Z:      .EQU    0F              ;F と書くとシンボルと見なされるので
                                ;先頭に 0 を付けています。
        ～
```

ソースプログラムの終わりとエントリポイント指定

.END

書　式　　△.END［△<シンボル>］

説　明　　ソースプログラムの終わりを示します。
.END が出現した時点でアセンブラはアセンブル処理を終了します。
オペランドに指定したシンボルをエントリポイントとします。
シンボルには外部定義シンボルを指定します。

例

```
        .EXPORT    START
        .SECTION   P,CODE,ALIGN=4
START:
        ～
        .END       START        ;ソースプログラムの終了を宣言しています。
                                ;シンボル START がエントリポイントになります。
```

指定したシンボルに対してスタック値を定義

.STACK

書　式　　△.STACK△<シンボル>=<スタック値>
ラベルは記述できません。

説　明　　シンボルに対して、CallWalker で参照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値は 1 度のみ有効とします。2 度以上指定した場合は、その定義を無効とします。また、指定できるスタック値は、H'00000000～H'FFFFFFFC の範囲の 4 の倍数のみとし、それ以外を指定した場合はその定義を無効とします。
スタック値は次のように指定します。
・定数値を指定する。
　かつ
・前方参照シンボル、外部参照シンボル、相対アドレスシンボルを使わずに指定する。

例

```
        ～
        .STACK  SYMBOL=H'100
        ～
```

11.5　ファイルインクルード機能

ファイルインクルードとはアセンブルするソースファイルに他のソースファイルを取り込む機能です(以下、取り込まれる側のソースファイルをインクルードファイルといいます)。

ファイルインクルード機能に関する制御文として.INCLUDE があります。

.INCLUDE を記述した位置に指定のインクルードファイルが取り込まれます。

■コーディング例

ソースプログラム

```
. INCLUDE    "FILE. H"

. SECTION   CD1, CODE, ALIGN=4
MOV   #ON, R0

～
```

インクルードファイル　FILE. H

```
ON:      . EQU    1
OFF:     . EQU    0
```

↓↓↓↓↓↓↓↓↓↓

ファイルインクルード結果（ソースリスト）

```
          . INCLUDE    "FILE. H"
ON:       . EQU    1
OFF:      . EQU    0

          . SECTION   CD1, CODE, ALIGN=4
          MOV   #ON, R0

          ～
```

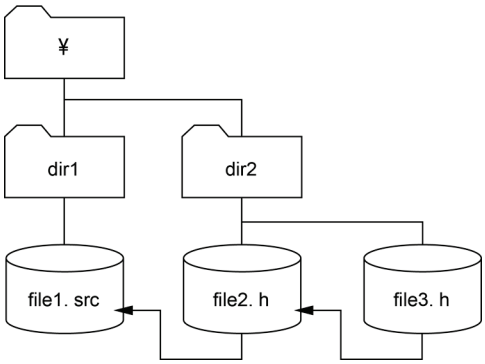
指定インクルードファイル

.INCLUDE

書　式　　△.INCLUDE△"<ファイル名>"
ラベルは記述できません。

説　明　　指定したインクルードファイルを取り込みます。
ファイル名として主ファイル名だけを指定した場合、ファイル拡張子なしのファイル名が有効となります (アセンブラによるファイル拡張子の仮定なし)。
ファイル名はフォルダを含めた形で指定することができます。
フォルダは絶対パス (ルートフォルダからの経路) または相対パス (カレントフォルダからの経路) で指定します。
インクルードファイルの中へさらに別のファイルを取り込むこともできます。インクルードは 30 段階までネストすることができます。
なお、ソースファイル中の.INCLUDE のカレントフォルダはアセンブラを起動したときのフォルダとなります。インクルードファイル中の.INCLUDE のカレントフォルダはそのインクルードファイルが存在するフォルダとなります。
.INCLUDE で指定したフォルダ名は include オプションで変更することができます。

例　　フォルダが下図のような構造になっているとき、以下のことを実行するとします。



ルートフォルダ (¥) からアセンブラを起動
入力ソースファイルは¥dir1¥file1.src
file1.src に file2.h をインクルード
file2.h に file3.h をインクルード
起動コマンドは次のようになります。

```
>asmsh ¥dir1¥file1.src [RET]
```

file1.src にはつぎのインクルード制御文が必要になります。

```
.INCLUDE "dir2¥file2.h"  ; ¥がカレントフォルダです (相対パス指定)。
または
.INCLUDE "¥dir2¥file2.h" ; 絶対パス指定
```

file2.h にはつぎのインクルード制御文が必要になります。

```
.INCLUDE "file3.h"   ; ¥dir2 がカレントフォルダです (相対パス指定)。
または
.INCLUDE "¥dir2¥file3.h" ; 絶対パス指定
```

11.6 条件つきアセンブリ機能

11.6.1 条件つきアセンブリ機能の概要

条件つきアセンブリ機能は次のようなアセンブルを簡単に実現します。

- ソースプログラムの文字列を他の文字列に置き換える
- ソースプログラムの一部分をアセンブルするか否か条件によって切り替える
- ソースプログラムの一部分を繰り返し展開してアセンブルする

(1) プリプロセッサ変数

アセンブル条件を記述するための変数をプリプロセッサ変数といいます。プリプロセッサ変数の型には整数型と文字型があります。

(a) 整数型プリプロセッサ変数

.ASSIGNA で定義します(再定義が可能)。プリプロセッサ変数の先頭にバックスラッシュ(¥)とアンパサンド(&)を付けることにより内容を参照できます。

■コーディング例

```
FLAG: .ASSIGNA  1
       ～
      .AIF ¥&FLAG EQ 1          ; FLAG が 1 のとき
      MOV    R0,R1              ; MOV R0,R1 をアセンブルします。
      .AENDI
       ～
```

(b) 文字型プリプロセッサ変数

.ASSIGNC で定義します(再定義が可能)。プリプロセッサ変数の先頭にバックスラッシュ(¥)とアンパサンド(&)を付けることにより内容を参照できます。

■コーディング例

```
FLAG: .ASSIGNC  "ON"
       ～
      .AIF "¥&FLAG" EQ "ON"     ; FLAG が"ON"のとき
      MOV    R0,R1              ; MOV R0,R1 をアセンブルします。
      .AENDI
       ～
```

(2) 置換シンボル

.DEFINE で定義します。ソースプログラムの一部分を指定によって置き換えることができます。コーディングは次のようになります。

■コーディング例

```
SYM1: .DEFINE "R1"
        ～
      MOV.L  SYM1,R0            ; MOV.L R1,R0 に置き換えられます。
        ～
```

(3) 条件つきアセンブル

ソースプログラムの一部分をアセンブルするか否か条件によって切り替えることができます。

条件つきアセンブリの条件には関係演算子で判別する比較型条件つきアセンブルと置換シンボルで判別する定義型条件つきアセンブルがあります。

(a) 比較型条件つきアセンブル

比較型条件つきアセンブルは条件の成立か不成立かによりアセンブルする範囲を切り替えます。コーディングは次のようになります。

```
          ～
.AIF      比較型条件
    条件が成立したときアセンブルする部分
.AELIF    比較型条件
    条件が成立したときアセンブルする部分          この部分は省略可能
.AELSE
    全ての条件が成立しないときアセンブルする部分
.AENDI
          ～
```

■コーディング例

```
  ～
.AIF  "¥&FLAG" EQ "ON"
MOV    R0,R10            ; FLAG が
MOV    R1,R11            ;   "ON"のとき
MOV    R2,R12            ;     アセンブルします。
.AELSE
MOV    R10,R0            ; FLAG が
MOV    R11,R1            ;   "ON"でないとき
MOV    R12,R2            ;     アセンブルします。
.AENDI
  ～
```

(b) 定義型条件つきアセンブル

定義型条件つきアセンブルは置換シンボルが定義されているか否かによりアセンブルする範囲を切り替えます。コーディングは次のようになります。

```
        ～
.AIFDEF   定義型条件
    条件の置換シンボルが定義されているとき
    アセンブルする部分
.AELSE
    条件の置換シンボルが定義されていないとき      この部分は省略可能
    アセンブルする部分
.AENDI
        ～
```

■コーディング例

```
  ～
.AIFDEF  FLAG
MOV    R0,R10            ; .AIFDEF で参照するより前に
MOV    R1,R11            ; FLAG が.DEFINE で定義されているとき
MOV    R2,R12            ; アセンブします。
.AELSE
MOV    R10,R0            ; .AIFDEF で参照するより前に
MOV    R11,R1            ; FLAG が.DEFINE で定義されていないとき
MOV    R12,R2            ; アセンブします。
.AENDI
  ～
```

(4) 繰り返し展開

ソースプログラムの一部分を指定の回数だけ繰り返し展開してアセンブルすることができます。コーディングは次のようになります。

```
        ～
.AREPEAT    繰り返し回数
    繰り返しの対象となる部分
.AENDR
        ～
```

■コーディング例

```
; 64 ビット÷32 ビットの除算を例に挙げます。
; R1:R2(64 ビット)÷R0(32 ビット)＝R2(32 ビット)：符号無し
    TST    R0,R0            ; ゼロ除算チェック
    BT     zero_div
    CMP/HS R0,R1            ; オーバフローチェック
    BT     over_div
    DIV0U                   ; フラグの初期化
    .AREPEAT  32
    ROTCL  R2               ; 32 回繰り返してアセンブします。
```

```
        DIV1   R0,R1              ;
        .AENDR
        ROTCL  R2                 ; R2=商
```

(5) 条件つき繰り返し展開

ソースプログラムの一部分を条件が成立している間、繰り返し展開してアセンブルすることができます。コーディングは次のようになります。

```
                  ～
        .AWHILE      繰り返し条件
            繰り返しの対象となる部分
        .AENDW
                  ～
```

■コーディング例

```
; 積和演算を例に挙げます。
TblSiz: .ASSIGNA  50            ; TblSiz：データテーブルの大きさ
        MOV     A_Tbl1,R1       ; R1：データテーブル 1 の先頭アドレス
        MOV     A_Tbl2,R2       ; R2：データテーブル 2 の先頭アドレス
        CLRMAC                  ; MAC レジスタの初期化
        .AWHILE ¥&TblSiz GT 0   ; TblSiz が 0 より大きい間、
        MAC.W  @R1+,@R2+        ; 積和演算を繰り返してアセンブルします。
TblSiz: .ASSIGNA  ¥&TblSiz-1    ; TblSiz から 1 を引きます。
        .AENDW
        STS     MACL,R0         ; 結果を R0 に得ます。
```

11.6.2 条件つきアセンブリ機能に関する制御文

条件つきアセンブリ機能の制御文には次のものがあります。

分類	ニーモニック	機能
変数定義に関するもの	.ASSIGNA	整数型プリプロセッサ変数を定義します。 再定義が可能です。
	.ASSIGNC	文字型プリプロセッサ変数を定義します。 再定義が可能です。
	.DEFINE	プリプロセッサ置換文字列を定義します。 再定義できません。
条件分岐に関するもの	.AIF .AELIF .AELSE .AENDI	ソースプログラムの一部分をアセンブルするか否か、条件によって切り替えます。 条件成立の場合は.AIF 以降、不成立の場合は.AELIF または.AELSE 以降のソースプログラムをアセンブルします。
	.AIFDEF .AELSE .AENDI	ソースプログラムの一部分をアセンブルするか否か、置換シンボルの定義によって切り替えます。 置換シンボルが定義されている場合は.AIFDEF 以降、定義されていない場合は.AELSE 以降のソースプログラムをアセンブルします。
繰り返し展開に関するもの	.AREPEAT .AENDR	ソースプログラムの一部分(.AREPEAT と.AENDR の間)を指定の回数だけ繰り返し展開してアセンブルします。
	.AWHILE .AENDW	ソースプログラムの一部分(.AWHILE と.AENDW の間)を条件が成立している間、繰り返し展開してアセンブルします。
	.EXITM	.AREPEAT、.AWHILE による繰り返し展開を中断します。
その他	.AERROR	プリプロセッサ展開時のエラー処理をします。
	.ALIMIT	プリプロセッサでの.AWHILE の展開の上限値を設定します。

整数型プリプロセッサ変数定義

.ASSIGNA

書　式　　<プリプロセッサ変数名>[:]△.ASSIGNA△<値>

説　明　　プリプロセッサ変数を定義します。
プリプロセッサ変数名の付け方はシンボルの名付け方と同じです。
また、プリプロセッサ変数名の最大文字数は 32 文字で英大文字と英小文字を区別します。
.ASSIGNA で定義したプリプロセッサ変数は.ASSIGNA によって再定義できます。
プリプロセッサ変数の値は次の形式で指定します。
・定数 (整数定数、文字定数)
・既に定義したプリプロセッサ変数
・上記を項とする式
定義したプリプロセッサ変数は本制御文以降のソースステートメントに対して有効です。
プリプロセッサ変数は以下の箇所で参照できます。
・.ASSIGNA
・.ASSIGNC
・.AIF
・.AELIF
・.AREPEAT
・.AWHILE
・マクロ本体(.MACRO～.ENDM 間のソースステートメント)
プリプロセッサ変数を参照する場合、前にバックスラッシュ (¥) とアンパサンド (&) を付けて記述してください。
　¥&プリプロセッサ変数名[']
アポストロフィ (') はプリプロセッサ変数名とソースステートメントの区別を明確にしたい場合に記述します。
オプションでプリプロセッサ文字列が定義されている場合、同名のプリプロセッサ変数に対する.ASSIGNA は無効となります。

例

```
; 汎用多ビットシフト命令を作ります
; SHIFT の値だけ右にシフトします
Rn:     .REG  R0                    ;Rn に R0 を設定します。
SHIFT: .ASSIGNA 27                  ;SHIFT に 27 を設定します。

       .AIF ¥&SHIFT GE 16           ;条件：SHIFT≧16
       SHLR16      Rn               ;条件成立なら Rn を右に 16 ビットシフト。
SHIFT: .ASSIGNA ¥&SHIFT-16          ;SHIFT から 16 を減じます。
       .AENDI

       .AIF ¥&SHIFT GE 8            ;条件：SHIFT≧8
       SHLR8       Rn               ;条件成立なら Rn を右に 8 ビットシフト。
SHIFT: .ASSIGNA ¥&SHIFT-8           ;SHIFT から 8 を減じます。
       .AENDI

       .AIF ¥&SHIFT GE 4            ;条件: SHIFT≧4
       SHLR2       Rn               ;条件成立なら Rn を右に 4 ビットシフト。
       SHLR2       Rn               ;
SHIFT: .ASSIGNA ¥&SHIFT-4           ;SHIFT から 4 を減じます。
       .AENDI

       .AIF ¥&SHIFT GE 2            ;条件：SHIFT≧2
       SHLR2       Rn               ;条件成立なら Rn を右に 2 ビットシフト。
SHIFT: .ASSIGNA ¥&SHIFT-2           ;SHIFT から 2 を減じます。
       .AENDI

       .AIF ¥&SHIFT EQ 1            ;条件：SHIFT=1
       SHLR        Rn               ;条件成立なら Rn を右に 1 ビットシフト。
       .AENDI
```

展開結果は次のようになります。

```
       SHLR16      R0               ;条件成立なら Rn を右に 16 ビットシフト。
       SHLR8       R0               ;条件成立なら Rn を右に 8 ビットシフト。
       SHLR2       R0               ;条件成立なら Rn を右に 2 ビットシフト。
       SHLR        R0               ;条件成立なら Rn を右に 1 ビットシフト。
```

文字型プリプロセッサ変数定義

.ASSIGNC

書　式　　<プリプロセッサ変数名>[:]△.ASSIGNC△"<文字列>"

説　明　　文字型プリプロセッサ変数を定義します。
プリプロセッサ変数名の付け方はシンボルの名付け方と同じです。
また、プリプロセッサ変数名の最大文字数は 32 文字で英大文字と英小文字を区別します。
.ASSIGNC で定義したプリプロセッサ変数は.ASSIGNC によって再定義できます。
文字列は文字および既に定義したプリプロセッサ変数をダブルクォーテーション(")で囲んで指定します。
定義したプリプロセッサ変数は本制御文以降のソースステートメントに対して有効です。
プリプロセッサ変数は以下の箇所で参照できます。

・.ASSIGNA
・.ASSIGNC
・.AIF
・.AELIF
・.AREPEAT
・.AWHILE
・マクロ本体(.MACRO～.ENDM 間のソースステートメント)

プリプロセッサ変数を参照する場合、前にバックスラッシュ(¥)とアンパサンド(&)を付けて記述してください。

¥&プリプロセッサ変数名[']

アポストロフィ(')はプリプロセッサ変数名とソースステートメントの区別を明確にしたい場合に記述します。
オプションでプリプロセッサ文字列が定義されている場合、同名のプリプロセッサ変数に対する.ASSIGNC は無効となります。

例

```
FLAG:  .ASSIGNC  "ON"                ;FLAG に"ON"を設定します。
       ～
       .AIF "¥&FLAG" EQ "ON"         ;FLAG が"ON"のとき、
       MOV        R0,R1              ;MOV R0,R1 をアセンブルします。
       .AENDI
       ～
FLAG:  .ASSIGNC  "¥&FLAG "           ; FLAG に空白文字(" ")を付加します。
FLAGA: .ASSIGNC  "OFF"               ; FLAGA に"OFF"を設定します。
FLAG:  .ASSIGNC  "¥&FLAG'AND ¥&FLAGA"
                   ;FLAG と AND の区別を明確にするため"'"を使います。
                   ;FLAG は結果的に"ON AND OFF"になります。
       ～
```

プリプロセッサ置換文字列定義

.DEFINE

書　式　　<シンボル>[:]△.DEFINE△"<置換文字列>"

説　明　　シンボルの対応する置換文字列に置き換えることを指定します。
.DEFINE と.ASSIGNC との違いは以下の点です。
・.ASSIGNC で定義したシンボルはプリプロセッサ文でしか使用できませんが、.DEFINE で定義したシンボルは任意のステートメントで使用できます。
・.ASSIGNA、.ASSIGNC で定義したシンボルは「¥&シンボル」の形式で参照しますが、.DEFINE で定義したシンボルは「シンボル」の形式で参照します。
.DEFINE で定義したシンボルは再定義できません。
オプションで置換シンボルが定義されている場合、同名のシンボルに対する.DEFINE は無効となります。
また、".AENDI",".AENDR",".AENDW",".AIFDEF",".END",".ENDM"の 6 制御命令は置換されません。

例

```
SYM1:  .DEFINE  "R1"
       ～
       MOV.L      SYM1,R0          ;MOV.L R1,R0 に置き換えられます。
       ～
```

先頭が a～f または A～F で始まる 16 進数は.DEFINE で同名のシンボルが定義された場合、置換対象になります。置換対象外にするには先頭に 0 を付加してください。

```
C0:  .DEFINE  "0"
       MOV.B      #H'C0,R0         ;MOV.B #H'0,R0 に置き換えられます。
       MOV.B      #H'0C0,R0        ;置き換えられません。
```

基数(B'、Q'、D'、H')は.DEFINE で同名のシンボルが定義された場合、置換対象になります。B、Q、D、H、b、q、d、h 一文字のシンボルを定義するときは注意してください。

```
B:  .DEFINE  "H"
       MOV.B      #B'10,R0         ;MOV.H #H'10,R0 に置き換えられます。
```

比較型条件つきアセンブル

.AIF, .AELIF, .AELSE, .AENDI

書　式

```
△.AIF△<項 1>△<関係演算子>△<項 2>
    <.AIF の条件成立時にアセンブルするソースステートメント>
[△.AELIF△<項 1>△<関係演算子>△<項 2>
    <.AELIF の条件成立時にアセンブルするソースステートメント>]
[△.AELSE
    <全ての条件不成立時にアセンブルするソースステートメント>]
 △.AENDI
```

説　明

.AIF～.AELIF～.AELSE～.AENDI 間に記述したソースステートメントのうち、条件が成立した部分をアセンブルします。
.AELIF と.AELSE は省略できます。
.AELIF は.AIF と.AELSE の間ならば繰り返し指定できます。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AIF	比較型条件
.AELIF	比較型条件
.AELSE .AENDI	記述不可能

<項 1>、<項 2>には値または文字列を記述します。ただし、値と文字列を比較すると常に条件不成立となります。
値は定数またはプリプロセッサ変数で指定します。
文字列は文字またはプリプロセッサ変数をダブルクォーテーション (") で囲んで指定します。
ダブルクォーテーション (") 自体を文字として指定する場合はダブルクォーテーションを 2 つ続けて記述 ("") します。

関係演算子の条件は以下のとおりです。

関係演算子	条件
EQ	項 1 ＝ 項 2
NE	項 1 ≠ 項 2
GT	項 1 ＞ 項 2
LT	項 1 ＜ 項 2
GE	項 1 ≧ 項 2
LE	項 1 ≦ 項 2

【注】 値は 32 ビット符号つき整数として比較します。
文字列の比較は EQ、NE のみ有効です。

例

```
～
.AIF ¥&TYPE EQ 1
MOV        R0,R3            ;TYPE が
MOV        R1,R4            ;  1 のとき
MOV        R2,R5            ;    アセンブルします。
.AELIF ¥&TYPE EQ 2
MOV        R0,R6            ;TYPE が
MOV        R1,R7            ;  2 のとき
MOV        R2,R8            ;    アセンブルします。
.AELSE
MOV        R0,R9            ;TYPE が
MOV        R1,R10           ;  1 でも 2 でもないとき
MOV        R2,R11           ;    アセンブルします。
.AENDI
～
```

定義型条件つきアセンブル

.AIFDEF, .AELSE, .AENDI

書　式

```
△.AIFDEF△<置換シンボル>
    <置換シンボルが定義されていた時にアセンブルするソースステートメント>
[△.AELSE
    <置換シンボルが定義されていない時にアセンブルするソースステートメント>]
 △.AENDI
```

ラベルは記述できません。

説　明

.AIFDIF、.AELSE、.AENDI はアセンブルするか否かを置換シンボルの定義によって切り替えます。.AELSE は省略できます。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AIFDEF	定義型条件
.AELSE	記述不可能
.AENDI	

置換シンボルは.DEFINE で定義します。
記述した置換シンボルがオプションで定義されている、または本制御文で参照するより前に定義している場合、条件成立となります。記述した置換シンボルが本制御文で参照した後に定義している、または定義がない場合、条件不成立となります。

例

```
～
.AIFDEF  FLAG
MOV        R0,R3            ;FLAG が.DEFINE で定義されて
MOV        R1,R4            ;いるときアセンブルします。
.AELSE
MOV        R0,R6            ;FLAG が.DEFINE で定義されて
MOV        R1,R7            ;いないときアセンブルします。
.AENDI
～
```

繰り返し展開

.AREPEAT, .AENDR

書　式

△.AREPEAT△<回数>
　<繰り返し展開してアセンブルするソースステートメント>
△.AENDR

ラベルは記述できません。

説　明

指定された回数だけ繰り返し展開してアセンブルします。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.AREPEAT	繰り返し展開する回数
.AENDR	記述不可能

.AREPEATで指定された回数だけ.AREPEAT～.AENDRの間に記述したソースステートメントを繰り返し展開してアセンブルします(ソースステートメントを繰り返しコピーするのと同じで実行時のループにはなりません)。
回数は定数またはプリプロセッサ変数で指定します。
回数に0以下の値を指定した場合は展開しません。

例

```
; 64ビット÷32ビットの除算を例に挙げます。
; R1:R2(64ビット)÷R0(32ビット)=R2(32ビット)：符号無し
        TST         R0,R0           ;ゼロ除算チェック
        BT          zero_div
        CMP/HS      R0,R1           ;オーバフローチェック
        BT          over_div
        DIV0U                       ;フラグの初期化
        .AREPEAT  32
        ROTCL       R2              ;32回繰り返してアセンブルします。
        DIV1        R0,R1           ;
        .AENDR
        ROTCL       R2              ;R2=商
```

条件つき繰り返し展開

.AWHILE, .AENDW

書　式　　△.AWHILE△<項 1>△<関係演算子>△<項 2>
　　<繰り返し展開してアセンブルするソースステートメント>
△.AENDW
ラベルは記述できません。

説　明　　条件が成立している間だけ繰り返し展開してアセンブルします。
.AWHILE で指定した条件が成立している間、.AWHILE～.AENDW の間に記述したソースステートメントを繰り返し展開してアセンブルします(ソースステートメントを繰り返しコピーするのと同じで実行時のループにはなりません)。
<項 1>、<項 2>は値または文字列を記述します。ただし、値と文字列を比較すると常に条件不成立となります。
値は定数またはプリプロセッサ変数で指定します。
文字列は文字またはプリプロセッサ変数をダブルクォーテーション(")で囲んで指定します。
ダブルクォーテーション(")自体を文字として指定する場合は、ダブルクォーテーションを 2 つ続けて記述("")します。
条件つき繰り返し展開は最終的に条件を不成立にして展開を終了します。
条件が不成立にならない場合は 65,535 回または.ALIMIT で指定した展開回数を繰り返しますので条件の指定にはよく注意してください。
関係演算子の条件は以下のとおりです。

関係演算子	条件
EQ	項 1 ＝ 項 2
NE	項 1 ≠ 項 2
GT	項 1 ＞ 項 2
LT	項 1 ＜ 項 2
GE	項 1 ≧ 項 2
LE	項 1 ≦ 項 2

【注】 値は 32 ビット符号つき整数として比較します。
　　文字列の比較は EQ、NE のみ有効です。

例

```
; 積和演算を例に挙げます。
TblSiz: .ASSIGNA  50                ;TblSiz：データテーブルの大きさ
        MOV       A_Tbl1,R1         ;R1：データテーブル 1 の先頭アドレス
        MOV       A_Tbl2,R2         ;R2：データテーブル 2 の先頭アドレス
        CLRMAC                      ;MAC レジスタの初期化
        .AWHILE ¥&TblSiz GT 0       ;TblSiz が 0 より大きい間、
        MAC.W     @R0+,@R1+         ;積和演算を繰り返してアセンブルします。
TblSiz: .ASSIGNA  ¥&TblSiz-1        ;TblSiz から 1 を引きます。
        .AENDW
        STS       MACL,R0           ;結果を R0 に得ます。
```

.EXITM

書　式　　△.EXITM
ラベルは記述できません。

説　明　　繰り返し展開(.AREPEAT～.AENDR)および条件つき繰り返し展開(.AWHILE～.AENDW)の展開を中断させます。
各展開では本制御文が出現した時点で展開を中断します。
本制御文はマクロ展開の中断終了にも使用します。マクロ命令と繰り返し展開を組み合わせて使用する場合は本制御文の位置に注意してください。

例

```
        ～
COUNT: .ASSIGNA  0                  ;COUNT に 0 を設定しています。
       .AWHILE 1 EQ 1               ;無限展開（常に条件成立）を指定しています。
       ADD        R0,R1
       ADD        R2,R3

COUNT: .ASSIGNA  ¥&COUNT+1          ;COUNT に 1 を加えます。
       .AIF ¥&COUNT EQ 2            ;条件は COUNT=2 です。
       .EXITM                       ;条件成立で.AWHILE を中断終了します。
       .AENDI
       .AENDW
       ～
```

COUNT が更新され、.AIF の条件が成立すると.EXITM がアセンブルされます。
.EXITM がアセンブルされた時点で.AWHILE の展開を中断終了します。
展開結果は以下のようになります。

```
       ADD        R0,R1    … COUNT が 0 のとき
       ADD        R2,R3
       ADD        R0,R1    … COUNT が 1 のとき
       ADD        R2,R3
```

この後、COUNT は 2 となり、展開は中断終了します。

プリプロセッサ展開時エラー処理

.AERROR

書　式　　△.AERROR
ラベルは記述できません。

説　明　　.AERROR をアセンブルするとエラー667 を発生し、アセンブラをエラー終了します。
.AERROR はプリプロセッサ変数の値のチェック等に使用することができます。

例

```
～
.AIF ¥&FLG EQ 1
MOV        R1,R10
MOV        R2,R11
.AELSE
.AERROR    ;¥&FLG が 1 以外の場合エラーとします。
.AENDI
～
```

展開上限値設定

.ALIMIT

書　式　　△.ALIMIT△<回数>
ラベルは記述できません。

説　明　　条件つき繰り返し展開(.AWHILE～.AENDW)で、ステートメントの展開回数の上限値を設定します。
<回数>の値は次の形式で指定します。
・定数(整数定数、文字定数)
・既に定義したプリプロセッサ変数
・上記を項とする式
.ALIMIT で指定した上限値を越えるとウォーニング 854 となり、展開を打ち切ります。
展開回数の限界値は.ALIMIT を指定しないとき、65,535 です。
繰り返し展開回数の上限値は、本制御命令で再指定することで値を変更できます。
上限値の再指定は、本制御命令以降のソースステートメントに対して有効です。

例

```
        .ALIMIT  20
        ～
FLG:  .ASSIGNA  0
        .AWHILE ¥&FLG EQ 0          ;20 回展開した後、展開を打ち切り
        NOP                         ;ウォーニングとします。
        .AENDW
        ～
```

11.7　マクロ機能

11.7.1　マクロ機能の概要

本アセンブリ言語ではプログラム中でよく使用する一連の処理に名前をつけ、1 つの命令(マクロ命令)として定義することができます。このような定義をマクロ定義といいます。マクロ定義の方法は次のとおりです。

```
            ～
.MACRO      マクロ名
    マクロ本体
.ENDM
            ～
```

マクロ名はマクロ命令につける名前、マクロ本体はマクロ命令の内容です。

定義したマクロ命令を呼び出して使用することをマクロコールといいます。マクロコールの方法は次のとおりです。

```
            ～
定義済みのマクロ名
            ～
```

マクロ定義とマクロコールの例を以下に示します。

■コーディング例

```
  ～
.MACRO  SUM              ; R0,R1,R2,R3 の合計を求める処理を
MOV     R0,R10           ; マクロ命令 SUM として定義します。
ADD     R1,R10
ADD     R2,R10
ADD     R3,R10
.ENDM
  ～

SUM                      ; マクロ命令 SUM を呼び出します。
                         ; マクロ本体
                         ;       MOV    R0,R10
                         ;       ADD    R1,R10
                         ;       ADD    R2,R10
                         ;       ADD    R3,R10
                         ; が展開されます。
```

定義したマクロ命令を一部変更して展開することも可能です。手順は次のとおりです。

(1) マクロ定義
.MACRO文で仮引数を定義(マクロ名につづいて記述)します。
マクロ本体の記述に仮引数を使います(仮引数の先頭にバックスラッシュ(¥)を付けます)。

(2) マクロコール
マクロパラメータを付けてマクロ命令を呼び出します。

マクロ命令展開の際、仮引数は対応するマクロパラメータに置き換えられます。

■コーディング例

```
  ～
.MACRO  SUM ARG1          ; 仮引数 ARG1 を定義します。
MOV     R0,¥ARG1          ; ARG1 を使ってマクロ本体を記述しています。
ADD     R1,¥ARG1
ADD     R2,¥ARG1
ADD     R3,¥ARG1
.ENDM
  ～
SUM     R10               ; マクロパラメータ R10 を付けてマクロ命令 SUM を呼び出します。
                          ; マクロ本体中の仮引数がマクロパラメータで置き換えられ、
                          ;       MOV    R0,R10
                          ;       ADD    R1,R10
                          ;       ADD    R2,R10
                          ;       ADD    R3,R10
                          ; が展開されます。
```

11.7.2　マクロ機能に関する制御文

マクロ機能の制御文には次のものがあります。

ニーモニック	機能
.MACRO .ENDM	マクロ命令を定義します。
.EXITM	マクロ命令の展開を中断します。 「11.6.2 .EXITM」を参照してください。

マクロ命令定義

.MACRO, .ENDM

書　式　　△.MACRO△<マクロ名>[△<仮引数>[,...]]
△.ENDM
<仮引数>：<仮引数名>[= <仮引数のデフォルト>]
ラベルは記述できません。

説　明　　マクロ定義(一連のソースステートメントに名前を付けてまとめて扱うこと)を行ないます。
.MACRO～.ENDM 間のソースステートメント(マクロ本体)をマクロ命令として名前を付けることを、マクロ命令を定義する(マクロ定義)といいます。
各オペレーションで指定できるオペランドは次のようになります。

オペレーション	オペランド
.MACRO	・マクロ命令 ・仮引数 デフォルトを記述(省略可)
.ENDM	記述不可能

(1) マクロ名
マクロ名はマクロ命令に付ける名前です。
マクロ命令を展開する際にマクロ本体の一部を置換して展開したい場合に指定します。
仮引数はマクロ展開を行なう(マクロコール)時に指定された文字列(マクロパラメータ)に置換されます。
マクロ本体では置換したい部分に仮引数名を記述します。マクロ本体での仮引数の参照方法は次のとおりです。
¥仮引数名['']
アポストロフィ('')は、仮引数名とソースステートメントの区別を明確にしたい場合に記述します。

(2) 仮引数
仮引数には仮引数のデフォルトを設定できます。仮引数のデフォルトにはマクロコール時にマクロパラメータを省略した場合に置換する文字列を指定します。
仮引数の書き方はシンボル名の書き方と同じです。
仮引数名の最大文字数は 32 文字で英大文字と英小文字を区別します。

(3) 仮引数のデフォルト
仮引数のデフォルトに次の文字を含む場合は文字列をダブルクォーテーション("))またはアングルブラケット(<>)で囲んでください。
・空白文字
・タブ
・カンマ(,)
・セミコロン(;)
・ダブルクォーテーション(")
・アングルブラケット(<>)
マクロ展開では文字列を囲んだダブルクォーテーション("))やアングルブラケット(<>)は取り除いて置換します。

(4) 制限

マクロ命令は次の場所では定義できません。

・マクロ本体(.MACRO～.ENDM)

・.AREPEAT～.AENDR の間

・.AWHILE～.AENDW の間

マクロ本体には.END を記述できません。

.ENDM のラベルにはシンボルを記述できません。

.ENDM のラベルにシンボルを記述した場合は.ENDM を無視します。この場合、エラーは表示しません。

例

```
～
.MACRO  SUM                 ;R0,R1,R2,R3 の合計を求める処理を
MOV         R0,R10          ;マクロ命令 SUM として定義します。
ADD         R1,R10
ADD         R2,R10
ADD         R3,R10
.ENDM
～
SUM                         ;マクロ命令 SUM を呼び出します。
                            ;マクロ本体
                            ;       MOV         R0,R10
                            ;       ADD         R1,R10
                            ;       ADD         R2,R10
                            ;       ADD         R3,R10
                            ;が展開されます。
```

11.7.3 マクロ本体

.MACRO と.ENDM の間に記述した一連のソースステートメントをマクロ本体と呼びます。マクロ本体はマクロコール (マクロ命令を呼び出すこと) により、展開してアセンブルされます。本節ではマクロ本体が持つ機能と記述方法を説明します。

(1) 仮引数の参照

マクロ展開でマクロパラメータと置換したい部分に仮引数を記述します。仮引数の参照方法は以下のとおりです。

¥仮引数 ['']

アポストロフィ(')は仮引数名とソースステートメントの区別を明確にしたい場合に記述します。

■コーディング例

```
      .MACRO  PLUS1 P,P1        ; P,P1 は仮引数です。
      ADD    #1,¥P1             ; 仮引数 P1 を参照しています。
      .SDATA "¥P'1"             ; 仮引数 P を参照しています。
      .ENDM
      PLUS1  R,R1               ; PLUS1 を展開します。
```

展開結果は次のようになります。

```
      ADD    #1,R1              ; 仮引数 P1 を参照しています。
      .SDATA "R1"               ; 仮引数 P を参照しています。
```

(2) プリプロセッサ変数の参照

マクロ本体ではプリプロセッサ変数を参照できます。プリプロセッサ変数の参照方法は次のとおりです。

¥&プリプロセッサ変数名 ['']

アポストロフィ(')はプリプロセッサ変数とソースステートメントの区別を明確にしたい場合に記述します。

■コーディング例

```
      .MACRO  PLUS1
      ADD    #1,R¥&V1           ; プリプロセッサ変数 V1 を参照しています。
      .SDATA "¥&V'1"            ; プリプロセッサ変数 V を参照しています。
      .ENDM
V:  .ASSIGNC  "R"               ; プリプロセッサ変数 V を定義しています。
V1: .ASSIGNA  1                 ; プリプロセッサ変数 V1 を定義しています。
      PLUS1                     ; PLUS1 を展開します。
```

展開結果は次のようになります。

```
      ADD    #1,R1              ; プリプロセッサ変数 V1 を参照しています。
      .SDATA "R1"               ; プリプロセッサ変数 V を参照しています。
```

(3) マクロ生成番号

マクロ本体にラベルがある場合などは、複数回マクロコールをするとシンボル名が重複してしまいます。このような事態を回避するためにはマクロ生成番号を使用してください。マクロ生成番号はマクロ展開で固有の 5 桁の 10 進数(00000～99999)を展開します。マクロ生成番号は次のように記述してください。

¥@

シンボル名の一部としてマクロ生成番号を記述しておくと、マクロコールのたびに固有のシンボル名となり、重複を避けることができます。1 つのマクロ本体に 2 つ以上のマクロ生成番号を記述できますが、1 回のマクロコールでは同じマクロ生成番号が展開されます。また、マクロ生成番号は数字に展開されるのでシンボル名の先頭には記述しないでください。

■コーディング例

```
     .MACRO  RES_STR STR,Rn
     MOV.L  #str¥@,¥Rn
     BRA    end_str¥@
     NOP
str¥@  .SDATA "¥STR"
     .ALIGN  2
end_str¥@
     .ENDM
     RES_STR "ONE",R0          ; RES_STR を展開するたびに
     RES_STR "TWO",R1          ; 異なるシンボルを生成します。
```

展開結果は次のようになります。

```
     MOV.L  #str00000,R0
     BRA    end_str00000
     NOP
str00000  .SDATA "ONE"
     .ALIGN  2
end_str00000
     MOV.L  #str00001,R1
     BRA    end_str00001
     NOP
str00001  .SDATA "TWO"
     .ALIGN  2
end_str00001
```

(4) マクロ処理除外

マクロ本体内にバックスラッシュ(¥)があるとマクロ置換処理の対象になります。したがって、バックスラッシュ(¥)を ASCII 文字として記述したい場合はマクロ置換処理から除外する必要があります。マクロ処理除外の書き方は次のとおりです。

¥(マクロ処理除外文字列)

マクロ展開ではバックスラッシュ(¥)とカッコは取り除きます。

■コーディング例

```
.MACRO BACK_SLASH_SET
¥(MOV  #"¥",R0)          ; ¥は ASCII 文字として展開されます。
.ENDM
```

展開結果は次のようになります。

```
MOV    #"¥",R0           ; ¥は ASCII 文字として展開されます。
```

(5) マクロ内コメント

マクロ本体のコメントをマクロ展開では展開したくない(リスティングファイルに同じコメントが何度も現れるのを避けたい)場合にマクロ内コメントを記述します。マクロ内コメントの書き方は次のとおりです。

¥;コメント

■コーディング例

```
.MACRO  PUSH Rn
MOV.L  ¥Rn,@-R15         ¥; ¥Rn はレジスタです。
.ENDM
PUSH   R0
```

展開結果は次のようになります(コメントは展開されません)。

```
MOV.L  R0,@-R15
```

(6) 文字列操作関数

マクロ本体には文字列操作関数を記述できます。文字列操作関数には次のものがあります。

.LEN 関数：文字列の文字数
.INSTR 関数：文字列の検索
.SUBSTR 関数：文字列の切り出し

11.7.4 マクロコール

マクロ定義により定義されたマクロ命令を展開することをマクロコールといいます。マクロコールの書き方は次のとおりです。

[<シンボル>[:]]△<マクロ名> [△<マクロパラメータ>[,…]]
<マクロパラメータ>:[= <仮引数名>] = <文字列>

マクロ名は、マクロコールする以前にマクロ定義(.MACRO)します。
マクロパラメータには、マクロ展開で置換する文字列を指定します。
この場合、マクロ名に対応するマクロ定義(.MACRO)で、仮引数を宣言しておく必要があります。

(1) マクロパラメータの指定方法

マクロパラメータの指定方法には、位置指定とキーワード指定があります。

(2) 位置指定

マクロ定義(.MACRO)で宣言した仮引数の並び順と、マクロパラメータの並び順を一致させて指定する方法です。

(3) キーワード指定

マクロ定義(.MACRO)で宣言した仮引数の仮引数名にイコール(=)で区切って指定する方法です。

(4) マクロパラメータの書き方

マクロパラメータに次の文字を含む場合は、文字列をダブルクォーテーション(")または、アングルブラケット(<>)で囲んでください。

- 空白文字
- タブ
- カンマ(,)
- セミコロン(;)
- ダブルクォーテーション(")
- アングルブラケット(<>)

マクロ展開では、文字列を囲んだダブルクォーテーションや、アングルブラケットは取り除いて置換します。

■コーディング例

```
        .MACRO  SUM FROM=0,TO=9  ; マクロ命令 SUM、仮引数 FROM,TO を定義します。
        MOV     R¥FROM,R10
COUNT   .ASSIGNA  ¥FROM+1
        .AWHILE ¥&COUNT LE ¥TO
        MOV     R¥&COUNT,R10        仮引数を用いてマクロ本体を記述しています。
COUNT   .ASSIGNA  ¥&COUNT+1
        .AENDW
        .ENDM
        SUM     0,5                 どちらも同じ展開結果になります。
        SUM     TO=5
```

マクロ本体中の仮引数がマクロパラメータで置き換えられ、展開結果は次のようになります。

```
        MOV     R0,R10
        MOV     R1,R10
        MOV     R2,R10
        MOV     R3,R10
        MOV     R4,R10
        MOV     R5,R10
```

11.7.5 文字列操作関数

マクロ本体で使用できる文字列操作関数には次のものがあります。

文字列操作関数	機能
.LEN	文字列の文字数を返します。
.INSTR	文字列の検索を行ないます。
.SUBSTR	文字列の切り出しを行ないます。

文字列の文字数

.LEN

書　式　　.LEN[△]("<文字列>")

説　明　　文字列の文字数を数え、基数を省略した 10 進数に置換します。
文字列は文字をダブルクォーテーション(")で囲んで指定します。
ダブルクォーテーション自体を文字として指定する場合は 2 つ続けて記述します。
文字列にはマクロの仮引数、プリプロセッサ変数を指定できます。
.LEN("¥仮引数名")
.LEN("¥&プリプロセッサ変数名")
本関数を記述できるのはマクロ本体(.MACRO～.ENDM)だけです。

例

```
～
.MACRO  RESERVE_LENGTH P1
.ALIGN  4
.SRES       .LEN("¥P1")
.ENDM
～
RESERVE_LENGTH ABCDEF
RESERVE_LENGTH ABC
```

展開結果は次のようになります。

```
.ALIGN  4
.SRES       6    ; "ABCDEF"の字数は 6 です。
.ALIGN  4
.SRES       3    ; "ABC"の字数は 3 です。
```

文字列の検索

.INSTR

書　式　　.INSTR[△]("<文字列１>","<文字列２>"[,<検索開始位置>])

説　明　　文字列 1 に文字列 2 が含まれているかを検索し、文字列の先頭を 0 とした検索位置を基数を省略した 10 進数に置換します。
文字列 1 に文字列 2 が含まれていない場合は-1 に置換します。
文字列は文字をダブルクォーテーション(")で囲んで指定します。
ダブルクォーテーション自体を文字として指定する場合は 2 つ続けて記述します。
検索開始位置は文字列 1 の先頭を 0 とした数値で指定します。省略した場合は 0 を設定します。
文字列、検索開始位置にはマクロの仮引数、プリプロセッサ変数を指定できます。

```
.INSTR("¥仮引数名", .... )
.INSTR("¥&プリプロセッサ変数名", .... )
```

本関数を記述できるのはマクロ本体(.MACRO～.ENDM)だけです。

例

```
～
.MACRO  FIND_STR P1
.DATA.W     .INSTR("ABCDEFG","¥P1",0)
.ENDM
～
FIND_STR   CDE
FIND_STR   H
```

展開結果は次のようになります。

```
.DATA.W     2    ; "ABCDEFG"の 2 文字目 (先頭を 0 とします) に"CDE"が
                   あります。
.DATA.W     -1   ; "ABCDEFG"の中に"H"はありません。
```

文字列の切り出し

.SUBSTR

書　式　　.SUBSTR[△]("<文字列>",<切り出しの開始位置>,<切り出しの長さ>)

説　明　　文字列の先頭を 0 とした切り出しの開始位置から、切り出しの長さ分の文字列を切り出し、ダブルクォーテーション(")で囲んだ文字列に置換します。
文字列は文字をダブルクォーテーションで囲んで指定します。
ダブルクォーテーション自体を文字として指定する場合は 2 つ続けて記述します。
切り出しの開始位置は 0 以上が指定できます。
切り出しの長さは 1 以上が指定できます。
切り出しの開始位置、切り出しの長さが不適当な場合には空文字("")に置換します。
切り出しの開始位置、切り出しの長さにはマクロの仮引数、プリプロセッサ変数を指定できます。

```
.SUBSTR("¥仮引数名", ....)
.SUBSTR("¥&プリプロセッサ変数名", ....)
```

本関数を記述できるのはマクロ本体(.MACRO～.ENDM)だけです。

例

```
～
.MACRO  RESERVE_STR P1=0,P2
.SDATA     .SUBSTR("ABCDEFG",¥P1,¥P2)
.ENDM
～
RESERVE_STR  2,2
RESERVE_STR  ,3                ;マクロパラメータ P1 を省略しています。
```

展開結果はそれぞれ次のようになります。

```
.SDATA     "CD"
.SDATA     "ABC"
```

11.8 リテラルプール自動生成機能

11.8.1 リテラルプール自動生成機能の概要

2 バイト長、4 バイト長の定数データ(以下、リテラルといいます)をレジスタへ転送するには、リテラルプール(リテラルの集まり)を確保し、PC 相対アドレス形式で参照しなければなりません。リテラルプールを配置する場合、次のような考慮をする必要があります。

- データ転送命令が参照できる範囲にデータが位置しているか？
- 2 バイト長のデータは 2 バイトアライメント､4 バイト長のデータは 4 バイトアライメントに位置しているか？
- 1 つのデータを複数のデータ転送命令で共有できないか？
- プログラム中のどこにリテラルプールを配置するか？

リテラルプール自動生成機能とは定数データのレジスタ転送に対応する PC 相対の MOV 命令(または MOVA 命令)と.DATA を 1 つの命令から自動的に生成する機能です。例えば、以下のプログラム(a)は本機能を用いれば(b)のように記述できます

■プログラム(a)

```
        MOV.L       DATA1,R0
        MOV.L       DATA2,R1
        ～
        .ALIGN      4
DATA1   .DATA.L     H'12345678
DATA2   .DATA.L     500000
```

■プログラム(b)

```
        MOV.L       #H'12345678,R0
        MOV.L       #500000,R1
        ～
```

11.8.2 リテラルプール自動生成機能に関する拡張命令

拡張命令(MOV.W #imm,Rn、MOV.L #imm,Rn、MOVA #imm,R0)の記述に対してアセンブラは必要なリテラルプールを自動生成し、PC 相対のディスプレースメント値を計算します。

表 11.34 に拡張命令のソースステートメントとその展開結果を示します。拡張命令のソースステートメントは 1 つの実行命令と 1 つのリテラルデータとに展開されます。

表 11.34 拡張命令の種類と展開結果

拡張命令		展開結果
MOV.W	#imm,Rn	MOV.W @(disp,PC),Rn と 2 バイト長のリテラルデータ
MOV.L	#imm,Rn	MOV.L @(disp,PC),Rn と 4 バイト長のリテラルデータ
MOVA	#imm,R0	MOVA @(disp,PC),R0 と 4 バイト長のリテラルデータ

11.8.3 リテラルプール自動生成機能のサイズモード

リテラルプール自動生成には、オペレーションサイズを指定したデータ転送命令(拡張命令)によりリテラルプールを自動生成するサイズ指定モード、オペレーションサイズの指定がないデータ転送命令をアセンブラが imm の値により適切な命令を選択するサイズ選択モードの 2 つのサイズモードがあります。

表 11.35 にデータ転送命令とサイズモードの関係を示します。

表 11.35 データ転送命令とサイズモードの関係

データ転送命令	サイズ指定モード	サイズ選択モード
MOV #imm,Rn	実行命令	アセンブラが選択
MOV.B #imm,Rn	実行命令	実行命令
MOV.W #imm,Rn	拡張命令	拡張命令
MOV.L #imm,Rn	拡張命令	拡張命令

(1) サイズ指定モード

サイズ指定モードではオペレーションサイズ指定がないデータ転送命令(MOV #imm,Rn)は通常の実行命令として扱われます。サイズ指定モードは auto_literal オプションが指定されていない場合に有効となります。

(2) サイズ選択モード

サイズ選択モードでは拡張命令のほかにオペレーションサイズを指定していないデータ転送命令(MOV #imm,Rn)の記述に対してアセンブラが imm の値の範囲を判定し、必要ならばリテラルプールを自動生成します。imm 値は符号付きの範囲で判定します。サイズ選択モードは auto_literal オプションを指定した場合に有効となります。

表 11.36 にサイズ選択モードで imm の値の範囲により選択される命令を示します。

表 11.36 サイズ選択モードで選択される命令

imm の指定方法	imm の値の範囲*	選択される命令
定数値 参照前に定義された定数シンボル 参照前に定義された絶対アドレスシンボル	H'FFFFFF80～H'0000007F (-128～127)	MOV.B #imm,Rn
	H'FFFF8000～H'FFFFFF7F (-32,768～-129) H'00000080～H'00007FFF (128～32,767)	MOV.W #imm,Rn 展開結果 (MOV.W @(disp,PC),Rn と 2 バイト長のリテラルデータ)
	H'80000000～H'FFFF7FFF (-2,147,483,648～-32,769) H'00008000～H'7FFFFFFF (32,768～2,147,483,647)	MOV.L #imm,Rn 展開結果 (MOV.L @(disp,PC),Rn と 4 バイト長のリテラルデータ)
相対アドレスシンボル 外部参照シンボル 参照後に定義された定数シンボル 参照後に定義された絶対アドレスシンボル	imm の値に依存しない	MOV.L #imm,Rn 展開結果 (MOV.L @(disp,PC),Rn と 4 バイト長のリテラルデータ)

【注】* ()内は 10 進表記

11.8.4 リテラルプールの出力

リテラルプールが出力されるポイントは次のどちらかです。

- 無条件分岐とそのディレイスロット命令に続く位置
- プログラマが .POOL を記述した位置

なお、出力ポイントは literal オプションで選択することができます。

アセンブラは拡張命令の記述位置以降で最も近い出力ポイントに対応するリテラルを出力します。アセンブラは同じポイントに出力するリテラルデータをまとめて 1 つのリテラルプールを生成します。

【注】ディレイスロット命令にラベルが指定されている場合、リテラルプール出力ポイントにはなりません。

(1) 無条件分岐を利用したリテラルプール出力

出力例を以下に示します。

■コーディング例

ソースプログラム

```
        . SECTION CD1, CODE, LOCATE=H' 0000F000
CD1_START:
        MOV. L      #H' FFFF0000, R0
        MOV. W      #H' FF00, R1
        MOV. L      #CD1_START, R2
        MOV         #H'FF, R3
        RTS
        MOV         R0, R10
        . END
```

↓↓↓↓

リテラルプール自動生成結果（ソースリスト）

```
 1  0000F000                          1          . SECTION CD1, CODE, LOCATE=H' 0000F000
 2  0000F000                          2     CD1_START
 3  0000F000  D003                    3          MOV. L      #H' FFFF0000, R0
 4  0000F002  9103                    4          MOV. W      #H' FF00, R1
 5  0000F004  D203                    5          MOV. L      #CD1_START, R2
 6  0000F006  E3FF                    6          MOV         #H'FF, R3
 7  0000F008  000B                    7          RTS
 8  0000F00A  6A03                    8          MOV         R0, R10
 9                                         ***** BEGIN-POOL *****
10  0000F00C  FF00                         DATA FOR SOURCE-LINE 4
11  0000F00E  0000                         ALIGNMENT CODE
12  0000F010  FFFF0000                     DATA FOR SOURCE-LINE 3
13  0000F014  0000F000                     DATA FOR SOURCE-LINE 5
14                                         ***** END-POOL *****
15                                    9          . END
```

(2) .POOL の位置へのリテラルプール出力

無条件分岐を利用したリテラルプール出力がディスプレースメントの範囲内に行なえない場合(プログラム中に無条件分岐が少ない場合など)アセンブラはエラー402 を出力します。このような場合、.POOL をディスプレースメントの範囲内に記述してください。

ディスプレースメントの範囲は次のとおりです。

- オペレーションサイズがワード(W)のとき：0～510 バイト
- オペレーションサイズがロング(L)のとき：0～1020 バイト

.POOL の位置へのリテラルプール出力ではそのリテラルプールを飛び越すための分岐命令も一緒に出力されます。

■コーディング例

ソースプログラム

```
        . SECTION CD1, CODE, LOCATE=H' 0000F000
CCD1_START
        MOV. L      #H' FFFF0000, R0
        MOV. W      #H' FF00, R1
        MOV. L      #CD1_START, R2
        MOV         #H'FF, R3
        . POOL
        . END
```

↓↓↓↓

リテラルプール自動生成結果（ソースリスト）

```
 1  0000F000                          1          . SECTION CD1, CODE, LOCATE=H' 0000F000
 2  0000F000                          2     CD1_START:
 3  0000F000  D003                    3          MOV. L      #H' FFFF0000, R0
 4  0000F002  9103                    4          MOV. W      #H' FF00, R1
 5  0000F004  D203                    5          MOV. L      #CD1_START, R2
 6  0000F006  E3FF                    6          MOV         #H' FF, R3
 7  0000F008                          7          . POOL
 8                                       ＊＊＊＊＊ BEGIN-POOL ＊＊＊＊＊
 9  0000F008  A006                          BRA TO END-POOL
10  0000F00A  0009                          NOP
11  0000F00C  FF00                          DATA FOR SOURCE-LINE 4
12  0000F00E  0000                          ALIGNMENT CODE
13  0000F010  FFFF0000                      DATA FOR SOURCE-LINE 3
14  0000F014  0000F000                      DATA FOR SOURCE-LINE 5
15                                       ＊＊＊＊＊ END-POOL ＊＊＊＊＊
16                                    8          . END
```

11.8.5 リテラルの共有

アセンブラは同一のリテラルプールに入る複数の拡張命令の imm を共有する処理を行ないます。以下に示すもので同一のものを共有します。

(1) シンボル
(2) 定数
(3) シンボル±定数

上記のほか、アセンブル時に同一の値を持つと判断できる式は共有する場合があります。imm の値が同じでも拡張命令のオペレーションサイズが異なる場合はリテラルデータを共有しません。また、出力先リテラルプールが異なる場合もデータは共有しません。複数の拡張命令が 1 つのリテラルデータを共有する例を以下に示します。

■コーディング例

ソースプログラム

```
        . SECTION CD1, CODE, LOCATE=H' 0000F000
CD1_START:
        MOV. L      #H' FFFF0000, R0
        MOV. W      #H' FF00, R1
        MOV. L      #H' FFFF0000, R2
        MOV         #H' FF, R3
        RTS
        MOV         R0, R10
        . END
```

↓↓↓↓

リテラルプール自動生成結果（ソースリスト）

```
 1  0000F000                          1          . SECTION CD1, CODE, LOCATE=H' 0000F000
 2  0000F000                          2      CD1_START:
 3  0000F000  D003                    3          MOV. L      #H' FFFF0000, R0
 4  0000F002  9103                    4          MOV. W      #H' FF00, R1
 5  0000F004  D202                    5          MOV. L      #H' FFFF0000, R2
 6  0000F006  E3FF                    6          MOV         #H' FF, R3
 7  0000F008  000B                    7          RTS
 8  0000F00A  6A03                    8          MOV         R0, R10
 9                                        ***** BEGIN-POOL *****
10  0000F00C  FF00                        DATA FOR SOURCE-LINE 4
11  0000F00E  0000                        ALIGNMENT CODE
12  0000F010  FFFF0000                    DATA FOR SOURCE-LINE 3, 5
13                                        ***** END-POOL *****
14                                    9          . END
```

11.8.6　リテラルプール出力の抑止

プログラム中に無条件分岐が多すぎると次のような問題が生じます。

- 小さなリテラルプールがいくつも出力される。
- リテラルを共有できない。

このような場合には次の方法でリテラルプール出力を抑止してください。

~
遅延分岐命令
ディレイスロット命令
.NOPOOL
~

■コーディング例

ソースプログラム

```
        ~
CASE1:
        MOV. L    #H' FFFF0000, R0   ----   拡張命令1
        RTS
        NOP                                 このポイントへの
        . NOPOOL                     ----   リテラルプール出力はありません。
CASE2:
        MOV. L    #H' FFFF0000, R0   ----   拡張命令2
        RTS
        NOP
                                     ----   リテラルプールの出力ポイントです。
        ~
```

↓↓↓↓

リテラルプール自動生成結果（ソースリスト）

```
        ~
20  0000F000                    20    CASE1:
21  0000F000  D002              21          MOV. L    #H' FFFF0000, R0
22  0000F002  000B              22          RTS
23  0000F004  0009              23          NOP
24                              24          . NOPOOL
25  0000F006                    25    CASE2:
26  0000F006  D001              26          MOV. L    #H' FFFF0000, R0
27  0000F008  000B              27          RTS
28  0000F00A  0009              28          NOP
29                                    ＊＊＊＊＊ BEGIN-POOL ＊＊＊＊＊
30  0000F00C  FFFF0000                    DATA FOR SOURCE-LINE 21, 26
31                                    ＊＊＊＊＊ END-POOL ＊＊＊＊＊
        ~
```

11.8.7 リテラルプール自動生成に関する注意事項

(1) 拡張命令を記述してエラーとなる場合
ディレイスロット命令に拡張命令を記述できません(エラー151)。
アライメント数2未満の相対アドレスセクションには拡張命令を記述できません(エラー152)。
アライメント数4未満の相対アドレスセクションにはMOV.L #imm,Rn、MOVA #imm,R0 を記述できません(エラー152)。

(2) .POOLを記述してエラーとなる場合
遅延分岐命令に続いて.POOLを記述できません(エラー522)。

(3) .NOPOOLを記述してエラーとなる場合
.NOPOOLはディレイスロット命令に続いて記述された場合に有効です。それ以外の位置に記述された場合、エラー521となります。

(4) 拡張命令を展開した結果、他の実行命令のディスプレースメントが範囲外となる場合
アセンブラはリテラルプールを生成し、ディスプレースメントが範囲外となる命令をエラー402とします。
.NOPOOLを使用するなどしてリテラルプールの出力ポイントを移動するか、エラーとなる命令の位置またはアドレッシングモードを変更してください。

(5) リテラルプールの出力位置が見つからない場合
拡張命令の位置からみて、次の条件を満足するリテラルプール出力ポイントが見つからない場合
・同ファイル
・同セクション
・拡張命令の記述された位置以降
アセンブラはその拡張命令が存在するセクションの最後にリテラルプールとそれを飛び越すBRA命令(ディレイスロット命令はNOP)を出力し、ウォーニング876を発行します。

(6) 拡張命令のディスプレースメントが範囲外となる場合
リテラルプールを生成したが、拡張命令からのディスプレースメントが範囲外となる場合、その拡張命令はエラー402となります。
.POOLを用いるなどして、範囲内となる場所にリテラルプールが生成されるようにしてください。

(7) サイズ指定モードとサイズ選択モードの相違
Ver.2.0のリテラルプール機能はサイズ指定モードのみのサポートでしたが、Ver.3.1以降ではサイズ選択モードを追加しました。Ver.2.0で作成したソースプログラムをVer.3.1以降のサイズ選択モードでアセンブルするとオペレーションサイズ指定のないデータ転送命令においてimm値がH'00000080～H'000000FF(128～255)の範囲で相違がでます。
サイズ指定モードとサイズ選択モードの出力例を次に示します。

■コーディング例

ソースプログラム

```
. SECTION CD1, CODE, LOCATE=H' 0000F000
MOV. L      #H' FF, R0
MOV. W      #H' FF, R1
MOV. B      #H' FF, R2
MOV         #H' FF, R3
RTS
MOV         R0, R10
. END
```

↓↓↓↓

サイズ指定モードのリテラルプール自動生成結果（ソースリスト）

```
 1  0000F000                          1          . SECTION CD1, CODE, LOCATE=H' 0000F000
 2  0000F000   D003                   2          MOV. L      #H' FF, R0
 3  0000F002   9103                   3          MOV. W      #H' FF, R1
 4  0000F004   E2FF                   4          MOV. B      #H' FF, R2
 5  0000F006   E3FF                   5          MOV         #H' FF, R3
 6  0000F008   000B                   6          RTS
 7  0000F00A   6A03                   7          MOV         R0, R10
 8                                        ***** BEGIN-POOL *****
 9  0000F00C   00FF                             DATA FOR SOURCE-LINE 3
10  0000F00E   0000                             ALIGNMENT CODE
11  0000F010   000000FF                         DATA FOR SOURCE-LINE 2
12                                        ***** END-POOL *****
13                                        8          . END
```

R3の内容は、H'FFFFFFFFFとなります。

サイズ選択モードのリテラルプール自動生成結果（ソースリスト）

```
 1  0000F000                          1          . SECTION CD1, CODE, LOCATE=H' 0000F000
 2  0000F000   D003                   2          MOV. L      #H' FF, R0
 3  0000F002   9103                   3          MOV. W      #H' FF, R1
 4  0000F004   E2FF                   4          MOV. B      #H' FF, R2
 5  0000F006   9301                   5          MOV         #H' FF, R3
 6  0000F008   000B                   6          RTS
 7  0000F00A   6A03                   7          MOV         R0, R10
 8                                        ***** BEGIN-POOL *****
 9  0000F00C   00FF                             DATA FOR SOURCE-LINE 3, 5
10  0000F00E   0000                             ALIGNMENT CODE
11  0000F010   000000FF                         DATA FOR SOURCE-LINE 2
12                                        ***** END-POOL *****
13                                        8          . END
```

R3の内容は、H'000000FFとなります。

11.9 リピートループ命令自動生成機能

11.9.1 リピートループ命令自動生成機能の概要

SH-DSP、SH3-DSP、SH4AL-DSP では、LDRS 命令と LDRE 命令によってリピート開始アドレスとリピート終了アドレスを RS レジスタと RE レジスタに設定します。そのアドレスの設定値はリピートループ間の命令数により異なります。アドレスを記述するとき、表 11.37 に示すような考慮をする必要があります。

表 11.37 リピートループ間の命令数とアドレスの設定値

命令数	1 命令	2 命令	3 命令	4 命令以上
RS レジスタ	s_addr0+8	s_addr0+6	s_addr0+4	s_addr
RE レジスタ	s_addr0+4	s_addr0+4	s_addr0+4	e_addr3+4

・s_addr0 ：リピート開始アドレスの 1 命令前のアドレス
・s_addr ：リピート開始アドレス
・e_addr3 ：リピート終了アドレスの 3 命令前のアドレス

リピートループ命令自動生成機能とは、リピートループ間の命令数からアドレス値を RS、RE レジスタに転送する PC 相対の LDRS、LDRE 命令と、リピート回数を設定する SETRC 命令を 1 つの命令から自動的に生成する機能です。

例えば、以下のプログラム(a)は本機能を用いると(b)のように記述できます。

■プログラム(a)

```
            LDRS s_addr0+6
            LDRE s_addr0+4
            SETRC #10
s_addr0:    NOP
            PADD A0,M0,A0      ;リピート開始アドレス
            PCMP x1,M0         ;リピート終了アドレス
```

■プログラム(b)

```
            REPEAT s_addr,e_addr,#10
            NOP
s_addr:     PADD A0,M0,A0      ;リピート開始アドレス
e_addr:     PCMP X1,M0         ;リピート終了アドレス
```

11.9.2 リピートループ命令自動生成機能に関する拡張命令

拡張命令(REPEAT s_label,e_ladel,#imm、REPEAT s_label,e_label,Rn、REPEAT s_label,e_label)の記述に対してアセンブラは必要な命令を自動生成し、PC 相対のディスプレースメント値を計算します。表 11.38 に拡張命令のソースステートメントとその展開結果を示します。拡張命令のソースステートメントは 2 つまたは 3 つの実行命令に展開されます。

表 11.38 拡張命令の種類と展開結果

拡張命令	展開結果
REPEAT s_label,e_label,#imm	LDRS@(disp,PC)と LDRE@(disp,PC)と SETRC#imm
REPEAT s_label,e_label,Rn	LDRS@(disp,PC)と LDRE@(disp,PC)と SETRC Rn
REPEAT s_label,e_label	LDRS@(disp,PC)と LDRE@(disp,PC)

11.9.3 REPEAT の記述方法

REPEAT の書き方は次のとおりです。

[<シンボル>[:]]△REPEAT△<開始アドレス>,<終了アドレス>[,<リピート回数>]

(1) ステートメントの要素

(a) <開始アドレス>、<終了アドレス>
リピートループの開始アドレスと終了アドレスをラベルで記述します。

(b) <リピート回数>
リピート回数をイミディエイトまたは汎用レジスタで記述します。

(2) 説明

(a) REPEATは開始アドレスから終了アドレスまでの命令をリピートするための実行命令(LDRS、LDRE)を自動的に生成します。

(b) リピート回数を指定した場合、SETRCを生成します。リピート回数を省略した場合、SETRCを生成しません。

11.9.4 コーディング例

(1) 基本例(リピートする命令数が 4 命令以上のとき)

```
            REPEAT RptStart,RptEnd,#5
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
            PADD A0,Y0,Y0  PMULS X1,Y1,A0
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
```

このプログラムは RptStart から RptEnd までの 5 命令を 5 回繰り返し実行します。

展開イメージ

```
            LDRS RptStart
            LDRE RptEnd3+4
            SETRC #5
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
RptEnd3:    PADD A0,Y0,Y0 PMULS X1,Y1,A0 ;ラベルは実際には生成されません
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
```

(2) リピートする命令数が 1 命令のとき

開始アドレスと終了アドレスを同じラベルで指定してください。

```
            REPEAT Rpt,Rpt,R0
            MOVX @R4+,X1  MOVY @R6,Y1
Rpt:        PADD A0,Y0,Y0 PMULS X1,Y1,A0  MOVX @R4+,X1  MOVY  @R6+,Y1
```

展開イメージ

```
            LDRS RptStart0+8
            LDRE RptStart0+4
            SETRC R0
RptStart0: MOVX @R4+,X1  MOVY @R6,Y1            ; ラベルは実際には生成されません
Rpt:        PADD A0,Y0,Y0 PMULS X1,Y1,A0  MOVX @R4+,X1  MOVY  @R6+,Y1
```

(3) リピートする命令数が 2 命令のとき

```
            REPEAT RptStart,RptEnd,#10
            PCLR Y0
RptStart:   MOVX @R4+,X1  MOVY  @R6+,Y1
RptEnd:     PADD A0,Y0,Y0 PMULS X1,Y1,A0
```

展開イメージ

```
            LDRS RptStart0+6
            LDRE RptStart0+4
            SETRC #10
RptStart0:  PCLR Y0           ; ラベルは実際には生成されません
RptStart:   MOVX @R4+,X1  MOVY  @R6+,Y1
RptEnd:     PADD A0,Y0,Y0 PMULS X1,Y1,A0
```

(4) リピートする命令数が 3 命令のとき

```
            REPEAT RptStart,RptEnd,R0
            PCLR Y0
RptStart:   MOVX @R4+,X1  MOVY  @R6+,Y1
            PMULS X1,Y1,A0
RptEnd:     PADD A0,Y0,Y0
```

展開イメージ

```
            LDRE RptStart0+4
            LDRS RptStart0+4
            SETRC R0
RptStart0:  PCLR Y0           ; ラベルは実際には生成されません
RptStart:   MOVX @R4+,X1  MOVY  @R6+,Y1
            PMULS X1,Y1,A0
RptEnd:     PADD A0,Y0,Y0
```

(5) リピート回数を省略したとき

リピート回数を省略するとアセンブラは SETRC を展開しません。LDRS、LDRE と SETRC を分離したいとき使用します。

```
            REPEAT RptStart,RptEnd
            ; この部分に LDRS, LDRE が展開されます
            MOV #10,R0
OuterLoop:
            SETRC #16
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
            PADD A0,Y0,Y0  PMULS X1,Y1,A0
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
            DT R0
            BF OuterLoop
```

11.9.5 REPEAT 拡張命令に関する注意事項

(1) 開始アドレス、終了アドレスに関する注意事項

開始アドレスおよび終了アドレスに指定できるラベルは同じセクション内のラベルまたは同じローカルブロック内のローカルラベルです。

開始アドレスは REPEAT 拡張命令より後のアドレスになければなりません。

また、終了アドレスは開始アドレスより後のアドレスになければなりません。

(2) ループ内に記述する命令に関する注意事項

(a) ループ内にデータまたはデータ領域を確保する制御命令または.ORGを記述した場合、アセンブラはウォーニングを出力し、1制御命令を1命令としてリピートする命令数をカウントします。.ALIGNを記述してアライメントが生成された場合、アセンブラはウォーニングを出力し、.ALIGNを1命令としてリピートする命令数をカウントします。該当する制御命令は次のとおりです。

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ALIGN、.ORG

(b) アセンブラはループ内ではリテラルプールの自動生成を抑止します。したがって、無条件分岐命令があってもリテラルプールの出力対象になりません。また、.POOLを記述した場合、アセンブラはウォーニングを出力し、.POOLを無視します。

(3) ループ直前の命令に関する注意事項

リピートする命令数が 3 命令以下の場合、ループ直前の命令は実行命令または DSP 命令でなければなりません。したがって、リピートする命令数が 3 命令以下で開始アドレスの直前が以下の場合、アセンブラはエラーを出力します。

(a) データまたはデータ領域を確保する制御命令または.ORG

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ORG

(b) リテラルプール自動生成機能で生成されたリテラルプール

開始アドレスの直前が無条件分岐命令+ディレイスロット命令または.POOLの場合、リテラルプールが自動的に生成される可能性があります。リテラルプールの生成を抑止するためにはディレイスロット命令の直後に.NOPOOLを記述してください。

(c) .ALIGNで1バイトのアライメントが生成された場合

.ALIGNを指定したとき、直前が奇数アドレスである場合、1バイトのアライメントが生成される場合があります(例えば、ロケーションカウンタが3で.ALIGN 4を指定したときなど)。この場合、実行命令でないデータがループの直前に出力されたことになり、エラーになります。2バイト以上のアライメントが出力された場合、直前の命令はNOPになり、正常に動作します。

(4) その他の注意事項

(a) REPEAT拡張命令～開始アドレスの間には1命令以上の実行命令またはDSP命令がなければなりません。1命令以上の実行命令またはDSP命令が存在しない場合、アセンブラはエラーを出力します。

(b) REPEAT拡張命令～終了アドレスの間に別のREPEAT拡張命令を記述することはできません。REPEAT拡張命令をネストして記述した場合、アセンブラはエラーを出力します。

(c) REPEAT拡張命令～終了アドレスの間に分岐命令、TRAPA命令(cpu=sh4aldsp未指定時)、SR、RS、REに対するロード命令を記述することはできません。上記の命令を記述した場合、アセンブラはエラーを出力します。

11.10 拡張リピートループ命令自動生成機能

11.10.1 拡張リピートループ命令自動生成機能の概要

SH4AL-DSP では、LDRS 命令と LDRE 命令によってリピート開始アドレスとリピート終了アドレスを RS レジスタと RE レジスタに設定します。

拡張リピートループ命令自動生成機能とは、リピート開始アドレスとリピート終了アドレスを RS、RE レジスタに転送する PC 相対の LDRS、LDRE 命令と、リピート回数を設定する LDRC 命令を 1 つの命令から自動的に生成する機能です。

例えば、以下のプログラム(a)は本機能を用いると(b)のように記述できます。

■プログラム(a)

```
            LDRS s_addr
            LDRE e_addr
            LDRC #10
            NOP
s_addr:     PADD A0,M0,A0      ;リピート開始アドレス
e_addr:     PCMP X1,M0         ;リピート終了アドレス
```

■プログラム(b)

```
            EREPEAT s_addr,e_addr,#10
            NOP
s_addr:     PADD A0,M0,A0      ;リピート開始アドレス
e_addr:     PCMP X1,M0         ;リピート終了アドレス
```

11.10.2 拡張リピートループ命令自動生成機能に関する拡張命令

拡張命令(EREPEAT s_label,e_label,#imm、EREPEAT s_label,e_label,Rn、EREPEAT s_label,e_label)の記述に対してアセンブラは必要な命令を自動生成します。

表 11.39 に拡張命令のソースステートメントとその展開結果を示します。拡張命令のソースステートメントは 2 つまたは 3 つの実行命令に展開されます。

表 11.39 拡張命令の種類と展開結果

拡張命令	展開結果
EREPEAT s_label,e_label,#imm	LDRS@(disp,PC)と LDRE@(disp,PC)と LDRC#imm
EREPEAT s_label,e_label,Rn	LDRS@(disp,PC)と LDRE@(disp,PC)と LDRC Rn
EREPEAT s_label,e_label	LDRS@(disp,PC)と LDRE@(disp,PC)

11.10.3 EREPEAT の記述方法

EREPEAT の書き方は次のとおりです。

[<シンボル>[:]]△EREPEAT△<開始アドレス>,<終了アドレス>[,<リピート回数>]

(1) ステートメントの要素

(a) <開始アドレス>、<終了アドレス>
リピートループの開始アドレスと終了アドレスをラベルで記述します。

(b) <リピート回数>
リピート回数をイミディエイトまたは汎用レジスタで記述します。

(2) 説明

(a) EREPEATは開始アドレスから終了アドレスまでの命令をリピートするための実行命令(LDRS、LDRE)を自動的に生成します。

(b) リピート回数を指定した場合、LDRCを生成します。リピート回数を省略した場合、LDRCを生成しません。

11.10.4 コーディング例

(1) 基本例

```
            EREPEAT RptStart,RptEnd,#5
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
            PADD A0,Y0,Y0  PMULS X1,Y1,A0
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
```

このプログラムは RptStart から RptEnd までの 5 命令を 5 回繰り返し実行します。

展開イメージ

```
            LDRS RptStart
            LDRE RptEnd3+4
            LDRC #5
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
            PADD A0,Y0,Y0 PMULS X1,Y1,A0
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
```

(2) リピート回数を省略したとき

リピート回数を省略するとアセンブラは LDRC を展開しません。LDRS、LDRE と LDRC を分離したいとき使用します。

```
            EREPEAT RptStart,RptEnd
            ; この部分に LDRS, LDRE が展開されます
            MOV #10,R0
OuterLoop:
            LDRC #16
            PCLR Y0
            PCLR A0
RptStart:   MOVX @R4+,X1  MOVY @R6+,Y1
            PADD A0,Y0,Y0  PMULS X1,Y1,A0
       DCT  PCLR A0
            AND  R0,R4
RptEnd:     AND  R0,R6
            DT R0
            BF OuterLoop
```

11.10.5 拡張 REPEAT 命令(EREPEAT 命令)に関する注意事項

(1) 開始アドレス、終了アドレスに関する注意事項

開始アドレスおよび終了アドレスに指定できるラベルは同じセクション内のラベルまたは同じローカルブロック内のローカルラベルです。

開始アドレスは EREPEAT 拡張命令より後のアドレスになければなりません。

また、終了アドレスは開始アドレスより後のアドレスになければなりません。

(2) ループ内に記述する命令に関する注意事項

(a) ループ内にデータまたはデータ領域を確保する制御命令または.ORGを記述した場合、アセンブラはウォーニングを出力し、1制御命令を1命令としてリピートする命令数をカウントします。.ALIGNを記述してアライメントが生成された場合、アセンブラはウォーニングを出力し、.ALIGNを1命令としてリピートする命令数をカウントします。該当する制御命令は次のとおりです。

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ALIGN、.ORG

(b) アセンブラはループ内ではリテラルプールの自動生成を抑止します。したがって、無条件分岐命令があってもリテラルプールの出力対象になりません。また、.POOLを記述した場合、アセンブラはウォーニングを出力し、.POOLを無視します。

(3) ループ直前の命令に関する注意事項

ループ直前の命令は実行命令または DSP 命令でなければなりません。したがって、開始アドレスの直前が以下の場合、アセンブラはエラーを出力します。

(a) データまたはデータ領域を確保する制御命令または.ORG

.DATA、.DATAB、.SDATA、.SDATAB、.SDATAC、.SDATAZ、.FDATA、
.FDATAB、.XDATA、.RES、.SRES、.SRESC、.SRESZ、.FRES、.ORG

(b) リテラルプール自動生成機能で生成されたリテラルプール

開始アドレスの直前が無条件分岐命令+ディレイスロット命令または.POOLの場合、リテラルプールが自動的に生成される可能性があります。リテラルプールの生成を抑止するためにはディレイスロット命令の直後に.NOPOOLを記述してください。

(c) .ALIGNで1バイトのアライメントが生成された場合

.ALIGNを指定したとき、直前が奇数アドレスである場合、1バイトのアライメントが生成される場合があります(例えば、ロケーションカウンタが3で.ALIGN 4を指定したときなど)。この場合、実行命令でないデータがループの直前に出力されたことになり、エラーになります。2バイト以上のアライメントが出力された場合、直前の命令はNOPになり、正常に動作します。

(4) その他の注意事項

(a) EREPEAT最終命令に別のEREPEAT拡張命令を記述することはできません。EREPEAT拡張命令をネストして記述した場合、アセンブラはエラーを出力します。

(b) EREPEAT最終命令に遅延分岐命令、SR、RS、REに対するロード命令を記述することはできません。上記の命令を記述した場合、アセンブラはエラーを出力します。

12.　コンパイラのエラーメッセージ

12.1　エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　(エラーレベル) エラーメッセージ
　　エラー内容

エラーレベルは、エラーの重要度に従い、5 種類に分類されます。

エラーレベル		動作
(I)	インフォメーション	処理を継続します。
(W)	ウォーニング	処理を継続します。
(E)	エラー	オプション解析処理を継続し、処理を中断します。
(F)	フェータル	処理を中断します。
(-)	インターナル	処理を中断します。

12.2　メッセージ一覧

C0001 (I) Character combination "文字列" in comment
　　コメントの中に、"文字列"があります。

C0002 (I) No declarator
　　宣言子のない宣言があります。

C0003 (I) Unreachable statement
　　実行されることのない文があります。

C0004 (I) Constant as condition
　　if 文または switch 文の条件を示す式として、定数式を指定しています。

C0005 (I) Precision lost
　　代入式において、右辺の式の値を左辺の型へ変換する時に、精度が失われる可能性があります。

C0006 (I) Conversion in argument
　　関数の引数の式が、関数原型で指定した引数の型に変換されます。

C0008 (I) Conversion in return
　　リターン文の式が、関数の返す値の型に変換されます。

C0010 (I) Elimination of needless expression
不要な式があります。

C0011 (I) Used before set symbol : "変数名" in "関数名"
値の設定されていない局所変数を参照しています。

C0012 (I) Unused variable "変数名"
使用していない変数があります。

C0015 (I) No return value
void 型以外の型を返す関数の中で、リターン文が値を返していません。

C0016 (I) Conversion in case constant expression
case ラベルの定数式が、制御式の汎整数拡張された型に変換されます。

C0017 (I) Missing return statement
void 型以外の型を返す関数の中で、リターン文のないコントロールパスがあります。

C0100 (I) Function "関数名" not optimized
関数のサイズが大きすぎるため、最適化できません。

C0101 (I) Optimizing range divided in function "関数名"
"関数名"の最適化範囲が複数に分割されました。

C0102 (I) Register is not allocated to "変数名" in "関数名"
register 記憶クラスを持つ変数にレジスタを割り付けることができませんでした。

C0200 (I) No prototype function
関数の関数原型がありません。

C1000 (W) Illegal pointer assignment
ポインタ型どうしの代入で、それぞれのポインタ型の指す型が異なっています。

C1001 (W) Illegal comparison in "演算子"
2 項演算子==または!=の被演算子が、一方がポインタ型で他方が値 0 以外のスカラ型を指しています。

C1002 (W) Illegal pointer for "演算子"
2 項演算子==、!=、>、<、>=または<=の被演算子が、同じ型へのポインタ型を指していません。

C1003 (W) Illegal pointer initialization
ポインタ型の初期値指定で、それぞれのポインタ型の指す型が異なっています。

C1005 (W) Undefined escape sequence
文字定数または文字列の中で、文法上定義していない拡張表記 (バックスラッシュとそれに続く文字) を用いています。

C1007 (W) Long character constant
文字定数の長さが 2 文字以上になっています。

C1008 (W) Identifier too long
識別子の長さが 8189 文字を超えています。8190 文字以降は無効となります。

C1010 (W) Character constant too long
文字定数の長さが 4 文字を超えています。

C1012 (W) Floating point constant overflow
浮動小数点定数の値が値の範囲を超えています。符号にしたがって+∞または-∞に対応する内部表現の値を仮定します。

C1013 (W) Integer constant overflow
整定数の値が unsigned long long 型のとり得る値の範囲を超えています。オーバフローした上位ビットを無視した値を仮定します。

C1014 (W) Escape sequence overflow
文字定数あるいは文字列の中でのビットパターンを示す拡張表記の値が 255 を超えています。下位 1 バイトの値を有効とします。

C1015 (W) Floating point constant underflow
浮動小数点定数の値の絶対値が表現できる最小値よりも小さな値となっています。定数の値を 0.0 と仮定します。

C1016 (W) Argument mismatch
関数原型の中の引数と関数呼び出しの対応する引数の型がポインタ型で、それぞれの指す型が異なっています。関数呼び出しにおける引数のポインタの内部表現をそのまま設定します。

C1017 (W) Return type mismatch
関数の返す型とリターン文の式の型がポインタ型で、それぞれの指す型が異なっています。リターン文の式のポインタの内部表現をそのまま設定します。

C1019 (W) Illegal constant expression
定数式において関係演算子<、>、<=または>=の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

C1020 (W) Illegal constant expression of "-"
定数式において 2 項演算子-の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

C1021 (W) Register saving pragma conflicts in interrupt function "関数名"
"関数名"で示す割り込み関数に対するレジスタ退避・回復を制御する#pragma が不適切です。#pragma 指定を無視します。

C1022 (W) First operand of "演算子" is not lvalue
第 1 オペランドの"演算子"は、左辺値になりません。

C1023 (W) Can not convert Japanese code "コード" to output type
日本語コードで指定の出力コードに変換できないものがあります。

C1024 (W) Out of float
浮動小数点定数の有効桁数が 17 桁を超えています。18 桁以降は無効となります。

C1026 (W) Address of packed member
pack=1 指定ありの構造体メンバのアドレスを取得しています。

C1027 (W) Invalid #pragma gbr_base/gbr_base1
gbr=auto と#pragma gbr_base、#pragma gbr_base1 が同時に指定されています。#pragma 指定を無視します。

C1028 (W)#pragma "識別子" has no effect
指定した#pragma 識別子は無効です。

C1029 (W)Function with ifunc calls "関数名" without ifunc
ifunc 指定ありの関数から ifunc 指定なしの関数を呼んでいます。

C1030 (W) Bit order mismatch
構造体と構造体メンバで bit_order が異なります。

C1031 (W) Multiple #pragma for one function
１つの関数に対して#pragma を重複して指定しています。

C1200 (W) Division by floating point zero
定数式の中で浮動小数点型 0.0 を除数とする割り算を行っています。符号にしたがって、+∞または-∞に対応する内部表現の値を仮定します。

C1201 (W) Ineffective floating point operation
定数式の中で∞-∞、0.0/0.0 等の無効演算を行っています。無効演算の結果を表す非数に対応する内部表現の値を仮定します。

C1300 (W) Command parameter specified twice
同じコンパイラオプションを 2 度以上指定しています。同じコンパイラオプションの中で最後に指定したものを有効とします。

C1301 (W) "オプション" option ignored
"オプション"を無視してコンパイルをします。

C1302 (W) "double=float" option ignored
double=float、cpu=sh2afpu|sh4|sh4a を同時に指定しています。double=float を無視し、fpu=single が指定されていると解釈してコンパイルをします。

C1304 (W) "マイコン種別 1" is interpreted as "マイコン種別 2"
cpu=<マイコン種別 1>は無効です。cpu=<マイコン種別 2>と解釈してコンパイルします。

C1308 (W) Duplicate number specified in option "オプション" : "番号"
"オプション"で同じ番号を指定しています。

C1309 (W) Section name "セクション名" specified
"セクション名"は、コンパイラが生成するセクション名と重なる場合があります。

C1310 (W) "repeat" option ignored
repeat オプションを無視してコンパイルをします。

C1311 (W) "softpipe" option ignored
softpipe オプションを無視してコンパイルをします。

C1312 (W) "fdiv" option ignored
fdiv オプションを無視してコンパイルをします。

C1313 (W) "bss_order=declaration" option ignored
bss_order=declaration オプションを無視してコンパイルをします。

C1314 (W) File_inline "ファイル名" ignored by #pragma global_register mismatch
#pragma global_register の指定が矛盾しています。
file_inline オプションを無視してコンパイルをします。

C1315 (W) File_inline "ファイル名" ignored by same file as source file
コンパイル対象のファイルが file_inline オプションで指定されています。
file_inline オプションを無視してコンパイルをします。

C1400 (W) Function "関数名" in #pragma inline is not expanded
#pragma inline で指定した関数がインライン展開されませんでした。
#pragma inline 指定を無視します。

C1402 (W) #pragma "識別子" ignored
#pragma "識別子"指定を無視します。

C1405 (W) Illegal #pragma syntax
キーワードとして存在しない#pragma を指定しています。

C1410 (W) A struct/union/class has different pack specifications
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。

C1501 (W) Division by zero
ゼロ除数が発生しました。

C1600 (W) Debugging information describing location of "名前" is lost
コンパイラが生成するシンボル割付情報の情報量が制限を超えたため、変数"名前" のシンボル割付情報を出力しません。このため、変数"名前" をウォッチウィンドウで参照できません。
本メッセージの原因の 1 つとして、メンバ数が多い構造体をローカルに定義していることが考えられます。該当する場合、以下のいずれかの方法で回避できる場合があります。
・変数 "名前" をポインタ参照に変更する
・変数 "名前" を static で定義する

C1700 (W) Memory qualifier ignored
メモリ型修飾子指定を無視します。

C1701 (W) Conversion from pointer without memory qualifier to pointer with memory qualifier
メモリ型修飾子のないポインタから、メモリ型修飾子のあるポインタへの変換が行われました。メモリ型修飾子は無効となります。

C1702 (W) Conversion from pointer with circular qualifier to pointer without circular qualifier
_ _circ 型修飾子のあるポインタから、_ _circ 型修飾子のないポインタへの変換が行われました。_ _circ 型修飾子は無効となります。

C1703 (W) Fixed point constant overflow
固定小数点定数の値が値の範囲を超えています。

C1704 (W) Out of Fixed point
固定小数点定数の有効桁数が 17 桁を超えています。

C1705 (W) Modulo addressing may be illegal in function "関数名"
"関数名"で示す関数で、モジュロアドレッシングが正しく行われない可能性があります。

C1800 (W) Variable "変数名" type mismatch in files
"変数名"で示す変数の型がファイル間で異なっています。
file_inline オプションの指定をやめてください。

C2000 (E) Illegal preprocessor keyword
プリプロセッサ文で、誤ったキーワードを使用しています。

C2001 (E) Illegal preprocessor syntax
プリプロセッサ文またはマクロ呼び出しの指定方法に誤りがあります。

C2002 (E) Missing ","
引数のある#define 文で引数の並びを区切るカンマ","が抜けています。

C2003 (E) Missing ")"
名前が#define 文で定義されているかどうかを判定する defined 式で名前の次の右括弧") "が抜けています。

C2004 (E) Missing ">"
#include 文のファイル名の指定でファイル名の次の > がありません。

C2005 (E) Cannot open include file "ファイル名"
#include 文で指定したファイル名のファイルがオープンできません。

C2006 (E) Multiple #define's
#define 文で同じマクロ名を再定義しています。

C2008 (E) Processor directive #elif mismatches
#elif 文に対応する#if 文、#ifdef 文、#ifndef 文、#elif 文がありません。

C2009 (E) Processor directive #else mismatches
#else 文に対応する#if 文、#ifdef 文、#ifndef 文がありません。

C2010 (E) Macro parameters mismatch
マクロ呼び出しの引数の数がマクロ定義の引数の数と異なっています。

C2011 (E) Line too long
マクロ展開後のソースプログラムの行が限界値を超えています。

C2012 (E) Keyword as a macro name
プリプロセッサで規定しているキーワードを#define 文または#undef 文のマクロ名として定義しています。

C2013 (E) Processor directive #endif mismatches
#endif 行に対応する#if 文、#ifdef 文、#ifndef 文がありません。

C2014 (E) Missing #endif
#if 文、#ifdef 文、#ifndef 文に対応する#endif 行がないままでファイルが終了しました。

C2016 (E) Preprocessor constant expression too complex
#if 文、#elif 文で指定した定数式の演算子と被演算子の合計が限界値を超えています。

C2017 (E) Missing "
#include 文のファイル名の指定で、ファイル名の次に"がありません。

C2018 (E) Illegal #line
#line 文で指定した行数が限界値を超えています。

C2019 (E) File name too long
ファイル名の文字数が限界値を超えています。

C2020 (E) System identifier "名前" redefined
組み込み関数と同名のシンボルを定義しています。

C2021 (E) Invalid number specified in option "オプション" : "番号"
"オプション"指定で無効な値を指定しています。値の範囲を確認してください。

C2022 (E) Error level message cannot be changed : "change_message"
Error レベルのメッセージは、メッセージレベルを変更できません。

C2027 (E) Cannot read specified file "ファイル名"
ファイルが正常に読み込めません。ファイルの指定が間違っていないか確認してください。

C2100 (E) Multiple storage classes
宣言の中で 2 つ以上の記憶域クラス指定子を指定しています。

C2101 (E) Address of register
register 記憶クラスを持つ変数に対して、単項演算子&を適用しています。

C2102 (E) Illegal type combination
型指定子の組み合わせが誤っています。

C2103 (E) Bad self reference structure
構造体または共用体のメンバの型を、親の構造体または共用体と同じ型で宣言しています。

C2104 (E) Illegal bit field width
ビットフィールド幅を示す定数式が整数型でありません。あるいはビットフィールド幅として負の整数を指定しています。

C2105 (E) Incomplete tag used in declaration
構造体または共用体で仮宣言されたタグ名、または未宣言のタグ名を typedef 宣言、ポインタを指す型あるいは関数の返す型以外の宣言で使用しています。

C2106 (E) Extern variable initialized
複文内で extern 記憶クラスを指定した宣言に対して初期値を指定しています。

C2107 (E) Array of function
要素の型が関数型となる配列型を指定しています。

C2108 (E) Function returning array
リターン値の型が配列型となる関数型を指定しています。

C2109 (E) Illegal function declaration
複文内の関数型の変数宣言において、extern 以外の記憶クラスを指定しています。

C2110 (E) Illegal storage class
外部定義の中で記憶クラスとして auto または register を指定しています。

C2111 (E) Function as a member
構造体または共用体のメンバの型に関数型を指定しています。

C2112 (E) Illegal bit field
ビットフィールドに誤った型を指定しています。ビットフィールドに許される型指定子は、char, unsigned char, short, unsigned short, int, unsigned int, long,unsigned long, long long, unsigned long long, bool, enum のいずれか、これらの型に const または volatile を組み合わせた型です。

C2113 (E) Bit field too wide
ビットフィールド幅が型指定子で指定したサイズ(8、16、32、64 ビット)を超えています。

C2114 (E) Multiple variable declarations
変数名を同じ有効範囲の中で重複して宣言しています。

C2115 (E) Multiple tag declarations
構造体、共用体、列挙型のタグ名を同じ有効範囲の中で重複して宣言しています。

C2117 (E) Empty source program
ソースプログラム内に外部定義が含まれていません。

C2118 (E) Prototype mismatch "関数名"
関数の型が以前になされている宣言で指定した型と一致しません。

C2119 (E) Not a parameter name "引数名"
関数の引数宣言列にない識別子に対して引数宣言を行っています。

C2120 (E) Illegal parameter storage class
関数の引数宣言で register 以外の記憶クラスを指定しています。

C2121 (E) Illegal tag name
構造体、共用体または列挙型とタグ名の組み合わせが、以前に宣言した型とタグ名の組み合わせと異なっています。

C2122 (E) Bit field width 0
メンバ名を指定しているビットフィールドの幅が 0 になっています。

C2123 (E) Undefined tag name
列挙型の宣言で未定義のタグ名を使用しています。

C2124 (E) Illegal enum value
列挙型のメンバに整数でない定数式を指定しています。

C2125 (E) Function returning function
リターン値の型が関数型となる関数型を指定しています。

C2126 (E) Illegal array size
配列の要素数の値が「1 以上 2147483647 以下の整数値」以外の値を指定しています。

C2127 (E) Missing array size
配列の要素数の指定がありません。

C2128 (E) Illegal pointer declaration for "*"
ポインタ型の宣言を示す*の直後に const、volatile 以外の型指定子を指定しています。

C2129 (E) Illegal initializer type
変数の初期値指定において初期値の型が変数に代入可能な型でありません。

C2130 (E) Initializer should be constant
構造体型、共用体型、配列型の変数の初期値、または静的に割り付けられる変数の初期値に定数式でないものを指定しています。

C2131 (E) No type nor storage class
外部データ定義において記憶クラスまたは型の指定がありません。

C2132 (E) No parameter name
関数の引数宣言列が空であるにもかかわらず引数の宣言を行っています。

C2133 (E) Multiple parameter declarations
(マクロ)関数定義の引数宣言列の中で同一名の引数を重複して宣言しているか、または引数宣言が関数宣言子の中と外の 2 ヶ所で行われています。

C2134 (E) Initializer for parameter
引数の宣言において初期値を指定しています。

C2135 (E) Multiple initialization
同一の変数に対して、初期化を重複して行っています。

C2136 (E) Type mismatch
extern あるいは static 記憶クラスを持つ変数あるいは関数を 2 度以上宣言しており、その型が一致していません。

C2137 (E) Null declaration for parameter
関数の引数宣言で識別子を指定していません。

C2138 (E) Too many initializers
構造体、共用体または配列の初期値指定において、構造体のメンバ数または配列の要素数より多く初期値の数を指定しています。あるいは、共用体の最初のメンバがスカラ型のときに 2 個以上の初期値を指定しています。

C2139 (E) No parameter type
関数宣言の引数宣言に型指定がありません。

C2140 (E) Illegal bit field
共用体にビットフィールドを指定しています。

C2141 (E) Struct has no member name
構造体の先頭のメンバに無名のビットフィールドを指定しています。

C2142 (E) Illegal void type
void型の指定方法に誤りがあります。void型を指定できるのは以下の3つの場合です。
(1)ポインタの指す先の型として指定する場合。
(2)関数の返す型として指定する場合。
(3)関数原型の関数が引数を持たないことを明示的に指定する場合。

C2143 (E) Illegal static function
ソースファイル内に定義のないstatic記憶クラスを持つ関数宣言があります。

C2144 (E) Type mismatch
extern記憶クラスを持つ同じ名前の変数あるいは関数の型が一致していません。

C2145 (E) Const/volatile specified for incomplete type
不完全型に対してconstまたはvolatileが指定されています。

C2200 (E) Index not integer
配列の添字の式が整数型ではありません。

C2201 (E) Cannot convert parameter "n"
関数呼び出しにおけるn番目の引数を対応する関数原型の引数の型に変換できません。

C2202 (E) Number of parameters mismatch
関数呼び出しにおける引数の数が関数原型の引数の数と一致しません。

C2203 (E) Illegal member reference for "."
演算子.の左側の式の型が構造体型または共用体型ではありません。

C2204 (E) Illegal member reference for "->"
演算子->の左側の式の型が構造体型または共用体型へのポインタではありません。

C2205 (E) Undefined member name
構造体、共用体への参照で宣言していないメンバ名を使用しています。

C2206 (E) Modifiable lvalue required for "演算子"
前置または後置演算子++、--を代入可能な左辺値(配列型、const型を除く左辺値)でない式に使用しています。

C2207 (E) Scalar required for "!"
単項演算子!をスカラ型でない式に使用しています。

C2208 (E) Pointer required for "*"
単項演算子*をポインタ型でない式か、または void 型へのポインタ型の式に使用しています。

C2209 (E) Arithmetic type required for "演算子"
単項演算子+または-を算術型でない式に使用しています。

C2210 (E) Integer required for "~"
単項演算子~をスカラ型でない式に使用しています。

C2211 (E) Illegal sizeof
sizeof 演算子をビットフィールドの指定のあるメンバ、関数型、void 型またはサイズの指定していない配列に使用しています。

C2212 (E) Illegal cast
キャスト演算子で指定している型が配列型、構造体型または共用体型です。あるいはキャスト演算子の被演算子が void 型、構造体型または共用体型で型変換できません。

C2213 (E) Arithmetic type required for "演算子"
2 項演算子*、/、*=または/=を算術型でない式に適用しています。

C2214 (E) Integer required for "演算子"
2 項演算子<<、>>、&、|、^、%、<<=、>>=、&=、|=、^=または%=をスカラ型でない式に適用しています。

C2215 (E) Illegal type for "+"
2 項演算子+の被演算子の型の組み合わせが許されていません。2 項演算子+の型の組み合わせで許されるのは、両辺とも算術型の場合か、または一方がポインタ型で他方が汎整数型の場合だけです。

C2216 (E) Illegal type for parameter
関数呼び出しの引数の型に void 型を指定しています。

C2217 (E) Illegal type for "-"
2 項演算子-の被演算子の型の組み合わせが許されていません。2 項演算子-の型の組み合わせで許されるのは、以下の 3 つの場合です。
(1) 両方の被演算子が算術型の場合。
(2) 両方の被演算子が同じ型へのポインタ型の場合。
(3) 第 1 被演算子がポインタ型で、第 2 被演算子がスカラ型の場合。

C2218 (E) Scalar required
条件演算子? :の第 1 被演算子の型がスカラ型でありません。

C2219 (E) Type not compatible in "? :"
条件演算子? :の第 2 被演算子と第 3 被演算子の型が合っていません。条件演算子? :の第 2 被演算子と第 3 被演算子の組み合わせで許されるのは、以下の 6 つの場合です。
(1) 両方とも算術型の場合。
(2) 両方とも void 型の場合。
(3) 両方とも同じ型へのポインタ型の場合。
(4) 一方がポインタ型で、他方が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。
(5) 一方がポインタ型で、他方が void 型へのポインタ型の場合。
(6) 両方とも同じ型の構造体または共用体の場合。

C2220 (E) Modifiable lvalue required for "演算子"
代入演算子=、*=、/=、%=、+=、-=、<<=、>>=、&=、^=または |= の左辺の式に代入可能な左辺値(配列型、const 型を除く左辺値)以外の式を指定しています。

C2221 (E) Illegal type for "演算子"
後置演算子++または--の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。

C2222 (E) Type not compatible for "="
代入演算子=の両辺の式の型が合っていません。代入演算子=の両辺の式の組み合わせで許されるのは、以下の 5 つの場合です。
(1) 両方とも算術型の場合。
(2) 両方とも同じ型へのポインタ型の場合。
(3) 左辺がポインタ型で、右辺が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。
(4) 一方がポインタ型で、他方が void 型へのポインタ型の場合。
(5) 両方とも同じ型の構造体または共用体の場合。

C2223 (E) Incomplete tag used in expression
構造体または共用体で仮宣言されたタグ名を式中で使用しています。

C2224 (E) Illegal type for assign
代入演算子+=または-=の両辺の型が正しくありません。

C2225 (E) Undeclared name "名前"
宣言していない名前を式の中で用いています。

C2226 (E) Scalar required for "演算子"
2 項演算子&&または||をスカラ型でない式に適用しています。

C2227 (E) Illegal type for equality
等値演算子==または!=の被演算子の型の組み合わせが許されていません。等値演算の被演算子の組み合わせで許されるのは、以下の 3 つの場合です。
(1) 両方とも算術型の場合。
(2) 両方とも同じ型へのポインタ型の場合。
(3) 一方がポインタ型で、他方が値 0 の整定数または void 型へのポインタ型である場合。

C2228 (E) Illegal type for comparison
関係演算子>、<、>=または<=の被演算子の型の組み合わせが許されていません。関係演算子の被演算子の組み合わせで許されるのは、以下の 2 つの場合です。
(1)両方とも算術型の場合。
(2)両方とも同じ型へのポインタ型の場合。

C2230 (E) Illegal function call
関数呼び出しにおいて、関数型あるいは関数型へのポインタ型でない式を用いています。

C2231 (E) Address of bit field
単項演算子&をビットフィールドに適用しています。

C2232 (E) Illegal type for "演算子"
前置演算子++または--の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。

C2233 (E) Illegal array reference
配列型、関数型または void 型を除くポインタ型以外の式を配列として使用しています。

C2234 (E) Illegal typedef name reference
typedef 宣言された名前を式の中で変数として使用しています。

C2235 (E) Illegal cast
ポインタを浮動小数点型または固定小数点型にキャストしています。

C2236 (E) Illegal cast in constant
定数式でポインタ型を char 型または short 型にキャストしています。

C2237 (E) Illegal constant expression
定数式の中でポインタ型の定数を整数型へキャストした結果に対して演算を行っています。

C2238 (E) Lvalue or function type required for "&"
単項演算子&を左辺値あるいは関数型以外の式に適用しています。

C2239 (E) Illegal section name
セクション名に使用できない文字があります。

C2240 (E) Illegal section naming
セクションの命名に誤りがあります。用途の異なるセクションに同じ名前を付けています。

C2300 (E) Case not in switch
case ラベルを switch 文以外に指定しています。

C2301 (E) Default not in switch
default ラベルを switch 文以外に指定しています。

C2302 (E) Multiple labels
1 つの関数内にラベル名を重複して定義しています。

C2303 (E) Illegal continue
continue 文を while 文、for 文または do 文以外に指定しています。

C2304 (E) Illegal break
break 文を while 文、for 文、do 文または switch 文以外に指定しています。

C2305 (E) Void function returns value
void 型を返す関数の中の return 文でリターン値を指定しています。

C2306 (E) Case label not constant
case ラベルの式がスカラ型の定数式ではありません。

C2307 (E) Multiple case labels
同一の値を持つ case ラベルを 1 つの switch 文の中に重複して指定しています。

C2308 (E) Multiple default labels
default ラベルを 1 つの switch 文の中に重複して指定しています。

C2309 (E) No label for goto
goto 文で指定した行き先のラベルがありません。

C2310 (E) Scalar required
while 文、for 文または do 文の制御式 (文の実行を判定する式) がスカラ型ではありません。

C2311 (E) Integer required
switch 文の制御式 (文の実行を判定する式) がスカラ型ではありません。

C2312 (E) Missing (
if 文、while 文、for 文、do 文または switch 文の制御式 (文の実行を判定する式) の左括弧"("がありません。

C2313 (E) Missing ;
do 文の最後のセミコロン";"がありません。

C2314 (E) Scalar required
if 文の制御式 (文の実行を判定する式) がスカラ型ではありません。

C2316 (E) Illegal type for return value
return 文の式の型を関数の返す型に変換することができません。

C2400 (E) Illegal character "文字"
不正な文字があります。

C2401 (E) Incomplete character constant
文字定数の途中に改行があります。

C2402 (E) Incomplete string
文字列の途中に改行があります。

C2403 (E) EOF in comment
コメントの途中でファイルが終了しました。

C2404 (E) Illegal character code "文字コード"
不正な文字コードがあります。

C2405 (E) Null character constant
文字定数の中に文字を指定していません。すなわち''という形式の文字定数を指定しています。

C2407 (E) Incomplete logical line
空でないソースファイルの最後の文字にバックスラッシュ"＼"、またはバックスラッシュのあとに改行文字"＼[RET]"を指定しています。

C2408 (E) Comment nest too deep
コメントのネストが 255 レベルを超えています。

C2500 (E) Illegal token "語句"
語句の並びが文法に合っていません。

C2501 (E) Division by zero
定数式中で整数型または固定小数点型データのゼロ除算が行われました。

C2600 (E) 文字列
nolistfile オプションが指定されていなければ、#error の文字列で指定されたエラーメッセージをリストファイルに表示します。

C2650 (E) Invalid pointer reference
指定されたアドレス値がアライメント数と一致しません。

C2700 (E) Function "関数名" in #pragma interrupt already declared
割り込み関数宣言#pragma interrupt で指定した関数が、すでに通常の関数として宣言されています。

C2701 (E) Multiple interrupt for one function
1 つの関数に対して割り込み関数宣言#pragma interrupt を重複して宣言しています。

C2702 (E) Multiple #pragma interrupt options
同種の割り込み仕様が重複して指定されています。

C2703 (E) Illegal #pragma interrupt declaration
割り込み関数宣言#pragma interrupt の仕様に誤りがあります。

C2704 (E) Illegal reference to interrupt function
割り込み関数を不正に参照しています。

C2705 (E) Illegal parameter in interrupt function
割り込み関数で使用する引数の型が一致していません。

C2706 (E) Missing parameter declaration in interrupt function
割り込み関数のオプション指定で使用する変数の宣言がありません。

C2707 (E) Parameter out of range in interrupt function
割り込み関数のパラメータ tn の値が 256 を超えています。

C2709 (E) Illegal section name declaration
#pragma section 指定に誤りがあります。

C2710 (E) Section name too long
指定したセクション名の文字数が 31 文字を超えています。

C2711 (E) Section name table overflow
指定したセクションの数が 1 ファイルで 64 個を超えています。

C2712 (E) GBR based displacement overflow
#pragma gbr_base、#pragma gbr_base1 で宣言した変数の領域がオーバフローしました。

C2713 (E) Illegal #pragma interrupt function type
#pragma interrupt 指定した関数の型が不正です。

C2799 (E) GBR used in-line function
gbr=auto を指定した時は、GBR 関連組み込み関数は使用できません。

C2800 (E) Illegal parameter number in in-line function
組み込み関数で使用する引数の数が一致しません。

C2801 (E) Illegal parameter type in in-line function
組み込み関数で引数の型が一致しません。

C2802 (E) Parameter out of range in in-line function
組み込み関数で引数の大きさが指定可能範囲を超えています。

C2803 (E) Invalid offset value in in-line function
組み込み関数で引数の指定が不適当です。

C2804 (E) Illegal in-line function
指定された cpu オプションでは使用できない組み込み関数があります。

C2805 (E) Function "関数名" in #pragma inline/inline_asm already declared
"関数名"で示す関数の本体が、#pragma 指定よりも前にあります。

C2806 (E) Multiple #pragma for one function
1 つの関数に対して複数の矛盾した#pragma 指定をしています。

C2807 (E) Illegal #pragma inline/inline_asm declaration
#pragma inline または#pragma inline_asm の指定方法に誤りがあります。

C2808 (E) Illegal option for #pragma inline_asm
#pragma inline_asm の指定があるにもかかわらず、code=machinecode を指定しています。

C2809 (E) Illegal #pragma inline/inline_asm function type
#pragma inline または#pragma inline_asm を指定した識別子の種類が誤っています。

C2810 (E) Global variable "変数名" in #pragma gbr_base/gbr_base1 already declared
"変数名"で示す変数の定義が#pragma 指定よりも前にあります。

C2811 (E) Multiple #pragma for one global variable
変数に対して複数の矛盾する#pragma が指定されています。

C2812 (E) Illegal #pragma gbr_base/gbr_base1 declaration
#pragma gbr_base、#pragma gbr_base1 の指定方法が誤っています。

C2813 (E) Illegal #pragma gbr_base/gbr_base1 global variable type
#pragma gbr_base、#pragma gbr_base1 を指定した識別子の種類が誤っています。

C2814 (E) Function "関数名" in #pragma noregsave/noregalloc/regsave already declared
"関数名"で示す関数の本体が、#pragma 指定よりも前にあります。

C2815 (E) Illegal #pragma noregsave/noregalloc/regsave declaration
#pragma noregsave、#pragma noregalloc、#pragma regsave の指定方法が誤っています。

C2816 (E) Illegal #pragma noregsave/noregalloc/regsave function type
#pragma noregsave、#pragma noregalloc、#pragma regsave を指定した識別子の種類が誤っています。

C2817 (E) Symbol "識別子" in #pragma abs16/abs20/abs28/abs32 already declared
"識別子"で示す名前の宣言が、#pragma 指定よりも前にあります。

C2818 (E) Multiple #pragma for one symbol
同一の識別子に対して、複数の矛盾した#pragma が指定されています。

C2819 (E) Illegal #pragma abs16/abs20/abs28/abs32 declaration
#pragma abs16/abs20/abs28/abs32 の指定方法が誤っています。

C2820 (E) Illegal #pragma abs16/abs20/abs28/abs32 symbol type
#pragma abs16/abs20/abs28/abs32 を指定した識別子の種類が誤っています。

C2821 (E) Global variable "変数名" in #pragma global_register already declared
#pragma global_register を指定した変数名はすでに定義されています。

C2822 (E) Illegal register "レジスタ" in #pragma global_register
#pragma global_register を指定したレジスタが不正です。

C2823 (E) Illegal #pragma global_register declaration
#pragma global_register の指定方法が誤っています。

C2824 (E) Illegal #pragma global_register type
#pragma global_register を指定できない変数が指定されています。

C2828 (E) Illegal #pragma entry declaration
#pragma entry 宣言に構文エラーがあります。

C2829 (E) Function "関数名" in #pragma entry already declared
#pragma entry 宣言の前に同名のシンボルがあるか、または既に#pragma 指定されています。

C2830 (E) Illegal #pragma entry function type
指定されたシンボルが関数ではありません。

C2831 (E) Multiple #pragma entry declaration
#pragma entry 宣言が複数存在しています。

C2832 (E) Illegal #pragma stacksize declaration
#pragma stacksize 宣言に構文エラーがあります。

C2833 (E) Multiple #pragma stacksize declaration
#pragma stacksize 宣言が複数存在しています。

C2840 (E) Illegal #pragma ifunc declaration
#pramga ifunc 宣言に構文エラーがあります。

C2841 (E) Illegal #pragma ifunc function type
#pragma ifunc で指定した識別子の種類が誤っています。

C2842 (E) Function "関数名" in #pragma ifunc already declared
"関数名"で示す名前の宣言が、#pragma 指定よりも前にあります。

C2843 (E) Illegal floating type used in function
関数内で float や double を使用しています。#pragma ifunc 宣言する場合は float や double を使用しないでください。

C2844 (E) Illegal #pragma pack/unpack declaration
#pragma pack/unpack 宣言に構文エラーがあります。

C2845 (E) Illegal #pragma bit_order declaration
#pragma bit_order 宣言に構文エラーがあります。

C2846 (E) Packed structure used in in-line function
組み込み関数に pack 指定の構造体を使っています。

C2847 (E) Illegal #pragma tbr declaration
#pragma tbr の指定方法が誤っています。

C2848 (E) Function "関数名" in #pragma tbr already declared
"関数名"で示す関数の本体が、#pragma tbr 指定よりも前にあります。

C2849 (E) Illegal offset in #pragma tbr
オフセット値の指定が不正です。

C2850 (E) Illegal #pragma tbr function type
指定されたシンボルが関数ではありません。

C2851 (E) Too many function in tbr
tbr 指定された関数の数が制限値を超えています。

C2852 (E) Variable "変数名" in #pragma address already declared
#pragma address を指定した変数名はすでに定義されています。

C2853 (E) Illegal #pragma address symbol type
#pragma address で構造体/共用体のメンバもしくは変数名以外のシンボルが指定されています。

C2854 (E) Illegal address in #pragma address
指定アドレスが以下のいずれかに該当しています。
(1) アライメント数 4 の変数、構造体に 4 の倍数以外のアドレスを指定している。
(2) アライメント数 2 の変数、構造体に 2 の倍数以外のアドレスを指定している。
(3) 異なる変数に対して、同一アドレスを指定している。
(4) 異なる変数に対して、変数のアドレスが重なっている。
(5) #pragma abs16/abs20/abs28 指定があり、その範囲内に絶対アドレスが入ってない。
(6) abs16/abs20/abs28 オプション指定があり、その範囲内に絶対アドレスが入ってない。

C2855 (E) All registers are used in #pragma global_register
#pragma global_register 指定によりレジスタの数が足りません。

C2856 (E) Illegal usage of in-line function "関数名"
組み込み関数"関数名"の使用方法に誤りがあります。

C2857 (E) Function "関数名" in #pragma already declared
#pragma 宣言を行う前に定義した関数は指定できません。

C2858 (E) Illegal #pragma <拡張子> function type
#pragma <拡張子>で指定したシンボルが関数ではありません。

C2859 (E) Illegal #pragma <拡張子> declaration
#pragma <拡張子>宣言に構文エラーがあります。

C2900 (E) Incompatible memory qualifiers
メモリ型修飾子が異なるポインタ型への変換をしようとしました。

C2901 (E) Illegal type qualifier
型修飾子の指定にエラーがあります。

C2902 (E) Illegal arithmetic conversion
算術変換にエラーがあります。

C2903 (E) Illegal circ specification
_circ 指定にエラーがあります。

C3000 (F) Statement nest too deep
if 文、while 文、for 文、do 文および switch 文のネストが、限界値を超えています。

C3001 (F) Block nest too deep
複文のネストが、限界値を超えています。

C3006 (F) Too many parameters
関数の宣言または呼び出しにおいて引数の数が限界値を超えています。

C3007 (F) Too many macro parameters
マクロの定義または呼び出しにおいて、引数の数が限界値を超えています。

C3008 (F) Line too long
マクロ展開後の 1 行の長さが限界値を超えています。

C3009 (F) String literal too long
文字列の文字数が 32766 文字を超えています。文字列の文字数は、連続して指定した文字列を連結した後のバイト数です。ここでいう文字列の文字数とは、ソースプログラム上の長さではなく文字列のデータに含まれるバイト数で、拡張表記も 1 文字に数えます。

C3013 (F) Too many switches
switch 文の数が限界値を超えています。

C3014 (F) For nest too deep
for 文のネストが限界値を超えています。

C3015 (F) Symbol table overflow
コンパイラが生成するシンボルの数が限界値を超えています。

C3016 (F) Internal label overflow
コンパイラが生成する内部ラベルの数が限界値を超えています。

C3017 (F) Too many case labels
1 つの switch 文の中の case ラベルの数が限界値を超えています。

C3018 (F) Too many goto labels
1 つの関数の中で定義している goto ラベルの数が限界値を超えています。

C3019 (F) Cannot open source file "ファイル名"
ソースファイルをオープンすることができません。

C3020 (F) Source file input error "ファイル名"
ソースファイルまたはインクルードファイルを読み込むことができません。

C3021 (F) Memory overflow
コンパイラが内部で使用するメモリ領域を割り当てることができません。

C3022 (F) Switch nest too deep
switch 文のネストが限界値を超えています。

C3023 (F) Type nest too deep
基本型を修飾する型 (ポインタ型、配列型、関数型) の数が 16 個を超えています。

C3024 (F) Array dimension too deep
配列の次元数が 6 次元を超えています。

C3025 (F) Source file not found
コマンドラインの中にソースファイル名の指定がありません。

C3026 (F) Expression too complex
式が複雑すぎます。

C3027 (F) Source file too complex
プログラムの文のネストが深いかあるいは、式が複雑すぎます。

C3030 (F) Too many compound statements
1 関数における複文の数が、2048 を超えています。

C3031 (F) Data size overflow
配列または構造体の大きさが、2147483647 バイトを超えています。

C3100 (F) Misaligned pointer access
アライメントが正しくないポインタを用いて参照または設定をしようとしています。

C3201 (F) Object size overflow
オブジェクトサイズが 4G バイトを超えています。

C3203 (F) Assembly source line too long
出力するアセンブリソースの 1 行が長すぎます。

C3204 (F) Illegal stack access
関数内で使用するスタックのサイズ(局所変数領域、レジスタ退避領域その他関数呼び出しのためのパラメータプッシュ領域等含む)または、その関数呼び出しのためのパラメータ領域が 2G バイトを超えています。

C3205 (F) Cannot apply repeat operation for a loop
ループに対して、DSP 拡張リピートループへの展開ができませんでした。ループの本体を小さくするか、repeat オプションの指定を外してください。

C3300 (F) Cannot open internal file
以下、3 つの場合のいずれかでエラーが起こっている可能性があります。
(1) 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。
(2) 中間ファイルと同じ名前のファイルが既に存在しています。
(3) コンパイラが内部で使用するファイルをオープンすることができません。

C3301 (F) Cannot close internal file
コンパイラが内部で生成する中間ファイルをクローズすることができません。コンパイラのインストール手順に誤りがないことを確認してください。

C3302 (F) Cannot input internal file
コンパイラが内部で生成する中間ファイルを読み込むことができません。コンパイラのインストール手順に誤りがないことを確認してください。

C3303 (F) Cannot output internal file
コンパイラが内部で生成する中間ファイルに書き込むことができません。ディスクの空き容量を増やしてください。

C3304 (F) Cannot delete internal file
コンパイラが内部で生成する中間ファイルを削除することができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。

C3305 (F) Invalid command parameter "オプション"
コンパイラオプションの指定方法が誤っています。

C3306 (F) Interrupt in compilation
コンパイル処理中に標準入力端末から(Ctrl)+C コマンドによる割り込みを検出しました。

C3307 (F) Compiler version mismatch
コンパイラを構成するファイル間のバージョンが一致していません。インストールガイドの組み込み方法を参照し、コンパイラ本体を再インストールしてください。

C3308 (F) Cannot create file "ファイル名"
コンパイラが生成するファイルを作成できません。

C3320 (F) Command parameter buffer overflow
コマンドラインの指定が 4096 文字を超えています。

C3321 (F) Illegal environment variable
以下(1)～(5)のいずれかに該当しています。
(1)環境変数 SHC_LIB が設定されていません。
(2)環境変数 SHCPU に、
"SH1","SH2","SH2E","SH2DSP","SHDSP","SH2A","SH2AFPU","SH3","SH3DSP","SH4","SH4A","SH4ALDSP"以外の設定がされています。
(3)環境変数 SHC_TMP が設定されていません。
(4)環境変数 SHC_TMP の設定で実在しないフォルダを指定しています。
(5)環境変数 SHC_TMP のパス名に" "(ダブルクォーテーション)を記述しています。

C3322 (F) Current directory cannot be read to get its name
カレントフォルダの情報を取得できません。

C4000-C4999 (-) Internal error
コンパイラの内部処理で何らかの障害が生じました。本コンパイラをお求めになった営業所あるいは代理店にエラーの発生状況をご連絡ください。

C5003 (F) #include file "ファイル名" includes itself
自分自身のファイル"ファイル名"をインクルードしています。

C5004 (F) Out of memory
コンパイルに必要なメモリが不足しています。システムのメモリを増やすか、他のアプリケーションを終了してください。

C5005 (F) Could not open source file "名前"
ファイル"名前"をオープンできませんでした。ファイル名が正しいか確認してください。

C5006 (E) Comment unclosed at end of file
コメントの終了指定*/がありません。

C5007 (E) (I) Unrecognized token
認識できない字句があります(マクロの場合は(I)となります)。

C5008 (E) (I) Missing closing quote
文字列の終了指定"がありません(マクロの場合は(I)となります)。

C5009 (I) Nested comment is not allowed
/* */コメントがネストしています。

C5010 (E) "#" not expected here
#が行の先頭、プリプロセッサ以外に指定されています。

C5011 (E) Unrecognized preprocessing directive
認識できないプリプロセッサのキーワードがあります。

C5012 (E) Parsing restarts here after previous syntax error
字句の解析を再開しました。

C5013 (E) (F) Expected a file name
ファイル名が必要です。
#include 文では(F)、#line 文では(E)となります。

C5014 (E) Extra text after expected end of preprocessing directive
プリプロセッサ文の後にさらにテキストが記述されています。

C5016 (F) "名前" is not a valid source file name
ファイル"名前"が有効でありません。

C5017 (E) Expected a "]"
"]"がありません。

C5018 (E) Expected a ")"
")"がありません。

C5019 (E) Extra text after expected end of number
数値の後ろにさらにテキストが記述されています。

C5020 (E) Identifier "名前" is undefined
シンボル"名前"の定義がありません。

C5021 (W) Type qualifiers are meaningless in this declaration
意味のない型限定子を指定しています。型限定子を無効にします。

C5022 (E) Invalid hexadecimal number
16 進数の記述に誤りがあります。

C5024 (E) Invalid octal digit
8 進数の記述に誤りがあります。

C5025 (E) Quoted string should contain at least one character
文字定数が空です。

C5026 (E) Too many characters in character constant
文字定数中の文字数が多すぎます。

C5027 (W) Character value is out of range
文字の値が範囲を超えています。超えた値は切り捨てられます。

C5028 (E) Expression must have a constant value
式の値が定数でありません。

C5029 (E) Expected an expression
式が必要です。

C5030 (E) Floating constant is out of range
浮動小数点型の値が範囲を超えています。

C5031 (E) Expression must have integral type
式の型は整数型でなければなりません。

C5032 (E) Expression must have arithmetic type
式の型は算術型でなければなりません。

C5033 (E) Expected a line number
#line 文には行番号が必要です。

C5034 (E) Invalid line number
#line 文の行番号が有効でありません。

C5035 (F) #error directive: "行番号"
#error 文が適用されました。

C5036 (E) The #if for this directive is missing
#if 文の指定方法に誤りがあります。

C5037 (E) The #endif for this directive is missing
#endif 行の指定方法に誤りがあります。

C5038 (W) Directive is not allowed -- an #else has already appeared
#else 文はすでに出現しました。本指定を読み飛ばします。

C5039 (W) Division by zero
ゼロ除算が発生しました。

C5040 (E) Expected an identifier
識別子が必要です。

C5041 (E) Expression must have arithmetic or pointer type
式の型は算術型またはポインタ型でなければなりません。

C5042 (E) Operand types are incompatible ("型 1" and "型 2")
"型 1"と"型 2"のオペランドの型が適合しません。

C5044 (E) Expression must have pointer type
式の型はポインタ型でなければなりません。

C5045 (W) #undef may not be used on this predefined name
システムで定義しているマクロ名を取り消すことはできません。#undef 指定を無効にします。

C5046 (W) This predefined name may not be redefined
システムで定義しているマクロ名を再定義することはできません。#define 指定を無効にします。

C5047 (W) Incompatible redefinition of macro "名前" (declared at line "行番号")
マクロ"名前"の再定義が以前の定義と異なります。再定義したマクロを有効にします。

C5049 (E) Duplicate macro parameter name
マクロのパラメータ名を 2 重定義しています。

C5050 (E) "##" may not be first in a macro definition
#define マクロの最初に##が指定されています。

C5051 (E) "##" may not be last in a macro definition
#define マクロの最後に##が指定されています。

C5052 (E) Expected a macro parameter name
#に続くマクロ引数がありません。

C5053 (E) Expected a ":"
":"が必要です。

C5054 (W) Too few arguments in macro invocation
マクロ展開時の実引数が足りません。

C5055 (W) Too many arguments in macro invocation
マクロ展開時の実引数が多すぎます。

C5056 (E) Operand of sizeof may not be a function
sizeof 演算のオペランドに関数を指定できません。

C5057 (E) This operator is not allowed in a constant expression
この演算子は定数式中に指定できません。

C5058 (E) This operator is not allowed in a preprocessing expression
この演算子はプリプロセッサの式中で指定できません。

C5059 (E) Function call is not allowed in a constant expression
定数式中で関数呼び出しはできません。

C5060 (E) This operator is not allowed in an integral constant expression
この演算子は整数型定数式中で指定できません。

C5061 (W) Integer operation result is out of range
整数演算の結果が値の範囲を超えました。オーバフローした上位ビットを無視した値を仮定します。

C5062 (W) Shift count is negative
シフトカウントが負の値です。指定された通りに演算します。

C5063 (W) Shift count is too large
シフトカウントが有効ビット数を超えています。指定された通りに演算します。

C5064 (W) Declaration does not declare anything
宣言を指定するシンボルがありません。宣言を無視します。

C5065 (E) Expected a ";"
";"が必要です。

C5066 (E) Enumeration value is out of "int" range
列挙型メンバの値が int 型の範囲を超えました。

C5067 (E) Expected a "}"
"}"が必要です。

C5068 (W) Integer conversion resulted in a change of sign
符号変換を伴った整数型変換が実施されました。ビット列をそのまま設定します。

C5069 (W) Integer conversion resulted in truncation
上位バイト側を切り捨てる整数型変換が実施されました。切り捨て後の値を設定します。

C5070 (E) Incomplete type is not allowed
不完全型が指定されています。

C5071 (E) Operand of sizeof may not be a bit field
sizeof 演算子のオペランドにビットフィールドが指定されています。

C5075 (E) Operand of "*" must be a pointer
*演算子のオペランドの型がポインタ型ではありません。

C5077 (E) This declaration has no storage class or type specifier
記憶クラスまたは型の指定がありません。

C5079 (E) Expected a type specifier
型指定子が必要です。

C5080 (E) A storage class may not be specified here
ここでは記憶クラスを指定することはできません。

C5081 (E) More than one storage class may not be specified
記憶クラスを複数指定することはできません。

C5082 (I) Storage class is not first
記憶クラス指定子が先頭にかかれていません。

C5083 (W) Type qualifier specified more than once
const/volatile 限定子を複数指定しています。余分な指定を無視します。

C5084 (E) Invalid combination of type specifiers
型の組み合わせが正しくありません。

C5085 (E) Invalid storage class for a parameter
仮引数に不当な記憶クラスを指定しています。

C5086 (E) Invalid storage class for a function
関数に不当な記憶クラスを指定しています。

C5087 (E) A type specifier may not be used here
型を指定することはできません。

C5088 (E) Array of functions is not allowed
関数を要素とする配列は指定できません。

C5089 (E) Array of void is not allowed
void 型を要素とする配列は指定できません。

C5090 (E) Function returning function is not allowed
関数型をリターン型とする関数は指定できません。

C5091 (E) Function returning array is not allowed
配列をリターン型とする関数は指定できません。

C5093 (E) Function type may not come from a typedef
typedef 宣言された関数型を使用することはできません。

C5094 (E) The size of an array must be greater than zero
配列のサイズは 0 より大きな値でなければなりません。

C5095 (E) Array is too large
配列のサイズが大きすぎます。

C5097 (E) A function may not return a value of this type
関数はこの型の値を返すことができません。

C5098 (E) An array may not have elements of this type
配列はこの型を要素とすることができません。

C5100 (E) Duplicate parameter name
仮引数の名前が重複しています。

C5101 (E) "名前" has already been declared in the current scope
同一スコープ内にすでに"名前"の宣言が存在します。

C5103 (E) Class is too large
クラスのサイズが大きすぎます。

C5105 (E) Invalid size for bit field
ビットフィールドのサイズが不正です。

C5106 (E) Invalid type for a bit field
ビットフィールドの型が不正です。

C5107 (E) Zero-length bit field must be unnamed
長さ 0 のビットフィールドには名前をつけられません。

C5108 (W) Signed bit field of length 1
符号付整数型の長さ 1 のビットフィールドが指定されています。指定された型で処理します。

C5109 (E) Expression must have (pointer-to-) function type
式は関数型へのポインタ型でなければなりません。

C5110 (E) Expected either a definition or a tag name
宣言の定義またはタグ名が必要です。

C5111 (I) Statement is unreachable
実行されない文です。最適化により削除される可能性があります。

C5112 (E) Expected "while"
while キーワードが必要です。

C5114 (E) Entity-kind "名前" was referenced but not defined
参照される"名前"の定義がありません。

C5115 (E) A continue statement may only be used within a loop
continue 文はループの中で有効です。

C5116 (E) A break statement may only be used within a loop or switch
break 文はループまたは switch 文の中で有効です。

C5117 (W) Non-void entity-kind "名前" should return a value
void 型でない関数がリターン値を返しません。リターン値は不定です。

C5118 (E) A void function may not return a value
void 型を返す関数はリターン値を返すことはできません。

C5119 (E) Cast to type "型" is not allowed
"型"へのキャストは指定できません。

C5120 (E) Return value type does not match the function type
リターン値と関数の型が合いません。

C5121 (E) A case label may only be used within a switch
case ラベルを switch 文以外で使用しています。

C5122 (E) A default label may only be used within a switch
default ラベルを switch 文以外で使用しています。

C5123 (E) Case label value has already appeared in this switch
case ラベルの値がすでに switch 文の中に存在します。

C5124 (E) Default label has already appeared in this switch
default ラベルの値がすでに switch 文の中に存在します。

C5125 (E) Expected a "("
"("が必要です。

C5126 (E) Expression must be an lvalue
式は左辺値でなければなりません。

C5127 (E) Expected a statement
文が必要です。

C5128 (I) Loop is not reachable from preceding code
実行されない繰り返し文です。

C5129 (E) A block-scope function may only have extern storage class
ブロック内で宣言された関数は extern 記憶クラスでなければなりません。

C5130 (E) Expected a "{"
"{"が必要です。

C5131 (E) Expression must have pointer-to-class type
式はクラスへのポインタ型でなければなりません。

C5132 (E) Expression must have pointer-to-struct-or-union type
式は構造体または共用体へのポインタ型でなければなりません。

C5133 (E) Expected a member name
メンバ名が必要です。

C5134 (E) Expected a field name
フィールド名が必要です。

C5135 (E) Entity-kind "名前" has no member "メンバ名"
"名前"は"メンバ名"を持ちません。

C5136 (E) Entity-kind "名前" has no field "フィールド名"
"名前"は"フィールド名"を持ちません。

C5137 (E) Expression must be a modifiable lvalue
式は修正可能な左辺値でなければなりません。

C5139 (E) Taking the address of a bit field is not allowed
ビットフィールドのアドレスを参照することはできません。

C5140 (E) Too many arguments in function call
関数呼び出しの実引数の数が多すぎます。

C5142 (E) Expression must have pointer-to-object type
式はオブジェクトへのポインタ型でなければなりません。

C5143 (F) Program too large or complicated to compile
プログラムが大きすぎるかまたは複雑すぎます。

C5144 (E) A value of type "型 1" cannot be used to initialize an entity of type "型 2"
初期値の"型 1"と変数の"型 2"が異なります。

C5145 (E) Entity-kind "名前" may not be initialized
"名前"を初期化することはできません。

C5146 (E) Too many initializer values
初期値の数が多すぎます。

C5147 (E) Declaration is incompatible with "名前" (declared at line "行番号")
前に宣言した"名前"の型が合致しません。

C5148 (E) Entity-kind "名前" has already been initialized
すでに"名前"の初期値が設定されています。

C5149 (E) A global-scope declaration may not have this storage class
大域的なスコープでの宣言にはこの記憶クラスを指定できません。

C5150 (E) A type name may not be redeclared as a parameter
型名を仮引数で再宣言することはできません。

C5151 (E) A typedef name may not be redeclared as a parameter
型名を仮引数で再宣言することはできません。

C5153 (E) Expression must have class type
式はクラス型でなければなりません。

C5154 (E) Expression must have struct or union type
式は構造体または共用体型でなければなりません。

C5157 (E) Expression must be an integral constant expression
式は整数型の定数式でなければなりません。

C5158 (E) Expression must be an lvalue or a function designator
式は左辺値または関数名でなければなりません。

C5159 (E) Declaration is incompatible with previous "名前" (declared at line "行番号")
前に使用した"名前"の型と合致しません。

C5160 (E) Name conflicts with previously used external name "名前"
前に使用した外部名"名前"と名前が重なります。

C5161 (I) Unrecognized #pragma
認識できない#pragma 指定があります。#pragma 指定を無視します。

C5163 (F) Could not open temporary file "名前"
テンポラリファイル"名前"をオープンできませんでした。コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5164 (F) Name of directory for temporary files is too long ("名前")
テンポラリファイルの"名前"が長すぎます。

C5165 (E) Too few arguments in function call
関数呼び出しの実引数の数が足りません。

C5166 (E) Invalid floating constant
浮動小数点定数の指定が不正です。

C5167 (E) Argument of type "型 1" is incompatible with parameter of type "型 2"
実引数の型"型 1"と仮引数の型"型 2"とが合致しません。

C5168 (E) A function type is not allowed here
関数型は許されません。

C5169 (E) Expected a declaration
宣言が必要です。

C5170 (W) Pointer points outside of underlying object
ポインタが指している領域がオブジェクトの範囲を超えています。

C5171 (E) Invalid type conversion
キャストの型が不正です。

C5172 (I) External/internal linkage conflict with previous declaration
前の宣言と外部/内部リンケージが異なります。内部リンケージが仮定されます。

C5173 (E) Floating-point value does not fit in required integral type
浮動小数点型の値を整数型に変換するときに値の範囲を超えました。

C5174 (I) Expression has no effect
効果のない式です。最適化で削除される可能性があります。

C5175 (W) Subscript out of range
配列のインデックスが範囲を超えています。指定されたインデックスで処理を継続します。

C5177 (W) (I) Entity-kind "entity" was declared but never referenced
参照されない宣言があります。

C5179 (W) Right operand of "%" is zero
%演算子の右辺が値 0 です。指定された式で評価します。

C5182 (F) Could not open source file "名前" (no directories in search list)
ファイル"名前"をオープンできませんでした。フォルダが存在するかどうか確認してください。

C5183 (E) Type of cast must be integral
キャストの型は整数型でなければなりません。

C5184 (E) Type of cast must be arithmetic or pointer
キャストの型は算術型またはポインタ型でなければなりません。

C5185 (I) Dynamic initialization in unreachable code
初期化式は実行されません。実行時に初期値は設定されません。

C5186 (W) Pointless comparison of unsigned integer with zero
0 と符号無し整数の無意味な比較をしています。指定された通りに式を評価します。

C5187 (I) Use of "=" where "==" may have been intended
"=="が意図される式で"="が使われています。指定された通りに式を評価します。

C5189 (F) Error while writing "ファイル名" file
ファイルの書き込みに失敗しました。

C5191 (W) Type qualifier is meaningless on cast type
キャストの型に意味のない型限定子を指定しています。指定された型を無視します。

C5192 (W) Unrecognized character escape sequence
認識できないエスケープシーケンス文字を指定しています。値をそのまま使用します。

C5193 (I) Zero used for undefined preprocessing identifier
プリプロセッサ文の式評価に値 0 が使われました。指定された通りに式を評価します。

C5219 (F) Error while deleting file "ファイル名"
ファイル"ファイル名"を削除することができません。

C5221 (W) Floating-point value does not fit in required floating-point type
浮動小数点型を要求された浮動小数点型に変換できません。無限大の値とみなします。

C5224 (W) The format string requires additional arguments
フォーマット文字列で要求する引数より実引数の数が足りません。

C5225 (W) The format string ends before this argument
フォーマット文字列が要求する引数より実引数の数が多すぎます。

C5226 (W) Invalid format string conversion
フォーマット変換の形式が実引数の型と異なります。

C5228 (W) Trailing comma is nonstandard
リストの最後の要素に与える値の直後にコンマをつけるのは標準形式ではありません。

C5229 (W) Bit field cannot contain all values of the enumerated type
ビットフィールドが列挙型全ての値を保持できません。結果は切り捨てられます。

C5235 (E) Variable "名前" was declared with a never-completed type
変数"名前"が不完全型のまま宣言されました。

C5236 (W) (I) Controlling expression is constant
制御式が定数です(I)。制御式がアドレス定数です(W)。指定された通りに式を評価します。

C5237 (I) Selector expression is constant
switch 文の制御式が定数です。

C5238 (E) Invalid specifier on a parameter
引数宣言で不正な指定子を使用しています。

C5239 (E) Invalid specifier outside a class declaration
クラス宣言外で不正な指定子を使用しています。

C5240 (E) Duplicate specifier in declaration
1 つの宣言内で指定子を重複して使用しています。

C5241 (E) A union is not allowed to have a base class
union 型は基底クラスを持つことはできません。

C5242 (E) Multiple access control specifiers are not allowed
アクセス指定子が重複して使われています。

C5243 (E) Class or struct definition is missing
class 定義の括弧の対応がとれません。

C5244 (E) Qualified name is not a member of class "型" or its base classes
限定名がクラスまたは基底クラスのメンバの"型"でありません。

C5245 (E) A nonstatic member reference must be relative to a specific object
非静的メンバの参照がオブジェクトに対応していません。

C5246 (E) A nonstatic data member may not be defined outside its class
非静的データメンバはクラス外で定義できません。

C5247 (E) Entity-kind "名前" has already been defined
"名前"はすでに定義されています。

C5248 (E) Pointer to reference is not allowed
リファレンス型へのポインタ型は許されません

C5249 (E) Reference to reference is not allowed
リファレンス型へのリファレンス型は許されません。

C5250 (E) Reference to void is not allowed
void 型へのリファレンス型は許されません。

C5251 (E) Array of reference is not allowed
リファレンス型の配列は許されません。

C5252 (E) Reference entity-kind "名前" requires an initializer
リファレンス型の定義"名前"には初期値が必要です。

C5253 (E) Expected a ","
カンマ","が必要です。

C5254 (E) Type name is not allowed
型名は許されません。

C5255 (E) Type definition is not allowed
型の定義は許されません。

C5256 (E) Invalid redeclaration of type name "名前" (declared at line "行番号")
型名"名前"を再定義することはできません。

C5257 (E) Const entity-kind "名前" requires an initializer
const 型の定義"名前"には初期値が必要です。

C5258 (E) "this" may only be used inside a nonstatic member function
"this"が非静的メンバ関数以外で使われています。

C5259 (E) Constant value is not known
const 型の値が不明です。

C5261 (I) Access control not specified ("名前" by default)
基底クラスのアクセス制御指定がありません。アクセス制御指定"名前"が仮定されます。

C5262 (E) Not a class or struct name
基底クラスで指定されたクラスまたは構造体がありません。

C5263 (E) Duplicate base class name
基底クラスを二重に指定しています。

C5264 (E) Invalid base class
基底クラスが不正です。

C5265 (E) Entity-kind "名前" is inaccessible
"名前"をアクセスすることはできません。

C5266 (E) "名前" is ambiguous
指定された"名前"があいまいです。

C5269 (E) Implicit conversion to inaccessible base class "型" is not allowed
アクセス不可能なクラスへの暗黙の型変換は許されません。

C5274 (E) Improperly terminated macro invocation
マクロ呼び出しの途中でファイルが終了しました。

C5276 (E) Name followed by "::" must be a class or namespace name
::に続く名前はクラス名または namespace 名でなければなりません。

C5277 (E) Invalid friend declaration
フレンド宣言の指定が正しくありません。

C5278 (E) A constructor or destructor may not return a value
コンストラクタやデストラクタはリターン値を持てません。

C5279 (E) Invalid destructor declaration
デストラクタの宣言が正しくありません。

C5280 (E)(W) Declaration of a member with the same name as its class
クラス名と同じ名前のメンバ名を宣言しています。
(W) 非 static 変数名
(E) static 変数名,typedef 名,enum メンバなど

C5281 (E) Global-scope qualifier (leading "::") is not allowed
グローバルなスコープ決定演算子は許されません。

C5282 (E) The global scope has no "名前"
"名前"がグローバルなスコープに宣言されていません。

C5283 (E) Qualified name is not allowed
限定名は許されません。

C5284 (W) NULL reference is not allowed
NULL へのリファレンスは許されません。指定された通りに式を評価します。

C5285 (E) Initialization with "{...}" is not allowed for object of type "型"
"型"のオブジェクトに{}形式の初期化は許されません。

C5286 (E) Base class "type" is ambiguous
基底クラスの型があいまいです。

C5287 (E) Derived class "type" contains more than one instance of class "型"
派生型が複数の同一クラス"型"を含みます。

C5288 (E) Cannot convert pointer to base class "型1" to pointer to derived class "型2" -- base class is virtual
仮想基底クラス"型1"のポインタ型を派生クラス"型2"のポインタ型に変換することはできません。

C5289 (E) No instance of constructor "名前" matches the argument list
コンストラクタ"名前"の引数が一致しません。

C5290 (E) Copy constructor for class "型" is ambiguous
クラス"型"のコピーコンストラクタがあいまいです。

C5291 (E) No default constructor exists for class "型"
クラス"型"のデフォルトコンストラクタは存在しません。

C5292 (E) "名前" is not a nonstatic data member or base class of class "型"
"名前"が非静的データメンバまたは基底クラス"型"でありません。

C5293 (E) Indirect nonvirtual base class is not allowed
仮想でない間接基底クラスは許されません。

C5294 (E) Invalid union member -- class "型" has a disallowed member function
union メンバに指定できないクラス"型"のメンバ関数があります。

C5297 (E) Expected an operator
演算子が必要です。

C5298 (E) Inherited member is not allowed
継承されたメンバを使用することはできません。

C5299 (E) Cannot determine which instance of entity-kind "名前" is intended
オーバーロード関数の"名前"を決定できません。

C5300 (E) A pointer to a bound function may only be used to call the function
メンバ関数へのポインタを関数呼び出し以外に使用しています。

C5302 (E) Entity-kind "名前" has already been defined
関数"名前"はすでに定義されています。

C5304 (E) No instance of entity-kind "名前" matches the argument list
関数"名前"の引数が一致しません。

C5305 (E) Type definition is not allowed in function return type declaration
関数のリターン型の宣言で型の定義をすることはできません。

C5306 (E) Default argument not at end of parameter list
デフォルト引数の宣言がパラメータリストの最後ではありません。

C5307 (E) Redefinition of default argument
デフォルト引数を再定義しています。

C5308 (E) More than one instance of entity-kind "名前" matches the argument list:
引数リストが一致するためオーバーロード関数"名前"があいまいです。

C5309 (E) More than one instance of constructor "名前" matches the argument list:
引数リストが一致するためコンストラクタ"名前"があいまいです。

C5310 (E) Default argument of type "型1" is incompatible with parameter of type "型2"
デフォルト値の"型1"が引数の"型2"に合致しません。

C5311 (E) Cannot overload functions distinguished by return type alone
リターン型が異なる関数をオーバーロードすることはできません。

C5312 (E) No suitable user-defined conversion from "型1" to "型2" exists
適切な利用者定義変換"型1"から"型2"が存在しません。

C5313 (E) Type qualifier is not allowed on this function
関数に型限定子(const,volatile)を指定することはできません。

C5314 (E) Only nonstatic member functions may be virtual
静的メンバ関数にvirtualを指定しています。

C5315 (E) The object has type qualifiers that are not compatible with the member function
オブジェクトの型限定子(const,volatile)がメンバ関数の型限定子と合致しません。

C5316 (E) Program too large to compile (too many virtual functions)
仮想関数の数が多すぎます。

C5317 (E) Return type is not identical to nor covariant with return type "型" of overridden virtual function entity-kind "名前"
仮想関数"名前"のリターン型"型"が異なります。

C5318 (E) Override of virtual entity-kind "名前" is ambiguous
仮想関数"名前"の置き換えがあいまいです。

C5319 (E) Pure specifier ("= 0") allowed only on virtual functions
純粋指定子"=0"を仮想関数以外に指定しています。

C5320 (E) Badly-formed pure specifier (only "= 0" is allowed)
純粋指定子の形式が正しくありません。"=0"だけが許されます。

C5321 (E) Data member initializer is not allowed
データメンバの初期化指定が正しくありません。

C5322 (E) Object of abstract class type "型" is not allowed:
抽象クラス"型"のオブジェクトは定義できません。

C5323 (E) Function returning abstract class "型" is not allowed:
抽象クラス"型"を返す関数は定義できません。

C5324 (I) Duplicate friend declaration
フレンド宣言が重複して指定されています。

C5325 (E) Inline specifier allowed on function declarations only
inline 指定子は関数宣言でのみ有効です。

C5326 (E) "inline" is not allowed
inline 指定は許されません。

C5327 (E) Invalid storage class for an inline function
inline 関数の記憶クラスが不正です。

C5328 (E) Invalid storage class for a class member
クラスメンバの記憶クラスが不正です。

C5329 (E) Local class member entity-kind "名前" requires a definition
局所クラスメンバ"名前"の定義がありません。

C5330 (E) Entity-kind "名前" is inaccessible
"名前"をアクセスできません。

C5332 (E) Class "type" has no copy constructor to copy a const object
クラス"型"に const 型オブジェクトをコピーするコピーコンストラクタがありません。

C5333 (E) Defining an implicitly declared member function is not allowed
暗黙宣言されたメンバ関数を定義することはできません。

C5334 (E) Class "型" has no suitable copy constructor
クラス"型"に適切なコピーコンストラクタが存在しません。

C5335 (E) Linkage specification is not allowed
リンケージ指定子を指定することはできません。

C5336 (E) Unknown external linkage specification
認識できないリンケージ指定が指定されました。

C5337 (E) Linkage specification is incompatible with previous "名前" (declared at line "行番号")
前に指定されたリンケージ指定子"名前"と合致しません。

C5338 (E) More than one instance of overloaded function "名前" has "C" linkage
C リンケージを持ったオーバーロード関数"名前"が複数あります。

C5339 (E) Class "型" has more than one default constructor
クラス"型"は複数のデフォルトコンストラクタを持っています。

C5340 (E) Value copied to temporary, reference to temporary used
値がローカルな領域にコピーされました。ローカルな領域への参照が使用されます。

C5341 (E) "operator 演算子" must be a member function
演算子関数"演算子"はメンバ関数でなければなりません。

C5342 (E) Operator may not be a static member function
静的メンバ関数の演算子関数は許されません。

C5343 (E) No arguments allowed on user-defined conversion
利用者定義変換に引数は許されません。

C5344 (E) Too many parameters for this operator function
演算子関数の引数の数が多すぎます。

C5345 (E) Too few parameters for this operator function
演算子関数の引数の数が足りません。

C5346 (E) Nonmember operator requires a parameter with class type
メンバ関数でない演算子関数はクラス型を引数に持つ必要があります。

C5347 (E) Default argument is not allowed
デフォルト引数は許されません。

C5348 (E) More than one user-defined conversion from "型 1" to "型 2" applies:
"型 1"から"型 2"への利用者定義型変換があいまいです。

C5349 (E) No operator "演算子" matches these operands
演算子関数"演算子"のオペランドが一致しません。

C5350 (E) More than one operator "演算子" matches these operands:
演算子関数"演算子"のオペランドがあいまいです。

C5351 (E) First parameter of allocation function must be of type "size_t"
operator new の第 1 パラメータは size_t 型でなければなりません。

C5352 (E) Allocation function requires "void *" return type
operator new のリターン型は void *型でなければなりません。

C5353 (E) Deallocation function requires "void" return type
operator delete のリターン型は void 型でなければなりません。

C5354 (E) First parameter of deallocation function must be of type "void *"
operator delete の第 1 パラメータは void *型でなければなりません。

C5356 (E) Type must be an object type
型はオブジェクト型でなければなりません。

C5357 (E) Base class "type" has already been initialized
基底クラスはすでに初期化されています。

C5359 (E) Entity-kind "名前" has already been initialized
"名前"はすでに初期化されています。

C5360 (E) Name of member or base class is missing
メンバ名または基底クラスに誤りがあります。

C5363 (E) Invalid anonymous union -- nonpublic member is not allowed
無名 union のメンバが公開メンバでありません。

C5364 (E) Invalid anonymous union -- member function is not allowed
無名 union にメンバ関数は許されません。

C5365 (E) Anonymous union at global or namespace scope must be declared static
グローバルまたは namespace スコープの無名 union は static 宣言が必要です。

C5366 (E) Entity-kind "名前" provides no initializer for:
"名前"に初期化指定はできません。

C5367 (E) Implicitly generated constructor for class "型" cannot initialize:
暗黙に生成されたクラス"型"のコンストラクタを初期化することはできません。

C5368 (W) Entity-kind "名前" defines no constructor to initialize the following:
"名前"は初期化のためのコンストラクタを定義していません。

C5369 (E) Entity-kind "名前" has an uninitialized const or reference member
"名前"の const またはリファレンスメンバが初期化されていません。

C5370 (W) Entity-kind "名前" has an uninitialized const field
"名前"の const フィールドが初期化されていません。

C5371 (E) Class "型" has no assignment operator to copy a const object
const オブジェクトをコピーするクラス"型"の代入演算子関数が定義されていません。

C5372 (E) Class "型" has no suitable assignment operator
クラス"型"に適当な代入演算が定義されていません。

C5373 (E) Ambiguous assignment operator for class "型"
クラス"型"の代入演算子関数があいまいです。

C5375 (E) Declaration requires a typedef name
typedef 名の宣言が必要です。

C5377 (E) "virtual" is not allowed
virtual を指定することはできません。

C5378 (E) "static" is not allowed
static を指定することはできません。

C5380 (E) Expression must have pointer-to-member type
式はメンバへのポインタ型でなければなりません。

C5381 (I) Extra ";" ignored
余分な";"を無視します。

C5382 (W) Nonstandard member constant declaration (standard form is a static const integral member)
const メンバの宣言が標準形式でありません。初期化は無効です。

C5384 (E) No instance of overloaded "名前" matches the argument list
オーバーロード関数"名前"の引数リストが一致しません。

C5386 (E) No instance of entity-kind "名前" matches the required type
要求される型のオーバーロード関数"名前"がありません。

C5388 (E) "operator->" for class "型 1" returns invalid type "型 2"
クラス"型 1"の operator->演算関数のリターン型"型 2"が正しくありません。

C5389 (E) A cast to abstract class "型" is not allowed:
抽象クラス"型"へのキャストは許されません。

C5391 (E) A new-initializer may not be specified for an array
配列を new によって初期化することはできません。

C5392 (E) Member function "名前" may not be redeclared outside its class
メンバ関数"名前"がクラスの外側で再宣言されました。

C5393 (E) Pointer to incomplete class type is not allowed
不完全クラスへのポインタ型は許されません。

C5394 (E) Reference to local variable of enclosing function is not allowed
ローカルクラスを囲む関数の局所変数へのリファレンスは許されません。

C5397 (E) Implicitly generated assignment operator cannot copy:
暗黙に生成された代入演算子関数がオブジェクトを正しくコピーすることができません。

C5399 (I) Entity-kind "名前" has an operator newxxxx() but no default operator deletexxxx()
"名前"が operator new を持ちますがデフォルトの operator delete を持ちません。

C5400 (I) Entity-kind "名前" has a default operator deletexxxx() but no operator newxxxx()
"名前"がデフォルトの operator delete を持ちますが operator new を持ちません。

C5401 (E) Destructor for base class "型" is not virtual
基底クラス"型"のデストラクタが virtual でありません。

C5403 (E) Entity-kind "名前" has already been declared
メンバ関数"名前"が再宣言されています。

C5404 (E) Function "main" may not be declared inline
main 関数を inline 宣言することはできません。

C5405 (E) Member function with the same name as its class must be a constructor
クラス名と同じ名前のメンバ関数はコンストラクタでなければなりません。

C5407 (E) A destructor may not have parameters
デストラクタは引数を持つことができません。

C5408 (E) Copy constructor for class "型 1" may not have a parameter of type "型 2"
クラス"型 1"のコピーコンストラクタは"型 2"の引数を持つことはできません。

C5409 (E) Entity-kind "名前" returns incomplete type "型"
関数"名前"のリターン型が不完全型"型"です。

C5410 (E) Protected entity-kind "名前" is not accessible through a "型" pointer or object
限定公開名"名前"は"型"へのポインタやオブジェクトを経由してアクセスすることはできません。

C5411 (E) A parameter is not allowed
仮引数は許されません。

C5412 (E) An "asm" declaration is not allowed here
asm 宣言は許されません。

C5413 (E) No suitable conversion function from "型 1" to "型 2" exists
"型 1"から"型 2"への適切な変換関数が存在しません。

C5414 (W) Delete of pointer to incomplete class
不完全型クラスへのポインタは削除されました。

C5415 (E) No suitable constructor exists to convert from "型 1" to "型 2"
"型 1"から"型 2"へ変換する適切なコンストラクタが存在しません。

C5416 (E) More than one constructor applies to convert from "型 1" to "型 2":
"型 1"から"型 2"へ変換するコンストラクタがあいまいです。

C5417 (E) More than one conversion function from "型 1" to "型 2" applies:
"型 1"から"型 2"への変換関数があいまいです。

C5418 (E) More than one conversion function from "型" to a built-in type applies:
"型"からビルトイン型への変換関数があいまいです。

C5424 (E) A constructor or destructor may not have its address taken
コンストラクタまたはデストラクタのアドレスを参照することはできません。

C5427 (E) Qualified name is not allowed in member declaration
限定名をメンバ宣言のなかで使用できません。

C5429 (E) The size of an array in "new" must be non-negative
new で指定された配列のサイズに負の値は許されません。

C5430 (W) Returning reference to local temporary
関数内にローカルな領域のリファレンスをリターン値にしています。

C5432 (E) "enum" declaration is not allowed
列挙型宣言は許されません。

C5433 (E) Qualifiers dropped in binding reference of type "型 1" to initializer of type "型 2"
const/volatile 限定の型"型 2"が参照型"型 1"の初期値に指定されました。

C5434 (E) A reference of type "型 1" (not const-qualified) cannot be initialized with a value of type "型 2"
const 型修飾されない型"型 1"へのリファレンスを"型 2"の値で初期化できません。

C5435 (E) A pointer to function may not be deleted
関数へのポインタを削除することはできません。

C5436 (E) Conversion function must be a nonstatic member function
変換関数は非静的メンバ関数でなければなりません。

C5437 (E) Template declaration is not allowed here
このスコープ内でテンプレート宣言は許されません。

C5438 (E) Expected a "<"
"<"が必要です。

C5439 (E) Expected a ">"
">"が必要です。

C5440 (E) Template parameter declaration is missing
テンプレートの引数宣言が正しくありません。

C5441 (E) Argument list for entity-kind "名前" is missing
テンプレート"名前"の実引数リストが正しくありません。

C5442 (E) Too few arguments for entity-kind "名前"
テンプレート"名前"の実引数が足りません。

C5443 (E) Too many arguments for entity-kind "名前"
テンプレートの実引数が多すぎます。

C5445 (E) Entity-kind "名前 1" is not used in declaring the parameter types of entity-kind "名前 2"
テンプレート"名前 1"の引数"名前 2"が使用されません。

C5449 (E) More than one instance of entity-kind "名前" matches the required type
オーバーロード関数"名前"があいまいです。

C5452 (E) Return type may not be specified on a conversion function
変換関数のリターン型が指定されていません。

C5456 (E) Excessive recursion at instantiation of entity-kind "名前"
テンプレート"名前"のインスタンスが再帰的に生成されます。

C5457 (E) "名前" is not a function or static data member
"名前"が関数または静的データメンバではありません。

C5458 (E) Argument of type "型 1" is incompatible with template parameter of type "型 2"
実引数の型"型 1"がテンプレートの引数"型 2"に合致しません。

C5459 (E) Initialization requiring a temporary or conversion is not allowed
初期化にテンポラリや変換を要求することは許されません。

C5461 (E) Initial value of reference to non-const must be an lvalue
const 型を持たないリファレンスの初期値は左辺値でなければなりません。

C5463 (E) "template" is not allowed
"template"指定は許されません。

C5464 (E) "型" is not a class template
"型"がクラステンプレートではありません。

C5466 (E) "main" is not a valid name for a function template
"main"は関数テンプレートの名前に使用できません。

C5467 (E) Invalid reference to entity-kind "名前" (union/nonunion mismatch)
"名前"の参照が不正です。

C5468 (E) A template argument may not reference a local type
テンプレートの実引数はローカルな型を参照できません。

C5469 (E) Tag kind of "名前 1" is incompatible with declaration of entity-kind "名前 2" (declared at line "行番号")
タグ名"名前 1"の種類と"名前 2"の宣言が合致しません。

C5470 (E) The global scope has no tag named "名前"
グローバルスコープにタグ名"名前"がありません。

C5471 (E) Entity-kind "名前 1" has no tag member named "名前 2"
"名前 1"はタグメンバ"名前 2"を持ちません。

C5473 (E) Entity-kind "名前" may be used only in pointer-to-member declaration
typedef 名"名前"はメンバへのポインタ型の宣言の中で使用されなければなりません。

C5475 (E) A template argument may not reference a non-external entity
テンプレートの実引数は外部名以外を参照できません。

C5476 (E) Name followed by "::~" must be a class name or a type name
::~に続く名前はクラス名または型名でなければなりません。

C5477 (E) Destructor name does not match name of class "型"
クラス名"型"とデストラクタ名が合致しません。

C5478 (E) Type used as destructor name does not match type "型"
デストラクタ名で使われた型と"型"が合致しません。

C5479 (I) Entity-kind "名前" redeclared "inline" after being called
関数が呼ばれたあとに inline"名前"を宣言しています。以降 inline 指定を有効にします。

C5481 (E) Invalid storage class for a template declaration
テンプレート宣言の記憶クラス指定が正しくありません。

C5484 (E) Invalid explicit instantiation declaration
テンプレートの実引数が不正です。

C5485 (E) Entity-kind "名前" is not an entity that can be instantiated
テンプレート"名前"を実体化できません。

C5486 (E) Compiler generated entity-kind "entity" cannot be explicitly instantiated
コンパイラが生成した関数を実体化することはできません。

C5487 (E) Inline entity-kind "名前" cannot be explicitly instantiated
インライン関数"名前"を実体化することはできません。

C5488 (E) Pure virtual entity-kind "名前" cannot be explicitly instantiated
純粋仮想関数"名前"を実体化することはできません。

C5489 (E) Entity-kind "名前" cannot be instantiated -- no template definition was supplied
テンプレート定義がないため"名前"を実体化することはできません。

C5490 (E) Entity-kind "名前" cannot be instantiated -- it has been explicitly specialized
"名前"を実体化することはできません。

C5493 (E) No instance of entity-kind "名前" matches the specified type
オーバーロード関数"名前"と指定された型が合致しません。

C5496 (E) Template parameter "名前" may not be redeclared in this scope
テンプレート引数"名前"がスコープ内で再宣言されています。

C5497 (W) Declaration of "名前" hides template parameter
"名前"の宣言はテンプレート引数を隠蔽します。

C5498 (E) Template argument list must match the parameter list
テンプレート実引数と仮引数が合致しません。

C5499 (E) Conversion function to convert from "型 1" to "型 2" is not allowed
"型 1"から"型 2"への変換関数は許されません。

C5500 (E) Extra parameter of postfix "operatorxxxx" must be of type "int"
後置演算関数の第 2 パラメータの型は int 型でなければなりません。

C5501 (E) An operator name must be declared as a function
演算子名は関数として宣言しなければなりません。

C5502 (E) Operator name is not allowed
演算子名は許されません。

C5503 (E) Entity-kind "名前" cannot be specialized in the current scope
スコープ内で"名前"があいまいです。

C5505 (E) Too few template parameters -- does not match previous declaration
テンプレートの引数が足りません。

C5506 (E) Too many template parameters -- does not match previous declaration
テンプレートの引数が多すぎます。

C5507 (E) Function template for operator delete(void *) is not allowed
operator delete(void *)の関数テンプレートは許されません。

C5508 (E) Class template and template parameter may not have the same name
クラステンプレートとテンプレートの引数が同じ名前です。

C5510 (E) A template argument may not reference an unnamed type
テンプレートの実引数が名前付けされていない型を参照しています。

C5511 (E) Enumerated type is not allowed
列挙型は許されません。

C5512 (W) Type qualifier on a reference type is not allowed
リファレンス型に const/volatile 修飾を指定することはできません。

C5513 (E) A value of type "型 1" cannot be assigned to an entity of type "型 2"
型不一致のため"型 1"の値を"型 2"の実体に代入することができません。

C5514 (W) Pointless comparison of unsigned integer with a negative constant
負の定数と符号無し整数を比較しています。

C5515 (E) Cannot convert to incomplete class "型"
不完全型"型"への型変換はできません。

C5516 (E) Const object requires an initializer
const 型のオブジェクトには初期値が必要です。

C5517 (E) Object has an uninitialized const or reference member
オブジェクトが未初期化の const 型メンバあるいはリファレンス型メンバを持ちます。

C5519 (E) Entity-kind "名前" may not have a template argument list
"名前"はテンプレート実引数を持つことができません。

C5520 (E) Initialization with "{...}" expected for aggregate object
集成型のオブジェクトは{...}の形式で初期化しなければなりません。

C5521 (E) Pointer-to-member selection class types are incompatible ("型 1" and "型 2")
メンバへのポインタ型のクラスの型が"型 1"と"型 2"で合致しません。

C5522 (W) Pointless friend declaration
自分自身へのフレンド宣言をしています。

C5526 (E) A parameter may not have void type
void 型の引数は指定できません。

C5529 (E) This operator is not allowed in a template argument expression
テンプレートの実引数式に指定された演算は許されません。

C5530 (E) Try block requires at least one handler
try 文に対応する catch 文がありません。

C5531 (E) Handler requires an exception declaration
catch 文の(...)には例外宣言が必要です。

C5532 (E) Handler is masked by default handler
デフォルトハンドラによってハンドラがマスクされました。

C5533 (E) Handler is potentially masked by previous handler for type "型"
"型"を持つ前のハンドラによってハンドラがマスクされる可能性があります。

C5534 (I) Use of a local type to specify an exception
ローカルな型を使用した例外処理が指定されています。

C5535 (I) Redundant type in exception specification
例外処理中に冗長な型の指定があります。

C5536 (E) Exception specification is incompatible with that of previous entity-kind "名前" (declared at line "行番号"):
例外処理指定が前の指定"名前"と合致しません。

C5540 (E) Support for exception handling is disabled
例外処理を行うオプション(exception)が指定されていません。

C5541 (W) Omission of exception specification is incompatible with previous entity-kind "名前" (declared at line "行番号")
例外処理の省略形が前の"名前"と合致しません。

C5542 (F) Could not create instantiation request file "名前"
テンプレートを実体化するのに使用するファイル"名前"を作成することができませんでした。

C5543 (E) Non-arithmetic operation not allowed in nontype template argument
対応するテンプレートの実引数に非算術型変換は許されません。

C5544 (E) Use of a local type to declare a nonlocal variable
ローカルでない変数にローカルな型を指定しています。

C5545 (E) Use of a local type to declare a function
関数宣言にローカルな型を指定しています。

C5546 (E) Transfer of control bypasses initialization of:
初期化処理が行われません。

C5548 (E) Transfer of control into an exception handler
例外ハンドラ処理が実行されます。

C5549 (I) Entity-kind "名前" is used before its value is set
"名前"に値を設定する前に使用しています。

C5550 (W) Entity-kind "名前" was set but never used
"名前"が使用されませんでした。

C5551 (E) Entity-kind "名前" cannot be defined in the current scope
"名前"はこのスコープ内で定義できません。

C5552 (W) Exception specification is not allowed
例外処理指定は許されません。例外処理を無効にします。

C5553 (W) External/internal linkage conflict for entity-kind "名前" (declared at line "行番号")
"名前"の外部/内部リンケージ指定が衝突します。外部リンケージを設定します。

C5554 (W) Entity-kind "名前" will not be called for implicit or explicit conversions
変換関数"名前"は暗黙的にも明示的にも呼ばれることはありません。

C5555 (E) Tag kind of "名前" is incompatible with template parameter of type "型"
タグ"名前"の種類とテンプレートの引数の"型"が合致しません。

C5556 (E) Function template for operator new(size_t) is not allowed
operator new(size_t)の関数テンプレートは許されません。

C5558 (E) Pointer to member of type "型" is not allowed
メンバへのポインタ型"型"が誤っています。

C5559 (E) Ellipsis is not allowed in operator function parameter list
省略指定(...)は演算子関数の引数リストに指定できません。

C5563 (F) Invalid preprocessor output file
プリプロセッサ出力に使用できないファイル名です。

C5598 (E) A template parameter may not have void type
テンプレートの引数にvoid型は指定できません。

C5601 (E) A throw expression may not have void type
throw式にvoid型は指定できません。

C5603 (E) Parameter of abstract class type "型" is not allowed:
抽象クラス"型"の引数は許されません。

C5604 (E) Array of abstract class "型" is not allowed:
抽象クラス"型"の配列は許されません。

C5610 (W) Entity-kind "名前1" does not match "名前2" -- virtual function override intended?
"名前1"と"名前2"が一致しません。指定された通りに処理を継続します。

C5611 (W) Overloaded virtual function "名前1" is only partially overridden in entity-kind "名前2"
"名前1"のオーバーロード仮想関数は"名前2"の中で一部の仮想関数だけが置き換えの対象になります。指定された通りに処理を継続します。

C5612 (E) Specific definition of inline template function must precede its first use
インライン指定されたテンプレート関数は呼び出しの前に定義しなければなりません。

C5614 (F) Invalid error number: "指定番号"
change_messageオプション指定で無効な値を指定しています。値の範囲を確認してください。

C5624 (E) "名前" is not a type name
"名前"は型の名前ではありません。

C5641 (F) "名前" is not a valid directory
"名前"が正しいフォルダでありません。

C5642 (F) Cannot build temporary file name
コンパイラが使用するテンポラリファイルを作成できません。

C5656 (E) Transfer of control into a try block
外側のブロックから try ブロックに制御が移ります。

C5657 (W) Inline specification is incompatible with previous "名前" (declared at line "行番号")
インライン指定が前の宣言"名前"と合致しません。

C5658 (E) Closing brace of template definition not found
テンプレート定義の閉じ括弧がありません。

C5660 (E) Invalid packing alignment value
pack の値が不正です。

C5662 (W) Call of pure virtual function
純粋仮想関数が関数を呼び出しています。

C5663 (E) Invalid source file identifier string
#pragma 指定の構文に誤りがあります。

C5664 (E) A class template cannot be defined in a friend declaration
フレンド宣言内でクラステンプレートを定義することはできません。

C5673 (E) A reference of type "型 1" cannot be initialized with a value of type "型 2"
const/volatile 型"型 1"のリファレンスは"型 2"の値で初期化できません。

C5674 (E) Initial value of reference to const volatile must be an lvalue
const/volatile 型のリファレンスの初期値は左辺値でなければなりません。

C5678 (I) Call of entity-kind "名前" (declared at line "行番号") cannot be inlined
関数呼び出し"名前"がインライン展開されませんでした。

C5679 (I) Entity-kind "名前" cannot be inlined
関数"名前"はインライン展開されません。

C5693 (E) <typeinfo> must be included before typeid is used
typeid を使うためには<typeinfo>をインクルードしなければなりません。

C5694 (E) "名前" cannot cast away const or other type qualifiers
"名前"のキャストの結果 const などの属性がなくなります。

C5695 (E) The type in a dynamic_cast must be a pointer or reference to a complete class type, or void *
dynamic_cast の型は完全クラス型へのポインタ型またはリファレンス型か void *型でなければなりません。

C5696 (E) The operand of a pointer dynamic_cast must be a pointer to a complete class type
dynamic_cast ポインタのオペランドは完全クラス型へのポインタ型でなければなりません。

C5697 (E) The operand of a reference dynamic_cast must be an lvalue of a complete class type
dynamic_cast のリファレンスのオペランドは完全クラス型の左辺値でなければなりません。

C5698 (E) The operand of a runtime dynamic_cast must have a polymorphic class type
実行時 dynamic_cast のオペランドはポリモフィックなクラス型でなければなりません。

C5701 (E) An array type is not allowed here
配列型は許されません。

C5702 (E) Expected an "="
代入式が必要です。

C5703 (E) Expected a declarator in condition declaration
宣言子が必要です。

C5704 (E) "名前", declared in condition, may not be redeclared in this scope
このスコープ内で"名前"を再宣言することはできません。

C5705 (E) Default template arguments are not allowed for function templates
関数テンプレートにデフォルトの実引数を指定することはできません。

C5706 (E) Expected a "," or ">"
","または">"が必要です。

C5707 (E) Expected a template parameter list
テンプレートの引数リストが必要です。

C5708 (W) Incrementing a bool value is deprecated
bool 型の値をインクリメントしています。値をインクリメントして処理を継続します。

C5709 (E) bool type is not allowed
bool 型の値をデクリメントすることはできません。

C5710 (E) Offset of base class "名前 1" within class "名前 2" is too large
クラス"名前 2"内の基底クラス"名前 1"のサイズが大きすぎます。

C5711 (E) Expression must have bool type (or be convertible to bool)
式の型は bool 型か bool 型へ変換可能な型でなければなりません。

C5717 (E) The type in a const_cast must be a pointer, reference, or pointer to member to an object type
const_cast の型はポインタ型、リファレンス型またはメンバへのポインタ型でなければなりません。

C5718 (E) A const_cast can only adjust type qualifiers; it cannot change the underlying type
const_cast は const/volatile 以外の型を調整することはできません。

C5719 (E) mutable is not allowed
mutable の指定は許されません。

C5720 (W) Redeclaration of entity-kind "名前" is not allowed to alter its access
"名前"の再宣言でアクセス指定を変更することはできません。前の宣言のアクセス指定を有効にします。

C5722 (W) Use of alternative token "<:" appears to be unintended
2 文字表記"<:"が使用されました。"["と解釈します。

C5723 (W) Use of alternative token "%:" appears to be unintended
2 文字表記"%:"が使用されました。"#"と解釈します。

C5724 (E) namespace definition is not allowed
namespace の定義はファイルスコープまたは namespace スコープ内で許されます。

C5725 (E) Name must be a namespace name
namespace の名前が正しくありません。

C5726 (E) Namespace alias definition is not allowed
namespace の別名定義はここでは許されません。

C5727 (E) namespace-qualified name is required
namespace の限定名が要求されます。

C5728 (E) A namespace name is not allowed
namespace 名は許されません。

C5730 (E) Entity-kind "名前" is not a class template
"名前"はクラステンプレートのメンバではありません。

C5732 (E) Allocation operator may not be declared in a namespace
operator new 関数が namespace 内で宣言されています。

C5733 (E) Deallocation operator may not be declared in a namespace
operator delete 関数が namespace 内で宣言されています。

C5734 (E) Entity-kind "名前 1" conflicts with using-declaration of entity-kind "名前 2"
名前"名前 1"が using 宣言名"名前 2"と衝突します。

C5735 (E) Using-declaration of entity-kind "名前 1" conflicts with entity-kind "名前 2" (declared at line "行番号")
using 宣言の名前が衝突します。

C5737 (W) Using-declaration ignored -- it refers to the current namespace
現在の namespace スコープの名前を using 宣言しています。using 宣言を無視します。

C5738 (E) A class-qualified name is required
クラスの限定名が要求されています。

C5741 (W) Using-declaration of entity-kind "名前" ignored
using 宣言"名前"は無効です。

C5742 (E) Entity-kind "名前 1" has no actual member "名前 2"
"名前 1"に"名前 2"のメンバは存在しません。

C5750 (E) Entity-kind "名前" (declared at line "行番号") was used before its template was declared
"名前"はテンプレートが宣言される前に使われました。

C5751 (E) Static and nonstatic member functions with same parameter types cannot be overloaded
同じ引数の型を持つ静的メンバ関数と非静的メンバ関数はオーバーロードすることはできません。

C5752 (E) No prior declaration of entity-kind "名前"
namespace テンプレート関数"名前"の宣言がありません。

C5753 (E) A template-id is not allowed
ここではテンプレート(template 名<template 実引数>)は許されません。

C5754 (E) A class-qualified name is not allowed
ここではクラス限定名は許されません。

C5755 (E) Entity-kind "名前" may not be redeclared in the current scope
このスコープ内で"名前"を再宣言することはできません。

C5756 (E) Qualified name is not allowed in namespace member declaration
namespace メンバの宣言で指定された限定名は許されません。

C5757 (E) Entity-kind "名前" is not a type name
"名前"は型名ではありません。

C5761 (E) Typename may only be used within a template
typename キーワードはテンプレート内でのみ使用できます。

C5766 (W) Exception specification for virtual entity-kind "名前 1" is incompatible with that of overridden entity-kind "名前 2"
仮想関数の例外指定"名前 1"が"名前 2"に合致しません。

C5767 (W) Conversion from pointer to smaller integer
ポインタをポインタサイズより小さい型に変換しています。

C5768 (W) Exception specification for implicitly declared virtual entity-kind "名前 1" is incompatible with that of overridden entity-kind "名前 2"
コンパイラが生成する暗黙の仮想関数"名前 1"の例外指定が"名前 2"に合致しません。

C5771 (E) "explicit" is not allowed
explicit はクラス宣言内のコンストラクタにのみ指定できます。

C5772 (E) Declaration conflicts with "名前" (reserved class name)
予約されたクラス名 type_info と衝突します。

C5773 (E) Only "()" is allowed as initializer for array entity-kind "名前"
配列"名前"の初期化指定が正しくありません。

C5774 (E) "virtual" is not allowed in a function template declaration
関数テンプレートに virtual 指定はできません。

C5775 (E) Invalid anonymous union -- class member template is not allowed
無名 union の指定が正しくありません。

C5776 (E) Template nesting depth does not match the previous declaration of entity-kind "名前"
テンプレートのパラメータのネストが前の宣言"名前"と合致しません。

C5777 (E) This declaration cannot have multiple "template <...>" clauses
この宣言に複数のテンプレート宣言はできません。

C5779 (E) "名前", declared in for-loop initialization, may not be redeclared in this scope
for 文の初期化式で宣言された"名前"をこのスコープ内で再宣言できません。

C5782 (E) Definition of virtual entity-kind "名前" is required here
ここに仮想関数の定義"名前"が要求されます。

C5784 (E) A storage class is not allowed in a friend declaration
フレンド宣言に記憶クラスを指定することはできません。

C5785 (E) Template parameter list for "名前" is not allowed in this declaration
この宣言内に"名前"のテンプレートの引数並びは許されません。

C5786 (E) entity-kind "名前" is not a valid member class or function template
"名前"は有効なメンバまたは関数テンプレートではありません。

C5787 (E) Not a valid member class or function template declaration
有効なメンバまたは関数テンプレート宣言ではありません。

C5788 (E) A template declaration containing a template parameter list may not be followed by an explicit specialization declaration
テンプレート関数の定義の後にテンプレート引数並びを含むテンプレート宣言は指定できません。

C5789 (E) Explicit specialization of entity-kind "名前 1" must precede the first use of entity-kind "名前 2"
明示的なテンプレートの実体の定義"名前 1"は最初のテンプレート"名前 2"を使用する前になければなりません。

C5790 (E) Explicit specialization is not allowed in the current scope
明示的なテンプレートの実体の定義はこのスコープでは許されません。

C5791 (E) Partial specialization of entity-kind "名前" is not allowed
テンプレート"名前"の部分的な定義は許されません。

C5792 (E) Entity-kind "名前" is not an entity that can be explicitly specialized
"名前"はテンプレートのインスタンスではありません。

C5793 (E) Explicit specialization of entity-kind "名前" must precede its first use
明示的なテンプレートの実体"名前"の定義は最初の使用より前になければなりません。

C5794 (W) Template parameter "テンプレート引数" may not be used in an elaborated type specifier
class 指定にテンプレート引数を使用することはできません。class 指定を無効にしてテンプレートを有効にします。

C5795 (E) Specializing entity-kind "名前" requires "template<>" syntax
"名前"のテンプレートの実体定義は tempalte<>形式が要求されます。

C5800 (E) This declaration may not have extern "C" linkage
この宣言は extern "C"リンケージを持つことはできません。

C5801 (E) "名前" is not a class or function template name in the current scope
"名前"はこのスコープ内ではクラステンプレートまたは関数テンプレートではありません。

C5802 (W) Specifying a default argument when redeclaring an unreferenced function template is nonstandard
未参照の関数テンプレートを再宣言するときにデフォルト引数を指定しています。
デフォルト引数を無視します。

C5803 (E) Specifying a default argument when redeclaring an already referenced function template is not allowed
すでに参照された関数テンプレートを再宣言するときにデフォルト引数を指定しています。

C5804 (E) Cannot convert pointer to member of base class "型 1" to pointer to member of derived class "型 2" -- base class is virtual
仮想基底クラス"型 1"のメンバポインタを派生クラス"型 2"のメンバポインタに変換することはできません。

C5805 (E) Exception specification is incompatible with that of entity-kind "名前" (declared at line "行番号"):
throw 例外指定は"名前"の例外指定と合致しません。

C5806 (W) Omission of exception specification is incompatible with entity-kind "名前" (declared at line "行番号")
throw 例外指定の省略は"名前"の例外指定と合致しません。"名前"を有効にします。

C5807 (E) The parse of this expression has changed between the point at which it appeared in the program and the point at which the expression was evaluated -- "typename" may be required to resolve the ambiguity
デフォルト実引数の式の評価が途中で終了しました。

C5808 (E) Default-initialization of reference is not allowed
リファレンス型のデフォルトの初期化は許されません。

C5809 (E) Uninitialized entity-kind "名前" has a const member
未初期化の"名前"が const 型メンバを持ちます。

C5810 (E) Uninitialized base class "型" has a const member
未初期化の基底クラス"型"が const 型メンバを持ちます。

C5811 (E) Const entity-kind "名前" requires an initializer -- class "型" has no explicitly declared default constructor
const 型の"名前"には初期化指定が必要です。クラス"型"が明示的に宣言されたデフォルトコンストラクタを持ちません。

C5812 (W) Const object requires an initializer -- class "型" has no explicitly declared default constructor
const 型オブジェクトには初期化指定が必要です。クラス"型"が明示的に宣言されたデフォルトコンストラクタを持ちません。

C5815 (I) Type qualifier on return type is meaningless
テンプレートで実体化されるリターン型に意味のない修飾型を指定しています。
修飾型を有効にします。

C5817 (E) Static data member declaration is not allowed in this class
局所クラスは静的データメンバを持つことはできません。

C5818 (E) Template instantiation resulted in an invalid function declaration
テンプレートで実体化された関数宣言が正しくありません。

C5822 (E) Invalid destructor name for type "型"
"型"のデストラクタ名が正しくありません。

C5824 (E) Destructor reference is ambiguous -- both entity-kind "名前 1" and entity-kind "名前 2" could be used
"名前 1"と"名前 2"が使われました。デストラクタの参照があいまいです。

C5825 (W) Virtual inline entity-kind "名前" was never defined
仮想インラインメンバ関数"名前"の定義がありません。

C5826 (W) Entity-kind "名前" was never referenced
関数の引数"名前"は参照されません。

C5827 (E) Only one member of a union may be specified in a constructor initializer list
共用体の一つのメンバのみをコンストラクタの初期化で指定できます。

C5831 (I) Support for placement delete is disabled
operator delete 関数の型が正しくありません。処理を継続します。

C5832 (E) No appropriate operator delete is visible
適当な operator delete 関数が見つかりません。

C5833 (E) Pointer or reference to incomplete type is not allowed
不完全型へのポインタまたはリファレンス型は許されません。

C5834 (E) Invalid partial specialization -- entity-kind "名前" is already fully specialized
すでに特別化された"名前"を部分特別化しています。

C5835 (E) Incompatible exception specifications
例外指定の型が合致しません。

C5836 (W) Returning reference to local variable
局所変数のリファレンスをリターン値に指定しています。指定された処理を継続します。

C5837 (W) Omission of explicit type is nonstandard ("int" assumed)
型指定がありません。int 型を仮定します。

C5838 (E) More than one partial specialization matches the template argument list of entity-kind "名前"
部分特別化テンプレート"名前"のテンプレート実引数があいまいです。

C5840 (E) A template argument list is not allowed in a declaration of a primary template
プライマリテンプレート宣言にテンプレート実引数は指定できません。

C5841 (E) Partial specializations may not have default template arguments
部分特別化テンプレートはデフォルトのテンプレート引数を持つことはできません。

C5842 (E) Entity-kind "名前 1" is not used in template argument list of entity-kind "名前 2"
部分特別化テンプレート"名前 1"は"名前 2"のテンプレート実引数に使用されません。

C5843 (E) The type of partial specialization template parameter entity-kind "名前" depends on another template parameter
部分特別化テンプレート"名前"のテンプレート仮引数が別のテンプレート仮引数に依存しています。

C5844 (E) The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter
部分特別化テンプレートのテンプレート実引数がテンプレート仮引数に依存する非型の実引数を含んでいます。

C5845 (E) This partial specialization would have been used to instantiate entity-kind "名前"
この部分特別化テンプレートはプライマリテンプレート"名前"を実体化しようとしています。

C5846 (E) This partial specialization would have been made the instantiation of entity-kind "名前" ambiguous
この部分特別化テンプレートは"名前"の実体化があいまいになります。

C5847 (E) Expression must have integral or enum type
式の型は整数型か列挙型でなければなりません。

C5848 (E) Expression must have arithmetic or enum type
式の型は算術型か列挙型でなければなりません。

C5849 (E) Expression must have arithmetic, enum, or pointer type
式の型は算術型、列挙型もしくはポインタ型でなければなりません。

C5850 (E) Type of cast must be integral or enum
キャストの型は整数型か列挙型でなければなりません。

C5851 (E) Type of cast must be arithmetic, enum, or pointer
キャストの型は算術型、列挙型もしくはポインタ型でなければなりません。

C5852 (E) Expression must be a pointer to a complete object type
式の型は完全オブジェクト型へのポインタ型でなければなりません。

C5853 (E) A partial specialization of a member class template must be declared in the class of which it is a member
メンバクラスの部分特別化テンプレートはそのメンバを含むクラスの中で宣言しなければなりません。

C5854 (E) A partial specialization nontype argument must be the name of a nontype parameter or a constant
部分特別化テンプレートの非型テンプレート実引数は非型の仮引数名か定数でなければなりません。

C5855 (E) Return type is not identical to return type "型" of overridden virtual function entity-kind "名前"
関数のリターン型がオーバーライドされた仮想関数"名前"のリターン型"型"と同一でありません。

C5857 (E) A partial specialization of a class template must be declared in the namespace of which it is a member
部分特別化テンプレートはそのメンバを含む namespace の中で宣言しなければなりません。

C5858 (E) Entity-kind "名前" is a pure virtual function
"名前"は純粋仮想関数です。

C5859 (E) Pure virtual entity-kind "名前" has no overrider
純粋仮想関数"名前"はオーバーライドされません。

C5861 (E) Invalid character in input line
行中に不正な文字が現れました。

C5862 (E) Function returns incomplete type "型"
関数のリターン型"型"が不完全型です。

C5864 (E) "名前" is not a template
"名前"はテンプレートでありません。

C5865 (E) A friend declaration may not declare a partial specialization
部分特別化テンプレートはフレンド宣言内で指定できません。

C5867 (W) Declaration of "size_t" does not match the expected type "型"
size_t 型が期待する"型"と異なります。

C5868 (E) Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator)
2 つのテンプレート実引数リストの最後に指定する">>"は間に空白が必要です。

C5870 (W) Invalid multibyte character sequence
不正な 2 バイト文字があります。

C5871 (E) Template instantiation resulted in unexpected function type of "型 1" (the meaning of a name may have changed since the template declaration -- the type of the template is "型 2")
"型 2"を持つテンプレートの実体化の結果、期待されない型"型 1"の関数が作られました。

C5873 (E) Non-integral operation not allowed in nontype template argument
非型のテンプレート実引数に非整数型の演算は許されません。

C5875 (E) Embedded C++ does not support templates
Embedded C++仕様はテンプレート機能をサポートしません。

C5876 (E) Embedded C++ does not support exception handling
Embedded C++仕様は例外処理機能をサポートしません。

C5877 (E) Embedded C++ does not support namespaces
Embedded C++仕様は namespace 機能をサポートしません。

C5878 (E) Embedded C++ does not support run-time type information
Embedded C++仕様はランタイム型情報機能をサポートしません。

C5879 (E) Embedded C++ does not support the new cast syntax
Embedded C++仕様は new のキャスト機能をサポートしません。

C5880 (E) Embedded C++ does not support using-declarations
Embedded C++仕様は using 宣言機能をサポートしません。

C5881 (E) Embedded C++ does not support "mutable"
Embedded C++仕様は mutable 機能をサポートしません。

C5882 (E) Embedded C++ does not support multiple or virtual inheritance
Embedded C++仕様は多重継承/仮想継承機能をサポートしません。

C5885 (E) "型 1" cannot be used to designate constructor for "型 2"
"型 1"はコンストラクタの"型 2"で使用することはできません。

C5891 (E) An explicit template argument list is not allowed on this declaration
この宣言内では明示的なテンプレート実引数は許されません。

C5894 (E) Entity-kind "名前" is not a template
"名前"はテンプレートではありません。

C5896 (E) Expected a template argument
テンプレートの実引数が期待されます。

C5898 (E) Nonmember operator requires a parameter with class or enum type
非メンバ演算子関数にはクラスまたは列挙型の仮引数が要求されます。

C5900 (E) Using-declaration of entity-kind "名前" is not allowed
"名前"の using 宣言は許されません。

C5901 (E) Qualifier of destructor name "型 1" does not match type "型 2"
"型 1"のデストラクタの限定名が"型 2"に一致しません。

C5902 (W) Type qualifier ignored
型限定名が不正です。型限定名を無効にします。

C5916 (E) Cannot convert pointer to member of derived class "型 1" to pointer
to member of base class "型 2" -- base class is virtual
派生クラス"型 1"のメンバへのポインタ型を仮想基底クラス"型 2"のメンバへのポインタ型に変換できません。

C5919 (F) Invalid output file: "名前"
テンプレート情報ファイルの"名前"が不正です。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5920 (F) Cannot open output file: "名前"
テンプレート情報ファイル"名前"をオープンすることができません。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5926 (F) Cannot open definition list file: "名前"
ファイル"名前"をオープンすることができません。
コンパイラの環境設定やホスト環境のファイルシステム異常がないか確認ください。

C5928 (E) Incorrect use of va_start
va_start の使用方法に誤りがあります。

C5929 (E) Incorrect use of va_arg
va_arg の使用方法に誤りがあります。

C5930 (E) Incorrect use of va_end
va_end の使用方法に誤りがあります。

C5935 (E) "typedef" may not be specified here
typedef を指定することはできません。

C5936 (W) Redeclaration of entity-kind "名前" alters its access
"名前"の再宣言でアクセス指定を変更しています。
再定義されたアクセス指定を有効にします。

C5937 (E) A class or namespace qualified name is required
クラスまたは namespace の限定名が要求されます。

C5940 (W) Missing return statement at end of non-void entity-kind "名前"
void 型以外をリターンする関数"名前"が return 文を持ちません。
return 値は不定になります。

C5941 (W) Duplicate using-declaration of "名前" ignored
using 宣言"名前"を重複指定しています。重複した using 宣言を無効にします。

C5946 (E) Name following "template" must be a member template
"template"に続く名前はメンバテンプレートでなければなりません。

C5947 (E) Name following "template" must have a template argument list
"template"に続く名前はテンプレート実引数でなければなりません。

C5952 (E) A template parameter may not have class type
テンプレート仮引数にクラス型名は指定できません。

C5953 (E) A default template argument cannot be specified on the declaration of a member of a class template
クラステンプレートのメンバ宣言にデフォルトのテンプレート実引数を指定できません。

C5954 (E) A return statement is not allowed in a handler of a function try block of a constructor
コンストラクタの try ブロックのハンドラ内にリターン文は許されません。

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "サイズ" bits
指定されたビット数がビットフィールドの型の"サイズ"を超えています。
ビット数をビットフィールドの型のサイズに合わせて処理を継続します。

C5960 (E) Type used as constructor name does not match type "型"
コンストラクタ名として使用された型が"型"と一致しません。

C5961 (W) Use of a type with no linkage to declare a variable with linkage
リンケージを持たない型を使用してリンケージを持つ変数として宣言しています。
リンケージを持つものとします。

C5962 (W) Use of a type with no linkage to declare a function
リンケージを持たない型を使用してリンケージを持つ関数として宣言しています。
リンケージを持つものとします。

C5963 (E) Return type may not be specified on a constructor
コンストラクタにリターン型を指定できません。

C5964 (E) Return type may not be specified on a destructor
デストラクタにリターン型を指定できません。

C5965 (E) Incorrectly formed universal character name
universal character の形式が正しくありません。

C5966 (E) Universal character name specifies an invalid character
universal character で指定された文字が不正です。

C5967 (E) A universal character name cannot designate a character in the basic character set
基本文字集合内で universal character を文字として指定することはできません。

C5968 (E) This universal character is not allowed in an identifier
識別子にこの universal character は許されません。

C5978 (E) A template friend declaration cannot be declared in a local class
テンプレートのフレンド関数は局所クラスで宣言できません。

C5979 (E) Ambiguous "?" operation: second operand of type "型 1" can be converted to third operand type "型 2", and vice versa
三項演算子"?:"の第 2 式の"型 1"と第 3 式の"型 2"が互いに変換可能な型であいまいです。

C5980 (E) Call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type
オブジェクトを呼び出していますが operator() 関数または関数へのポインタ型変換関数が定義されていません。

C5982 (E) There is more than one way an object of type "型" can be called for the argument list
実引数リストから呼ぶことができる"型"のオブジェクトが 2 つ以上あります。

C5986 (E) Expected a section name string
_ _sectop/_ _secend/_ _secsize にセクション名がありません。

C5988 (E) Invalid pragma declaration
#pragma の構文が不正です。

C5989 (E) "名前" has already been specified by other pragma
このシンボルは既に他の#pragma 指定がされています。

C5990 (E) Pragma may not be specified after definition
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。

C5991 (E) Invalid kind of pragma is specified to this symbol
不正な#pragma を指定しました。

C5994 (W) Operator new and operator delete cannot be given internal linkage
operator new/operator delete が static で定義されています。

C5995 (E) Storage class "mutable" is not allowed for anonymous unions
mutable を無名共用体に指定することはできません。

C5997 (E) Abstract class type "型" is not allowed as catch type:
抽象クラスを catch で受けることはできません。

C5998 (E) A qualified function type cannot be used to declare a nonmember function or a static member function
修飾付き関数型を非メンバ関数や static メンバ関数の宣言に使用することはできません。

C5999 (E) A qualified function type cannot be used to declare a parameter
修飾付き関数型を関数パラメータ指定に使用することはできません。

C6000 (E) Cannot create a pointer or reference to qualified function type
修飾付き関数型へのポインタ型や参照型を作ることはできません。

C6001 (W) Extra braces are nonstandard
集合型の初期化子リストに余分な'{'があります。

C6002 (E) An empty template parameter list is not allowed in a template template parameter declaration
空テンプレートパラメータを持つテンプレートをテンプレートパラメータに指定することはできません。

C6005 (E) Expected "class"
テンプレートパラメータに指定するクラステンプレートはクラスを必要とします。

C6006 (E) The "class" keyword must be used when declaring a template template parameter
テンプレートパラメータに指定するクラステンプレートは構造体ではいけません。

C6007 (W) "関数名 1" is hidden by "関数名 2" -- virtual function override intended?
"関数名 1"が仮想関数"関数名 2"を隠しています。

C6008 (E) A qualified name is not allowed for a friend declaration that is a function definition
friend 指定付き関数定義において、名前空間の名前付き関数名を指定することはできません。

C6009 (E) "型 1" is not compatible with "型 2"
指定したクラステンプレートはテンプレートパラメータと形式が一致しません。

C6010 (W) A storage class may not be specified here
ここには記憶域クラス指定子を指定することはできません。

C6011 (E) Class member designated by a using-declaration must be visible in a direct base class
クラスメンバの using 指定は参照可能な直接基底クラスでなければなりません。

C6016 (E) A template template parameter cannot have the same name as one of its template parameters
テンプレートパラメータに指定するクラステンプレート名が、それ自身のテンプレートパラメータ名と同じです。

C6017 (E) Recursive instantiation of default argument
テンプレート関数のデフォルト引数のインスタンスが再帰的に生成されます。

C6018 (E) A parameter of a template template parameter cannot depend on the type of another template parameter
テンプレートパラメータに指定するテンプレートが持つ非型パラメータの型は、他の型パラメータに依存してはいけません。

C6019 (E) "インスタンス名" is not an entity that can be defined
実体のないインスタンスを生成しようとしています。

C6023 (E) A qualified friend template declaration must refer to a specificpreviously declared template
限定フレンドテンプレートは参照前に定義しておく必要があります。

C6028 (E) "クラス名" has no member class "メンバ名"
クラスにないメンバを使っています。

C6029 (E) The global scope has no class named "クラス名"
クラス内の名前にファイルスコープ演算子を使っています。

C6030 (E) Recursive instantiation of template default argument
テンプレートのデフォルト引数で再帰的にインスタンスを生成します。

C6031 (E) Access declarations and using-declarations cannot appear in unions
union で using 指定は使えません。

C6032 (E) "名前" is not a class member
クラスのメンバではありません。

C6038 (W) Invalid redeclaration of nested class
クラス内でクラスを 2 重定義しています。

C6045 (E) "テンプレート名" cannot be declared in this scope
このスコープではテンプレートを宣言することができません。

12.3　C 標準ライブラリ関数のエラーメッセージ

ライブラリ関数の中には、ライブラリ関数を実行中にエラーが発生した場合、標準ライブラリのヘッダファイル<errno.h>で定義しているマクロ errno にエラー番号を設定するものがあります。
エラー番号には対応するエラーメッセージが定義してあり、エラーメッセージを出力することができます。エラーメッセージを出力するプログラム例を以下に示します。

例

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

main()
{
    FILE *fp;

    fp=fopen("file", "w");
    fp=NULL;

    fclose(fp);                    /* error occurred     */

    printf("%s¥n", strerror(errno));  /* print error message */
}
```

説明

(1) fclose 関数に値 NULL のファイルポインタを実引数として渡しているので、エラーとなります。このとき errno に対応するエラー番号が設定されます。

(2) strerror 関数は、エラー番号を実引数として渡すと、対応するエラーメッセージの文字列のポインタを返します。printf 関数の文字列出力指定によりエラーメッセージを出力します。

標準ライブラリエラーメッセージ一覧

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1100 (ERANGE)	Data out of range オーバフローが発生しました。	frexp, ldexp, modf, ceil, floor, fmod, atof, atoi, atol, atoll, atolfixed, atolaccum, strtod, strtol, strtoul, strtoll, stroull, strtolfixed, strtolaccum, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceilf, cos, cosf, cosh, coshf, exp, expf, floorf, fmodf, ldexpf, log, log10, log10f, logf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexpf
1101 (EDOM)	Data out of domain 数学関数の引数に対する結果の値が定義されません。	acos, acosf, asin, asinf, atan, atan2, atan2f, atanf, ceil, ceilf, cos, cosf, cosh, coshf, exp, expf, floor, floorf, fmod, fmodf, ldexp, ldexpf, log, log10, log10f, logf, modf, modff, pow, powf, sin, sinf, sinh, sinhf, sqrt, sqrtf, tan, tanf, tanh, tanhf, fabs, fabsf, frexp, frexpf
1104 (ESTRN)	Too long string 文字列の文字数が 512 文字を超えています。	atof, atoi, atol, atoll, atolfixed, atolaccum, strtod, strtol, strtoul, strtoll, stroull, strtolfixed, strtolaccum
1106 (PTRERR)	Invalid file pointer ファイルポインタの値に NULL ポインタ定数を指定しています。	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 (ECBASE)	Invalid radix 基数の指定が誤っています。	strtol, strtoul, strtoll, strtoull
1202 (ETLN)	Number too long 数値を表現する文字列の文字数が有効桁数を超えています。	atof, atolfixed, atolaccum, strtod, strtolfixed, strtolaccum, fscanf, scanf, sscanf
1204 (EEXP)	Exponent too large 指数部の桁数が 3 桁を超えています。	atof, strtod, fscanf, scanf, sscanf
1206 (EEXPN)	Normalized exponent too large 文字列を一度 IEEE 規格の 10 進形式に正規化したとき指数部の桁数が 3 桁を超えています。	atof, strtod, fscanf, scanf, sscanf
1210 (EFLOATO)	Overflow out of float float 型の 10 進数値が,float 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1220 (EFLOATU)	Underflow out of float float 型の 10 進数値が,float 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf
1250 (EDBLO)	Overflow out of double double 型の 10 進数値が，double 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1260 (EDBLU)	Underflow out of double double 型の 10 進数値が，double 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1270 (ELDBLO)	Overflow out of long double long double 型の 10 進数値が，long double 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1280 (ELDBLU)	Underflow out of long double long double 型の 10 進数値が，long double 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf
1300 (NOTOPN)	File not open ファイルがオープンされていません。	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen
1302 (EBADF)	Bad file number 入力専用ファイルに対して出力関数,あるいは出力専用ファイルに対して入力関数を発行しています。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 (ECSPEC)	Error in format 書式付き入出力関数で指定している書式が誤っています。	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror
1400 (EFIXEDO)	Overflow out of _ _fixed _ _fixed 型の 10 進数値が，_ _fixed 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1410 (EFIXEDU)	Underflow out of _ _fixed _ _fixed 型の 10 進数値が，_ _fixed 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf
1420 (EACCUMO)	Overflow out of _ _accum _ _accum 型の 10 進数値が, _ _accum 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1430 (EACCUMU)	Underflow out of _ _accum _ _accum 型の 10 進数値が, _ _accum 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf
1440 (ELFIXEDO)	Overflow out of long _ _fixed long _ _fixed 型の 10 進数値が, long _ _fixed 型の範囲を超えています(オーバフロー)。	fscanf, scanf, sscanf
1450 (ELFIXEDU)	Underflow out of long _ _fixed long _ _fixed 型の 10 進数値が, long _ _fixed 型の範囲を超えています(アンダフロー)。	fscanf, scanf, sscanf
1460 (ELACCUMO)	Overflow out of long _ _accum long _ _accum 型の 10 進数値が, long _ _accum 型の範囲を超えています (オーバフロー)。	fscanf, scanf, sscanf
1470 (ELACCUMU)	Underflow out of long _ _accum long _ _accum 型の 10 進数値が, long _ _accum 型の範囲を超えています (アンダフロー)。	fscanf, scanf, sscanf

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
2100 (EMALRESM)	Error in waiting semaphore malloc 用セマフォ資源確保に失敗しました。	calloc, free, malloc, realloc, calloc_ _X, free_ _X,malloc_ _X,realloc_ _X, calloc_ _Y, free_ _Y,malloc_ _Y,realloc_ _Y
2101 (EMALFRSM)	Error in signaling semaphore malloc 用セマフォ資源解放に失敗しました。	calloc, free, malloc, realloc calloc_ _X, free_ _X,malloc_ _X,realloc_ _X, calloc_ _Y, free_ _Y,malloc_ _Y,realloc_ _Y
2110 (ETOKRESM)	Error in waiting semaphore strtok 用セマフォ資源確保に失敗しました。	strtok
2111 (ETOKFRSM)	Error in signaling semaphore strtok 用セマフォ資源解放に失敗しました。	strtok
2120 (EIOBRESM)	Error in waiting semaphore _iob 用セマフォ資源確保に失敗しました。	fopen
2121 (EIOBFRSM)	Error in signaling semaphore _iob 用セマフォ資源解放に失敗しました。	fopen

13.　アセンブラのエラーメッセージ

13.1　エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　(エラーレベル)　エラーメッセージ
　　エラー内容

エラーレベルは、エラーの重要度に従い、3 種類に分類されます。

エラーレベル		動作
(W)	ウォーニング	処理を継続します。
(E)	エラー	処理を継続します。
(F)	フェータル	処理を中断します。

13.2　メッセージ一覧

10 (E) NO INPUT FILE SPECIFIED
　入力ソースファイルの指定がありません。
　入力ソースファイルを指定してください。

20 (E) CANNOT OPEN FILE ファイル名
　指定のファイルをオープンできません。
　ファイル名、フォルダ名などを見直してください。

30 (E) INVALID COMMAND PARAMETER
　オプションに誤りがあります。
　オプションを見直してください。

40 (E) CANNOT ALLOCATE MEMORY
　処理中にメモリが足りなくなりました。
　ユーザが使用できるメモリ量が極端に少ない場合に発生します。
　他に実行中の処理があればその処理を終了してからアセンブラを再起動してください。
　それでも本エラーが発生する場合はホストシステムのメモリ管理の方法を見直してください。

50 (E) INVALID FILE NAME ファイル名
　フォルダを含めたファイル名が長すぎるか、ファイル名に誤りがあります。
　ファイル名を見直してください。
　このときアセンブラが出力するオブジェクトモジュールはデバッガで扱えない可能性があります。

101 (E) SYNTAX ERROR IN SOURCE STATEMENT
ソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

102 (E) SYNTAX ERROR IN DIRECTIVE
アセンブラ制御命令のソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

104 (E) LOCATION COUNTER OVERFLOW
ロケーションカウンタ値が最大値を超えています。
プログラムを縮小してください。

105 (E) ILLEGAL INSTRUCTION IN STACK SECTION
スタックセクション内に実行命令、DSP 命令、拡張命令、データを確保するアセンブラ制御命令を記述しています。
実行命令、DSP 命令、拡張命令、データを確保するアセンブラ制御命令を削除してください。

106 (E) TOO MANY ERRORS
エラーの数が多いので表示を打ち切りました。
ソースステートメント全体を見直してください。

108 (E) ILLEGAL CONTINUATION LINE
複数行にわたって記述したソースステートメントに誤りがあります。
記述方法を見直してください。

150 (E) INVALID DELAY SLOT INSTRUCTION
ディレイスロット命令 (遅延分岐命令の直後にくる実行命令) が不当です。
実行命令を記述する順序を変更するなどして、ディレイスロット命令が不当とならないようにしてください。

151 (E) ILLEGAL EXTENDED INSTRUCTION POSITION
ディレイスロット命令に拡張命令を記述しています。
ディレイスロット命令には実行命令を記述してください。

152 (E) ILLEGAL BOUNDARY ALIGNMENT VALUE
拡張命令を記述しているセクションのアライメント数が不当です。
拡張命令を記述するセクションのアライメント数は 2 以上を指定してください。

160 (E) REPEAT LOOP NESTING
REPEAT 拡張命令～終了アドレス間に別の REPEAT 拡張命令が存在します。
REPEAT 拡張命令の位置を訂正してください。

161 (E) ILLEGAL START ADDRESS FOR REPEAT LOOP
REPEAT 拡張命令～開始アドレス間に実行命令または DSP 命令が存在しません。
REPEAT 拡張命令と開始アドレスの間に 1 つ以上の実行命令または DSP 命令を記述してください。

162 (E) ILLEGAL DATA BEFORE REPEAT LOOP
REPEAT 拡張命令で指定したループの直前に不当なデータが存在します。
ループ直前がアセンブラ制御命令の場合、アセンブラ制御命令を訂正してください。
ループ直前がリテラルプールの場合、.NOPOOL でリテラルプールの出力を抑止してください。
リピートする命令が 3 命令以下の場合、ループの直前は実行命令または DSP 命令でなければなりません。

163 (E) ILLEGAL INSTRUCTION IN REPEAT LOOP
リピートループ内に不当な命令を配置しています。
REPEAT 拡張命令～終了アドレス間に分岐命令、TRAPA 命令 (cpu=sh4aldsp 未指定時)、SR、RS、RE に対するロード命令は配置できません。

164 (E) ILLEGAL INSTRUCTION IN REPEAT LOOP
リピート最終命令に不当な命令を配置しています。
リピート最終命令には遅延分岐命令、SR、RS、RE に対するロード命令は配置できません。

200 (E) UNDEFINED SYMBOL REFERENCE
参照しているシンボルが定義されていません。
シンボルを定義してください。

201 (E) ILLEGAL SYMBOL OR SECTION NAME
シンボル (セクション名を含む) としてキーワードを指定しています。
シンボル (セクション名を含む) を訂正してください。

202 (E) ILLEGAL SYMBOL OR SECTION NAME
シンボル (セクション名を含む) に誤りがあります。
シンボル (セクション名を含む) を訂正してください。

203 (E) ILLEGAL LOCAL LABEL
ローカルラベルの指定に誤りがあります。
ローカルラベルの指定を訂正してください。

300 (E) ILLEGAL MNEMONIC
オペレーションに誤りがあります。
オペレーションを訂正してください。

301 (E) TOO MANY OPERANDS OR ILLEGAL COMMENT
実行命令のオペランドが多すぎるか、コメントに誤りがあります。
オペランドまたはコメントを訂正してください。

304 (E) LACKING OPERANDS
オペランドが足りません。
オペランドを訂正してください。

307 (E) ILLEGAL ADDRESSING MODE
オペランドに許されないアドレス形式を指定しています。
オペランドを訂正してください。

308 (E) SYNTAX ERROR IN OPERAND
オペランドに文法上の誤りがあります。
オペランドを訂正してください。

309 (E) FLOATING POINT REGISTER MISMATCH
単精度のオペレーションに対して倍精度の浮動小数点レジスタを指定したか、倍精度のオペレーションに対して単精度浮動小数点レジスタを指定しています。
オペレーションサイズまたは浮動小数点レジスタを訂正してください。

350 (E) SYNTAX ERROR IN SOURCE STATEMENT (ニーモニック)
DSP 命令のソースステートメントに構文上の誤りがあります。
ソースステートメントを見直してください。

351 (E) ILLEGAL COMBINATION OF MNEMONICS (ニーモニック, ニーモニック)
許されていない DSP 演算命令の組み合わせを指定しています。
命令の組み合わせを訂正してください。

352 (E) ILLEGAL CONDITION (ニーモニック)
DSP 演算命令に対するコンディションを不正に指定しています。
コンディションの指定を削除するか、DSP 演算命令を変更してください。

353 (E) ILLEGAL POSITION OF INSTRUCTION (ニーモニック)
DSP 演算命令の指定位置が誤っています。
DSP 演算命令を正しい位置に指定してください。

354 (E) ILLEGAL ADDRESSING MODE (ニーモニック)
DSP 演算命令のアドレス形式が誤っています。
オペランドを訂正してください。

355 (E) ILLEGAL REGISTER NAME (ニーモニック)
DSP 演算命令のレジスタ指定に誤りがあります。
レジスタ名を訂正してください。

357 (E) ILLEGAL COMBINATION OF MNEMONICS (ニーモニック)
データ転送命令の指定が不正です。
データ転送命令を訂正してください。

371 (E) ILLEGAL COMBINATION OF MNEMONICS (ニーモニック, ニーモニック)
データ転送命令の組み合わせが誤っています。
データ転送命令の組み合わせを訂正してください。

372 (E) ILLEGAL ADDRESSING MODE (ニーモニック)
データ転送命令のオペランドとして許されないアドレス形式を指定しています。
オペランドを訂正してください。

373 (E) ILLEGAL REGISTER NAME (ニーモニック)
データ転送命令のレジスタの指定が誤っています。
レジスタ名を訂正してください。

400 (E) CHARACTER CONSTANT TOO LONG
文字定数が 4 文字を超えています。
文字定数を訂正してください。

402 (E) ILLEGAL VALUE IN OPERAND
オペランドとして範囲外の値です。
値を変更してください。

403 (E) ILLEGAL OPERATION FOR RELATIVE VALUE
相対アドレスに対して乗除算または論理演算を指定しています。
演算内容を訂正してください。

404 (E) ILLEGAL IMMEDIATE DATA
イミディエイト値に不当な値(前方参照シンボル、外部参照シンボル、相対アドレスシンボル)を指定しています。
イミディエイト値を訂正してください。

407 (E) MEMORY OVERFLOW
式の計算中、計算用のメモリが足りなくなりました。
演算内容を簡単にしてください。

408 (E) DIVISION BY ZERO
0 除算を指定しています。
演算内容を変更してください。

409 (E) REGISTER IN EXPRESSION
式の中にレジスタ名が現れました。
演算内容を訂正してください。

411 (E) INVALID STARTOF/SIZEOF OPERAND
STARTOF 演算または SIZEOF 演算で誤ったセクション名を指定しています。
セクション名を訂正してください。

412 (E) ILLEGAL SYMBOL IN EXPRESSION
シフト数に相対アドレスを指定しています。
演算内容を訂正してください。

450 (E) ILLEGAL DISPLACEMENT VALUE
ディスプレースメント値が不当です(負の値を指定しています)。
ディスプレースメント値を訂正してください。

452 (E) ILLEGAL DATA AREA ADDRESS
PC 相対データ転送命令のデータ領域のアドレスが不当です。
各命令のオペレーションサイズに合った境界のアドレスにアクセスしてください
(MOV.L, MOVA 命令は 4 バイトアライメント、MOV.W は 2 バイトアライメントです)。

453 (E) LITERAL POOL OVERFLOW
リテラル未出力の拡張命令が 510 個を超えています。
.POOL を用いるなどしてリテラルプールを出力してください。

460 (E) ILLEGAL SYMBOL
REPEAT 拡張命令のオペランドに前方参照でないラベル、未定義シンボル、ラベル以外のシンボルを指定したか、開始アドレスが終了アドレスより大きな値になっています。
オペランドを訂正してください。

461 (E) SYNTAX ERROR IN OPERAND
不当なオペランドを指定しています。
オペランドを訂正してください。

462 (E) ILLEGAL VALUE IN OPERAND
REPEAT 拡張命令とラベルの距離が範囲外です。
REPEAT 拡張命令またはラベルの位置を訂正してください。

463 (E) NO INSTRUCTION IN REPEAT LOOP
リピートループ内に命令が存在しないか、終了アドレスに指定した位置に命令が存在しません。
開始アドレス～終了アドレス間に命令を記述するか、命令の存在するアドレスを終了アドレスに指定してください。

500 (E) SYMBOL NOT FOUND
ラベルが必要なアセンブラ制御命令のソースステートメントにラベルがありません。
ラベルを記述してください。

501 (E) ILLEGAL ADDRESS VALUE IN OPERAND
セクションの先頭アドレスの指定が誤っているか、ロケーションカウンタ値の指定が誤っています。
先頭アドレスまたはロケーションカウンタ値を訂正してください。

502 (E) ILLEGAL SYMBOL IN OPERAND
オペランドに不当な値(前方参照シンボル、外部参照シンボル、相対アドレスシンボル、未定義シンボル)を指定しています。
オペランドを訂正してください。

503 (E) UNDEFINED EXPORT SYMBOL
ファイル内で定義していないシンボルを外部定義シンボルとして宣言しています。
シンボルを定義するか、外部定義シンボルとしての宣言を取りやめてください。

504 (E) INVALID RELATIVE SYMBOL IN OPERAND
オペランドに不当な値(前方参照シンボル、外部参照シンボル)を指定しています。
オペランドを訂正してください。

505 (E) ILLEGAL OPERAND
オペランドの名称に誤りがあります。
オペランドを訂正してください。

506 (E) ILLEGAL OPERAND
オペランドとして許されない要素を指定しています。
オペランドを訂正してください。

508 (E) ILLEGAL VALUE IN OPERAND
オペランドに範囲外の値を指定しています。
オペランドを訂正してください。

510 (E) ILLEGAL BOUNDARY VALUE
アライメント数の指定に誤りがあります。
アライメント数を訂正してください。

512 (E) ILLEGAL EXECUTION START ADDRESS
実行開始アドレスに誤りがあります。
実行開始アドレスを訂正してください。

513 (E) ILLEGAL REGISTER NAME
レジスタ名に誤りがあります。
レジスタ名を訂正してください。

514 (E) INVALID EXPORT SYMBOL
外部定義できないシンボルを外部定義シンボルとして宣言しています。
外部定義シンボルとしての宣言を取りやめてください。

516 (E) EXCLUSIVE DIRECTIVES
制御命令の指定内容が矛盾しています。
関連する制御命令を含めて見直してください。

517 (E) INVALID VALUE IN OPERAND
オペランドに不当な値(前方参照シンボル、外部参照シンボル、他セクションの相対アドレスシンボル)を指定しています。
オペランドを訂正してください。

518 (E) INVALID IMPORT SYMBOL
ファイル内で定義しているシンボルを外部参照シンボルとして宣言しています。
外部参照シンボルとしての宣言を取りやめてください。

519 (E) ILLEGAL SYMBOL IN OPERAND
マイコン種別が SH2A、SH2A-FPU の時、定数値を指定するオペランドにアドレスを値に持つシンボル、またはロケーションカウンタ値を指定しています。
マイコン種別が SH2A、SH2A-FPU の時、オペランドにアドレスを値に持つシンボル、ロケーションカウンタ値は指定しないでください。

520 (E) ILLEGAL .CPU DIRECTIVE POSITION
.CPU がプログラムの先頭にないか、複数回指定しています。
.CPU はプログラムの先頭に 1 回だけ指定してください。

521 (E) ILLEGAL .NOPOOL DIRECTIVE POSITION
.NOPOOL の記述位置が不当です。
.NOPOOL はディレイスロット命令に続けて記述してください。

522 (E) ILLEGAL .POOL DIRECTIVE POSITION
.POOL を遅延分岐命令に続けて記述しています。
遅延分岐命令の後にはディレイスロット命令を記述してください。

523 (E) ILLEGAL OPERAND
.LINE のオペランドに誤りがあります。
.LINE のオペランドを訂正してください。

525 (E) ILLEGAL .LINE DIRECTIVE POSITION
.LINE をマクロ展開または条件付き繰り返し展開内に指定しています。
.LINE の指定位置を変えてください。

526 (E) STRING TOO LONG
オペランドの文字列が 255 文字を超えています。
.SDATA、.SDATAB、.SDATAC、.SDATAZ のオペランドに指定する文字列は 255 文字以内としてください。

527 (E) CANNOT SUPPORT COMMON SECTION SINCE VERSION 5
セクション属性に COMMON を指定しています。
コモンセクションは使用できなくなりました。
最適化リンケージエディタの start オプションでコロン(:)を用いて複数セクションを同一アドレスに配置できます。

528 (E) SPECIFICATION OF THE ADDRESS OVERLAPS
セクション内のアドレス割付けが重複しています。
.SECTION、.ORG の指定内容を見直してください。

529 (E) THE ADDRESS BETWEEN SECTIONS OVERLAPS
セクション間のアドレス割付けが重複しています。
.SECTION、.ORG の指定内容を見直してください。

530 (E) ILLEGAL OPERAND
.FILE のオペランドに誤りがあります。
.FILE のオペランドを訂正してください。

531 (E) ILLEGAL .FILE DIRECTIVE POSITION
.FILE をマクロ展開または条件付き繰り返し展開内に指定しています。
.FILE の指定位置を変えてください。

532 (E) ILLEGAL OPERAND
.STACK のオペランドに誤りがあります。
スタック値を 4 の倍数に訂正してください。

533 (E) ILLEGAL .STACK DIRECTIVE POSITION
.STACK をマクロ展開または条件付き繰り返し展開内に指定しています。
.STACK の指定位置を変えてください。

600 (E) INVALID CHARACTER
ソースプログラムに不当な文字があります。
不当な文字を訂正してください。

601 (E) INVALID DELIMITER
区切り文字が不当です。
区切り文字を訂正してください。

602 (E) INVALID CHARACTER STRING FORMAT
文字列に誤りがあります。
文字列を訂正してください。

603 (E) SYNTAX ERROR IN SOURCE STATEMENT
ソースステートメントに構文上の誤りがあります。
ソースステートメント全体を見直してください。

604 (E) ILLEGAL SYMBOL IN OPERAND
制御命令のオペランドが不当です。
本制御命令のオペランドにシンボルおよびロケーションカウンタ ($) は記述できません。

610 (E) MULTIPLE MACRO NAMES
.MACRO で定義しようとしているマクロ命令は既に定義されています。
マクロ名を訂正してください。

611 (E) MACRO NAME NOT FOUND
.MACRO のオペランドにマクロ名がありません。
マクロ名を記述してください。

612 (E) ILLEGAL MACRO NAME
.MACRO のマクロ名に誤りがあります。
マクロ名を訂正してください。

613 (E) ILLEGAL .MACRO DIRECTVE POSITION
マクロ本体(.MACRO～.ENDM 間)、.AREPEAT～.AENDR 間、.AWHILE～.AENDW 間に.MACRO があります。
.MACRO を削除してください。

614 (E) MULTIPLE MACRO PARAMETERS
マクロ定義(.MACRO)の仮引数の宣言で仮引数名が重複しています。
仮引数名を訂正してください。

615 (E) ILLEGAL .END DIRECTIVE POSITION
マクロ本体(.MACRO～.ENDM 間)に.END があります。
.END を削除してください。

616 (E) MACRO DIRECTIVES MISMATCH
.ENDM が.MACRO に対応していないか、.EXITM がマクロ本体(.MACRO～.ENDM 間)、.AREPEAT～.AENDR 間、.AWHILE～.AENDW 間以外にあります。
.ENDM または.EXITM を削除してください。

618 (E) MACRO EXPANSION TOO LONG
マクロ展開で 1 行の文字数が 8,192 文字を超えています。
8,192 文字以下になるように訂正してください。

619 (E) ILLEGAL MACRO PARAMETER
マクロコールでマクロパラメータの仮引数名に誤りがあるか、マクロ本体(.MACRO～.ENDM 間)の仮引数名に誤りがあります。
仮引数名を訂正してください。
マクロ本体の仮引数名が誤りの場合はマクロ展開時にエラーになります。

620 (E) UNDEFINED PREPROCESSOR VARIABLE
参照しているプリプロセッサ変数が定義されていません。
プリプロセッサ変数を定義してください。

621 (E) ILLEGAL .END DIRECTIVE POSITION
マクロ展開中に.END があります。
.END を削除してください。

622 (E) ')' NOT FOUND
マクロ処理除外の閉じカッコがありません。
マクロ処理除外の閉じカッコを記述してください。

623 (E) SYNTAX ERROR IN STRING FUNCTION
文字列操作関数に構文上の誤りがあります。
文字列操作関数を見直してください。

624 (E) MACRO PARAMETERS MISMATCH
マクロコールで位置指定のマクロパラメータの数が多すぎます。
マクロパラメータの数を訂正してください。

631 (E) END DIRECTIVE MISMATCH
対になる制御文で終了の制御文が一致しません。
制御文を見直してください。

640 (E) SYNTAX ERROR IN OPERAND
条件つきアセンブリ制御文のオペランドに構文上の誤りがあります。
ソースステートメント全体を見直してください。

641 (E) INVALID RELATIONAL OPERATOR
条件つきアセンブリ制御文のオペランドの関係演算子に誤りがあります。
関係演算子を訂正してください。

642 (E) ILLEGAL .END DIRECTIVE POSITION
.AREPEAT～.AENDR 間、.AWHILE～.AENDW 間に.END があります。
.END を削除してください。

643 (E) DIRECTIVE MISMATCH
.AREPEAT、.AWHILE に対する.AENDR、.AENDW が対になっていません。
制御文を見直してください。

644 (E) ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION
.AIF～.AENDI 間に.AENDR、.AENDW があります。
.AENDR、.AENDW を削除してください。

645 (E) EXPANSION TOO LONG
.AREPEAT、.AWHILE 展開で 1 行の文字数が 8,192 文字を超えています。
8,192 文字以下になるように訂正してください。

650 (E) INVALID INCLUDE FILE
.INCLUDE のファイル名に誤りがあります。
ファイル名を訂正してください。

651 (E) CANNOT OPEN INCLUDE FILE
.INCLUDE のファイルをオープンできません。
ファイル名を訂正してください。

652 (E) INCLUDE NEST TOO DEEP
ファイルインクルードのネストが 30 レベルを超えています。
ネストを 30 レベル以下にしてください。

653 (E) SYNTAX ERROR IN OPERAND
.INCLUDE のオペランドに構文上の誤りがあります。
オペランドを訂正してください。

660 (E) .ENDM NOT FOUND
.MACRO に対する.ENDM がありません。
.ENDM を記述してください。

662 (E) ILLEGAL .END DIRECTIVE POSITION
.AIF～.AENDI 間に.END があります。
.END を削除してください。

663 (E) ILLEGAL .END DIRECTIVE POSITION
インクルードファイル中に.END があります。
.END を削除してください。

664 (E) ILLEGAL .END DIRECTIVE POSITION
.AIF～.AENDI 間に.END があります。
.END を削除してください。

665 (E) EXPANSION TOO LONG
.DEFINE で 1 行の文字数が 8,192 文字を超えています。
8,192 文字以下になるように訂正してください。

667 (E) SUCCESSFUL CONDITION .AERROR
.AERROR を含むステートメントが.AIF の条件によって処理されています。
.AERROR を処理しないように条件式を見直してください。

668 (E) ILLEGAL VALUE IN OPERAND
.AIFDEF のオペランドに誤りがあります。
本制御命令のオペランドは.DEFINE のシンボルで指定してください。

669 (E) STRING TOO LONG
オペランドの文字列が 255 文字を超えています。
.ASSIGNC、.DEFINE、文字列操作関数(.LEN、.INSTR、.SUBSTR)のオペランドに指定する文字列は 255 文字以内としてください。

670 (E) ILLEGAL SYMBOL IN OPERAND
マイコン種別が SH2A、SH2AFPU の時、プリプロセッサ制御命令にプリプロセッサ変数以外のシンボルを指定しました。
シンボルを訂正してください。

700 (W) ILLEGAL VALUE IN OPERAND (ニーモニック)
DSP 演算命令のオペランドが値の範囲を超えています。
値を訂正してください。

701 (W) MULTIPLE REGISTER IN DESTINATION (ニーモニック, ニーモニック)
DSP 演算命令のデスティネーションオペランドに複数の同一レジスタを指定しています。
レジスタの指定を訂正してください。

702 (W) ILLEGAL OPERATION SIZE (ニーモニック)
DSP 演算命令またはデータ転送命令のオペレーションサイズが誤っています。
オペレーションサイズを訂正または削除してください。

703 (W) MULTIPLE REGISTER IN DESTINATION (ニーモニック, ニーモニック)
DSP 演算命令とデータ転送命令で同一のデスティネーションレジスタを指定しています。
レジスタの指定を訂正してください。

704 (W) A PRIVILEGED INSTRUCTION "ニーモニック" IS USED
chkmd オプション指定で、特権モード命令が存在します。

705 (W) A LDTLB INSTRUCTION IS USED
chktlb オプション指定で、LDTLB 命令が存在します。

706 (W) A CACHE INSTRUCTION "ニーモニック" IS USED
chkcache オプション指定で、キャッシュ関連命令が存在します。

707 (W) A DSP INSTRUCTION "ニーモニック" IS USED
chkdsp オプション指定で、DSP 関連命令が存在します。

708 (W) A FPU INSTRUCTION "ニーモニック" IS USED
chkfpu オプション指定で、FPU 関連命令が存在します。

800 (W) SYMBOL NAME TOO LONG
プリプロセッサ変数が 32 文字を超えています。
プリプロセッサ変数を訂正してください。
アセンブラは 33 文字目以降を無視します。

801 (W) MULTIPLE SYMBOLS
定義済みのシンボルを再び定義しています。
シンボルの再定義を取りやめてください。
アセンブラは 2 度目以降の定義を無視します。

807 (W) ILLEGAL OPERATION SIZE
オペレーションサイズに誤りがあります。
オペレーションサイズを訂正してください。
アセンブラはオペレーションサイズの指定を無視します。

808 (W) ILLEGAL CONSTANT SIZE
整数定数の記述の一部に誤りがあります。
記述を訂正してください。
アセンブラが整数定数を誤って(プログラマの意図しない値として)解釈する可能性があります。

810 (W) TOO MANY OPERANDS
アセンブラ制御命令のオペランドが多すぎるか、コメントに誤りがあります。
オペランドまたはコメントを訂正してください。
アセンブラは余分なオペランドの指定を無視します。

811 (W) ILLEGAL SYMBOL DEFINITION
ラベルを記述できないアセンブラ制御命令のソースステートメントにラベルを記述しています。
ラベルを削除してください。
アセンブラはラベルを無視します。

813 (W) SECTION ATTRIBUTE MISMATCH
セクションの再開で異なる種類のセクションを指定しているか、絶対アドレスセクションの再開でセクションの先頭アドレスを再び指定しています。
セクションを再開する場合は異なる種類のセクションや先頭アドレスを指定しないでください。
セクションを開始したときの指定がそのまま有効です。

814 (W) ILLEGAL OBJECT CODE SIZE
確保サイズに誤りがあります。
確保サイズに指定できるのは:12 のみです。

815 (W) MULTIPLE MODULE NAMES
オブジェクトモジュール名を再設定しています。
オブジェクトモジュールの設定は 1 度だけにしてください。
アセンブラは 2 度目以降の設定を無視します。

816 (W) ILLEGAL DATA AREA ADDRESS
データまたはデータ領域の配置が不当です。
ワード単位のデータやデータ領域は先頭アドレスが偶数になるように確保してください。
ロングワードまたは単精度単位のデータやデータ領域は先頭アドレスが 4 の倍数になるように確保してください。
倍精度単位のデータやデータ領域は先頭アドレスが 8 の倍数になるように確保してください。
アセンブラはデータまたはデータ領域を指定どおりに配置します。

817 (W) ILLEGAL BOUNDARY VALUE
コードセクションまたはスタックセクションのアライメント数が 4 未満です。
指定は有効です。
ただし、実行命令、DSP 命令、拡張命令を奇数アドレスに記述している場合はウォーニング 882 になります。
コードセクションまたはスタックセクションのアライメント数に 1 を指定する場合は特に注意してください。

818 (W) COMMANDLINE OPTION MISMATCH FOR FLOATING DIRECTIVE
マイコン種別が SH2E のときは、round=nearest または denormalize=on を、マイコン種別が SH2AFPU のときは、denormalize=on を指定しています。
round オプションまたは denormalize オプションの指定を変更してください。
アセンブラは round オプションまたは denormalize オプションの指定どおりにオブジェクトコードを生成します。

825 (W) ILLEGAL INSTRUCTION IN DUMMY SECTION
ダミーセクションに実行命令、DSP 命令、拡張命令、データを確保するアセンブラ制御命令を記述しています。
実行命令、DSP 命令、拡張命令、データを確保するアセンブラ制御命令を削除してください。
アセンブラは実行命令、拡張命令、DSP 命令、データを確保するアセンブラ制御命令の記述を無視します。

826 (W) ILLEGAL PRECISION
浮動小数点定数の精度がオペレーションサイズで指定した精度と異なります。
オペレーションサイズで指定した精度または浮動小数点定数の精度を訂正してください。
アセンブラはオペレーションサイズで指定した精度を有効として処理します。

832 (W) MULTIPLE 'P' DEFINITIONS
デフォルトセクション名としての P が他のシンボルである P と重複しています。
P が重複しないようにしてください。
アセンブラは P をデフォルトセクション名とみなし、他のシンボル P の定義を無効とします。

835 (W) ILLEGAL VALUE IN OPERAND
実行命令のオペランドに範囲外の値を指定しています。
値を訂正してください。
アセンブラは値を範囲内に補正してオブジェクトコードを生成します。

836 (W) ILLEGAL CONSTANT SIZE
整数定数の記述の一部に誤りがあります。
記述を訂正してください。
アセンブラが整数定数を誤って (プログラマの意図しない値として) 解釈する可能性があります。

837 (W) SOURCE STATEMENT TOO LONG
ソースステートメントの 1 行の長さが 8,192 バイトを超えています。
コメント文を短くするなどして 1 行を 8,192 バイト以内に納めてください。
または、ソースステートメントを複数行に分けて記述してください。

838 (W) ILLEGAL CHARACTER CODE
コメント、文字列以外にシフト JIS、EUC または LATIN1 コードを指定したか、sjis オプション、euc オプション、または latin1 オプションの指定がありません。
シフト JIS コード、EUC コードまたは LATIN1 コードはコメント、文字列内に指定してください。もしくは、sjis オプション、euc オプション、latin1 オプションを指定してください。

839 (W) ILLEGAL FIGURE IN OPERAND
固定小数点データでワードサイズの場合は 6 桁以上、ロングワードサイズの場合は 11 桁以上のデータを指定しました。
余分な桁を削除してください。

840 (W) OPERAND OVERFLOW
浮動小数点データがオーバフローしました。
値を変更してください。
正の値の場合は+∞、負の値の場合は-∞になります。

841 (W) OPERAND UNDERFLOW
浮動小数点データがアンダフローしました。
値を変更してください。
正の値の場合は+0、負の値の場合は-0 になります。

842 (W) OPERAND DENORMALIZED
浮動小数点データに非正規化数を指定しました。
浮動小数点データを確認してください。
アセンブラは指定どおりにオブジェクトコードを生成します (非正規化数を設定します)。

843 (W) INEFFECTIVE FLOATING POINT OPERATION
定数式の中で∞－∞、0.0/0.0 等の無効演算を行っています。
無効演算の結果を表す非数に対応する内部表現の値を仮定します。

844 (W) DIVISION BY FLOATIONG POINT ZERO
定数式の中でゼロ除算を行っています。
符号に従って＋∞または－∞に対応する内部表現の値を仮定します。

845 (W) ILLEGAL IMMEDIATE VALUE
イミディエイト値が不正です。下位 8 ビットが 0 となっていません。
アセンブラはイミディエイト値の下位 8 ビットを 0 に補正します。

850 (W) ILLEGAL SYMBOL DEFINITION
ラベルフィールドにシンボルを指定しました。
シンボルを削除してください。

851 (W) MACRO SERIAL NUMBER OVERFLOW
マクロ生成番号が 99,999 を超えています。
マクロコールの回数を減らしてください。

852 (W) UNNECESSARY CHARACTER
オペランドの終了後に文字があります。
オペランドを訂正してください。

854 (W) .AWHILE ABORTED BY .ALIMIT
展開回数が.ALIMIT で設定した上限値に達したため展開を中断しました。
繰り返しを展開する条件を見直してください。

856 (W) MULTIPLE SYMBOLS
同一シンボルに対してスタック値を再び定義しています。
スタック値の再定義をやめてください。
アセンブラは 2 度目以降の定義を無視します。

870 (W) ILLEGAL DISPLACEMENT VALUE
ディスプレースメント値が不当です。
オペレーションサイズがワードのときにディスプレースメントが偶数となっていません。
オペレーションサイズがロングワードのときにディスプレースメントが 4 の倍数となっていません。
アセンブラがディスプレースメントを補正することを考慮してください。
アセンブラはオペレーションサイズに応じてディスプレースメントを補正してオブジェクトコードを生成します。
オペレーションサイズがワードのときはディスプレースメントが偶数になるように切り捨てます。
オペレーションサイズがロングワードのときはディスプレースメントが 4 の倍数になるように切り捨てます。

874 (W) CANNOT CHECK DATA AREA BOUNDARY
PC 相対データ転送命令において、データ領域の境界をチェックできません。
リンクする際にはデータ領域の境界に十分注意してください。
データ転送命令が相対アドレスセクションに属している場合や、データ領域が外部参照シンボルである場合などにアセンブラは本ウォーニングを出力します。

875 (W) CANNOT CHECK DISPLACEMENT SIZE
PC 相対データ転送命令において、ディスプレースメントの大きさをチェックできません。
リンクする際にはデータ転送命令とデータ領域の距離に十分注意してください。
データ転送命令が相対アドレスセクションに属している場合や、データ領域が外部参照シンボルである場合などにアセンブラは本ウォーニングを出力します。

876 (W) ASSEMBLER OUTPUTS BRA INSTRUCTION
アセンブラが自動的に BRA 命令を出力しました。
.POOL などを用いてリテラルプールの出力位置を指定するか、アセンブラが自動的に生成した BRA 命令でプログラムが正常に動作するかどうかを確認してください。
リテラルプールの出力ポイントが見つからないため、リテラルプールとそれを飛び越すための BRA 命令を自動出力したことを示します。

880 (W) .END NOT FOUND
プログラムに.END がありません。
.END を記述してください

881 (W) ILLEGAL DIRECTIVE IN REPEAT LOOP
リピートループ内に不正なアセンブラ制御命令を指定しています。
不正なアセンブラ制御命令を削除してください。
リピートループ内にデータまたはデータ領域を確保する制御命令、.ALIGN、.ORG を記述した場合、1 制御命令を 1 命令としてリピートする命令数をカウントします。

882 (W) ILLEGAL ADDRESS
実行命令、拡張命令を奇数アドレスに記述しています。
実行命令、拡張命令は偶数アドレスに記述してください。

883 (W) MULTIPLE FILE NAMES
.FILE によるファイル指定が複数回あります。
2 回目以降の指定を無視します。

884 (W) "マイコン種別 1" IS INTERPRETED AS " マイコン種別 2"
cpu=<マイコン種別 1>は無効です。
cpu=<マイコン種別 2>と解釈してアセンブルします。

885 (W) CANNOT SUPPORT "マイコン種別"
.CPU で指定したマイコン種別はサポートしていません。
cpu オプションで指定したマイコン種別として解釈します。

901 (F) SOURCE FILE INPUT ERROR
ソースファイルの入力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

902 (F) MEMORY OVERFLOW
メモリ不足です(中間語に関する情報を処理できません)。
プログラムを分割してください。

903 (F) LISTING FILE OUTPUT ERROR
リストファイルの出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

904 (F) OBJECT FILE OUTPUT ERROR
オブジェクトファイルの出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

905 (F) MEMORY OVERFLOW
メモリ不足です(ソースプログラムの行に関する情報を処理できません)。
プログラムを分割してください。

906 (F) MEMORY OVERFLOW
メモリ不足です(シンボルに関する情報を処理できません)。
プログラムを分割してください。

907 (F) MEMORY OVERFLOW
メモリ不足です(セクションに関する情報を処理できません)。
プログラムを分割してください。

908 (F) SECTION OVERFLOW
セクションの個数が多すぎます。
デバッグ情報を出力するときは 62,265 までです。
デバッグ情報を出力しないときは 65,274 個までです。
プログラムを分割してください。

933 (F) ILLEGAL ENVIRONMENT VARIABLE
マイコン種別に誤りがあります。
マイコン種別を訂正してください。

935 (F) SUBCOMMAND FILE INPUT ERROR
サブコマンドファイル入力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

950 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

951 (F) LITERAL POOL OVERFLOW
リテラルプール用の内部シンボルの個数が 100,000 個を超えています。
ソースプログラムを分割してください。

952 (F) LITERAL POOL OVERFLOW
リテラルプールの容量があふれています。
リテラルプールがあふれる前に無条件分岐を挿入してください。

953 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

954 (F) LOCAL BLOCK NUMBER OVERFLOW
ローカルラベルの有効範囲であるローカルブロックの個数が 100,000 個を超えています。
ソースプログラムを分割してください。

956 (F) EXPAND FILE INPUT/OUTPUT ERROR
プリプロセッサ展開出力のファイル出力時にエラーが発生しました。
ディスクの空き容量を確認してください。
ディスク上の不要なファイルを削除するなどして必要な空き容量を確保してください。

957 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

958 (F) MEMORY OVERFLOW
メモリ不足です。
ソースプログラムを分割してください。

964 (F) MEMORY OVERFLOW
メモリ不足です(シンボルに関する情報を処理できません)。
ソースプログラムを分割してください。

970 (F) MEMORY OVERFLOW
メモリ不足です(セクションのサイズが大きすぎます)。
.ORG でロケーションカウンタに大きなオフセットを与えたり、.DATAB 等で大きなデータ領域を確保した可能性があります。
セクションを分割するか、データ領域を小さくしてください。

14.　最適化リンケージエディタのエラーメッセージ

14.1　エラー形式とエラーレベル

本章では、以下の形式で出力するエラーメッセージとエラー内容を説明します。

エラー番号　(エラーレベル) エラーメッセージ
　　エラー内容

エラーレベルは、エラーの重要度に従い、5 種類に分類されます。

	エラーレベル		動作
L0000 - L0999 P0000 - P0999	(I)	インフォメーション	処理を継続します。
L1000 - L1999 P1000 - P1999	(W)	ウォーニング	処理を継続します。
L2000 - L2999 P2000 - P2999	(E)	エラー	オプション解析処理を継続し、処理を中断します。
L3000 - L3999 P3000 - P3999	(F)	フェータル	処理を中断します。
L4000 - P4000 -	(-)	インターナル	処理を中断します。

L で始まるエラー番号は、最適化リンケージエディタ出力メッセージです。

P で始まるエラー番号は、プレリンカ出力メッセージです。P で始まるエラー番号は、nomessage オプションや change_message オプションで指定できません。

14.2　エラーの返値

最適化リンケージエディタは処理を終了した際、処理結果によって次の値を OS に返します。

返値	内容
0	正常終了、インフォメーションメッセージ出力後終了、およびウォーニング出力後終了します。
1	エラー、フェータル、インターナルおよび強制終了します。

14.3 メッセージ一覧

L0001 (I) Section "セクション" created by optimization "最適化"

"最適化"の最適化によって、"セクション"を作成しました。

L0002 (I) Symbol "シンボル" created by optimization "最適化"

"最適化"の最適化によって、"シンボル"を作成しました。

L0003 (I) "ファイル"-"シンボル" moved to "セクション" by optimization variable_access の最適化によって、"ファイル"内の"シンボル"を移動しました。

L0004 (I) "ファイル"-"シンボル" deleted by optimization 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。

L0005 (I) 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"

"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。

L0100 (I) No inter-module optimization information in "ファイル"

"ファイル"内にモジュール間最適化情報がありません。"ファイル"をモジュール間最適化の対象外にします。モジュール間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。ただし、asmsh には goptimize オプションはありません。

L0101 (I) No stack information in "ファイル"

"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または SYSROF->ELF コンバートファイルの可能性があります。最適化リンケージエディタ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。

L0102 (I) Stack size "サイズ" specified to the undefined symbol "シンボル" in "ファイル"

"ファイル"内の未定義シンボル"シンボル"に、スタックサイズ "サイズ" が指定されています。

L0103 (I) Multiple stack sizes specified to the symbol "シンボル"

シンボル"シンボル"は、複数のスタックサイズが指定されています。

L0300 (I) Mode type "モード種別１" in "ファイル" differ from "モード種別２"

異なるモード種別のファイルを入力しました。

L0400 (I) Unused symbol "ファイル"-"シンボル"

"ファイル"内の"シンボル"は使用されていません。

L0500 (I) Generated CRC code at "アドレス"

"アドレス"に CRC コードを出力しました.

L0510 (I) Section "セクション" was moved other area specified in option "cpu=<メモリ属性>"

セクションを分割せずに cpu=<メモリ属性>にしたがって“セクション”を配置しました。

L0511 (I) Sections "セクション名","分割後のセクション名" are Non-contiguous

〝セクション名〞のセクションを分割し、〞分割後のセクション名〞のセクションを生成しました。

L1000 (W) Option "オプション" ignored

"オプション"は無効です。"オプション"を無視します。

L1001 (W) Option "オプション 1" is ineffective without option "オプション 2"

"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。

L1002 (W) Option "オプション 1" cannot be combined with option "オプション 2"

"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。

L1003 (W) Divided output file cannot be combined with option "オプション"

"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。

L1004 (W) Fatal level message cannot be changed to other level : "番号"

Fatal レベルメッセージはレベル変更できません。"番号"の指定を無視します。change_message オプションで変更できるエラーは、Information/Warning/Error レベルです。

L1005 (W) Subcommand file terminated with end option instead of exit option

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。

L1006 (W) Options following exit option ignored

exit オプションの後のオプションを無視しました。

L1007 (W) Duplicate option : "オプション"

"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。

L1008 (W) Option "オプション" is effective only in cpu type "マイコン種別"

"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。

L1010 (W) Duplicate file specified in option "オプション" : "ファイル名"

"オプション"で同じファイルを 2 度指定しました。2 度目の指定を無視します。

L1011 (W) Duplicate module specified in option "オプション" : "モジュール"

"オプション"で同じモジュールを 2 度指定しました。2 度目の指定を無視します。

L1012 (W) Duplicate symbol/section specified in option
"オプション" : "名前"

"オプション"で同じシンボル名またはセクション名を 2 度指定しました。2 度目の指定を無視します。

L1013 (W) Duplicate number specified in option "オプション" : "番号"

"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。

L1100 (W) Cannot find "名前" specified in option "オプション"

"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。

L1101 (W) "名前" in rename option conflicts between symbol and section

rename 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。

L1102 (W) Symbol "シンボル" redefined in option "オプション"

"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。

L1103 (W) Invalid address value specified in option
"オプション" : "アドレス"

"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。

L1104 (W) Invalid section specified in option "オプション" : "セクション"

"オプション"で無効なセクションを指定しています。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。
- -jump_entries_for_picオプションは、プログラムセクション以外を指定できません。

L1110 (W) Entry symbol "シンボル" in entry option conflicts

entry オプションで指定した"シンボル"以外のシンボルがコンパイル、アセンブル時にエントリシンボルとして指定されています。オプション指定を優先します。

L1120 (W) Section address is not assigned to "セクション"

"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。

optlnk 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。

L1121 (W) Address cannot be assigned to absolute section "セクション" in start option

"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。

L1122 (W) Section address in start option is incompatible with alignment : "セクション"

start オプションで指定した"セクション"のアドレスはアライメント数と矛盾しています。アライメント数に合わせてセクションアドレスを補正します。

L1130 (W) Section attribute mismatch in rom option :
"セクション 1,セクション 2"

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、アライメント数が異なります。"セクション 2"のアライメント数はどちらか大きい方を有効とします。

L1140 (W) Load address overflowed out of record-type in option "オプション"

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 形式を超える範囲は、別の record 形式で出力します。

L1141 (W) Cannot fill unused area from "アドレス" with the specified value

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。

L1150 (W) Sections in "オプション" option have no symbol

"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。

L1160 (W) Undefined external symbol "シンボル"

未定義の"シンボル"を参照しています。

L1170 (W) Specified SBR addresses conflict

異なる複数の SBR アドレスが指定されました。SBR=USER として処理します。

L1171 (W) Least significant byte in SBR="定数" ignored

SBR オプションで指定されたアドレス"定数"の下位 8bit は無効です。

L1180 (W) Directive command "制御命令" is duplicated in "ファイル"

複数のソースファイルに、"制御命令"を記述しています。

"制御命令"は、複数記述することはできません。

L1181 (W) Fail to write "出力コード種別"

出力ファイルへの、"出力コード種別"の書き込みが失敗しました。

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:

ID コード書き込み失敗時・・・「"ID Code"」

→ L1181 Fail to write "ID Code"

PROTECT/OFSREG コード書き込み失敗時・・・「"Protect Code" or "OFSREG Code"」

→ L1181 Fail to write "Protect Code" or "OFSREG Code"

CRC コード書き込み失敗時・・・「"CRC Code"」

→ L1181 Fail to write "CRC Code"

L1182 (W) Cannot generate vector table section "セクション"

入力ファイル内に、ベクタテーブル"セクション"があります。リンカは、"セクション"を自動生成しません。

L1183 (W) Interrupt number "ベクタ番号" of "セクション" is defined in input file

VECTN オプションで記述したベクタ番号は、入力ファイル内で定義済みです。入力ファイルの内容を優先して、処理を継続します。

L1190 (W) Section "セクション" was moved other area specified in option "cpu=<メモ属性>"

外部変数アクセス最適化によりオブジェクトサイズが変更されたため、次の cpu 指定範囲の”セクション”を移動しました。

L1191 (W) Area of "FIX" is within the range of the area specified by “cpu=<メモリ属性>” :"<start>-<end>"

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。

L1192 (W) Bss Section "セクション名" is not initialized

初期値なしのデータセクション"セクション名"は、初期設定プログラムで初期化できません。-cpu 指定範囲、ポインタ変数のサイズ指定を見直してください。

L1193 (W) Section "セクション名" specified in option "オプション" is ignored

-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。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。

L1194 (W) Section "セクション" in relocation "ファイル"-"セクション"-"オフセット" is changed.

"セクション""ファイル""オフセット"　の位置にある"セクション"を参照していたリロケーションが、分割した後半セクションを参照するよう変更しました。分割しないようにするには、"セクション"を contiguous_section オプションで指定してください。

L1200 (W) Backed up file "ファイル 1" into "ファイル 2"

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。

L1300 (W) No debug information in input files

入力ファイル内にデバッグ情報がありません。debug,sdebug,compress オプション指定を無視します。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。

L1301 (W) No inter-module optimization information in input files

入力ファイル内に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。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。

L1302 (W) No stack information in input files

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または SYSROF->ELF コンバートファイルの場合は、stack オプションは無効です。

L1303 (W) No rts information in input files

.rts ファイルを生成可能な入力ファイルがありません。

.rts ファイルを生成せずに処理を終了します。

L1304 (W) No utl information in input files

utl ファイルを生成するための情報が入力されませんでした。

L1305 (W) Entry address in "ファイル" conflicts : "アドレス"
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。

L1310 (W) "セクション" in "ファイル" is not supported in this tool
"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。

L1311 (W) Invalid debug information format in "ファイル"
"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。

L1320 (W) Duplicate symbol "シンボル" in "ファイル"
"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。

L1321 (W) Entry symbol "シンボル" in "ファイル" conflicts
エントリシンボル定義のあるオブジェクトファイルを複数入力しました。先に入力したファイル内のエントリシンボルを有効にします。

L1322 (W) Section alignment mismatch : "セクション"
アライメント数の異なる同名セクションを入力しました。アライメント数は最大の指定を有効にします。

L1323 (W) Section attribute mismatch : "セクション"
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。

L1324 (W) Symbol size mismatch : "シンボル" in "ファイル"
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。

L1325 (W) Symbol attribute mismatch : "シンボル" : "ファイル"
“ファイル”内の”シンボル”が、他のファイルの同名シンボルと属性が一致していません。シンボルを確認してください。

L1326 (W) Reserved symbol "シンボル” is defined in "ファイル”
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。

L1327 (W) Section alignment in option "aligned_section" is small : "セクション"
aligned_section オプション指定時のアライメント数 16 の方が、"セクション"のアライメント数より小さいため、指定セクションに対するオプション指定を無視します。

L1330 (W) Cpu type "マイコン種別１" in "ファイル" differ from "マイコン種別２"
異なるマイコン種別のファイルを入力しました。マイコン種別を H8SX として処理を継続します。

L1400 (W) Stack size overflow in register optimization
レジスタ最適化で、スタックアクセスコードがコンパイラのスタック量制限値を超えました。レジスタ最適化指定を無視します。

L1401 (W) Function call nest too deep
関数の呼び出しネストが深すぎるため、レジスタ最適化を実施できません。

L1402 (W) Parentheses specified in option "start" with optimization
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。

L1410 (W) Cannot optimize "ファイル"-"セクション" due to multi label relocation operation
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。

L1420 (W) "ファイル" is newer than "プロファイル"
"ファイル"は"プロファイル"より後に更新されました。プロファイル情報を無視します。

L1430 (W) Cannot generate effective bls file for compiler optimization
無効な bls ファイルが生成されました。コンパイル時に、外部変数アクセス最適化(map オプション)を指定しても、この最適化は実施できません。
コンパイラの外部変数アクセス最適化(map オプション)には、以下の制限があります。該当する内容がないかを確認し、セクション配置を見直してください。
コンパイル時に base オプションを使用している場合、プログラムセクションの直後にデータセクションを配置すると、外部変数アクセス最適化が実施できない場合があります。
※bls ファイルは"外部シンボル割り付け情報ファイル"を指します。コンパイラの map オプションに使用するための情報ファイルです。

L1500 (W) Cannot check stack size

スタックセクションがないため、コンパイル時の stack オプションで指定したスタックサイズの整合性をチェックできません。コンパイル時の stack オプションの整合性をチェックするためにはコンパイル時、アセンブル時に goptimize オプション指定が必要です。

L1501 (W) Stack size overflow : "スタックサイズ"

スタックセクションサイズが、コンパイル時に stack オプションで指定した"スタックサイズ"を超えました。コンパイル時のオプションを変更するか、スタック量を削減できるようにプログラムを変更してください。

L1502 (W) Stack size in "ファイル" conflicts with that in another file

複数のファイルで異なるスタックサイズを指定されています。コンパイル時のオプションを確認してください。

L1510 (W) Input file was compiled with option "smap" and option "map" is specified at linkage

"smap を指定してコンパイルしたファイルがあります。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。

P1600 (W) An error occurred during name decoding of "インスタンス"

"インスタンス"はデコードできませんでした。エンコード名でメッセージ出力します。

L2000 (E) Invalid option : "オプション"

"オプション"はサポートしていません。

L2001 (E) Option "オプション" cannot be specified on command line

"オプション"はコマンドライン上では指定できません。サブコマンドファイル内で指定してください。

L2002 (E) Input option cannot be specified on command line

コマンドライン上で input オプションを指定しました。コマンドライン上での入力ファイル指定は input オプション無しで指定してください。

L2003 (E) Subcommand option cannot be specified in subcommand file

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。

L2004 (E) Option "オプション 1" cannot be combined with option "オプション 2"
"オプション 1"と"オプション 2"は同時に指定できません。

L2005 (E) Option "オプション" cannot be specified while processing
"プロセス"
"プロセス"処理に対して"オプション"は指定できません。

L2006 (E) Option "オプション 1" is ineffective without option "オプション 2"
"オプション 1"は"オプション 2"が必要です。

L2010 (E) Option "オプション" requires parameter
"オプション"はパラメータ指定が必要です。

L2011 (E) Invalid parameter specified in option "オプション" : "パラメータ"
"オプション"で無効なパラメータを指定しました。

L2012 (E) Invalid number specified in option "オプション" : "値"
"オプション"指定で無効な値を指定しました。値の範囲を確認してください。

L2013 (E) Invalid address value specified in option
"オプション" : "アドレス"
"オプション"で指定した"アドレス"は無効な値です。0～FFFFFFFF の間の 16 進数で指定してください。

L2014 (E) Illegal symbol/section name specified in "オプション" : "名前"
"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。セクション/シンボル名で使用できるのは数字、英字、_、$(先頭は数字以外)です。

L2016 (E) Invalid alignment value specified in option "オプション" : "アライメント数"
"オプション"で指定した"アライメント数"は無効な値です。
1,2,4,8,16 または 32 を指定してください。

L2017 (E) Cannot output “セクション” specified in option “オプション”
"オプション"で指定した"セクション"のコードの一部を出力できません。命令コードのエンディアンを変換したことにより、"セクション"内命令コードの一部が非連続となりました。非連続部分の命令コードが属しているセクションは、リンケージリストからセクションアドレスを 4 バイト境界で確認の上、出力するセクションがどのセクションとエンディアン変換を行っているか確認してください。

L2020 (E) Duplicate file specified in option "オプション" : "ファイル"
"オプション"指定で同じファイルを 2 度指定しました。

L2021 (E) Duplicate symbol/section specified in option
"オプション" : "名前"
"オプション"指定で同じシンボル名またはセクション名を 2 度指定しました。

L2022 (E) Address ranges overlap in option "オプション" : "アドレス範囲"
"オプション"で指定した"アドレス範囲"が重複しています。

L2100 (E) Invalid address specified in cpu option : "アドレス"
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。

L2101 (E) Invalid address specified in option "オプション" : "アドレス"
"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超えました。

L2110 (E) Section size of second parameter in rom option is not 0 : "セクション"
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。

L2111 (E) Absolute section cannot be specified in "オプション" option : "セクション"
"オプション"で絶対アドレスセクションを指定しました。

L2112 (E) "セクション 1" and "セクション 2" cannot mapped as ROM/RAM in "ファイル"
"ファイル名"で指定された"セクション 1"と"セクション 2"は ROM/RAM 対応となりません。

L2113 (E) Option "rom" and internal information in the file are conflicted
rom オプションの指定と内部情報が矛盾しています。

L2120 (E) Library "ファイル" without module name specified as input file
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。

L2121 (E) Input file is not library file : "ファイル(モジュール)"
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。

L2130 (E) Cannot find file specified in option "オプション" : "ファイル"
"オプション"で指定したファイルが見つかりません。

L2131 (E) Cannot find module specified in option "オプション" : "モジュール"
"オプション"で指定したモジュールがありません。

L2132 (E) Cannot find "名前" specified in option "オプション"
"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。

L2133 (E) Cannot find defined symbol "名前" in option "オプション"
"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。

L2140 (E) Symbol/section "名前" redefined in option "オプション"
"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。

L2141 (E) Module "モジュール" redefined in option "オプション"
"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。

L2142 (E) Interrupt number "ベクタ番号" of "セクション" has multiple definition
ベクタテーブル"セクション"の、ベクタ番号定義が複数入力されました。ベクタ番号には、ひとつのアドレスしか設定できません。ソースファイルの記述を見直してください。

L2143 (E) Invalid vector number specified : "number"
number で示すベクタ番号は指定できません。
#pragma special で指定したベクタ番号を見直してください。

L2200* (E) Illegal object file : "ファイル"
ELF フォーマット以外を入力しました。
* P2200 と表示される場合があります。

L2201 (E) Illegal library file : "ファイル"
"ファイル"はライブラリファイルではありません。

L2202 (E) Illegal cpu information file : "ファイル"
"ファイル"はマイコン情報ファイルではありません。

L2203 (E) Illegal profile information file : "ファイル"
"ファイル"はプロファイル情報ファイルではありません。

L2210 (E) Invalid input file type specified for option "オプション" : "ファイル(種別)"
"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。

L2211 (E) Invalid input file type specified while processing "プロセス" : "ファイル(種別)"
"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。

L2212 (E) "オプション" cannot be specified for inter-module optimization information in "ファイル"
"ファイル"内に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。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。

L2220 (E) Illegal mode type "モード種別" in "ファイル"
異なる"モード種別"のファイルを入力しました。

L2221 (E) Section type mismatch : "セクション"
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。

L2223 (E) Cpu type "CPU 種別 1" in "ファイル" is incompatible with "CPU 種別 2"
異なる CPU 種別を入力しました。
一部の仕様に互換性がないため、リンクしても動作が保証できません。

L2300 (E) Duplicate symbol "シンボル" in "ファイル"
"シンボル"は重複しています。

L2301 (E) Duplicate module "モジュール" in "ファイル"
"モジュール"は重複しています。

L2310 (E) Undefined external symbol "シンボル" referenced in "ファイル"
"ファイル"内で未定義の"シンボル"を参照しています。

L2311 (E) Section "セクション 1" cannot refer to overlaid section : "セクション 2"-"シンボル"
同一アドレスを指定したオーバーレイセクション間でシンボル参照がありました。
"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。

L2320 (E) Section address overflowed out of range : "セクション"
"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。

L2321 (E) Section "セクション１" overlaps section "セクション２"
"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。start オプションのアドレス指定を変更してください。

L2322 (E) Section size too large: "セクション"
セクション"セクション"のサイズが大きすぎます。
$TBR セクションのサイズは 1024 バイト以内でなければなりません。

L2323 (E) Section "セクション 1(アドレス範囲)" overlaps with section
"セクション 2(アドレス範囲)" in physical space
物理メモリの配置上で、"セクション 1"と"セクション 2"が重複しています。
各セクションの配置アドレスを見直してください。
<アドレス範囲> : <セクションの開始アドレス>-<セクションの終端アドレス>

L2330 (E) Relocation size overflow : "ファイル"-"セクション"-"オフセット"
リロケーション演算結果がリロケーションサイズを超えました。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。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。

L2331 (E) Division by zero in relocation value calculation :
"ファイル"-"セクション"-"オフセット"
リロケーション演算に 0 除算が発生しました。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。

L2332 (E) Relocation value is odd number :
"ファイル"-"セクション"-"オフセット"
リロケーション演算結果が奇数になりました。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。

L2340 (E) Symbol name "ファイル"-"セクション"-"シンボル..." is too long
"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超えました。
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。

L2400 (E) Global register in "ファイル" conflicts : "シンボル","レジスタ"

"ファイル"内で指定したグローバルレジスタにはすでに別のシンボルが割り付いています。

L2401 (E) near8,near16 symbol "シンボル" is outside near memory area

"シンボル"は near8、near16 の範囲に割りついていません。start 指定を変更するか、コンパイル時の near 指定を外して、正しいアドレス計算ができるようにしてください。

L2402 (E) Number of register parameter conflicts with that in another file : "関数"

"関数"は複数のファイルで異なるレジスタパラメータ数を指定されています。

L2403 (E) Fast interrupt register in "ファイル" conflicts with that in another file

"ファイル"内で指定した高速割り込み用汎用レジスタ番号が、他ファイルと統一されていません。高速割り込み用汎用レジスタ番号を他ファイルに合わせて、再度コンパイルして下さい。

L2404 (E) Base register "ベースレジスタ種別" in "ファイル" conflicts with that in another file

"ファイル"内で指定した"ベースレジスタ種別"用のレジスタ番号が、他ファイルと統一されていません。ベースレジスタ番号を他ファイルに合わせて、再度コンパイルして下さい。

L2405 (E) Option "コンパイルオプション" conflicts with that in other files

"コンパイルオプション"の指定が入力ファイル間で統一されていません。
コンパイルオプションを見直してください。

L2410 (E) Address value specified by map file differs from one after linkage as to "シンボル"

"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。下記の (1)～(3) 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) optlnk 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、optlnk の最適化オプションを無効にしてください。

(3) tbr オプションまたは#pragma tbr 使用時、コンパイラの最適化によって、コンパイル時の map オプション指定後のシンボルが削除されることがあります。コンパイル時 map オプションを無効にするか、tbr オプションまたは#pragma tbr を無効にしてください。

L2411 (E) Map file in "ファイル" conflicts with that in another file

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。

L2412 (E) Cannot open file : "ファイル"

"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。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。

L2413 (E) Cannot close file : "ファイル"

"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。

L2414 (E) Cannot read file : "ファイル"

"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。

L2415 (E) Illegal map file : "ファイル"

"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。ファイル名が正しいか確認してください。

L2416 (E) Order of functions specified by map file differs from one after linkage as to "関数名"

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、他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。

L2417 (E) Map file is not the newest version: "ファイル名"

bls ファイルが最新バージョンではありません。

L2420 (E) "ファイル 1" overlap address "ファイル 2" : "アドレス"

ファイル 1 とファイル 2 のアドレスが重複しています。

P2500 (E) Cannot find library file : "ファイル"

ライブラリとして指定した"ファイル"がありません。

P2501 (E) "インスタンス" has been referenced as both an explicit specialization and a generated instantiation

すでに定義が存在しているインスタンスに対して、インスタンス生成を要求しています。
"インスタンス"を使用しているファイルに対して、form=relocate でリロケータブルファイルを作成していないか確認してください。

P2502 (E) "インスタンス" assigned to "ファイル 1" and "ファイル 2"

"ファイル 1"と"ファイル 2"に"インスタンス"定義が重複しています。
"インスタンス"を使用しているファイルに対して、form=relocate でリロケータブルファイルを作成していないか確認してください。

L3000 (F) No input file

入力ファイルがありません。

L3001 (F) No module in library

ライブラリ内のモジュール数が 0 になりました。

L3002 (F) Option "オプション 1" is ineffective without option "オプション 2"

"オプション 1"は"オプション 2"が必要です。

L3004 (F) Unsupported inter-module optimization information type 〝タイプ〞 in 〝ファイル〞

ファイル内にサポートしていないモジュール間最適化情報〞タイプ〞がありました。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。

P3007 (F) Cannot create instantiation request file "ファイル"

インスタンス生成処理用の中間ファイルを作成できません。
オブジェクト作成フォルダ以下のアクセス権が正しいか確認してください。

P3008 (F) Cannot change to directory "フォルダ"

"フォルダ"に移動できません。"フォルダ"が存在するか確認してください。

P3009 (F) File "ファイル" is read-only

"ファイル"は読み取り専用です。アクセス権を変更してください。

L3100 (F) Section address overflow out of range : "セクション"

"セクション"のアドレスが使用可能な上限の領域を超えました。
start オプションのアドレス指定を変更してください。
アドレス空間の詳細については各マイコンのハードウェアマニュアルを参照してください。

L3102 (F) Section contents overlap in absolute section "セクション"

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。ソースプログラムを修正してください。

L3110 (F) Illegal cpu type "マイコン種別" in "ファイル"

異なるマイコン種別のファイルを入力しました。

L3111 (F) Illegal encode type "エンディアン種別" in "ファイル"

異なるエンディアン種別のファイルを入力しました。

L3112 (F) Invalid relocation type in "ファイル"

"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。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。

L3120 (F) Illegal size of the absolute code section : "セクション" in "ファイル"

"ファイル"に存在する絶対アドレスプログラムセクション"セクション" のサイズが不正です。CPU 種別が RX ファミリでビッグエンディアンの場合は、絶対アドレスプログラムセクションのサイズが 4 の倍数になるように変更してください。

L3200 (F) Too many sections

セクション数が翻訳限界を超えました。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。

L3201 (F) Too many symbols

シンボル数が翻訳限界を超えました。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。

L3202 (F) Too many modules

モジュール数が翻訳限界を超えました。ライブラリを分けて作成してください。

L3203 (F) Reserved module name “optlnk_generates”

optlnk_generates_** (**は、01～99 までの数値)は、最適化リンケージエディタ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。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。

L3300* (F) Cannot open file : "ファイル"

"ファイル"をオープンできません。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。

* P3300 と表示される場合があります。

L3301 (F) Cannot close file : "ファイル"

"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。

L3302 (F) Cannot write file : "ファイル"

"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。

L3303* (F) Cannot read file : "ファイル"

"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。

* P3303 と表示される場合があります。

L3310* (F) Cannot open temporary file

中間ファイルをオープンできません。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。

* P3310 と表示される場合があります。

L3311 (F) Cannot close temporary file

中間ファイルをクローズできません。ディスク容量に空きがない可能性があります。

L3312 (F) Cannot write temporary file

中間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。

L3313 (F) Cannot read temporary file

中間ファイルを読めません。ディスク容量に空きがない可能性があります。

L3314 (F) Cannot delete temporary file

中間ファイルを削除できません。ディスク容量に空きがない可能性があります。

L3320* (F) Memory overflow

最適化リンケージエディタが内部で使用するメモリが不足しています。メモリを増やしてください。

* P3320 と表示される場合があります。

L3400 (F) Cannot execute "ロードモジュール"

"ロードモジュール"を起動できません。"ロードモジュール"のパスが設定されているか確認してください。

L3410 (F) Interrupt by user

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。

L3420 (F) Error occurred in "ロードモジュール"

"ロードモジュール"実行中にエラーが発生しました。

P3500 (F) Bad instantiation request file -- instantiation assigned to more than one file

インスタンス生成処理用の中間ファイルに誤りがあります。

リンク対象ファイルを再コンパイルしてください。

P3505 (F) corrupted template information file or instantiation request file

テンプレート処理用中間ファイル、またはインスタンス生成処理用の中間ファイルのデータが誤っています。これらのファイルの編集はしないでください。

L4000* (-) Internal error : ("内部エラー番号") "ファイル 行番号" / "コメント"

最適化リンケージエディタの処理中に内部的な問題が発生しました。

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、販売元のサポートセンターまでご連絡ください。

* P4000 と表示される場合があります。

15.　翻訳限界

15.1　コンパイラの翻訳限界

コンパイラの翻訳限界を表 15.1 に示します。
ソースプログラムを作成する際は、この翻訳限界の範囲で作成してください。

表 15.1　コンパイラの翻訳限界

	分類	項目	翻訳限界
1	起動	define オプションで指定可能なマクロ名総数	制限なし(メモリ容量に依存)
2		ファイル名の文字数	制限なし(OS に依存)
3	ソースプログラム	1 行の文字数	32768 文字
4		1 ファイルあたりのソースプログラムの行数	制限なし(メモリ容量に依存)
5		コンパイル可能なソースプログラムの総行数	制限なし(メモリ容量に依存)
6	プリプロセッサ	#include 文のネストの深さ	制限なし(メモリ容量に依存)
7		#define 文のマクロ名総数	制限なし(メモリ容量に依存)
8		マクロ定義、マクロ呼び出しのパラメータの個数	制限なし(メモリ容量に依存)
9		マクロ名の再置き換えの数	制限なし(メモリ容量に依存)
10		条件コンパイルのネストのレベル数	制限なし(メモリ容量に依存)
11		#if, #elif 文で指定可能な演算子、非演算子の合計数	制限なし(メモリ容量に依存)
12	宣言	関数定義の個数	制限なし(メモリ容量に依存)
13		外部結合となる識別子(外部名)の数	制限なし(メモリ容量に依存)
14		1 関数内で有効な識別子(内部名)の数	制限なし(メモリ容量に依存)
15		基本型を修飾するポインタ、配列、および関数宣言子の数	16 個
16		配列の次元数	6 次元
17		配列・構造体のサイズ	2147483647 バイト
18	文	複文のネストの深さ	制限なし(メモリ容量に依存)
19		繰り返し文(while 文、do 文、for 文)、選択文(if 文、switch 文)の組み合わせによるネストの深さ	4096 レベル
20		1 関数内で指定可能な goto ラベルの数	2147483646 個
21		switch 文の数	2048 個
22		switch 文のネストの深さ	2048 レベル
23		1 つの switch 文内で指定可能な case ラベルの数	2147483646 個
24		for 文のネストの深さ	2048 レベル
25	式	文字列の文字数	32766 文字
26		関数定義、関数呼び出しでパラメータの個数	2147483646 個
27		1 つの式で指定可能な演算子と非演算子の合計数	約 500 個
28	標準ライブラリ	open 関数で一度にオープンできるファイルの数	128
29	セクション	セクション名長*1	8192 文字
30		1 ファイルあたりの#pragma section で指定できるセクション数	2045 個

【注】*1　コンパイラがオブジェクト生成時に作成する長さです。#pragma section や section オプションで指定できる長さはこれより小さくなります。

15.2 アセンブラの翻訳限界

アセンブラの翻訳限界を表 15.2 に示します。

表 15.2 アセンブラの翻訳限界

	項目	翻訳限界
1	1 行文字数	8192 文字
2	文字定数	4 文字
3	シンボル長	制限なし(メモリ容量に依存)*1
4	シンボル数	制限なし(メモリ容量に依存)
5	外部参照シンボル数	制限なし(メモリ容量に依存)
6	外部定義シンボル数	制限なし(メモリ容量に依存)
7	セクションの最大サイズ	H'FFFFFFFF バイト
8	セクション数	H'FEF1 個(デバッグあり)、H'FEFA 個(デバッグなし)
9	ファイルインクルード	ネストは 30 レベル
10	文字列長	255 文字
11	ファイル名の文字数	制限なし(OS に依存)

【注】*1 プリプロセッサ変数名、マクロ名および、マクロ仮引数名は 32 文字までです。
.DEFINE で指定できる置換シンボルの文字数は制限なしです。ただし、置換文字列は 255 文字まで、1 行に記述できる文字数は 8192 文字までです。

16.　バージョンアップにおける注意事項

16.1　バージョンアップ時の注意事項

旧バージョン(「SuperH RISC engine C/C++コンパイラパッケージ」Ver.8.x 以前)からバージョンアップして使用する場合の注意事項を説明します。

16.1.1　プログラムの動作保証

コンパイラをバージョンアップしてプログラム開発する場合、プログラムの動作が変わることがあります。プログラムを作成する際は、以下の点に注意して、お客様のプログラムを十分にテストしてください。

(1)　プログラムの実行時間やタイミングに依存するプログラム

言語仕様は、プログラムの実行時間については何も規定していません。したがってコンパイラのバージョンの違いによりプログラムの実行時間と I/O 等周辺機器のタイミングのずれ、あるいは割り込み処理のような非同期処理の時間の差等により、プログラムの動作が変わる場合があります。

(2)　1 つの式に 2 個以上の副作用が含まれているプログラム

1 つの式に 2 個以上の副作用が含まれている場合、コンパイラのバージョンにより、動作が変わる可能性があります。

例：

```
a[i++]=b[i++];          /* i のインクリメント順序は不定です。          */
f(i++,i++)  ;           /* インクリメントの順序でパラメータの値が変わります。*/
                        /* i の値が 3 の時 f(3,4) または f(4,3) になります。 */
```

(3)　結果がオーバフローや不当演算に依存するプログラム

オーバフローが生じた場合や、不当演算を実施した場合、結果の値は保証しません。したがって、バージョンが変わると動作が変わる可能性があります。

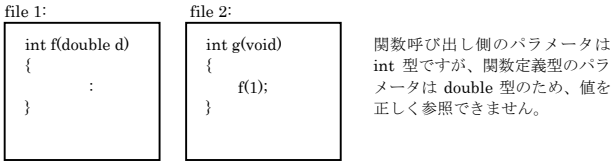
例：

```
int a, b;
x=(a*b)/10;   /* a と b の値の範囲によってはオーバフローする可能性があります */
```

(4)　変数の初期化抜け、型の不一致

変数が初期化されていない場合や、パラメータやリターン値の型が呼び出し側と呼び出される側で対応していない場合、不正な値をアクセスすることになります。したがって、コンパイラのバージョンによって動作が変わる場合があります。

例：



上記に記載された情報が全ての起こりうる状況を示したわけではありません。したがって、お客様の責任で本コンパイラを正しくご使用の上、お客様のプログラムを十分にテストしてください。

16.1.2 旧バージョンとの互換性

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)およびリンケージエディタ(Ver.6.x 以前)出力のオブジェクトファイル、ライブラリファイルとリンクする場合、または旧バージョンで使用していたデバッガをそのまま使用する場合に注意すべき点を説明します。

(1) フォーマットコンバータについて

Ver.9.04 以降のコンパイラパッケージは、フォーマットコンバータを同梱しておりません。旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)出力オブジェクトファイル(SYSROF)を最適化リンケージエディタに入力する必要がある場合は、弊社サポートセンターにお問合せください。

(2) インクルードファイルの基点

フォルダ相対形式で指定されたインクルードファイル検索時、旧バージョンではコンパイラ起動フォルダを基点に検索していましたが、ソースファイルのあるフォルダを基点に検索するように変更しました。

(3) C++プログラム

エンコード規則、実行方式を変更しましたので、旧バージョンのコンパイラで作成した C++オブジェクトファイルはリンクできません。必ずリコンパイルしてから使用してください。

また実行環境の設定で用いる、グローバルクラスオブジェクト初期処理/後処理のライブラリ関数名も変更になりました。「9.2.2 実行環境の設定」を参照し、修正してください。

(4) .END のエントリ指定(アセンブリプログラム)

.END でエントリ指定できるシンボルは外部定義シンボルだけになりました。

(5) 最適化リンケージエディタがサポートするオブジェクトについて

最適化リンケージエディタはバージョンにより、サポートするコンパイラ/アセンブラが異なります。

以下に各リンケージエディタがサポートするツールのバージョンを示します。記載されていないバージョンのオブジェクトファイルのリンケージ処理については保証しません。

- 最適化リンケージエディタ Ver.7： コンパイラ Ver.7 以前、アセンブラ Ver.5 以前
- 最適化リンケージエディタ Ver.8： コンパイラ Ver.8 以前、アセンブラ Ver.6 以前
- 最適化リンケージエディタ Ver.9： コンパイラ Ver.9 以前、アセンブラ Ver.7 以前

16.1.3 旧バージョンのオブジェクトとの互換性

Ver.6 オブジェクトと Ver.7 以降のオブジェクトをリンクするためには、以下のオプションを指定してください。

- gbr = user
- pack = 4　(Ver.8 以降)
- bit_order = left　(Ver.8 以降)

以下のオプションは、コンパイル時、ライブラリ構築時に必ず同じオプションを指定する必要があります。Ver.6 のオブジェクト作成時に以下のオプションを指定している場合は、Ver.7 以降でも同じオプションを指定してください。

- endian = big | little　(SH-2,SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP)
- pic = 0 | 1　(SH-1 以外)
- fpu = single | double　(SH2A-FPU,SH-4,SH-4A)
- fpscr = safe | aggressive　(SH2A-FPU,SH-4,SH-4A)
- round = zero | nearest　(SH2A-FPU,SH-4,SH-4A)
- denormalize = on | off　(SH-4,SH-4A)
- double = float　(SH-4,SH-4A 以外)
- exception | noexception
- rtti = on | off
- rtnext | nortnext
- macsave

アセンブリプログラムについては「9.3.2 関数呼び出し規約」の関数呼び出し規約に準拠していればリンクできます。

【注】 1 マニュアルに記述のない内容については、バージョンアップによる互換性は保証していません。
レジスタの退避・回復順序などコンパイラの出力コードに依存するアセンブリコードを記述している場合は、Ver.6 オブジェクトと Ver.7 以降のオブジェクトをリンクできません。
2 OS やミドルウェア等とのリンクについては、購入元にお問い合わせください。

16.1.4 コマンドラインインタフェース

(1) アセンブラ、最適化リンケージエディタコマンドライン規約

ファイル名、オプション間に、空白文字が必須になりました。
また、ファイル名、オプションの指定順序に制限がなくなりました。

(2) 最適化リンケージエディタオプション

会話形式のオプション指定サポートを廃止しました。
また、旧バージョンのモジュール間最適化ツール(optlnksh)とリンケージエディタ(lnk)、ライブラリアン(lbr)、オブジェクトコンバータ(cnvs)を統合しました。これに伴い、コマンドライン仕様が大幅に変更になりました。変更したコマンド一覧を表 16.1、表 16.2 に示します。

表 16.1 リンケージコマンド変更一覧

No.	コマンド名	V6	V7	備考
1	start	start= セクション(アドレス)	start= セクション/アドレス	—
		短縮形 st	短縮形 star	
2	rom	rom=(rom セクション, ram セクション)	rom=rom セクション= ram セクション	—
3	define	define=外部名(定義値)	define=外部名=定義値	—
4	rename	rename= ed=変更前(変更後), er=変更前(変更後), un=変更前(変更後)	rename= (変更前=変更後), (変更前=変更後), —	オブジェクト形式変更によりユニットの概念廃止
		短縮形 re	短縮形 ren	
5	delete	delete= ed=ユニット.シンボル un=ユニット	delete=(シンボル) —	オブジェクト形式変更によりユニットの概念廃止
6	print / noprint	print noprint	list —	ファイル名省略可
7	mlist	mlist	list	—
8	information	information	message	—
9	directory	directory	HLNK_DIR(環境変数)	—
10	form	短縮形 f	短縮形 fo	—
11	output / nooutput	短縮形 o nooutput 指定可	短縮形 ou nooutput 指定不可	output のみ指定可
12	cpu	短縮形 c	短縮形 cp	直接範囲指定可
13	elf / sysrof / sysrofplus	elf / sysrof / sysrofplus	廃止	常に ELF
14	exclude / noexclude	exclude / noexclude	廃止	常に exclude
15	align_section	align_section	廃止	常に有効*
16	check_section	check_section	廃止	常に有効*
17	cpucheck	cpucheck	廃止	常に有効*
18	udf / noudf	udf / noudf	廃止	常に出力*

No.	コマンド名	V6	V7	備考
19	udfcheck	udfcheck	廃止	常に有効*
20	echo / noecho	echo / noecho	廃止	常に抑止
21	exchange	exchange	廃止	オブジェクト形式変更によりユニットの概念廃止
22	autopage	autopage	廃止	対象 cpu なし
23	abort	abort	廃止	会話形式廃止
24	list	list	廃止	V7 の list オプションとは別
25	library / nolibrary	nolibrary 指定可	nolibrary 指定不可	library のみ指定可
26	exit	省略不可	省略可	—
27	debug / nodebug	省略時：nodebug	省略時： 入力ファイルの debug 情報有無に依存	—

【注】*　change_message オプションで無効にすることができます。

表 16.2　ライブラリアンコマンド変更一覧

No.	コマンド名	V2	V7	備考
1	add	add	input	—
2	directory	directory	HLNK_DIR(環境変数)	—
3	slist	slist	list show	—
4	list	list　(s)	list show	—
5	delete	短縮形 d	短縮形 del	—
6	create	create (s | u)	output form=library(s | u)	—
7	output	output (s|u)	output form=library(s | u)	—
		短縮形 o	短縮形 ou	
8	replace	短縮形 r	短縮形 rep	—
9	abort	abort	廃止	会話形式廃止
10	exit	省略不可	省略可	—

16.1.5 提供内容

「SuperH RISC engine C/C++コンパイラパッケージ」の提供内容のうち、以下のファイルが変更になりました。

(1) 標準ライブラリファイル

関数インタフェースや最適化オプションを任意に指定できるようにするため、従来の標準ライブラリファイル提供から、標準ライブラリ構築ツール提供に変更しました。

(2) ヘッダファイル

標準ライブラリとして提供する初期設定用ルーチン_INITSCT 関数、_CALL_INIT 関数、_CALL_END 関数用のヘッダファイル_h_c_lib.h を追加しました。

固定小数点数の内部表現に関する各種制限値を定義したヘッダファイル fixed.h を追加しました。

16.1.6 リストファイル仕様

(1) 最適化リンケージエディタ

従来のリンケージマップリスト、ライブラリリストのフォーマットを一新しました。

16.2 追加・改善内容

16.2.1 共通の追加・改善(パッケージ Ver.6)

(1) 制限値の緩和

ソースプログラムやコマンドラインの制限を大幅に緩和しました。

- ファイル名の文字数：251 バイト → 無制限
- シンボル長：251 バイト → 無制限
- シンボル数：32767 個 → 無制限
- ソースプログラム行数：C/C++:32767 行、ASM:65535 行 → 無制限
- C プログラム文字列長：512 文字 → 32766 文字
- アセンブリプログラム行長：255 文字 → 8192 文字
- サブコマンドファイル行長：ASM:300 バイト、optlnk:512 バイト → 無制限
- 最適化リンケージエディタ rom オプションのパラメータ数:64 個 → 無制限

(2) フォルダ名、ファイル名のハイフン(-)

フォルダ名、ファイル名にハイフン(-)を指定できるようになりました。

(3) コピーライト表示抑止

logo/nologo オプション指定により、コピーライト表示有無を指定できるようになりました。

(4) エラーメッセージのプリフィックス

統合開発環境でのエラーヘルプ機能サポートに伴い、コンパイラ、最適化リンケージエディタのエラーメッセージの先頭にプリフィックスを付与しました。

16.2.2 コンパイラの追加・改善機能

(1) Ver.7 の主な追加・改善機能

(a) 外部変数アクセス最適化機能(map オプションサポート)

リンク時の変数、関数の割り付けアドレスに基づいた外部変数アクセス、関数分岐命令の最適化を行います。一度目のリンク時に最適化リンケージエディタより出力(map オプション指定)された外部シンボル割り付け情報ファイルを用いてリコンパイルすることにより、最適化を実施します。

(b) GBR 相対アクセスコード自動生成(gbr オプションサポート)

gbr=auto を指定した場合、コンパイラが GBR の設定および GBR 相対アクセスコードを自動生成します。関数呼び出し前後で GBR の値を保証します。ただし、GBR 関連の組み込み関数は使用できません。

(c) speed/size 選択オプションの強化

speed/size 選択オプション(shift、blockcopy、division、approxdiv オプション)を追加し、より細かな size/speed 調整を可能にしました。

(d) 組み込み向け機能の強化

- 組み込み関数の追加
  倍精度乗算、SWAP 命令、LDTLB 命令、NOP 命令

- #pragma の追加、変更
  #pragma entry　　エントリ関数指定、SP の設定
  #pragma stacksize　　スタックサイズの指定
  #pragma interrupt　　sp=<変数>+<定数>、sp=&<変数>+<定数>のサポート
-
- セクション演算子のサポート
  セクションのアドレス、サイズ参照を C 言語で記述できる機能をサポートしました。
-
- アドレスキャストのエラー緩和
  外部変数初期化時のアドレス初期化に対するキャスト式のエラーを緩和しました。

(e) ライブラリの改善

- リエントラントライブラリサポート
  ライブラリ構築ツールで reent オプションを指定した場合、リエントラントライブラリが生成されます。

- malloc 確保サイズ単位、入出力ファイル数の可変化
  C/C++ライブラリ関数の初期設定において、_sbrk_size で malloc 確保サイズを、_nfiles で入出力ファイル数を指定できるようになりました。これにより、RAM 容量を節約できます。指定を省略した場合、malloc 確保サイズは 1024、入出力ファイル数は 20 になります。

- 簡易 I/O のサポート
  ライブラリ構築ツールで nofloat オプションを指定した場合、浮動小数点変換をサポートしない、サイズの小さい I/O ルーチンを生成します。

(2) Ver.7.1 での主な追加・改善機能

(a) 最適化を制御するオプションの追加

以下のオプションを追加し、最適化レベルのより細かな調整を可能にしました。

- global_volatile
- opt_range
- del_vacant_loop
- max_unroll
- infinite_loop
- global_alloc
- struct_alloc
- const_var_propagate
- const_load
- schedule

(3) Ver.8 での主な追加・改善機能

(a) 新マイコンのサポート

SH-4A,SH4AL-DSP をサポートしました。

(b) 言語仕様拡張・変更

- DSP-C 言語をサポートしました。
- long long、unsigned long long 型をサポートしました。

(c) 組み込み向け機能強化

- DSP 向け組み込み関数の追加
  - 絶対値、MSB 検出、算術シフト、丸め演算、ビットパターンコピー
  - モジュロアドレッシング設定、モジュロアドレッシング解除、CS ビットの設定
  -
- SH-4A、SH4AL-DSP 向け組み込み関数の追加
  - 正弦・余弦の計算、平方根の逆数、命令キャッシュブロックの無効化、
  - 命令キャッシュブロックのプリフェッチ、データ操作の同期
- #pragma の追加、変更
  - #pragma ifunc　浮動小数点レジスタの退避/回復抑止
  - #pragma bit_order　ビットフィールド並び順指定
  - #pragma pack　構造体、共用体、クラスのアライメント数指定

(d) 列挙型サイズの自動選択(auto_enum オプションサポート)

列挙型が収まる最小型として列挙型を処理します。

(e) 構造体、共用体、クラスメンバのアライメント数を指定(pack オプションサポート)

構造体、共用体、クラスメンバのアライメント数を指定できます。

(f) ビットフィールド並び順指定(bit_order オプションサポート)

ビットフィールドのメンバの並び順を指定できます。

(g) エラーレベルの変更(change_message オプションサポート)

インフォメーション、ウォーニングレベルのメッセージは、個別にエラーレベルを変更できます。

(h) 制限値の緩和

switch 文の数を 2048 個に拡張しました。

(i) DSP ライブラリの固定小数点型サポート

DSP ライブラリを固定小数点サポートしました。

(4) Ver.9 での主な追加・改善機能

(a) 新マイコンのサポート

SH-2A、SH2A-FPU をサポートしました。
また、SH-2A、SH2A-FPU の TBR を活用する機能(tbr オプション、#pragma tbr)を追加しました。

(b) 言語仕様拡張・変更

- 以下の項目を ANSI 準拠しました。
  - ・配列のインデックス
    ```
    int iarray[10], i=3;
    i[iarray] = 0;   /* iarray[i] = 0;と同じになる  */
    ```
  - ・union のビットフィールド指定を可能にする
    ```
    union u {
        int a:3;
    };
    ```
  - ・定数演算
    ```
    static int i=1||2/0;   /*  ゼロ除算を Error から Warning に変更  */
    ```
  - ・ライブラリ、マクロ追加
    strtoul,FOPEN_MAX

- strict_ansi オプションを指定することにより、以下項目が ANSI 準拠になります。
  これにより、Ver.8 までの結果と異なる場合があります。
  - ・unsigned int と long 型演算
  - ・浮動小数点演算の結合則

- enable_register オプションを指定することにより、register 記憶クラスを指定した変数を優先的にレジスタに割り付けます。

(c) 組み込み向け機能強化

- **SH-2A、SH2A-FPU 向け組み込み関数の追加**

  飽和演算、TBR 設定・参照

- **C 言語で記述できない命令の組み込み関数サポート**

  T ビット参照・設定、連結レジスタの中央切り出し、キャリー付き加算、ボロー付き減算、符号反転、1 ビット除算、回転、シフト

(d) 制限値の緩和

以下の制限値を緩和しました。

- 繰り返し文(while 文、do 文、for 文)、選択文(if 文、switch 文)の組み合わせによるネストの深さ：32 レベル → 4096 レベル
- 1 関数内で指定可能な goto ラベルの数：511 個 → 2147483646 個
- switch 文のネストの深さ：16 レベル → 2048 レベル
- 1 つの switch 文内で指定可能な case ラベルの数：511 個 → 2147483646 個
- 関数定義、関数呼び出しでパラメータの個数：63 個 → 2147483646 個
- セクション名長：31 バイト → 8192 バイト
- 1 ファイルあたりの#pragma section で指定できるセクション数：64 個 → 2045 個

(e) メモリ空間配置指定の拡張

メモリ空間配置指定をより詳細に指定できます。

- abs16/abs20/abs28/abs32 オプション
- #pragma abs16/abs20/abs28/abs32

(f) 変数の絶対アドレス指定(#pragma address サポート)

外部変数のアドレスを絶対アドレス指定できます。

(g) 外部変数アクセス最適化機能拡張(smap オプションサポート)

コンパイル対象ファイル内で定義された外部変数について外部変数アクセス最適化を実施します。
map オプションのようなリコンパイルは必要ありません。

(h) 数学関数ライブラリ精度向上

数学関数ライブラリの演算精度が向上しました。
これにより、Ver.8 までの結果と異なる場合があります。

(5) Ver.9.01 での主な追加・改善機能

(a) デバッグ情報出力モードの追加 (optimize=debug_only オプションサポート)

optimize=debug_only オプションを指定することにより、ローカル変数の情報を常に参照できるようにしました。

(b) 割り込み仕様追加 (SH-3, SH3-DSP, SH-4, SH-4A, SH4AL-DSP)

以下の#pragma interrupt の割り込み仕様および組み込み関数を追加し、実行効率の良い割り込み関数を記述できるようにしました。

- 割り込み仕様

  #pragma interrupt sr_rts　レジスタバンク切り替え、RTS 命令リターン指定
  #pragma interrupt bank　割り込みハンドリング関数指定
  #pragma interrupt rts　RTS 命令リターン指定

- 組み込み関数

  sr_jsr()　多重割り込み制御

(c) 浮動小数点数-整数変換時の範囲チェック省略機能の追加 (simple_float_conv オプションサポート SH-2E, SH2A-FPU, SH-4, SH-4A)

simple_float_conv オプションを指定することにより、符号なし整数型と浮動小数点型の間の型変換に対して、変換対象の値の範囲のチェックを省略したコードを生成できるようにしました。

(d) 既存機能の仕様追加・変更

- division=cpu=inline | runtime オプションを、SH-2A, SH2A-FPU で指定できるようにしました。
- キャッシュブロック操作のための組み込み関数 ocbi(), ocbp(), ocbwb() を、SH-4 でも使用できるようにしました。
- #pragma inline を指定した関数を、inline オプションに関係なくインライン展開するように変更しました。
- subcommand オプションと listfile オプションを指定した場合に、subcommand オプションで指定したファイルの内容をコンパイルリストに出力するよう変更しました。これにより、ルネサス統合開発環境を用いてリストファイルを出力した際に、リストファイルにオプションが表示されるようになりました。

(e) 数学関数ライブラリ改善 (SH-1, SH-2, SH2-DSP, SH-2A, SH-3, SH3-DSP, SH4AL-DSP)

浮動小数点数用の数学関数 sinf, cosf, tanf, expf, logf, sqrtf, atanf について、オブジェクトサイズを削減し、演算速度および演算精度が向上しました。これにより、これらの数学関数の演算結果は Ver.9.00 と異なる場合があります。

(6) Ver.9.02 での主な追加・改善機能

(a) $G0,$G1 セクション変数の配置指定機能

stuff_gbr オプションにより、#pragma gbr_base または#pragma gbr_base1 で宣言した変数をサイズに応じたセクションに配置できるようにしました。

(b) 分岐先アドレスの 4 バイト整合機能

align4 オプションおよび#pragma align4 により、分岐命令の分岐先アドレスを 4 バイト整合にできるようにしました。

(c) inline 指定子付き関数のインライン展開抑止機能

cpp_noinline オプションにより、C++で inline 指定子付きの関数及びメンバ関数のインライン展開を抑止できるようにしました。

(7) Ver.9.03 での主な追加・改善機能

(a) const 修飾かつ volatile 修飾された変数の配置先選択機能

const_volatile オプションにより、const 修飾かつ volatile 修飾された初期値付き変数を、初期化データ領域と定数領域のどちらに割り付けるかを選択可能にしました。

(b) ビット操作組み込み関数の追加(SH-2A, SH2A-FPU)

以下の組み込み関数により、メモリ上のビット操作を必ず SH-2A コアのビット操作命令により行うようにしました。

- bset() ：メモリ上の指定ビットに 1 を設定
- bclr() ：メモリ上の指定ビットに 0 を設定
- bcopy() ：メモリ上のあるビットの値を別のビットにコピー
- bnotcopy() ：メモリ上のあるビットの値を別のビットに反転してコピー

(8) Ver.9.04 での主な追加・改善機能

(a) ポインタ指示先の型を考慮した最適化の追加

alias=ansi オプションにより、ANSI 規格に基づいてポインタ指示先の型を考慮した最適化を行うことが可能になりました。

(b) DIVS, DIVU 命令出力抑止機能(SH-2A, SH2A-FPU)

nouse_div_inst オプションにより、SH-2A コアの DIVS,DIVU 命令を出力を抑止できるようになりました。

(c) 浮動小数点演算式の演算順序変更最適化

float_order オプションにより、浮動小数点演算式の演算順序変更の最適化を積極的に行えるようになりました。

16.2.3 アセンブラの追加・改善機能

(1) Ver.7 での主な追加・改善機能

(a) 新マイコンのサポート

SH-2A、SH2A-FPU をサポートしました。

(b) 制限値の緩和

define オプション、.DEFINE 制御命令の制限値を 32 文字から無制限に拡張しました。

(c) .STACK 制御命令サポート

.STACK 制御命令をサポートしました。
これによりアセンブラで記述した関数のスタックサイズを CallWalker に反映することができます。

16.2.4 最適化リンケージエディタの追加・改善機能

(1) Ver.7 での主な追加・改善機能

(a) ワイルドカードのサポート

入力ファイルや start オプションのセクション名でワイルドカードを指定できます。

(b) サーチパス

環境変数(HLNK_DIR)により、複数の入力ファイル、ライブラリファイルのサーチパスを指定できます。

(c) ロードモジュール分割出力

アブソリュートロードモジュールファイルを分割出力できます。

(d) エラーレベルの変更

インフォメーション、ウォーニング、エラーレベルのメッセージは、個別にエラーレベルや出力有無を変更できます。

(e) バイナリ、インテル(拡張)HEX サポート

バイナリファイルを入出力できるようになりました。
また、インテル(拡張)HEX タイプの出力も選択できるようになりました。

(f) stack 使用量情報の出力

stack オプションにより、CallWalker 用情報ファイルを出力できます。

(g) デバッグ情報削除機能

strip オプションにより、ロードモジュールファイルやライブラリファイル内のデバッグ情報だけを削除できます。

(h) デバッグ情報圧縮機能

compress オプション指定により、デバッグ情報の圧縮が可能になりました。

(2) Ver.7.1 での主な追加・改善機能

(a) 外部シンボル割り付け情報ファイルの出力(map オプションサポート)

map オプションを指定した場合、コンパイラが外部変数アクセス最適化で使用する外部シンボル割り付け情報ファイルを生成します。

(3) Ver.8 での主な追加・改善機能

(a) 空きエリア出力指定

space オプション指定により、空きエリアへ指定値を埋め込むことができます。

(b) メモリ使用量指定

memory オプション指定により、内部のメモリ使用量を指定することができます。

(c) セクションアドレス重複時のエラーレベル変更

リンク時にセクションのアドレスが重なった場合のエラーレベルが Ver.7 では Fatal だったのを、Ver.8 では Error に変更しました。これにより、セクションアドレスが重複するような状況でも change_message オプション指定により、ユーザ責任において処理の継続が可能です。

(4) Ver.9 での主な追加・改善内容

(a) 新マイコンのサポート

マイコン種別が SH-2A、SH2A-FPU のオブジェクトファイルの入力をサポートしました。

(b) binary オプション入力セクションへのアライメント数指定

binary オプションに指定するセクションに対して、アライメント数の指定ができます。

(c) クロスリファレンス情報出力

show=xreference オプション指定により、クロスリファレンス情報がリンケージリスト内に出力されます。これにより、変数/関数がどこから参照されているのかを知ることができます。

(d) 参照されないシンボルの通知オプション

msg_unused オプション指定により、最適化を使用しない状況でも参照されないシンボルの存在を知ることができます。

(5) Ver.9.01 での主な追加・改善内容

(a) セクション単位の最適化抑止機能

新規オプション(section_forbid)を追加しました。
section_forbid オプションを使用することにより、モジュール間最適化を
セクション単位で抑止することができます。

(b) オーバーレイ機能の強化

start オプションに括弧"()"を用いた記法を導入しました。
旧バージョンよりも複雑なオーバーレイ配置を記述することができます。

(c) ライブラリ内の同名シンボル通知

リンク時に使用するライブラリファイルに、複数個の同名シンボルがある場合には
下記メッセージにて警告するように機能追加しました。
** L1320 (W) Duplicate Symbol "シンボル" in "ライブラリ(モジュール)"

(6) Ver.9.02 での主な追加・改善内容

(a) 物理空間配置時の重複検出機能

新規オプション(ps_check)を追加しました。
ps_check オプションを使用することにより、物理空間上でのオブジェクトの重なりを検出することができます。

(b) データレコードのバイト数指定機能

新規オプション(byte_count)を追加しました。
byte_count オプションを使用することにより、インテル HEX 形式ファイルのデータレコードの最大バイト数を変更することができます。

(c) 空きエリアへの乱数充填機能

space=random オプションを使用することにより、空きエリアへ乱数を埋め込むことが可能になりました。

(d) ライブラリ作成時のメモリ使用量削減機能

メモリ使用量削減機能(memory=low)が、ライブラリファイル作成時にも使用できるようになりました。

(7) Ver.9.03 の追加・新規機能

(a) 種別ごとのセクション合計サイズ出力機能

total_size オプションにより、標準出力へ下記種別ごとのセクション合計サイズが出力可能になりました。

- 実行可能セクション
- ROM 配置データセクション
- RAM 配置データセクション

(b) セクション合計サイズのリンケージリスト出力機能

total_size オプションで出力した種別ごとのセクションの合計サイズを、show=total_size オプションによりリンケージリストに出力可能になりました。

(8) Ver.9.04 の追加・新規機能

(a) CRC 計算結果埋め込み機能

crc オプションにより、指定した範囲の CRC(Cyclic Redundancy Check)演算を行い、計算結果を指示したアドレスのメモリに埋め込めが可能になりました。

(9) Ver.9.05 の追加・新規機能

(a) 指定領域内に割り付けられないセクションの分割機能

cpu=stride オプションにより、セクションがセクションの割り付けアドレスに対して、割り付けるメモリ範囲に収まらない場合に、次の同メモリ種別のセクションに配置、または、そのセクションを分割して配置することが可能になりました。

(b) 分割対象外セクション指定機能

contiguous_section オプションにより、cpu=stride オプションが有効なときに、分割せずに同メモリ種別の割り付け可能なアドレス領域に割り付けるセクションが指定可能になりました。

(c) リスト内容出力機能強化

show=all オプションにより、全てのリスト内容出力が可能になりました。

(d) エラー時のリンケージマップ出力

エラー終了時にリンケージマップを出力可能になりました。

(e) 指定入力ファイルの緩和

リロケータブルファイル出力時にバイナリファイルを入力ファイルに指定可能になりました。

(10) Ver.10.00 の追加・新規機能

(a) セクションサイズをアライメント数の倍数にする機能

padding オプションにより、セクションのサイズをアライメント数の倍数にすることが可能になりました。

(11) Ver.10.01 の追加・新規機能

(a) 境界調整機能

aligned_section オプションにより、指定したセクションの境界調整数をリンク単位で 16 バイトに変更することが可能になりました。

(b) バイナリデータのセクション属性指定機能

binary=<セクション属性>オプションにより、バイナリデータのセクション属性指定が可能になりました。

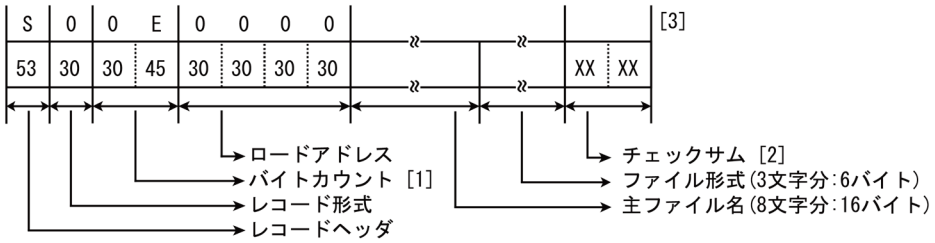
17.　付録

17.1　モトローラ S 形式、インテル HEX 形式ファイル

本節では、最適化リンケージエディタによって出力されるモトローラ S 形式ファイルおよび、インテル HEX 形式ファイルについて説明します。

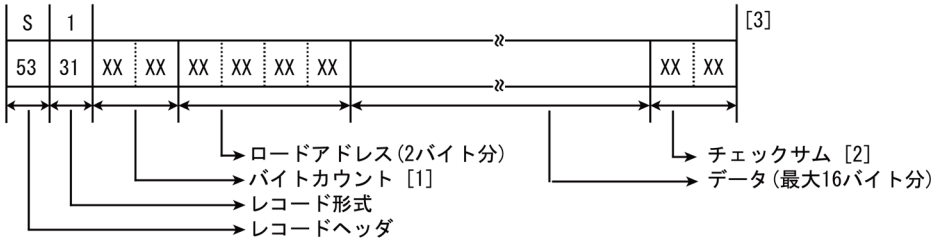
17.1.1　モトローラ S 形式ファイル

(a) ヘッダレコード(S0レコード)

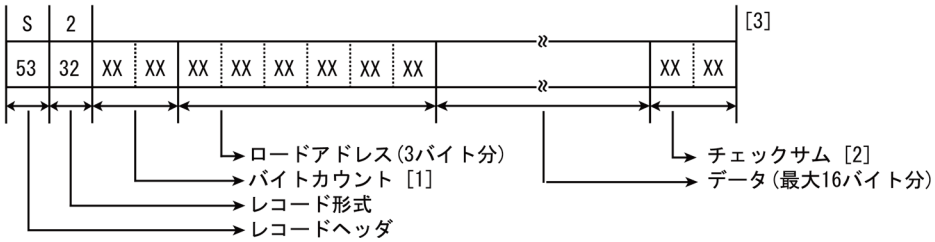


(b) データレコード(S1, S2, S3レコード)

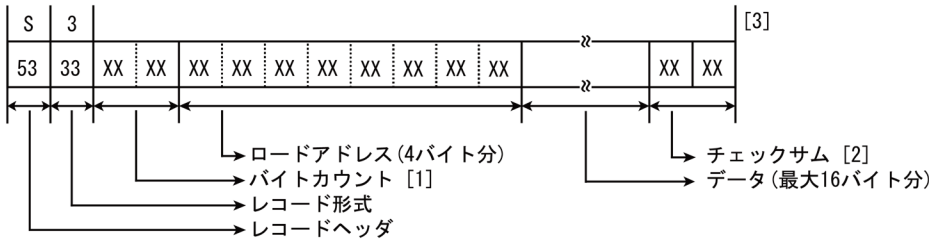
(i) ロードアドレスが0 ～ FFFFFの場合



(ii) ロードアドレスが10000 ～ FFFFFFの場合

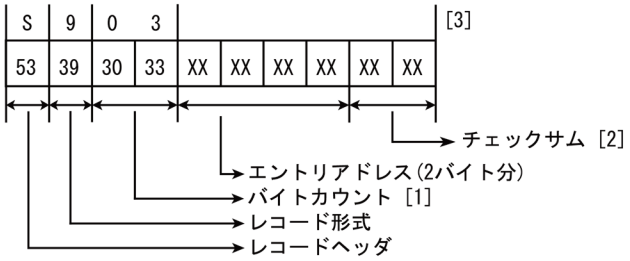


（iii）ロードアドレスが1000000 ～ FFFFFFFFの場合

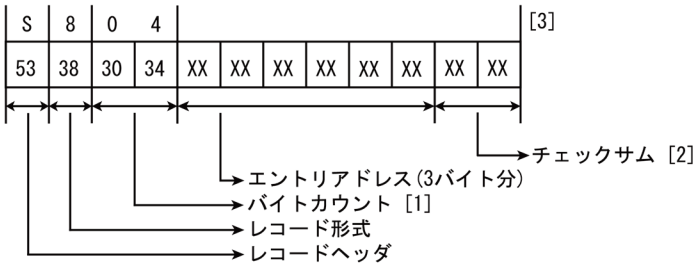


(c) エンドレコード(S9, S8, S7レコード)

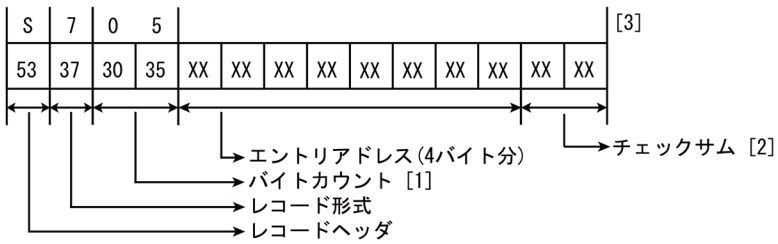
（i） エントリアドレスが0 ～ FFFFの場合



（ii）エントリアドレスが10000 ～ FFFFFFの場合



（iii）エントリアドレスが1000000 ～ FFFFFFFFの場合



【注】 [1] ロードアドレス(またはエントリアドレス)からチェックサムまでのバイト数
[2] バイトカウンタからチェックサムの前までのデータ値をバイト単位に加算した結果の1の補数
[3] チェックサムの直後に改行コードが付加される

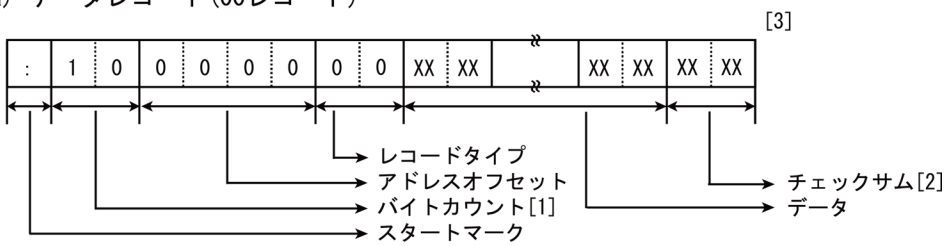
17.1.2　インテル HEX 形式ファイル

各データレコードの実行アドレスは以下のように求めます。

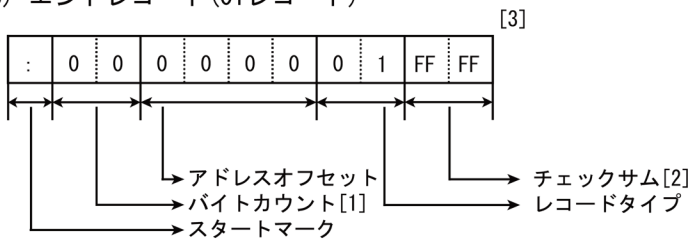
(1)　セグメントアドレスの場合
(セグメントベースアドレス << 4) + (データレコードのアドレスオフセット)

(2)　リニアアドレスの場合
(リニアベースアドレス << 16) + (データレコードのアドレスオフセット)

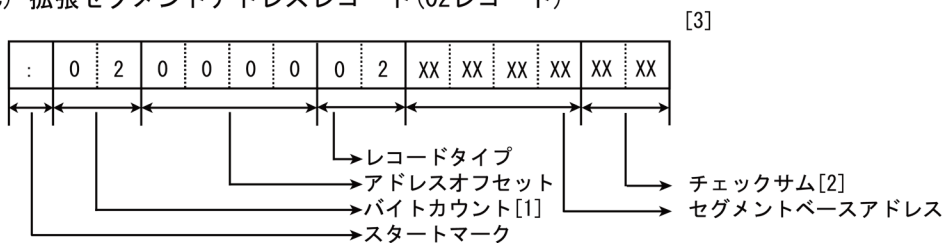
(a) データレコード(00レコード)



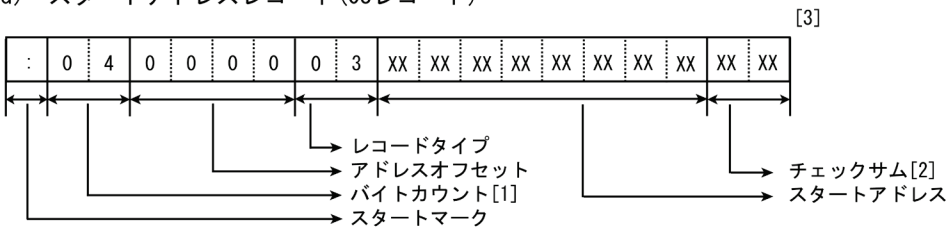
(b) エンドレコード(01レコード)



(c) 拡張セグメントアドレスレコード(02レコード)

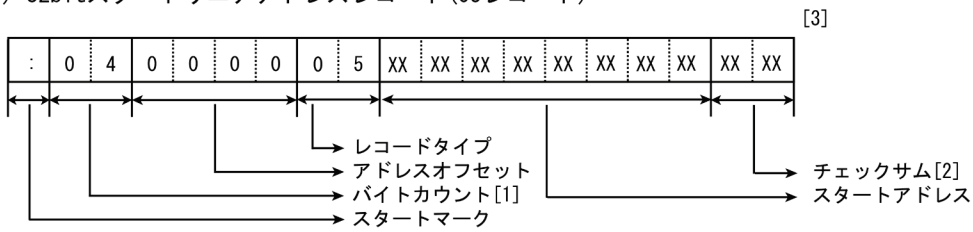


(d) スタートアドレスレコード(03レコード)



(e) 拡張リニアアドレスレコード(04レコード)

(f) 32bitスタートリニアアドレスレコード(05レコード)



【注】 [1] レコードタイプの次のデータから、チェックサムの前までのバイト数
[2] バイトカウンタからチェックサムの前までのデータを、16進数で加算した結果の2の補数(下位8bitが有効)
[3] チェックサムの直後に改行コードが付加される

17.2　ASCII コード一覧表

表 17.1　ASCII コード一覧表

下位 4 ビット	上位 4 ビット							
	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	＋	;	K	[	k	{
C	FF	FS	,	＜	L	＼	l	|
D	CR	GS	－	＝	M	]	m	}
E	SO	RS	．	＞	N	^	n	～
F	SI	US	／	?	O	_	o	DEL

ルネサスマイクロコンピュータ開発環境システム
ユーザーズマニュアル
SuperH™ RISC engine C/C++コンパイラ、アセンブラ、
最適化リンケージエディタ コンパイラパッケージ V.9.04
発行年月日　2010 年 7 月 9 日　第 1 版
2011 年 7 月 6 日　Rev.1.01
2022 年 3 月 1 日　Rev.1.02
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）

SuperH™ RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタ コンパイラパッケージ V.9.04 ユーザーズマニュアル

ルネサス エレクトロニクス株式会社

R20UT0704JJ0102