User's Manual

M16C シリーズ,R8C ファミリ用
C/C++コンパイラパッケージ V.6.00
C/C++コンパイラユーザーズマニュアル

お知らせ:
本資料 下記の箇所に訂正、追加がございます。
P.36 「(1) 標準入出力関数を使用しないときの設定」に記載のソース例
P.108 - Wlarge_to_small(-WLTS) 注意事項
P.160 「B.6.3 組み込み機器に関する拡張機能の使用方法」記載事項
P.167 「スペシャルページサブルーチン呼び出し関数宣言機能」記載事項
P.180 「表C.1 コンパイラの翻訳限界 (1/2)」記載事項

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00 2011.01.16

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

NC30 は、ルネサス 16 ビットマイクロコンピュータ M16C シリーズ, R8C ファミリ用の C/C++コンパイラです。NC30 は、C/C++言語で記述したプログラムを M16C シリーズ, R8C ファミリ用のアセンブリ言語ソースファイルに変換します。
またコンパイルオプションを指定することによって、アセンブル/リンクを実行してマイクロコンピュータに書き込み可能な 16 進数形式ファイルを生成することができます。なお、記載されている注意事項をよくお読みの上、ご使用ください。

(1) 用語の使い分けの説明

本ユーザーズマニュアルでは表現上、以下に示す用語を使い分けています。

用語	意味
NC30	本製品に含まれるコンパイラシステムを意味します。
nc30	コンパイルドライバ、又はその実行ファイルを意味します。
AS30	本システムに含まれるアセンブラシステムを意味します。
as30	リロケータブルマクロアセンブラ、又はその実行ファイルを意味します。

(2) 使用する記号の説明

本マニュアルでは、以下に示す記号を使用します。

記号	表示内容
A>	MS-Windows (TM)のプロンプトを示します。
<RET>	リターンキーの入力を示します。
< >	< >の中の部分は必須項目を示します。
[]	[ ]の中の部分は省略可能であることを示します。
△	スペース又はタブコードを示します（必須）。
▲	スペース又はタブコードを示します（省略可能）。
: (省略) :	ファイルの表示中の省略を示します。

なお、その他の記号を使用するときは、適宜説明します。

目次

第 1 章 NC30 の処理概要 1
1.1 NC30 の構成 1
1.2 NC30 の処理フロー 2
1.2.1 nc30 3
1.2.2 rcfrt 3
1.2.3 ccom30 3
1.2.4 aopt30 3
1.2.5 sbauto 3
1.2.6 as30 3
1.2.7 optlnk 3
1.2.8 lbg30 3
1.2.9 CallWalker 3
1.2.10 utl30 3
1.3 注意事項 4
1.3.1 コンパイラのバージョンアップ等についての注意事項 4
1.3.2 マイコンの機種依存部に関する注意事項 4
1.3.3 RAM データの参照についての注意事項 4
1.4 プログラム開発例 5
1.5 NC30 の出力ファイル 7
1.5.1 出力ファイルの概要 7
1.5.2 プリプロセッサ展開出力ファイル 8
1.5.3 アセンブラソースファイル 10
1.5.4 コンパイラが使用する一時ファイル 12
第 2 章 コンパイラの基本的な使い方 13
2.1 コンパイラの起動 13
2.1.1 コンパイルドライバのコマンドの入力書式 13
2.1.2 コマンドファイル 14
2.1.3 起動オプションに関する注意事項 15
2.1.4 nc30 の起動オプション 17
2.2 アセンブラスタートアッププログラムの準備 26
2.2.1 アセンブラスタートアッププログラムのサンプル 26
2.2.2 アセンブラスタートアッププログラムのカスタマイズ 35
2.2.3 メモリ配置のカスタマイズ 39
2.3 C 言語スタートアッププログラムの準備 46
2.3.1 生成ファイル 46
2.3.2 各生成ファイルの処理 47
2.3.3 C 言語スタートアップの生成方法 52
第 3 章 プログラミング 58
3.1 注意事項 58
3.1.1 コンパイラのバージョンアップ等についての注意事項 58
3.1.2 マイコンの機種依存部に関する注意事項 58
3.1.3 最適化について 59
3.1.4 register 変数の使用に関する注意事項 61
3.2 生成コードの向上のために 62
3.2.1 コード効率の良いプログラミング方法 62
3.2.2 スタートアップ処理を高速化する方法 63
3.3 アセンブリ言語プログラムとの結合方法 65
3.3.1 C 言語プログラムからアセンブラ関数の呼び出し方法 65
3.3.2 アセンブラ関数の記述方法 68
3.3.3 アセンブラ関数の記述に関する注意事項 71

3.4 その他 ... 73
3.4.1 NC シリーズコンパイラ間の移植に関する注意事項 ... 73
付録 A コマンドオプションリファレンス ... 74
A.1 コンパイルドライバの入力書式 ... 74
A.2 起動オプション ... 75
A.2.1 コンパイルドライバの制御に関するオプション ... 75
A.2.2 出力ファイル指定オプション ... 79
A.2.3 バージョン情報及びコマンドライン表示オプション ... 80
A.2.4 デバッグ用オプション ... 81
A.2.5 最適化オプション ... 82
A.2.6 生成コード変更オプション ... 94
A.2.7 ライブラリ指定オプション ... 106
A.2.8 警告オプション ... 107
A.2.9 アセンブル / リンクオプション ... 112
A.3 起動オプションに関する注意事項 ... 113
A.3.1 起動オプションの記述に関する注意事項 ... 113
A.3.2 オプションの優先順位 ... 113
付録 B 拡張機能リファレンス ... 114
B.1 near/far 修飾子 ... 116
B.1.1 near/far 修飾子の概要 ... 116
B.1.2 変数の宣言書式 ... 116
B.1.3 ポインタ型変数の宣言書式 ... 117
B.1.4 関数の宣言 ... 119
B.1.5 nc30 の起動オプションによる near/far の制御 ... 119
B.1.6 near から far への型変換機能 ... 120
B.1.7 far から near ポインタへの代入の検査機能 ... 120
B.1.8 near/far によるクラスの宣言 ... 121
B.1.9 テンプレート関数と near/far 宣言 ... 121
B.1.10 複数の宣言で near/far の確定を行う機能 ... 122
B.1.11 near/far 属性に関する注意事項 ... 123
B.2 asm 関数 ... 124
B.2.1 asm 関数の概要 ... 124
B.2.2 auto 変数の FB オフセット値の指定 ... 125
B.2.3 レジスタ変数のレジスタ名の指定 ... 128
B.2.4 extern 変数及び static 変数のシンボル名の指定 ... 129
B.2.5 記憶クラスに依存しない指定 ... 132
B.2.6 最適化の部分的な抑止方法 ... 133
B.2.7 asm 関数に関する注意事項 ... 133
B.3 日本語文字サポート ... 136
B.3.1 日本語文字の概要 ... 136
B.3.2 日本語文字を記述するための設定 ... 136
B.3.3 文字列中の日本語文字 ... 137
B.3.4 文字定数としての日本語文字 ... 138
B.4 関数のデフォルト引数宣言 ... 139
B.4.1 関数のデフォルト引数宣言の概要 ... 139
B.4.2 関数のデフォルト引数宣言の書式 ... 139
B.4.3 関数のデフォルト引数宣言の規定事項 ... 141
B.5 inline 関数宣言 ... 142
B.5.1 inline 記憶クラスの概要 ... 142
B.5.2 inline 記憶クラスの宣言書式 ... 142
B.5.3 inline 記憶クラスの規定事項 ... 143
B.6 #pragma 拡張機能 ... 146
B.6.1 #pragma 拡張機能の一覧 ... 146
B.6.2 メモリ配置に関する拡張機能 ... 152
B.6.3 組み込み機器に関する拡張機能の使用方法 ... 160
B.6.4 その他の拡張機能の使用方法 ... 168
B.7 アセンブラマクロ関数 ... 172

B.7.1 アセンブラマクロ関数の概要 172
B.7.2 アセンブラマクロ関数の記述例 172
B.7.3 アセンブラマクロ関数で記述可能な命令 173
付録C 翻訳限界 180
付録D C/C++言語実装仕様 182
D.1 言語仕様 182
D.2 データの内部表現 184
D.2.1 整数型 184
D.2.2 浮動小数点型 185
D.2.3 列挙型 186
D.2.4 ポインタ型 186
D.2.5 配列型 186
D.2.6 構造体型 187
D.2.7 共用体型 187
D.2.8 ビットフィールド型 188
D.2.9 クラス型 (C++言語) 189
D.2.10 参照型およびメンバへのポインタ型 192
D.3 符号拡張規則 193
D.4 関数呼び出し規則 194
D.4.1 戻り値に関する規則 194
D.4.2 引数渡しに関する規則 195
D.4.3 関数のアセンブリ言語シンボルへの変換規則 196
D.4.4 関数間のインターフェース 201
D.5 auto 変数の領域確保 207
D.6 レジスタの退避 208
D.7 プリプロセッサ仕様 208
D.7.1 インクルードファイルの読み込み方法 208
D.7.2 プリデファインドマクロ 208
D.7.3 #assert 208
D.8 C++言語コンパイルの注意事項 209
D.8.1 const 修飾した変数に関する注意事項 209
D.8.2 new/delete 演算子関数の注意事項 209
D.8.3 char 型に関する注意事項 210
D.8.4 複数の宣言で near/far を確定する記述に関する注意事項 210
D.8.5 メンバ配置属性の near/far に関する注意事項 211
D.8.6 インライン関数に関する注意事項 211
D.8.7 参照型を持つ変数の配置属性の near/far に関する注意事項 211
付録E C/C++ライブラリ 212
E.1 標準ヘッダファイルの機能と、その仕様の詳細 212
E.1.1 標準ヘッダファイルの概要 212
E.1.2 標準ヘッダファイルリファレンス 213
E.2 標準関数リファレンス 220
E.2.1 標準関数ライブラリの概要 220
E.2.2 標準関数ライブラリ機能別一覧 221
E.2.3 標準関数リファレンス 227
E.2.4 標準関数ライブラリの使用に関する注意事項 293
E.3 標準入出力関数ライブラリのカスタマイズ方法 294
E.3.1 入出力関数の構成 294
E.3.2 入出力関数の変更手順 295
E.4 EC++クラスライブラリ 302
付録F コンパイラのエラーメッセージ 365
F.1 エラー形式とエラーレベル 365
F.2 メッセージ一覧 365
付録G SBDATA 宣言&SPECIAL ページ関数宣言ユーティリティ(utl30) 467
G.1 utl30 の概要 467
G.1.1 utl30 の処理概要 467
G.2 utl30 の起動方法 469

G.2.1 入力書式 469
G.2.2 出力情報の切り替え 469
G.2.3 オプションリファレンス 470
G.3 制限事項 472
G.4 utl30 が処理対象とする変数および関数 472
G.4.1 SBDATA 宣言が処理対象とする変数 472
G.4.2 SPECIAL ページ関数宣言が処理対象とする関数 472
G.5 utl30 の使用例 473
G.5.1 SBDATA 宣言の場合 473
G.5.2 SPECIAL ページ関数宣言の場合 475
G.6 utl30 のエラーメッセージ 476
G.6.1 エラーメッセージ 476
付録 H ライブラリジェネレータ 478
H.1 コマンド記述形式 478
H.2 ご使用にあたっての注意点 478
H.3 ライブラリジェネレータオプション 479
H.4 ライブラリジェネレータに指定可能なコンパイラオプション 481
付録 I NC30 の C 言語の振舞い 482
I.1 ANSI 規格未定義の動作 482
I.2 処理系定義の動作 491
I.2.1 変換（Translation） 491
I.2.2 環境（Envi ronment） 491
I.2.3 識別子（Identifiers） 491
I.2.4 文字（Characters） 491
I.2.5 整数（Integers） 492
I.2.6 浮動小数点（Floating Point） 493
I.2.7 配列とポインタ（Arrays and Pointers） 493
I.2.8 レジスタ（Registers） 494
I.2.9 構造体、共用体、ビットフィールド（structures , unions , enumerators , and Bit - fields） 494
I.2.10 修飾子（qualifiers） 495
I.2.11 宣言子（Declarators） 495
I.2.12 文（Statements） 495
I.2.13 前処理指令（Preprocessing Directives） 495
I.2.14 ライブラリ関数（Library Functions） 496
I.3 文化圏固有動作 499
付録 J ELF フォーマットコンバータ ELFCONV 500
J.1 概要 500
J.2 起動方法 500
J.2.1 書式 500
J.2.2 オプション 501
J.3 使用上の注意事項 501
J.4 ELFCONV のメッセージ 503
付録 K バージョンアップ内容と移行方法 504
K.1 バージョンアップ内容 504
K.1.1 C++言語対応 504
K.1.2 統合開発環境(High-performance Embedded Workshop)プロジェクト変換 504
K.1.3 オブジェクトフォーマット変更 504
K.1.4 拡張子の変更 504
K.1.5 ライブラリアンの変更 505
K.1.6 リンケージエディタの変更 506
K.1.7 ロードモジュールコンバータの変更 507
K.1.8 スタック使用量算出ユーティリテイの変更 507
K.1.9 マップ情報を表示するツールの変更 507
K.1.10 ライブラリジェネレータの使用 507
K.1.11 表示メッセージの変更 508
K.1.12 コンパイルオプションの追加 508
K.1.13 アセンブラ指示命令の追加 508

K.1.14 アセンブラオプションの追加 508
K.1.15 外部分岐最適化の設定方法変更 508
K.1.16 廃止された機能 509
K.2 移行時の注意事項 510
K.2.1 -R8C オプションで生成したオブジェクトのリンクについて 510
K.2.2 -R8CE オプションで生成したオブジェクトのリンクについて 510
K.2.3 複数のライブラリファイル中に同名のシンボル名が存在する場合の仕様変更について 510
K.2.4 #pragma ASM/ENDASM に.section の記述の取り扱いについて 510
K.2.5 モジュール間最適化時のウォーニング表示について 510
K.2.6 標準ライブラリの sprintf、vsprintf, sscanf 関数だけを使用する場合 511
K.2.7 アセンブラスタートアップの変更 511
K.3 オブジェクトフォーマット変換後の実行コードの比較検証方法について 513
K.3.1 検証方法の考え方 513
K.3.2 検証方法の手順 514
K.3.3 注意事項 516
K.3.4 Ｓフォーマットファイル比較ツール玄１号について 516
付録 L 注意事項 518
L.1 機種依存部に関する注意事項 518
L.1.1 SFR 領域のアクセスに関する注意事項 518
L.1.2 M16C/62 4M 拡張モードに関して 518
L.1.3 オンチップデバッガ選択時の FirmRam_NE セクションと SB レジスタの値に関して 518
L.2 コンパイラ、アセンブラ、リンケージエディタ及びユーティリティに関する注意事項 520
L.2.1 -ffar_pointer(-fFP)について 520
L.2.2 標準入出力関数について 520
L.2.3 インラインアセンブル機能(#pragma ASM～#pragma ENDASM、asm 関数)に関する注意事項 520
L.2.4 メモリ管理関数 malloc()、calloc()および realloc()の注意事項 520
L.3 MISRA C ルール適合に関して 521
L.3.1 標準関数ライブラリ 521
L.3.2 ルール違反の要因 521
L.3.3 ルール違反となった検査番号 521
L.3.4 評価環境 521
L.3.5 統合開発環境(High-performance Embedded Workshop)が自動生成するソースコード 521
L.3.6 ルール違反の要因 521
L.3.7 ルール違反となった検査番号 522
L.3.8 評価環境 522
L.3.9 C スタートアップ中で使用する#pragma 拡張機能(Misra C ルール 99) 523

第1章 NC30 の処理概要

この章では、NC30 が行うコンパイル処理の概要と、NC30 を使用したプログラム開発の事例を説明します。

1.1 NC30 の構成

NC30 は、以下に示す 10 個の実行ファイルで構成されています。

(1)	nc30	コンパイルドライバ
(2)	rcfrt	プリプロセッサ
(3)	ccom30	コンパイラ本体
(4)	aopt30	アセンブラオプティマイザ
(5)	sbauto	SB レジスタ自動切換えユーティリティ
(6)	as30	アセンブラドライバ
(7)	optlnk	最適化リンケージエディタ
(8)	utl30	SBDATA 宣言& SPECIAL ページ関数宣言ユーティリティ
(9)	lbg30	ライブラリジェネレータ
(10)	CallWalker	スタック表示ツール

1.2 NC30 の処理フロー

NC30 の処理フローを【図 1.1】に示します。

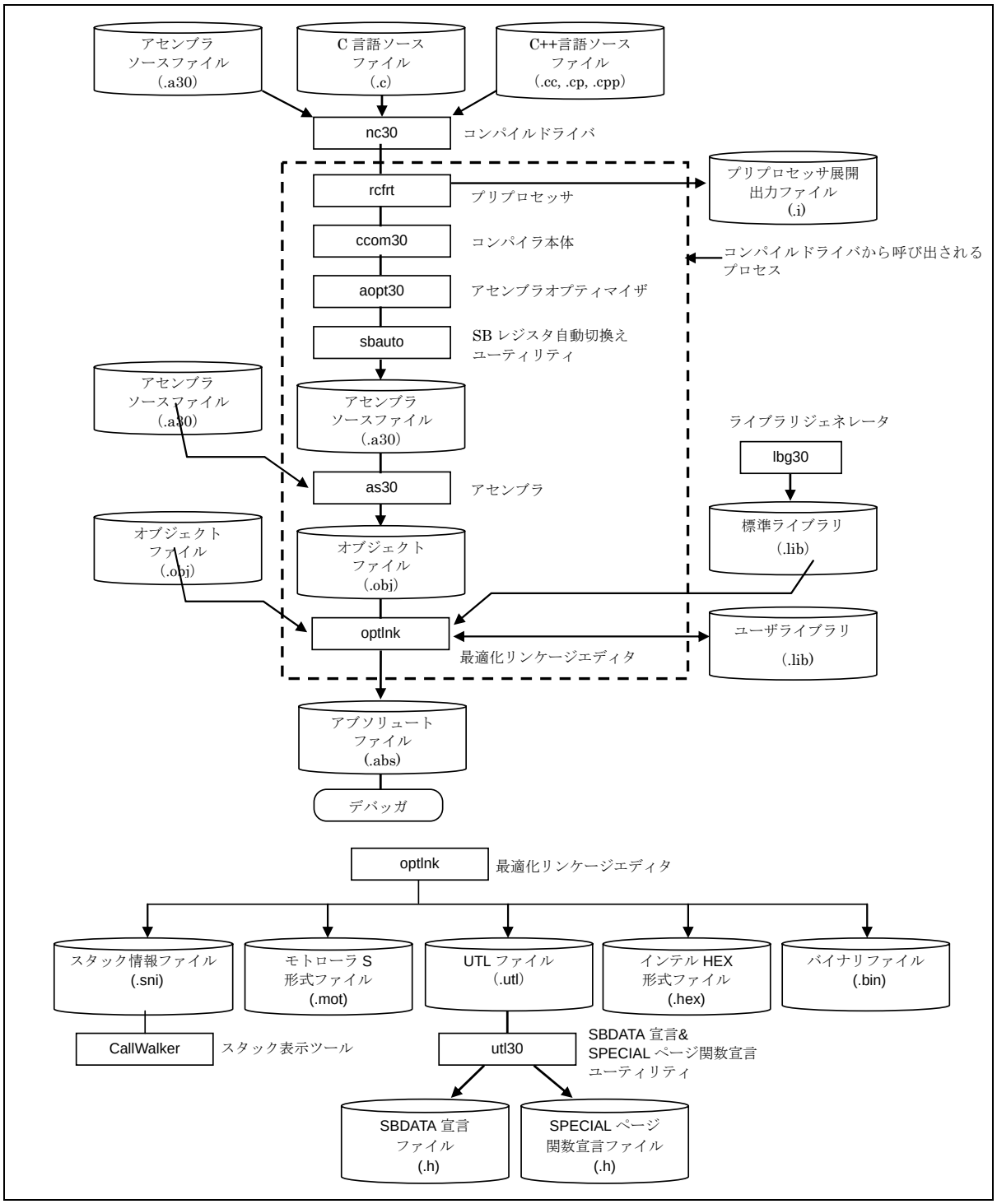


図1.1　NC30 の処理フロー

1.2.1 nc30

nc30 は、コンパイルドライバの実行ファイルです。
nc30 は、オプションの指定によりコンパイルからリンクまでの処理を連続して行うことができます。また、nc30 の起動オプション "-as30" に続けてリロケータブルマクロアセンブラ as30 のオプションを指定でき、"-lnkcmd=" に続けて最適化リンケージエディタ optlnk のコマンドファイルを指定することができます。

1.2.2 rcfrt

rcfrt は、プリプロセス処理を行います。
#で始まるプリプロセスコマンドに対して、ヘッダファイルの内容、マクロの展開、及び条件コンパイルの判定、等の処理を行います。

1.2.3 ccom30

ccom30 は、コンパイルを行います。
アセンブラソースプログラムを生成します。

1.2.4 aopt30

aopt30 は、アセンブラオプティマイザです。
ccom30 が出力したアセンブラソースに対して、最適化を行います。

1.2.5 sbauto

sbauto は、関数内における外部変数の参照回数を解析し、最適な SB 相対を出力します。

1.2.6 as30

アセンブラ(as30)は、アセンブラソースファイルを読み込み、アセンブルしてオブジェクトファイルを生成します。

1.2.7 optlnk

最適化リンケージエディタ(optlnk)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ファイルを入力し、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。

1.2.8 lbg30

標準ライブラリ構築ツール(lbg30)は、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ツールです。

1.2.9 CallWalker

スタック解析ツール(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)を読み込んでスタック使用量を表示します。
なお、Call Walker が表示するスタックサイズは、厳密な値ではありません。したがいまして、Call Walker が表示するスタックサイズは、スタック領域のサイズを検討する場合の参考値として扱ってください。
Call Walker が表示するスタックサイズを基にスタック領域のサイズを決定した際には、十分な評価を行ってください。

1.2.10 utl30

utl30 は、SBDATA 宣言 および SPECIAL ページ関数宣言を生成するユーティリティの実行ファイルです。
utl30 は、最適化リンケージエディタが生成した UTL ファイルを読み込み、SBDATA 宣言を行なったファイル(使用頻度の高いものから SB 領域に配置)および SPECIAL ページ関数宣言を行なったファイル(使用頻度の高いものから SPECIAL ページ領域に配置)を生成します。

1.3 注意事項

本マニュアルに記載の製品データ、図、表に示す技術的な内容、プログラムおよびアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサスエレクトロニクス株式会社は、適用可否に対する責任は負いません。

1.3.1 コンパイラのバージョンアップ等についての注意事項

本コンパイラが生成する機械語命令(アセンブリ言語)は、コンパイル時に指定する起動オプション、バージョンアップの内容等により変化します。したがって、起動オプションの変更又はコンパイラのバージョンアップを行った場合は再度アプリケーションプログラムの動作評価を必ず行ってください。

1.3.2 マイコンの機種依存部に関する注意事項

SFR 領域のレジスタへの書き込み、または読み出しには特定の命令を使用しなければならないことがあります。この特定の命令は機種ごとに異なりますので、詳しくは各マイコン機種のユーザーズマニュアルを参照してください。

本コンパイラは、SFR領域のレジスタへの書き込み、読み出しには使用できない命令を生成する場合があります。【図 1.2】のようなC言語記述でSFR領域にアクセスすると、SFR領域で使ってはいけない命令を生成するため、割り込み要求ビットの判定が正常に行われず意図しない動作を行う可能性があります。

C/C++言語で SFR 領域のレジスタへの書き込み、読み出しをする場合は、asm 関数を使用してプログラム中に直接命令を記述してください。この場合、コンパイラのバージョン、オプションの有無に関わらず、生成されたコードに問題が無いことを必ず確認してください。

```
#pragma   ADDRESS   TA0IC   0055h        /* M16C/62 タイマ A0 割込み制御レジスタ */

struct {
            char        ILVL : 3;
            char        IR : 1;          /* 割込み要求ビット */
            char        dmy : 4;
} TA0IC;

void        wait_until_IR_is_ON(void)
{
            while (TA0IC.IR == 0)        /* 1 になるまで待つ */
            {
                        ;
            }
            TA0IC.IR = 0;                /* 1 になったら 0 に戻す */
}
```

図1.2　SFR 領域に対する C ソースコード記述

1.3.3 RAM データの参照についての注意事項

割込み処理プログラムと被割込み処理プログラム間、リアルタイム OS 上のタスク間等で、同じ RAM データを参照し内容を変更する場合は、必ず volatile 指定や排他制御を行ってください。また、ビットフィールド構造体において、メンバ名が異なっている場合においても、同一の RAM 上に確保される場合は、同様に volatile 指定や排他制御を行ってください。

1.4 プログラム開発例

NC30 を使用したプログラム開発例の流れを【図 1.3】に示します。(項目の(1)～(4)は【図 1.3】の(1)～(4)に対応します)。

(1) C 言語もしくは C++言語ソースプログラム(AA.c)を nc30 でコンパイル、アセンブラ as30 でアセンブルし、オブジェクトファイル(AA.obj)を作成します。

(2) スタートアッププログラム ncrt0.a30 とセクション情報を記述したインクルードファイル sect30.inc 及び nc_define.inc を組み込むシステムに合わせて、セクションの配置/セクションサイズ/割り込みベクタテーブルの設定などを変更します。

(3) 変更したスタートアッププログラムをアセンブルします。この結果、オブジェクトファイル(ncrt0.obj)を作成します。

(4) 2 つのオブジェクトファイル、AA.obj と ncrt0.obj を nc30 から実行される最適化リンケージエディタ optlnk でリンクし、アブソリュートファイル(AA.abs)を作成します。

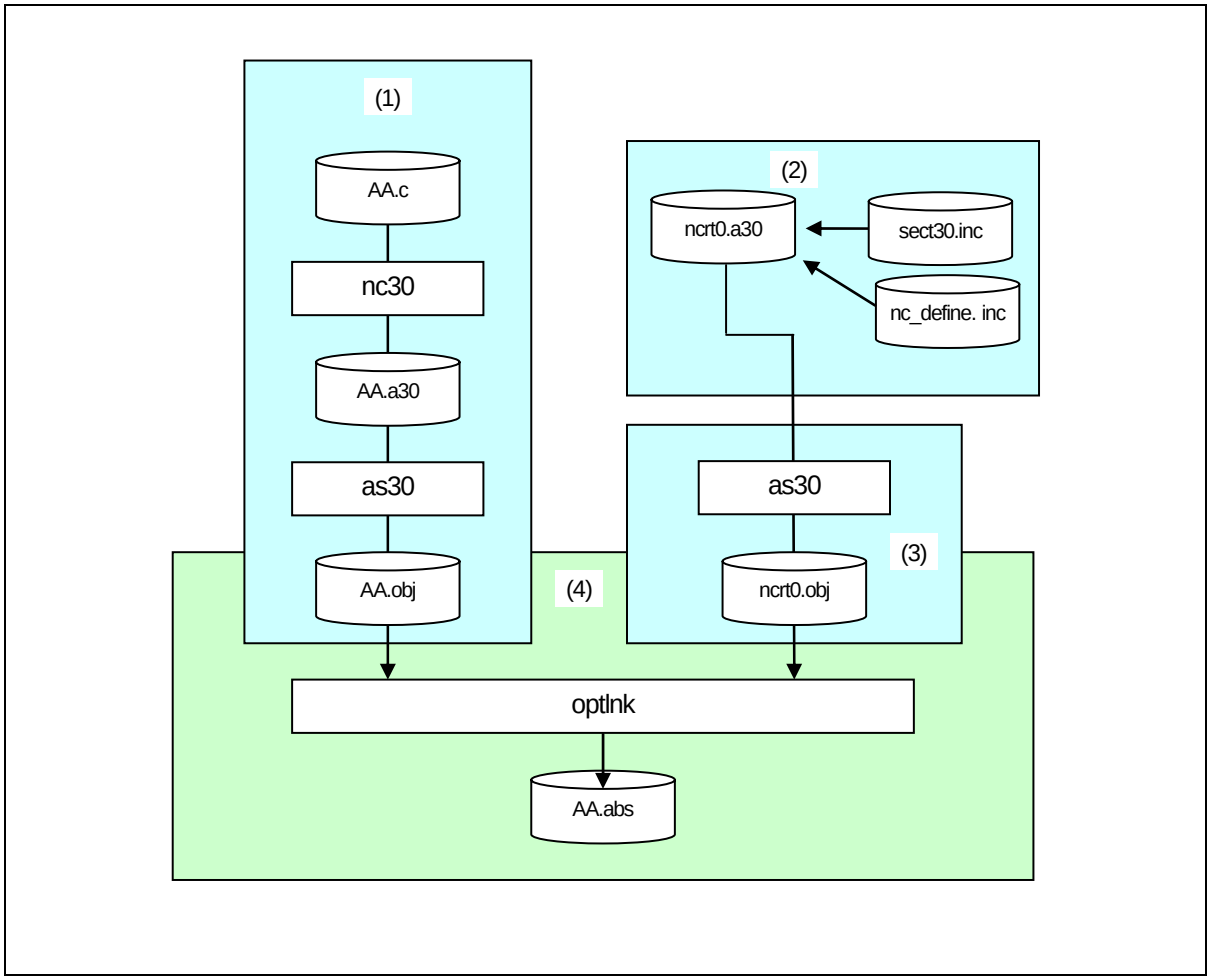


図1.3　プログラム開発フロー

【図 1.3】に示した一連の処理を記述した実行手順ファイル(makefile)の例を【図 1.4】に示します。

```
AA.abs: ncrt0.obj AA.obj
        nc30 -oAA ncrt0.obj AA.obj

ncrt0.obj: ncrt0.a30
        as30 ncrt0.a30

AA.obj: AA.c
        nc30 -c AA.c
```

図1.4　実行手順ファイル(makefile)の記述例

また、コンパイルドライバnc30 では、【図 1.4】と同様の処理をコマンドラインから【図 1.5】に示すように入力できます。

```
% nc30  -oAA  ncrt0.a30  AA.c<RET>

%：プロンプトを示します。
<RET>：リターンキーの入力を示します。
```

図1.5　nc30 コマンドの入力例

1.5 NC30 の出力ファイル

NC30 の出力ファイルを説明します。

1.5.1 出力ファイルの概要

コンパイルドライバnc30 は、起動オプションによって【図 1.6】に示すファイルを出力します。次項から【図 1.7】に示すC言語ソースファイルsample.cをコンパイル/アセンブル/リンクした結果、出力された個々のファイル例と表示内容を説明します。

C++プログラムとしてコンパイルして作成したオブジェクトファイル(.obj)を別ディレクトリに移動した後にリンクすると問題が発生する場合がありますので、オブジェクトファイルを移動せずにリンクを行ってください。

as30,optlnk については「アセンブラ/最適化リンケージエディタユーザーズマニュアル」を参照してください。

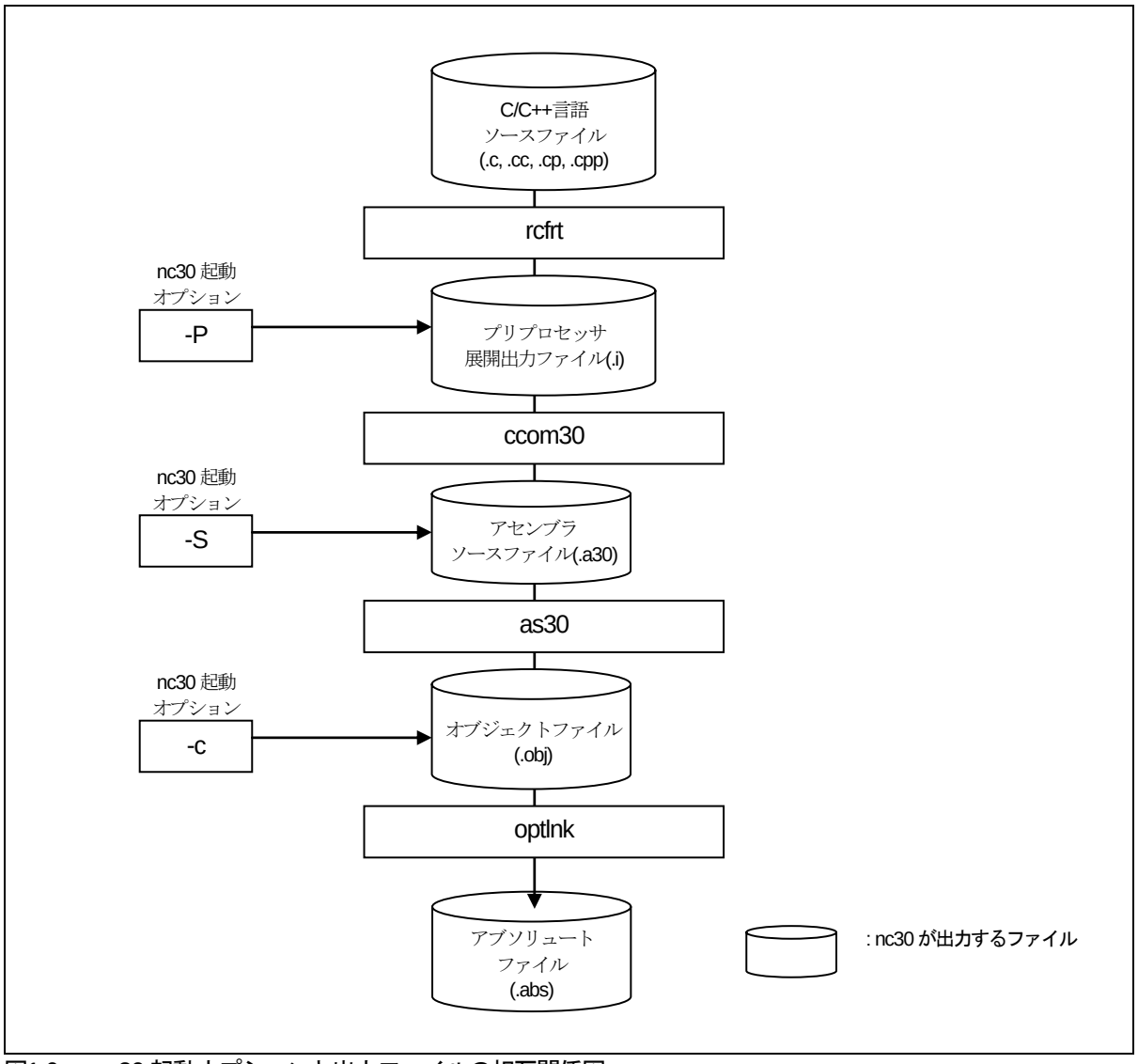


図1.6　nc30 起動オプションと出力ファイルの相互関係図

```
#include <stdio.h>
#define CLR   0
#define PRN   1

void        main(void)
{
            int         flag;

            flag = CLR;
#ifdef PRN
            printf( "flag = %d¥n", flag );
#endif
}
```

図1.7　C 言語ソースファイル例 (sample.c)

1.5.2　プリプロセッサ展開出力ファイル

プリプロセッサ rcfrt は、#で始まるプリプロセスコマンドに対して、ヘッダファイルの内容、マクロの展開、及び条件コンパイルの判定、等の処理を行います。

プリプロセッサ展開出力ファイルは、rcfrtがC/C++言語ソースファイルを処理した結果を格納しています。したがって、このファイルには、#pragma、#line 以外のプリプロセス行は出力されません。このファイルの内容を参照することによりコンパイラが処理を行うプログラムの内容を確認することができます。ファイルの拡張子は.i です。ファイルの出力例を【図 1.8】、【図 1.9】に示します。

```
typedef struct _iobuf {                                                    (1)
            char        _buff;
            int         _cnt;
            int         _flag;
            int         _mod;
            int         (*_func_in)(void);
            int         (*_func_out)(int);
} FILE;
                :
            (省略)
                :
typedef     long                    fpos_t;

typedef     unsigned int            size_t;

extern FILE _iob[];
```

図1.8　プリプロセッサ展開出力ファイル例 (1)

```
extern int    getc(FILE _far *);                                                    (1)
extern int    getchar(void);
extern int    putc(int, FILE _far *);
extern int    putchar(int);
extern int    feof(FILE _far *);
extern int    ferror(FILE _far *);
extern int    fgetc(FILE _far *);
extern char _far *fgets(char _far *, int, FILE _far *);
extern int    fputc(int, FILE _far *);
extern int    fputs(const char _far *, FILE _far *);
extern size_t          fread(void _far *, size_t, size_t, FILE _far *);
                :
              (省略)
                :
extern int    printf(const char _far *, ...);
extern int    fprintf(FILE _far *, const char _far *, ...);
extern int    sprintf(char _far *, const char _far *, ...);
                :
              (省略)
                :
extern int    init_dev(FILE _far *, int);
extern int    speed(int, int, int, int);
extern int    _sget(void);
extern int    _sput(int);
extern int    _pput(int);
extern const char _far * _print(int(*)(), const char _far *, int _far * _far *, int _far *);
```

```
void          main(void)                                                            (2)
{
              int          flag;
              flag = 0 ;                          ← (3)
              printf( "flag = %d¥n", flag );      ← (4)
}
```

図1.9　プリプロセッサ展開出力ファイル例 (2)

プリプロセッサ展開出力ファイルの内容を以下に説明します。項目番号の(1)～(4)は【図 1.8】、【図 1.9】中の(1)～(4)にそれぞれ対応しています。

(1)　#include で指定したヘッダファイル stdio.h の展開部を示します。
(2)　マクロを展開した結果の C 言語ソースプログラムを示します。
(3)　#define で指定された CLR を 0 として展開していることを示します。
(4)　#define で指定された PRN が定義されているためコンパイル条件が有効となり、printf 関数が出力されていることを示します。

1.5.3　アセンブラソースファイル

このファイルは、コンパイラ本体 ccom30 がプリプロセッサ展開出力ファイルを AS30 で処理可能なアセンブラソースに変換したファイルです。ここで出力されるファイルは、拡張子.a30 で示されるアセンブラソースファイルです。ファイルの出力例を【図 1.10】、【図 1.11】に示します。

```
                .__LANG     'C','X.XX.XX.XXX','REV.X'

;##       C Compiler                    OUTPUT
;## ccom30 Version X.XX.XX.XXX
;## Copyright (C) XXXX (XXXX - XXXX) Renesas Electronics Corporation
;## and Renesas Solutions Corp. All rights reserved.
;## Compile Start Time XXX XXX XX XX:XX:XX XXXX

;## COMMAND_LINE: ccom30    -finfo -gnone -dS -o sample.a30 sample.i

;## Normal Optimize              OFF                                    (1)
;## ROM size Optimize            OFF
;## Speed Optimize               OFF
;## Default ROM is               far
;## Default RAM is               near

                .FB         0
                :
                (省略)
                :
;## #           FUNCTION main
;## #           FRAME    AUTO      (      flag)   size   2,     offset -2
;## #           ARG Size(0)        Auto Size(2)          Context Size(5)

                .SECTION  program,CODE,align
                .__inspect   'U', 2, "program", "program", 0
                .__file      'sample.c'
                .__inspect   'F', 's', "main", "_main", 'G', 7
                .__block     1h,1h
                .__line      6
;## # C_SRC :    {
                .glb         _main
_main:
                enter        #02H
                .__line      9
;## # C_SRC :                 flag = CLR;
                mov.w        #0000H,-2[FB]          ;    flag
                .__line      11
```

図1.10　アセンブラソースファイル例 (1/2) (sample.a30)

```
;## # C_SRC :                    printf( "flag = %d¥n", flag );          ← (2)
                push.w      -2[FB]      ;   flag
                ._inspect   'S', 'p', 2
                push.w      #___T0>>16
                push.w      #(___T0&0FFFFH)
                ._inspect   'S', 'p', 4
                ._inspect   'S', 'c', "printf", "_printf", 'G', 0, 11
                ._block     2h,2h
                jsr         _printf
                ._eblock    2h,3h
                ._inspect   'S', 'p', -6
                add.b       #06H,SP
                ._line      13
;## # C_SRC :   }
                exitd
E1:
                .align
                ._eblock    1h,4h
                :
                （省略）
                :
                .glb        _printf
                :
                （省略）
                :
                .SECTION    rom_FO,ROMDATA
                ._inspect   'U', 4, "rom_FO", "rom_FO", 0
___T0:
                .byte       66H         ;   'f'
                .byte       6cH         ;   'l'
                .byte       61H         ;   'a'
                .byte       67H         ;   'g'
                .byte       20H         ;   ' '
                .byte       3dH         ;   '='
                .byte       20H         ;   ' '
                .byte       25H         ;   '%'
                .byte       64H         ;   'd'
                .byte       0aH
                .byte       00H
                :
                （省略）
                :
                .END

;## Compile End Time XXX XXX XX XX:XX:XX XXXX
```

図1.11　アセンブラソースファイル例 (2/2) (sample.a30)

アセンブラソースファイルの内容を以下に説明します。項目番号の(1)～(2)は【図 1.10】中の(1)～(2)に対応しています。

(1) 最適化オプションの状態と ROM 及び RAM に対する near/far 属性の初期設定の情報を示しています。

(2) ソースファイルの内容がコメントで表示されます。

1.5.4 コンパイラが使用する一時ファイル

コンパイラは内部で一時ファイルを使用します。そのため、以下のファイルがあれば上書きされたり、削除されますので注意してください。

- ソースファイル名＋拡張子.fnm
- ソースファイル名＋拡張子.db1
- ソースファイル名＋拡張子.db2
- ソースファイル名＋拡張子.dbs

第2章 コンパイラの基本的な使い方

この章では、コンパイルの起動方法と起動オプションの機能を説明します。起動オプションの説明では、コンパイルドライバから起動できるアセンブラと最適化リンケージエディタの起動オプションを併せて記載しています。

2.1 コンパイラの起動

2.1.1 コンパイルドライバのコマンドの入力書式

コンパイルドライバは、コンパイラの各モジュールとアセンブラ及び最適化リンケージエディタを起動します。このコンパイルドライバを起動するためには、以下の情報(入力パラメータ)が必要となります。

(1) C/C++言語ソースファイル
(2) アセンブラソースファイル
(3) オブジェクトファイル
(4) 起動オプション(必要に応じて記述する項目)

これらの項目をコマンド行に入力します。

項目(1)、(2)、(3)、のいずれか一つは入力が必要です。【図 2.1】に入力書式を、【図 2.2】に入力例を示します。入力例では、

(1) スタートアッププログラム ncrt0.a30 をアセンブル
(2) C/C++言語ソースプログラム sample.c をコンパイルの後アセンブル
(3) オブジェクトファイル ncrt0.obj と sample.obj をリンク

を行い、アブソリュートファイル sample.abs を作成するときの記述例を示します。

起動オプションには、

- アブソリュートファイル名 sample.abs の指定..... -o オプション
- アセンブル時のリストファイル(拡張子.lst)の出力指定..... -as30 オプション
- リンク時のリストファイル(拡張子.map)の出力指定..... -lnkcmd オプション

を行っています。

```
% nc30△[起動オプション]△<[アセンブラソースファイル名]△
                          [オブジェクトファイル名]△[C/C++言語ソースファイル名]>

% : プロンプトを示します。
<> : 必須項目を示します。
[] : 必要に応じて記述する項目を示します。
△ : スペースを示します。
```

図2.1 コンパイルドライバコマンドの入力書式

```
% nc30 -osample -as30 "-l" -lnkcmd=command.txt ncrt0.a30 sample.c <RET>

<RET>：リターンキーの入力を示します。
```

図2.2　コンパイルドライバコマンドの入力例

入力ファイル（アセンブラソースファイル名、オブジェクトファイル名、C/C++言語ソースファイル名）のパス指定を除いた部分が同じファイルは、入力ファイルとして同時に指定できません。

2.1.2　コマンドファイル

コンパイルドライバは、複数のコマンドオプションを記述したファイル(コマンドファイル)を読み込んでコンパイル処理を行うことができます。

コマンドファイルを使用することにより、Microsoft Windows のコマンド行文字数制限を回避することができます。

a.　コマンドファイルの入力書式

```
% nc30△[起動オプション]△<@ファイル名>△[起動オプション]

% : プロンプトを示します。
< > : 必須項目を示します。
[] : 必要に応じて記述する項目を示します。
△ : スペースを示します。
```

図2.3　コマンドファイルの入力書式

```
% nc30　@test.cmd　<RET>

<RET>：リターンキーの入力を示します。
```

図2.4　コマンドファイルの入力例

コマンドファイルの記述は以下のようになります。

```
ncrt0.a30
sample1.c sample2.obj
-osample
-lnkcmd=command.txt
```

図2.5　コマンドファイルの記述例

b. コマンドファイルの記述規定

コマンドファイルの記述には以下の規定があります。

- 一度に指定できるコマンドファイルは、1 ファイルのみです。同時に複数のコマンドファイルを指定することはできません。
- コマンドファイル内にコマンドファイルを指定できません。
- コマンドファイル内には、コマンド行を複数行にわたって記述できます。
- コマンドファイル内の改行は、空白文字に置き換えられます。
- コマンドファイルの一行に記述可能な文字数は、2048 文字以下です。2048 文字を越えた場合はエラーとなります。

c. コマンドファイル使用時の注意事項

コマンドファイル名にディレクトリパスを指定できます。指定したディレクトリパスにファイルが存在しない場合はエラーとなります。

同時に 2 ファイル以上のコマンドファイルは指定できません。複数ファイルを指定した場合は、"Too many command files. "のエラーメッセージを表示し終了します。

2.1.3 起動オプションに関する注意事項

a. 起動オプションの記述に関する注意事項

コンパイルドライバの起動時オプションは、アルファベットの大文字と小文字を区別します。誤って入力した場合、誤って入力した場合は警告を出して、そのオプションは指定されていないものとして処理を続行します。

b. コンパイルドライバの制御に関するオプションの優先順位

コンパイルドライバの制御に関するオプションには、以下の優先順位があります。

```
-E      -P      -S      -c
← 高      優先順位      低 →
```

例えば、

- "-c"：オブジェクトファイル(拡張子.obj)を作成して処理を終える
- "-S"：アセンブラソースファイル(拡張子.a30)を作成して処理を終える

を同時に指定した場合は"-S"オプションが優先されます。この場合、コンパイルドライバは、アセンブラ以後の処理を行いません。アセンブラソースファイルのみが生成されます。

オブジェクトファイルを作成し、かつ、アセンブラソースファイルも同時に作成したい場合は、"-dsource (短縮形 -dS) を使用してください。

c. コンパイルドライバから optlnk を呼び出す場合の注意事項

コンパイルドライバは optlnk を呼び出す際に、次のオプションを自動的に付加し起動します。これらのオプションを指定すると重複の警告が出て無視されるので注意してください。

-nologo, -library=nc30lib.lib, -output=<ファイル名>, -total_size

また、コンパイルドライバは optlnk を呼び出す際に、ユーザー指定のオプションに応じて、次のオプションを自動的に、または変更し付加し起動します。ユーザーがこれらのオプションを指定すると重複の警告が出て無視されるので注意してください。

-fno_lib オプションは標準ライブラリのリンク指定を抑止できます。

なお、optlnk を直接起動する場合には標準ライブラリ等を適切にリンクする必要があります。

ユーザ指定オプション	コンパイルドライバが optlnk に渡すオプション
-dir	-output=<ディレクトリ名>¥<ファイル名>
-o	-output=<ファイル名>
-OS_MAX	-optimize=branch
-OR_MAX	-optimize=branch
-g	-debug -list -show=all
-fsizet_16	-library="%LIB30%¥nc30s16.lib"
-fptrdifft_16	-library="%LIB30%¥nc30s16.lib"
-R8C	-library="%LIB30%¥r8clib.lib"
-R8C -fsizet_16	-library="%LIB30%¥r8cs16.lib"
-R8C -fptrdifft_16	-library="%LIB30%¥r8cs16.lib"
-R8CE	-library="%LIB30%¥r8celib.lib"
-R8CE -fsizet_16	-library="%LIB30%¥r8ces16.lib"
-R8CE -fptrdifft_16	-library="%LIB30%¥r8ces16.lib"
-finfo	-debug -list -show=all
-fSB_auto	-debug -list -show=all
-Wstop_at_link	-change_message=error
-Wno_used_function	-message -msg_unused

2.1.4 nc30 の起動オプション

a. コンパイルドライバの制御に関するオプション

【表 2.1】【表 2.2】にコンパイルドライバの制御に関する起動オプションを示します。各オプションの注意事項等詳細は、付録Aを参照ください。

表2.1 コンパイルドライバ制御オプション (1/2)

オプション	機能
-c	オブジェクトファイル(拡張子.obj)を作成し、処理を終了します。[1]
- D *識別子名*	識別子を定義します。#define と同じ機能です。
-dsource (短縮形 -dS)	アセンブラソースファイル(拡張子".a30")を生成します。 本オプションで生成されたアセンブラソースファイルをアセンブルしないでください。
-dsource_in_list (短縮形 -dSL)	アセンブラリストファイル(.lst)を生成します。
-E	プリプロセスコマンドのみを処理し結果を標準出力に出力します。
-I *ディレクトリ名*	プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。
-P	プリプロセスコマンドのみを処理し、ファイル（拡張子".i"）を作成します。
-S	アセンブラソースファイル(拡張子.a30)を作成し、処理を終了します。 なお、テンプレート関数を static 関数として.a30 ファイル中に出力します。 本オプションで生成されたアセンブラソースファイルをアセンブルすると、C/C++言語レベルのデバッグ情報が失われますので、注意してください。
-silent	起動時のコピーライトメッセージを出力しません。
-U *プリデファインドマクロ名*	指定したプリデファインドマクロを未定義にします。
-lang={c|cpp|ecpp}	c を指定すると C(C89)言語ソースファイルとしてコンパイルします。 cpp を指定すると C++言語ソースファイルとしてコンパイルします。 ecpp を指定すると EC++言語ソースファイルとしてコンパイルします。 -exception と-rtti は ecpp と同時に選択できません。 本オプション省略時は、拡張子が.cpp,.cc,.cp の時には C++言語ソースファイルとしてコンパイルします。それ以外の場合は C(C89)言語ソースファイルとしてコンパイルします。ただし、拡張子が.a30 の場合は本オプションの指定にかかわらずアセンブラソースファイルとして扱います。

[1] 起動オプション-c、-E、-P、及び-Sを指定しない場合、nc30 はoptlnkまで制御を行い、アブソリュートファイル(拡張子.abs)まで作成します。

表2.2　コンパイルドライバ制御オプション (2/2)

オプション	機能
-preinclude=ファイル名[･･･]	指定したファイルをインクルードします。ファイルが複数ある場合にはコンマで区切って指定することができます。その場合には左に指定したものから順に検索を行います。本オプションを複数指定した場合には全てのファイルが取り込み対象となります。 なお、検索順は#include "ファイル名"と同じ仕様になります。
-exception	例外処理機能を有効にします。 C 言語(C89)としてコンパイルする場合は本オプションは警告が出て無効となります。-lang=ecpp と同時に選択できません。 本オプションを指定した場合、本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用して下さい。
-noexception	例外処理機能を無効にします。 -noexception がデフォルトになります。
-rtti=on	C++言語実行時型情報機能(dynamic_cast,typeid)を有効にします。C 言語(C89)としてコンパイルする場合は本オプションは警告が出て無効となります。 -lang=ecpp と同時に選択できません。 本オプションを指定した場合、本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用して下さい。
-rtti=off	C++言語実行時型情報機能(dynamic_cast,typeid)を無効にします。 -rtti=off がデフォルトになります。 -lang=ecpp と同時に選択できません。

b. 出力ファイル指定オプション

【表 2.3】に出力するアブソリュートファイルの名称を指定する起動オプションを示します。

表2.3　出力ファイル指定オプション

オプション	機能
-dir *ディレクトリ名*	コンパイラが生成するファイルの出力先ディレクトリを指定できます。
-o *ファイル名*	optlnk が生成するファイルの名称を指定します。また、ディレクトリ名を含んだパス名も指定できます。 ファイルの拡張子は必ず省略してください。

c. バージョン及びコマンドライン情報表示オプション

【表 2.4】に使用するクロスツールのバージョン及びコマンドラインを表示する起動オプションを示します。

表2.4　バージョン情報及びコマンドライン表示オプション

オプション	機能
-v	実行中のコマンドプログラム名及びコマンドラインを表示します。
-V	コンパイラの各プログラムの起動時メッセージを表示し、処理を終了します(コンパイル処理は行いません)。

d. デバッグ用オプション

【表 2.5】にC/C++言語レベルデバッグ情報を出力するデバッグの起動オプションを示します。

表2.5　デバッグ用オプション

オプション	機能
-g	デバッグ情報をオブジェクトファイルに出力します。
-genter	関数呼び出し時に必ず enter 命令を出力します。 デバッガのスタックトレース機能を使用するときは、必ずこのオプションを指定してください。
-gno_reg	レジスタ変数に関するデバッグ情報の出力を抑止します。

e. 最適化オプション

【表 2.6】にプログラムの実行速度を最大、及びROM容量を最小にする最適化を行う起動オプションを示します。

表2.6　最適化オプション

オプション	短縮形	機能
-O[1～5]	なし	レベル毎に速度及び ROM 容量ともに効果がある最適化を行います。
-OR	なし	ROM 容量を重視した最適化を行います。
-OS	なし	速度を重視した最適化を行います。
-OR_MAX	-ORM	ROM 容量を優先する最大限の最適化を行います。
-OS_MAX	-OSM	速度を優先する最大限の最適化を行います。
-Ocompare_byte_to_word	-OCBTW	連続した領域のバイト単位の比較をワード単位で行います。
-Oconst	-OC	const 修飾子で宣言した外部変数の参照を定数に置き換える最適化を行います。
-Oforward_function_to_inline	-OFFTI	全てのインライン関数に対して、インライン展開を行います。
-Oloop_unroll[=ループ回数]	-OLU	ループ文を回さずに、ループ回数分コードを展開します。"ループ回数"は省略可能、省略時は最大 5 回のループ文が対象となります。
-Ono_asmopt	-ONA	アセンブラオプティマイザ"aopt30" による最適化を抑止します。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。 本最適化は、Cプログラムとしてコンパイルする場合のみ有効です。
-Ono_logical_or_combine	-ONLOC	論理 OR をまとめる最適化を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。 これにより ROM 容量を削減することができます。 ただし、使用するスタック量が多くなる可能性があります。
-Ostack_frame_align	-OSFA	スタックフレームの偶数アライメントを行います。
-Ostatic_to_inline	-OSTI	static 宣言された関数を、inline 宣言扱いにします。
-O5OA	なし	最適化オプション"-O5"選択時におけるビット操作命令(BTSTC、 BTSTS)を使用したコード生成を抑止します。
-goptimize	なし	モジュール間最適化用付加情報を出力します。

f. 生成コード変更オプション

【表 2.7】に本コンパイラが生成するアセンブリ言語を制御する起動オプションを示します。

表2.7　生成コード変更オプション（1/3）

オプション	短縮形	機能
-fansi	なし	予約語と演算方法を ANSI 準拠にします。 C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、asm と inline はキーワードになり、char 型データを int 型に拡張し演算します。
-fchar_enumerator	-fCE	enumerator(列挙子)の型を int 型ではなく unsigned char 型で扱います。
-fconst_not_ROM	-fCNR	const で指定した型を ROM データとして扱いません。
-fdouble_32	-fD32	double 型を float 型として処理します。 本オプションを使用する場合、C++言語プログラムで float 型と double 型の多重定義はできません。 本オプションを付けると-Wnon_prototype が同時に有効になります。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fextend_to_int	-fETI	char型データをint型に拡張し演算します(ANSI規格で定められた拡張を行います)。[2] 本オプションは C(C89)言語の場合に有効です。 C++言語の時は警告となり無視されます。 C++プログラムとしてコンパイルする場合、必ず char 型データを int 型に拡張し演算をします。
-ffar_RAM	-fFRAM	RAM データのデフォルト属性を far にします。
-finfo	なし	インスペクタ情報を出力します。
-fbit	-fB	near 領域に配置した外部変数全てに対して 1 ビット命令が使用できると仮定してコード生成を行います。
-fno_carry	-fNC	far ポインタ間接でのデータアクセス時のキャリーフラグの加算を抑止します。
-fauto_128	-fA1	使用するスタックフレームを最大 128 バイトに制限します。
-ffar_pointer	-fFP	ポインタ型のデフォルト属性を far 属性とします。near 修飾子で定義されたポインタ変数は本オプションに関わらず near 属性になります。
-fnear_ROM	-fNROM	ROM データのデフォルト属性を near にします。
-fno_align	-fNA	関数の先頭アドレスのアライメントを行いません。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべて odd 属性のセクションに配置します。
-fno_switch_table	-fNST	switch 文に対し、比較を行ってから分岐するコードを生成します。
-fnot_address_volatile	-fNAV	#pragma ADDRESS で指定した変数を、volatile で指定された変数として扱いません。
-fnot_reserve_asm	-fNRA	asm を予約語にしません("_asm"のみ有効になります)。 C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、asm はキーワードになります。

[2] ANSI規格ではchar型データ又はsigned char型データを評価する時に必ずint型に拡張します。
これは、「c1 = c2 * 2 / c3;」にような char 型の演算を行う場合に、演算の途中で char 型をオーバーフローし、結果が予期せぬ値になるのを防ぐためです。

表2.8　生成コード変更オプション（2/3）

オプション	短縮形	機能
-fnot_reserve_far_and_near	-fNRFAN	far、near を予約語にしません(_far、_near のみ有効になります)。
-fnot_reserve_inline	-fNRI	inline を予約語にしません(_inline のみ予約語となります)。C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、inline はキーワードになります。
-fsmall_array	-fSA	総サイズが不明な far 型の配列を参照する場合、その総サイズが 64K バイト以内であると仮定し、添字の計算を 16 ビットで行います。
-fswitch_other_section	-fSOS	switch 文に対するジャンプテーブルをプログラムセクションとは別セクションに出力します。
-fchange_bank_always	-fCBA	毎回バンク切り替えを行います。
-fauto_over_255	-fAO2	１つの関数内で確保可能なスタックフレームのサイズを、最大 64K バイトに変更します。
-fsizet_16	-fS16	型定義 size_t を unsigned long 型から unsinged int 型に変更します。 本オプションを指定した場合は、標準ライブラリ nc30s16.lib 、 r8cs16.lib(-R8C 指定時)もしくは r8ces16.lib(-R8CE 指定時)を指定してください。または本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用してください。
-fptrdifft_16	-fP16	型定義 ptrdiff_t を signed long 型から singed int 型に変更します。 本オプションを指定した場合は、標準ライブラリ nc30s16.lib 、 r8cs16.lib(-R8C 指定時)もしくは r8ces16.lib(-R8CE 指定時)を指定してください。または本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用してください。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。
-fuse_MUL	-fUM	乗算に対する生成コードを変更します。
-R8C	なし	R8C ファミリ MCU に対応したコードを生成します。 本オプションを指定すると、キーワード far と_far が無視されます。本オプションを指定した場合は、標準ライブラリ r8clib.lib もしくは r8cs16.lib(-fsizet_16 もしくは -fptrdifft_16 指定時)を指定してください。または本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用してください。 リンクする全てのプログラムに本オプションを付けてください。
-R8CE	なし	R8C ファミリ MCU(ROM64K 以上)に対応したコードを生成します。 本オプションを指定した場合は、標準ライブラリ r8celib.lib もしくは r8ces16.lib(-fsizet_16 もしくは -fptrdifft_16 指定時)を指定して下さい。または、本オプションをつけてライブラリジェネレータで生成した標準ライブラリを使用して下さい。 リンクする全てのプログラムに本オプションを付けてください。

表2.9　生成コード変更オプション（3/3）

オプション	短縮形	機能
-fSB_auto	-fSBA	全ての関数に対して SB 相対アドレッシングの自動生成を実施します。 関数単位で外部変数の参照回数を解析し、最適な SB 相対アドレッシングを生成します。 (1) SB 相対の基点となったシンボルのアドレスを SB レジスタへ格納します。 (2) 関数の出入り口で SB レジスタの退避/復帰のコードを生成します。 (3) 外部変数のみ有効です。 (4) -OR, -OS, -OR_MAX(-ORM), -OS_MAX(-OSM)と併用できません。 (5) 本オプションを使用したオブジェクトファイルと以下の機能を使用したオブジェクトファイルをリンクしたプログラムの動作を保証しません。 ● #pragma SBDATA ● コンパイラオプション-fauto_over_255(-fAO2)

g. ライブラリ指定オプション

【表 2.10】にライブラリファイルを指定する起動オプションを示します。

表2.10　ライブラリ指定オプション

オプション	機能
-l ライブラリファイル名	リンク時に使用するライブラリを指定します。
-fno_lib	コンパイルドライバから optlnk を呼び出す時に-library オプションを自動的に付加する機能を抑止します。

h. 警告オプション

【表 2.11】に本コンパイラの言語仕様に関する記述の間違いに対して警告(ウォーニングメッセージ)を出力する起動オプションを示します。

表2.11 ウォーニングオプション

オプション	短縮形	機能
-Wall	なし	検出可能なウォーニング("-Wlarge_to_small"、" -Wno_used_argument "で出力されるウォーニングを除く)をすべて表示します。
-Wccom_max_warnings =*ウォーニング回数*	-WCMW	ccom30 の出力するウォーニングの回数の上限を指定できます。 本機能は、C プログラムとしてコンパイルする場合のみ有効です。
-Wlarge_to_small	-WLTS	大きいサイズから小さいサイズへの暗黙の代入に対して、ウォーニングを出力します。
-Wnesting_comment	-WNC	コメント中に"/*"を記述した場合にウォーニングを出します。
-Wno_stop	-WNS	エラーが発生してもコンパイル作業を停止しません。 C++プログラムとしてコンパイルする場合、本オプションの選択の有無にかかわらず、100 個のエラーが出力されると停止します。
-Wno_used_argument	-WNUA	引数を持つ関数を定義した場合に、使用していない引数に対してウォーニングを出力します。
-Wno_used_function	-WNUF	リンク時に未使用のグローバル関数を表示します。 リンカオプション-msg_unused と-message を使用している場合、本オプションは必要ありません。
-Wno_used_static_function	-WNUSF	コード生成が不要な static 関数名を表示します。
-Wno_warning_stdlib	-WNWS	"-Wnon_prototype"指定時や"-Wall"指定時に本オプションを指定すると、「関数原型宣言されていない標準ライブラリに対するウォーニング」を抑止します。 C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、関数原型宣言がなければメッセージを出します。
-Wnon_prototype	-WNP	関数原型宣言されていない関数を使用した場合、ウォーニングを出します。 C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、関数原型宣言がなければメッセージを出します。
-Wstop_at_link	-WSAL	リンク時にウォーニングが発生した場合、アブソリュートファイルの生成を抑止します。
-Wstop_at_warning	-WSAW	ウォーニング発生時にコンパイル処理を停止します。
-Wundefined_macro	-WUM	#if の中で未定義のマクロを使用した場合にウォーニングします。
-Wuninitialize_variable	-WUV	初期化されていない auto 変数に対してウォーニングを出力します。
-Wunknown_pragma	-WUP	サポートしていない #pragma を使用した場合、ウォーニングを出します。

i. アセンブル / リンクオプション

【表 2.12】にas30 及びoptlnkのオプションを指定する起動オプションを示します。

表2.12 アセンブル/リンクオプション

オプション	機能
-as30△< オプション>	アセンブルコマンド as30 のオプションを指定します。2 個以上のオプションを渡す場合は、"(ダブルクオーテーション)で囲んでください。
-lnkcmd=<ファイル名>	optlnk にコマンドファイルを指定します。

2.2 アセンブラスタートアッププログラムの準備

C/C++言語で記述したプログラムを ROM 化するために、マイコンの初期設定、セクションの配置、割り込みベクタテーブル、等を設定するアセンブリ言語もしくは C 言語で記述したスタートアッププログラムが必要です。スタートアッププログラムはお客様がご使用のマイコン機種、システムに合わせて変更していただく必要があります。

ここでは、アセンブラ言語で記述したアセンブラスタートアッププログラムの説明と、そのカスタマイズの方法について説明します。なお、統合開発環境(High-performance Embedded Workshop)を起動し、新規プロジェクトの作成でプロジェクトタイプに Application を選択すると、アセンブラスタートアッププログラムの雛型がフォルダに自動生成されますので、それを元に変更してください。

2.2.1 アセンブラスタートアッププログラムのサンプル

アセンブラスタートアッププログラムは、以下の 3 つのファイルで構成しています。

- ncrt0.a30
  リセット直後に実行されるプログラムを記述します。
- nc_define.inc
  スタックサイズ、ヒープサイズ、可変ベクタ及びスペシャルページベクタのアドレスを定義します。
- sect30.inc
  このファイルは、ncrt0.a30 からインクルードされ、セクションの配置(メモリの配置)を定義します。

ncrt0.a30 のソースプログラムリストを次に示します。

```
;/***********************************************************************
;*
;* Device        : M16C/60,30,20,10
;*
;* File Name     : ncrt0.a30
;*
;* Abstract      : Startup Program for M16C/60,30,20,10.
;*
;* History       : X.XX  (xxxx-xx-xx)
;*
;* Copyright(c) 2010 Renesas Electronics Corporation
;*                       And Renesas Solutions Corp.,All Rights Reserved.
;*
;***********************************************************************/

;------------------------------------------------------------------
; include files
;------------------------------------------------------------------
                        .list                   OFF
                        .include    nc_define.inc
                        .include    sect30.inc                          ← (1)
                        .list                   ON

;------------------------------------------------------------------
; BankSelect definition for 4M mode
;------------------------------------------------------------------
;                       .glb                    __BankSelect
;__BankSelect           .equ        0BH
```

図2.6　アセンブラスタートアッププログラム ncrt0.a30 リスト抜粋 (1/5)

```
;======================================================================
; Interrupt section start
;----------------------------------------------------------------------
                        .glb            start
                        .section interrupt,CODE,ALIGN
                        .insf           start,G,0
start:                                                          ← (2)
;----------------------------------------------------------------------
; after reset,this program will start
;----------------------------------------------------------------------
                        ldc             #((topof istack)+(sizeof istack)),isp ;set istack pointer
                        mov.b           #02h,0ah
                        mov.b           #00h,04h                ← (3)
                        mov.b           #00h,0ah
.if __STACKSIZE__ != 0
                        ldc             #0080h,flg              ← (4)
                        ldc             #((topof stack)+(sizeof stack)),sp ;set stack pointer
.else
                        ldc             #0000h,flg
.endif
                        ldc             #__SB__,sb              ;set sb register

                        ; If the destination is INTBL or INTBH,
                        ; make sure that bytes are transferred in succession.
                        ldc             #((topof vector)>>16)&0FFFFh,INTBH  ← (5)
                        ldc             #(topof vector)&0FFFFh,INTBL

;======================================================================
; NEAR area initialize.
;----------------------------------------------------------------------
; bss zero clear                                                ← (6)
;----------------------------------------------------------------------
                        N_BZERO         (topof bss_SE),bss_SE
                        N_BZERO         (topof bss_SO),bss_SO
                        N_BZERO         (topof bss_NE),bss_NE
                        N_BZERO         (topof bss_NO),bss_NO
```

図2.7　アセンブラスタートアッププログラム ncrt0.a30 リスト抜粋 (2/5)

```
;------------------------------------------------------------------
; initialize data section                                    ← (7)
;------------------------------------------------------------------
                        N_BCOPY         (topof data_SEI),(topof data_SE),data_SE
                        N_BCOPY         (topof data_SOI),(topof data_SO),data_SO
                        N_BCOPY         (topof data_NEI),(topof data_NE),data_NE
                        N_BCOPY         (topof data_NOI),(topof data_NO),data_NO

;======================================================================
; FAR area initialize.
;------------------------------------------------------------------
; bss zero clear                                             ← (8)
;------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                        BZERO           (topof bss_FE),bss_FE
                        BZERO           (topof bss_FO),bss_FO.
.endif

;------------------------------------------------------------------
; initialize data section                                    ← (9)
;------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                        BCOPY           (topof data_FEI),(topof data_FE),data_FE
                        BCOPY           (topof data_FOI),(topof data_FO),data_FO
.if __STACKSIZE__ != 0
                        ldc             #((topof stack)+(sizeof stack)),sp.
.else
                        ldc             #((topof istack)+(sizeof istack)),isp.
.endif
                        .stk            -40.
.endif

;======================================================================
; heap area initialize                                       ← (10)
;------------------------------------------------------------------
.if __HEAPSIZE__ != 0
                        .glb            __mnext
                        .glb            __msize
                        mov.w           #((topof heap_NE)&0FFFFH),__mnext
                        mov.w           #((topof heap_NE)>>16),__mnext+2
                        mov.w           #(__HEAPSIZE__&0FFFFH),__msize
                        mov.w           #(__HEAPSIZE__>>16),__msize+2.
.endif
```

図2.8　アセンブラスタートアッププログラム ncrt0.a30 リスト抜粋 (3/5)

```
;======================================================================
; Initialize standard I/O                                        ← (11)
;----------------------------------------------------------------------
.if __STANDARD_IO__ == 1
                .glb            __init
                .call           __init,G
                jsr.a           __init
.endif

;======================================================================
; Call main() function                                           ← (12)
;----------------------------------------------------------------------
                ldc             #0h,fb                  ; for debuger

; Remove the comment when you use global class object            ← (13)
; Sections C$INIT will be generated
;               .glb            __CALL_INIT
;               .call           __CALL_INIT,G
;               jsr.a           __CALL_INIT

                .glb            _main
                .call           _main,G
                jsr.a           _main

;======================================================================
; exit() function                                                ← (14)
;----------------------------------------------------------------------
                .glb            _exit
                .glb            $exit
                .glb            __exit_loop
_exit:
$exit:

; Remove the comment when you use global class object            ← (15)
; Sections C$INIT will be generated
;               .glb            __CALL_END
;               .call           __CALL_END,G
;               jsr.a           __CALL_END

__exit_loop:                                            ; End program
                jmp             __exit_loop
                .einsf
```

図2.9　アセンブラスタートアッププログラム ncrt0.a30 リスト抜粋 (4/5)

```
;======================================================================
; dummy interrupt function
;----------------------------------------------------------------------
                .glb    dummy_int                       ← (16)
dummy_int:
                reit

                .end
```

(1) sect30.inc をインクルードします。
(2) リセット直後はラベル start からスタートします。
(3) プロセッサ動作モードを設定します。
(4) 割り込み許可レベル及び各種フラグの設定を行います。
(5) 割り込みベクタテーブルの開始アドレスを定義します。
(6) near 領域の bss セクションのゼロクリアを行います。
(7) near 領域の data セクションの初期値を RAM 領域に転送します。
(8) far 領域の bss セクションのゼロクリア処理を行います。
(9) far 領域の data セクションの初期値を RAM 領域に転送します。
(10) heap 領域の初期化を行います。メモリ管理関数を使用しない場合にはコメントにします。
(11) 標準入出力の初期化を行う init 関数を呼び出します。入出力関数を使用しない場合にはコメントにします。
(12) グローバルクラスオブジェクト（C++言語）を使うと、C$INIT セクションが生成されることがあります。その際にはこのコメントを外した上で再リンクしてください。
(13) main 関数の呼び出しを行います。
※ main 関数呼び出し時には、割り込みは禁止になっています。割り込みを使用する際には"FSET"命令により割り込みを許可してください。
(14) exit 関数部です。
(15) グローバルクラスオブジェクト（C++言語）を使うと、C$INIT セクションが生成されることがあります。その際にはこのコメントを外した上で再リンクしてください。
(16) ダミーの割り込み処理関数です。

図2.10　アセンブラスタートアッププログラム ncrt0.a30 リスト抜粋 (5/5)

nc_define.inc のソースプログラムリストを次に示します。

```
__FAR_RAM_FLG__   .equ    0       ; FAR RAM flag definition          ← (1)
__STANDARD_IO__   .equ    0       ; STANDARD I/O flag definition     ← (2)
__HEAPSIZE__      .equ    0300H   ; HEEP SIZE definition             ← (3)
__STACKSIZE__     .equ    0300H   ; STACK SIZE definition            ← (4)
__ISTACKSIZE__    .equ    0300H   ; INTERRUPT STACK SIZE definition  ← (5)
```

(1) アドレス 0x10000 以降にある RAM を使用する場合非ゼロにします。
(2) 標準入出力を使用する場合非ゼロにします。
(3) ヒープサイズを定義します。
(4) ユーザースタックサイズを定義します。
(5) 割り込みスタックサイズを定義します。

図2.11　アセンブラスタートアッププログラム nc_define.inc リスト抜粋

sect30.inc のソースプログラムリストを次に示します。

```
;/*********************************************************************
;*
;* Device        : M16C/60,30,20,10
;*
;* File Name     : sect30.inc
;*
;* Abstract      : Section definition for M16C/60,30,20,10
;*
;* History       : x.xx  (xxxx-xx-xx)
;*
;* Copyright(c) 2010 Renesas Electronics Corporation
;*                   And Renesas Solutions Corp.,All Rights Reserved.
;*
;*********************************************************************/

;=====================================================================
;
;               Definition of section
;
;---------------------------------------------------------------------
; Near RAM data area
;---------------------------------------------------------------------
; SBDATA area
                        .section    data_SE,DATA,ALIGN
                        .section    bss_SE,DATA,ALIGN
                        .section    data_SO,DATA
                        .section    bss_SO,DATA

; SBDATA area definition                                              ← (1)
; Sets the top address (__SB__) of the SBDATA area
; (it is accessing area to used the SBrelative addressing mode).
                        .glb                __SB__
__SB__      .equ                400H

; near RAM area
                        .section    data_NE,DATA,ALIGN
                        .section    bss_NE,DATA,ALIGN
                        .section    data_NO,DATA
                        .section    bss_NO,DATA

;---------------------------------------------------------------------
; Stack area
;---------------------------------------------------------------------
.if __STACKSIZE__ != 0
                        .section    stack,DATA,ALIGN
                        .blkb               __STACKSIZE__               ← (2)
.endif
```

図2.12　アセンブラスタートアッププログラム sect30.inc リスト抜粋 (1/5)

```
                    .section    istack,DATA,ALIGN
                    .blkb               __ISTACKSIZE__                  ← (3)

;------------------------------------------------------------------------
; heap section
;------------------------------------------------------------------------
.if __HEAPSIZE__ != 0
                    .section    heap_NE,DATA,ALIGN
                    .blkb               __HEAPSIZE__                    ← (4)
.endif

;------------------------------------------------------------------------
; Far RAM data area
;------------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                    .section    data_FE,DATA,ALIGN
                    .section    bss_FE,DATA,ALIGN
                    .section    data_FO,DATA
                    .section    bss_FO,DATA
.endif

;------------------------------------------------------------------------
; Initial data of 'data' section
;------------------------------------------------------------------------
                    .section    data_SEI,ROMDATA
                    .section    data_SOI,ROMDATA
                    .section    data_NEI,ROMDATA
                    .section    data_NOI,ROMDATA
.if __FAR_RAM_FLG__ != 0
                    .section    data_FEI,ROMDATA
                    .section    data_FOI,ROMDATA
.endif

;------------------------------------------------------------------------
; variable vector section
;------------------------------------------------------------------------
                    .section    vector,ROMDATA

; When you use "#pragma interrupt" with "vect=",                        ← (5)
; you need not define interrupt vector.
;
; When you use "#pragma interrupt" without "vect=",
; you must define all interrupt vectors like the following example.
; You define dummy_int for interrupt vector not used.
```

図2.13　アセンブラスタートアッププログラム sect30.inc リスト抜粋 (2/5)

```
;                       .lword      dummy_int           ; vector   0
;                       .lword      dummy_int           ; vector   1
;                       .lword      dummy_int           ; vector   2
;                                     :
;                       .lword      dummy_int           ; vector 63

;------------------------------------------------------------------------
; for User Boot Code Area
; Please custumize this data for your setting.
;------------------------------------------------------------------------
.if 0
        .section _UB_section_FE,ROMDATA
        .org    013ff0H
        .byte 0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh,0FFh ; User boot code
        .word 0FFFFh                                  ; Port address
        .byte 0FFh                                    ; Port bit
        .byte 0FFh                                    ; Boot level
        .byte 0FFh,0FFh,0FFh,0FFh                     ; Reserved
.endif

;------------------------------------------------------------------------
; fixed vector section
;------------------------------------------------------------------------
                        .section    fvector,ROMDATA
                        .org                0fffdcH
UDI:
                        .lword              dummy_int
OVER_FLOW:
                        .lword              dummy_int
BRKI:
                        .lword              dummy_int
ADDRESS_MATCH:
                        .lword              dummy_int
SINGLE_STEP:
                        .lword              dummy_int
WDT:
                        .lword              dummy_int
DBC:
                        .lword              dummy_int
NMI:
                        .lword              dummy_int
RESET:
                        .lword      start
```

図2.14　アセンブラスタートアッププログラム sect30.inc リスト抜粋 (3/5)

```
;======================================================================
; ID code & ROM code protect
;----------------------------------------------------------------------
; ID code check function
                        .id "#FFFFFFFFFFFFFF"

; ROM code protect control address
                        ; .protect 00H

;======================================================================
; Initialize Macro declaration
;----------------------------------------------------------------------
N_BZERO .macro          TOP_,SECT_
                        mov.b           #00H,R0L
                        mov.w           #(TOP_ & 0FFFFH),A1
                        mov.w           #sizeof SECT_,R3
                        sstr.b
                        .endm

N_BCOPY .macro          FROM_,TO_,SECT_
                        mov.w           #(FROM_ & 0FFFFH),A0
                        mov.b           #(FROM_ >> 16),R1H
                        mov.w           #TO_,A1
                        mov.w           #sizeof SECT_,R3
                        smovf.b
                        .endm

BZERO   .macro          TOP_,SECT_
                        push.w          #sizeof SECT_ >> 16
                        push.w          #sizeof SECT_   & 0ffffh
                        pusha           TOP_ >> 16
                        pusha           TOP_ & 0ffffh
                        .stk            8
                        .glb            _bzero
                        .call           _bzero,G
                        jsr.a           _bzero
                        .endm

BCOPY   .macro          FROM_ ,TO_ ,SECT_
                        push.w          #sizeof SECT_   >> 16
                        push.w          #sizeof SECT_   & 0ffffh
                        pusha           TO_ >> 16
                        pusha           TO_   & 0ffffh
                        pusha           FROM_ >> 16
                        pusha           FROM_   & 0ffffh
                        .stk            12
                        .glb            _bcopy
                        .call   _bcopy,G
                        jsr.a           _bcopy
                        .endm
```

図2.15　アセンブラスタートアッププログラム sect30.inc リスト抜粋 (4/5)

(1) SBDATA 領域の先頭アドレスを定義します。
(2) ユーザスタックサイズを定義します。
(3) 割り込みスタックサイズを定義します。
(4) 使用する heap 領域を定義します。
(5) ベクタテーブルを定義します。

図2.16 アセンブラスタートアッププログラム sect30.inc リスト抜粋 (5/5)

2.2.2 アセンブラスタートアッププログラムのカスタマイズ

a. アセンブラスタートアッププログラムの処理概要

(1) ncrt0.a30 について

このプログラムは、プログラムの開始時又はリセット直後に実行されます。
このプログラムは主に以下の処理を行います。

- プロセッサ動作モードを設定します。
- スタックポインタ（ISP レジスタと USP レジスタ）の初期化を行います。
- SB レジスタの初期化を行います。
- INTB レジスタの初期化を行います。
- データの near 領域の初期化を行います。
  bss_SE、bss_SO、bss_NE、bss_NO セクションをゼロクリアします。
  また初期値が格納された ROM 領域（data_SEI、data_SOI、data_NEI、data_NOI)の初期値を RAM 領域(data_SE、data_SO、data_NE、data_NO)に転送する処理を行います。
- データの far 領域の初期化を行います。
  bss_FE、bss_FO セクションをゼロクリアします。
  また、初期値が格納された ROM 領域(data_FEI、data_FOI)の初期値を RAM 領域(data_FE、data_FO)に転送する処理を行います。
- heap 領域の初期化を行います。
- 標準入出力関数ライブラリの初期化を行います。
- 静的オブジェクトの動的初期化のための呼び出しを行います。
- main 関数の呼び出しを行います。

b. アセンブラスタートアッププログラムの変更手順

アセンブラスタートアッププログラムを、組み込むシステムに合わせて変更する方法の手順を次に示します。

- c. 注意を要するアセンブラスタートアップの変更例
- d. スタックセクションのサイズの設定
- e. heap 領域のサイズの設定
- f. 割り込みベクタテーブルの設定
- g. プロセッサモードレジスタの設定

c. 注意を要するアセンブラスタートアップの変更例

(1) 標準入出力関数を使用しないときの設定

_init関数[3]は、標準入出力関数ライブラリの入出力の初期化を行います。_init関数は、ncrt0.a30 内でmain関数を呼び出す前に呼び出されます。【図 2.17】に_init関数の呼び出し部を示します。

アプリケーションプログラム中で標準入出力関数を使用しないときは、nc_define.inc 内の__STANDARD_IO__マクロを 0 にしてください。

```
;=====================================================================
; Initialize standard I/O
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
                .glb            __init
                .call           __init,G
                jsr.a           __init
.endif
```

図2.17　_init 関数呼び出し部 (ncrt0.a30)

sprintf 関数、vsprintf 関数、sscanf 関数のみを使う場合は、_init 関数を呼び出す必要がありません。この場合に、リンク時に__sget、__iob、$_fp、$_sput シンボルが未定義エラーになる場合がありますので、次のようなダミーのスタブ関数を作成してリンクしてください。

```
.if     __PROGRAM_NO_ALIGN__==1
   .section program,code
.else
   .section program,code,align
.endif
   .glb $__fp, $__pc, ___fs, ___sc
$__fp:
$__pc:
___fs:
___sc:
   rts
   .end
```

[3] _init関数は、標準入出力関数のためのマイコン（ハードウエア）の初期化も行っています。標準入出力関数を使用する場合は、組み込むシステムにより_init関数等を修正する必要があります。_init関数のソースファイルは、統合開発環境(High-performance Embedded Workshop)により、プロジェクトタイプ「C source startup Application」を選択し、「I/Oライブラリ使用」を有効にして作成したプロジェクトに生成します。

(2) メモリ管理関数を使用しないときの設定

メモリ管理関数(calloc、malloc、等)を使用するために、heap 領域の確保に加えて、ncrt0.a30 中で以下の設定を行っています。

(1) 外部変数 char * __mnext の初期化
heap 領域の先頭アドレスを表すラベル topof heap_NE で初期化します。

(2) 外部変数 unsigned long __msize の初期化
「2.2.2 e. heap 領域のサイズの設定」で設定した__HEAPSIZE__で初期化します。

【図 2.18】にncrt0.a30 内の初期化部を示します。

```
.if __HEAPSIZE__ != 0
  .glb  __mnext
  .glb  __msize
  mov.w  #((topof heap_NE)&0FFFFH),__mnext
  mov.w  #((topof heap_NE)>>16),__mnext+2
  mov.w  #(__HEAPSIZE__&0FFFFH),__msize
  mov.w  #(__HEAPSIZE__>>16),__msize+2
.endif
```

図2.18　メモリ管理関数を使用するときの初期化部 (ncrt0.a30)

メモリ管理関数を使用しない場合は、nc_define.inc 内の__HEAPSIZE__マクロの値を 0 にしてください。コメントにすることで、不要なライブラリがリンクされず ROM 容量を節約できます。C++プログラムを使用する場合、malloc 関数をランタイムライブラリとして使用するため、この初期化部をコメントにしないで下さい。

(3) 独自の初期化プログラムを記述するときの注意事項

独自の初期化プログラムをアセンブラスタートアッププログラム中に追加する場合は、以下の点に注意してください。

(1) 独自の初期化プログラムにおいて、U、B フラグを変更した場合は、初期化プログラムの出口で U、B フラグを元の状態に戻してください。また、SB レジスタの内容を変更しないでください。

(2) 独自の初期化プログラムから C 言語で記述されたサブルーチンを呼び出す場合は、以下の 2 項に注意してください。

- B、D フラグはクリアした状態で呼び出してください。
- U フラグはセットした状態で呼び出してください。

(4) グローバルクラスオブジェクトの初期化および終了処理関数の呼び出し

グローバルクラスオブジェクトを使った C++言語をコンパイル、リンクを実行すると、C$INIT セクションが見つからない、というリンクエラーが発生することがあります。その際には、main 関数の呼び出しの直前にグローバルクラスオブジェクトの初期化関数__CALL_INIT を呼び出す必要があります。さらに、main 関数の呼び出し直後にグローバルクラスオブジェクトの終了処理関数__CALL_END を呼び出す必要があります。

ncrt0.a30 ではこの部分をコメントにしてありますので、必要に応じてコメントを外して、この部分をコンパイル対象にしてください。

d. stack セクションのサイズの設定

stack セクションは、ユーザースタック用に使用する領域と割り込みスタック用に使用する領域が含まれます。
nc_define.inc において、シンボル__ISTACKSIZE__に割り込みスタックサイズを設定して下さい。
また、割り込みスタックとは別にユーザースタックを使用する場合には、nc_define.inc において、シンボル__STACKSIZE__にユーザースタックサイズを設定して下さい。

e. heap 領域のサイズの設定

heap 領域を使用する際には、シンボル__HEAPSIZE__に必要なサイズを設定してください。
heap 領域は物理的な RAM 領域を越えないように設定してください。
C++プログラムを使用する場合、malloc 関数をランタイムライブラリとして使用するため、必要な heap 領域を確保して下さい。
RAM データポインタのデフォルト属性が near 属性の C++プログラムオブジェクトファイルをリンクする場合、heap 領域を near 領域に配置する必要があります。

```
__HEAPSIZE__        .equ        0300H       ; HEEP SIZE definition
```

図2.19　heap サイズの設定例 (nc_define.inc)

f. 割り込みベクタテーブルの設定

リンカオプション-start でセクション vector のアドレスを設定します。セクション vector の先頭アドレスで、INTB レジスタが初期化されます。

g. プロセッサモードレジスタの設定

ncrt0.a30 中の【図 2.20】で示す部分において 04H番地(プロセッサモードレジスタ)に、組み込むシステムに合わせたプロセッサ動作モードを設定します。

```
;------------------------------------------------------------------
; after reset,this program will start
;------------------------------------------------------------------
                :
        (省略)
                :
        mov.b       #00h,04h                ;set processer mode
                :
        (省略)
                :
```

図2.20　プロセッサモードレジスタ設定例 (ncrt0.a30)

プロセッサモードレジスタの詳細については、マイコンのユーザーズマニュアルを参照してください。

2.2.3 メモリ配置のカスタマイズ

a. セクションの構成

ネイティブな環境のコンパイラの場合、コンパイラが生成した実行ファイルは UNIX 等のオペレーティングシステムによってメモリ配置が決定されます。しかし、本コンパイラのようなクロス環境のコンパイラでは、ユーザーがメモリ配置を決定する必要があります。

本コンパイラは、変数の記憶クラス、初期値を持つ変数、初期値を持たない変数、文字列データ、割り込み処理プログラム、割り込みベクタテーブル等プログラムの機能ごとにセクションという単位でマイコンのメモリ上に配置します。

セクションを表すセクション名は、セクションベース名とその属性により、【図 2.21】で構成されます。

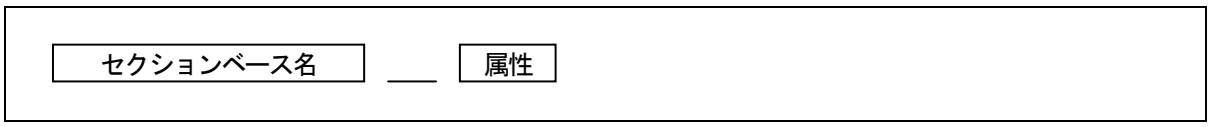


図2.21　セクション名

セクション名を【表 2.13】に属性を【表 2.14】に示します。

表2.13　セクションベース名

セクションベース名	内容
data	初期値のあるデータを格納します。
bss	初期値のないデータを格納します。
rom	文字列、const 修飾子で指定されたデータを格納します。

表2.14　属性

属性	意味		対象セクションベース名
I	データの初期値を保持するセクション		data
N/F/S	N	near属性[4]	data, bss, rom
	F	far 属性	
	S	SBDATA 属性	data, bss
E/O	E	データサイズが偶数	data, bss, rom
	O	データサイズが奇数	

前述の命名規則に準じたセクション以外のセクションの内容を【表 2.15】に示します。

[4] near、farとは、NC30 固有の修飾子です。この修飾子を使用することによりアドレッシングモードを明示的に指定できます。
near.....アクセス範囲は 00000H 番地から 00FFFFH 番地まで
far.....アクセス範囲は 00000H 番地から 0FFFFFH 番地まで

表2.15　セクションの名称

セクション名	内容
fvector	マイコンの固定ベクタの内容を格納します。
heap_NE	メモリ管理関数(malloc, new)により、プログラム実行中に動的に確保されるメモリ領域です。 このセクションはマイコンの任意の RAM 領域に配置できます。 RAM データポインタのデフォルト属性が near 属性の C++プログラムオブジェクトファイルをリンクする場合、heap 領域を near 領域に配置する必要があります。
program	プログラムを格納します。
program_S	#pragma SPECIAL で指定したプログラムを格納します。
stack	スタックとして使用する領域です。 ユーザースタックポインタレジスタ(USP)で参照します。 アドレス 0400H から 0FFFFH の間に配置してください。
istack	スタックとして使用する領域です。 割り込みスタックポインタレジスタ(ISP)で参照します。 アドレス 0400H から 0FFFFH の間に配置してください。
switch_table	switch 文に対するジャンプテーブルを格納します。 このセクションは、コンパイルオプション"-fswitch_other_section(-fSOS)"を使用した場合のみ生成されます。
vector	マイコンの割り込みベクタテーブルの内容を格納します。割り込みベクタテーブルは INTB レジスタ相対によりマイコンの全メモリ空間に任意に配置できます。詳しくはマイコンのユーザーズマニュアルを参照してください。
C$INIT	グローバルクラスオブジェクトに対して呼び出されるコンストラクタおよびデストラクタのアドレスを格納します。これは ROM に配置しなければなりません。
C$VTBL	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します。これは ROM に配置しなければなりません。
interrupt	ncrt0.a30 で定義されたエントリシンボル（start）を含むスタートアップ用セクションです。 エントリシンボルを含む本セクションを、program セクションより前に配置することで、効果的に最適化リンケージエディタの最適化をかけることができます。

これらのセクションの配置は、リンク時に-startオプションで指定します。セクション配置例を【図 2.22】に示します。

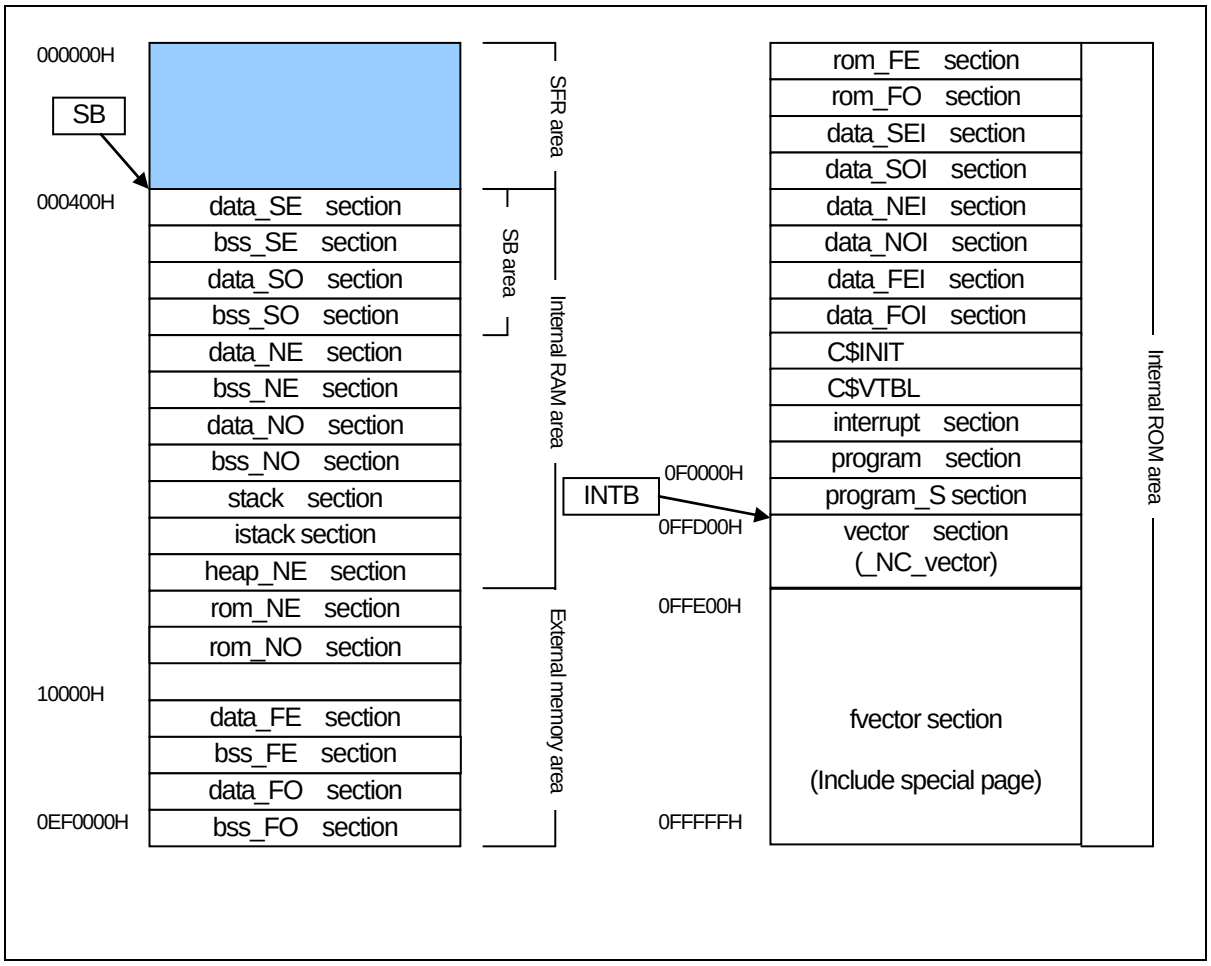


図2.22　セクション配置例

(1) セクションの配置規則

セクションには、 マイコンのメモリ属性(RAM/ROM)に影響されるため、配置できる領域が限られているセクションがあります。セクションの配置は、以下の規則に従ってください。

(1) RAM 領域に配置するセクション

- stack セクション
- data_SE セクション
- data_NE セクション
- bss_SE セクション
- bss_NE セクション
- bss_FE セクション
- heap_NE セクション
- data_SO セクション
- data_NO セクション
- bss_SO セクション
- bss_NO セクション
- bss_FO セクション

(2) ROM に配置するセクション

- program セクション
- fvector セクション
- rom_NO セクション
- rom_FO セクション
- data_SOI セクション
- data_NOI セクション
- data_FOI セクション
- C$VTBL セクション
- interrupt セクション
- rom_NE セクション
- rom_FE セクション
- data_SEI セクション
- data_NEI セクション
- data_FEI セクション
- C$INIT セクション

また、セクションはマイコンのメモリ空間に配置できる領域が限られているセクションがあります。

(1) 0H～0FFFFH(near 領域)にのみ配置できるセクション

- data_NE セクション
- data_SE セクション
- bss_NE セクション
- bss_SE セクション
- rom_NE セクション
- stack セクション
- data_NO セクション
- data_SO セクション
- bss_NO セクション
- bss_SO セクション
- rom_NO セクション

(2) 0F0000H～0FFFFFH にのみ配置できるセクション

- program_S セクション

(3) M16C/60 シリーズの全メモリ空間に配置できるセクション

- program セクション
- data_NEI セクション
- data_FE セクション
- data_FEI セクション
- data_SEI セクション
- bss_FE セクション
- rom_FE セクション
- C$INIT セクション
- vector セクション
- data_NOI セクション
- data_FO セクション
- data_FOI セクション
- data_SOI セクション
- bss_FO セクション
- rom_FO セクション
- C$VTBL セクション

また、以下のデータに関するセクションのサイズが 0 の場合、必ずしも定義する必要はありません。

- data_SE セクション
- data_SO セクション
- data_NE セクション
- data_NO セクション
- data_FE セクション
- data_FO セクション
- bss_NE セクション
- bss_FE セクション
- bss_SE セクション
- rom_NE セクション
- rom _FE セクション
- data_SEI セクション
- data_SOI セクション
- data_NEI セクション
- data_NOI セクション
- data_FEI セクション
- data_FOI セクション
- bss_NO セクション
- bss_FO セクション
- bss_SO セクション
- rom _NO セクション
- rom_FO セクション

b. 割り込みベクタテーブルの設定

割り込み処理を使用するプログラムは、割り込みベクタテーブル(セクション vector)に割り込み関数アドレスを設定する必要があります。ベクタ番号を使用した#pragma INTERRUPT や#pragma INTCALL を使用して割り込み関数を定義する場合、リンカがセクション vector を生成します。しかし、ベクタ番号を使用しない、#pragma INTERRUPT を使用する場合、sect30.inc で割り込みベクタテーブルを設定する必要があります。

割り込みベクタの内容は、マイコンの機種により異なります。使用するマイコン機種に合わせて設定する必要があります。詳細については、各機種のユーザーズマニュアルを参照してください。

(1) sect30.inc で割り込みベクタテーブルの設定

割り込み処理を使用するプログラムでは、sect30.inc 中の vector セクションの割り込みベクタテーブルを変更します。

【図 2.23】に割り込みベクタテーブル例を示します。

```
;------------------------------------------------------------
; variable vector section
;------------------------------------------------------------
                .section    vector,ROMDATA              ; variable vector table
                .org        VECTOR_ADR

                .lword      dummy_int                   ; BRK (software int 0)
                   :
                (省略)
                   :
                .lword      dummy_int                   ; DMA0 (software int 8)
                .lword      dummy_int                   ; DMA1 (software int 9)
                .lword      dummy_int                   ; DMA2 (software int 10)
                   :
                (省略)
                   :
                .lword      dummy_int                   ; uart1 transe (software int 19)
                .lword      dummy_int                   ; uart1 receive (software int 20)
                .lword      dummy_int                   ; TIMER B0 (software int 21)
                   :
                (省略)
                   :
                .lword      dummy_int                   ; INT5 (software int 26)
                .lword      dummy_int                   ; INT4 (software int 27)
                   :
                (省略)
                   :
                .lword      dummy_int                   ; uart2 transe/NACK (software int 33)
                .lword      dummy_int                   ; uart2 receive/ACK (software int 34)
                   :
                (省略)
                   :
                .lword      dummy_int                   ; software int 63

※ dummy_int はダミーの割り込み処理関数です
```

図2.23　割り込みベクタテーブルの設定例

次の手順で sect30.inc 中の vector セクションの割り込みベクタテーブルを変更します。

(1) 割り込み処理関数名をアセンブラの指示命令.GLB で外部参照宣言します。

(2) 本コンパイラで作成した割り込み処理関数への登録ラベル名は、関数名の前に_(アンダースコア)が付加されます。したがって、ここで宣言する割り込み処理関数名にもアンダースコアを付加して記述します。

(3) 該当する割り込みベクタテーブルのダミー割り込み関数名 dummy_int から使用する割り込み処理関数名に置き換えます。

【図 2.24】にUART1 送信割り込み処理関数uarttrnの設定例を示します。

```
.lword    dummy_int        ; uart0 receive (for user)
.glb      _uarttrn                                    ← 上記(1)の処理
.lword    _uarttrn         ; uart1 transmit (for user) ← 上記(2)の処理

(以下省略)
```

図2.24　割り込みベクタテーブルの設定例

2.3 C言語スタートアッププログラムの準備

C言語で記述したプログラムを ROM 化するために、マイコンの初期設定、セクションの配置、割り込みベクタテーブルを設定するアセンブリ言語もしくは C 言語で記述したスタートアッププログラムが必要です。スタートアッププログラムはお客様がご使用のマイコン機種、システムに合わせて変更していただく必要があります。

ここでは、C 言語で記述した C 言語スタートアッププログラムの説明と、そのカスタマイズの方法について説明します。

なお、統合開発環境(High-performance Embedded Workshop)を起動し、新規プロジェクトの作成でプロジェクトタイプを C source startuop Application を選択すると、C 言語スタートアッププログラムの雛型がフォルダに自動生成されますので、それを元に変更してください。

2.3.1 生成ファイル

C 言語スタートアッププログラムには、以下のファイルがあります。

(1) resetprg.c
マイコンの初期設定を行います。

(2) initsct.c
各セクションの初期化(ゼロクリア、初期値転送)を行います。

(3) heap.c
ヒープ領域を確保します。

(4) fvector.c
固定ベクタテーブルの定義を行います。

(5) intprg.c
可変ベクタ割り込みのエントリ関数を宣言します。

(6) firm.c/firm_ram.c　(変更の必要はありません)
OnChipDedebugger 選択時の FoUSB/E8 の firm が使用するプログラム領域及びワークスペース領域をダミーとして確保します。

(7) cstartdef.h
スタックサイズ、ヒープサイズ等の各 define 値を定義しています。

(8) initsct.h　(変更の必要はありません)
各セクションを初期化する処理（アセンブラマクロ）を記述しています。

(9) resetprg.h
各ヘッダファイルをインクルードしています。

(10) typedefine.h　(変更の必要はありません)
各型に対して typedef 宣言をしています。

(11) sfrXX.h,sfrXX.inc
プロジェクト作成時に選択した CPU に対応して sfr 定義ヘッダファイルをワークスペースへ登録します。

2.3.2 各生成ファイルの処理

- resetprg.c（必須）
  本ファイルは、選択した MCU（M16C もしくは、R8C）によって内容が変わります。

```
#pragma section program interrupt   ------------------------------------------------------------   (1)

void start(void)                    ------------------------------------------------------------   (2)
{
        _isp_       = &_istack_top;         // set interrupt stack pointer      ----------   (3)
        protect = 0x02U;                    // change protect mode register ----------   (4)
        pmode0      = 0x00U;                // set processor mode register   ----------   (5)
        protect = 0x00U;                    // change protect mode register ----------   (6)
        _flg_       = __F_value__;          // set flag register                ----------   (7)
        _sp_        = &_stack_top;          // set user stack pointer           ----------   (8)
        _sb_        = 0x400U;  // 400H fixation (Do not change)                 ----------   (9)

        // set variable vector's address
        _asm("      ldc         #((topof vector)>>16)&0FFFFh,INTBH");   ----------   (10)
        _asm("      ldc         #(topof vector)&0FFFFh,INTBL");

        initsct();              // initlalize each sections                  ----------   (11)
#if __HEAPSIZE__ != 0
        heap_init();            // initialize heap                           ----------   (12)
#endif
#if __STANDARD_IO__ != 0
        _init();                            // initialize standard I/O       ----------   (13)
#endif
        _fb_ = 0U;  // initialize FB registe for debugger
//      _CALL_INIT();           // Remove the comment when you use global class object
        main();                             // call main routine             ----------   (14)

        _exit();                // call exit
}
```

(1)　スタート関数は interrupt セクションに配置します。
(2)　CPU 初期化関数 start()本体を宣言します。
(3)　割り込みスタックポインタを初期化します。
(4)　プロテクトレジスタを“書き込み許可”に設定します。
(5)　プロセッサモードレジスタを“シングルチップモード”に設定します。
モードを変更する場合は、この式を変更する必要があります。
(6)　プロテクトレジスタを“書き込み禁止”に設定します。
(7)　U フラグを設定します。
ワークスペース作成ウィザードで“ユーザスタックを使用する”を選択した場合は、ユーザスタックポインタを設定します。
(8)　ワークスペース作成ウィザードで“ユーザスタックを使用する“を選択した場合に、ユーザスタックポインタを初期化します。
(9)　SB レジスタを 0x400 番地に設定（RAM の先頭アドレスを設定）します。
(10)　可変ベクタアドレスを INTB レジスタへの設定します。

(11) 各セクションの初期化 (ゼロクリア、初期値転送) を行います。
(12) ヒープ領域の初期化を行います。
メモリ管理関数を使用する場合は、本関数の呼び出しを有効にする必要があります。
(13) 標準入出力関数用を初期化を行います。
標準入出力関数を使用する場合は、本関数の呼び出しを有効にする必要があります。
(14) main 関数を呼び出します。

- initsct.c (必須)

本ファイルは、選択した MCU(M16C/R8C)により内容は変わります。

```
void initsct(void)
{
        sclear("bss_SE","data","align");    ----------    (1)
        sclear("bss_SO","data","noalign");
        sclear("bss_NE","data","align");
        sclear("bss_NO","data","noalign");

        sclear_f("bss_FE","data","align");  ----------    (2)
        sclear_f("bss_FO","data","noalign");

        // add new sections. refer to the above-mentioned.

        scopy("data_SE","data","align");    ----------    (3)
        scopy("data_SO","data","noalign");
        scopy("data_NE","data","align");
        scopy("data_NO","data","noalign");

        scopy_f("data_FE","data","align");  ----------    (4)
        scopy_f("data_FO","data","noalign");
}
```

(1) sclear：near 領域の bss セクションをゼロクリアします。
#pragma SECTION bss 機能を用いて bss セクション名を変更、追加した場合は、NE/NO をセットで変更、追加が必要

```
sclear(“セクション名_NE”,” data.align”);
sclear(“セクション名_NO”,” data,noalign”);
```

例) #pragma section bss　bss2 でセクション追加した場合

```
sclear(“bss2_NE”,”data,align”);
sclear(“bss2_NO”,”data,noalign”);
```

を initsct.c へ追加します。

(2) sclear_f：far 領域の bss セクションをゼロクリアします。

(3) scopy: near 領域の data セクションに対して初期値を転送します。
#pragma SECTION data 機能を用いて data セクション名を変更、追加した場合は、NE/NO をセットで変更、追加が必要

```
scopy("セクション名_NE","data,align");
scopy("セクション名_NO","data,noalign");
```

例) #pragma section data data2 でセクション追加した場合

```
scopy(“data2_NE”,”data,align”);
scopy(“data2_NO”,”data,noalign”);
```

を initsct.c へ追加します。

(4) scopy_f : far 領域の data セクションに初期値を転送します。

- heap.c (malloc などのメモリ管理関数を使用する場合のみ必要)

```
#pragma SECTION      bss       heap      ----------      (1)

_UBYTE heap_area[__HEAPSIZE__];      ----------      (2)
```

(1) heap 領域を heap_NE セクションに配置します。
※ ヒープサイズを奇数バイトにした場合は、heap_NO セクションになります。

(2) ヒープ領域を__HEAPSIZE__で定義されたサイズ分確保します。

- fvector.c (必須)

```
#pragma sectaddress   fvector,ROMDATA      0xffdc    ----------    (1)

//////////////////////////////////////////////////////////////////

#pragma interrupt/v _dummy_int    //udi                ----------    (2)
#pragma interrupt/v _dummy_int    //over_flow
#pragma interrupt/v _dummy_int    //brki
#pragma interrupt/v _dummy_int    //address_match
#pragma interrupt/v _dummy_int    //single_step
#pragma interrupt/v _dummy_int    //wdt
#pragma interrupt/v _dummy_int    //reserved
#pragma interrupt/v _dummy_int    //reserved
#pragma interrupt/v start                            ----------    (3)
```

(1)　固定ベクタテーブルのセクションとアドレスを出力します。
※本 pragma は、スタートアップ用ですので、通常は使用できません。

(2)　リセット以外の固定ベクタをダミー関数（_dummy_int）で埋めます。
#pragma interrupt/v 関数名は、関数名をベクタに登録します。関数の本体を記述する場合は、本宣言とは別に#pragma interrupt を用いて関数を定義してください。

(3)　エントリ関数を定義します。
リセット時の実行関数を固定ベクタへ登録します。

- intprg.c（マイコン品種毎、必要に応じて）

```
// DMA0 (software int 8)
#pragma interrupt       _dma0(vect=8)
void _dma0(void){}

// DMA1 (software int 9)
#pragma interrupt       _dma1(vect=9)
void _dma1(void){}

// DMA2 (software int 10)
#pragma interrupt       _dma2(vect=10)
void _dma2(void){}

// DMA3 (software int 11)
#pragma interrupt       _dma3(vect=11)
void _dma3(void){}

  （省略）
```

(1)　可変ベクタ割り込み関数を宣言します。
各可変ベクタ割り込み関数に対応した関数を宣言します。同時に可変ベクタテーブルを生成します。

(2)　可変ベクタ割り込み関数を定義します。
使用する割り込みベクタ番号に対応する関数に処理を記述してください。

例)　割り込みベクタ番号９番（DMA1）を使用する場合

```
#pragma interrupt       _dma1(vect=9)
void _dma1(void)
{
        //処理を記述
}
```

(3)　intprg.c が不要な場合
ファイルの登録から削除してリンク対象からはずしてください。

- firm.c/firm_ram.c (OnChipDebugger FoUSB/E8 選択時)

  本ファイルの内容は変更しないでください。

  本ファイルの内容は、マイコン及び FOUSB/E8 選択により変更されます。

```
#ifdef __E8__              // for E8                                  ----------    (1)

#pragma section bss FirmRam                                          ----------    (2)

#ifndef __WORK_RAM__
#define __WORK_RAM__           0x80
#endif

_UBYTE _workram[__WORK_RAM__];                                       ----------    (3)

#pragma section bss FirmArea                                         ----------    (4)
_far _UBYTE _firmarea[0x800];      // dummy for monitor              ----------    (5)

#else          // for FoUSB

#pragma section bss FirmRam                                          ----------    (6)
_UBYTE _workram[0x80];             // for Firmware's workram         ----------    (7)

#pragma section bss FirmArea                                         ----------    (8)
_far _UBYTE _firmarea[0x600];      // dummy for monitor              ----------    (9)
#endif
```

(1) E8 を使用する場合に有効にします。
(2) E8 のファームウエアが使用する work ram 領域を FirmRam_NE セクションに確保します。
(3) work ram 領域を__WORK_RAM__で定義されたサイズ分確保します。
(4) E8 のファームウエアプログラムを FirmArea セクションに配置します。
(5) ファームウエアプログラムのサイズを指定します。
(6) FoUSB のファームウエアが使用する work ram 領域を FirmRam_NE セクションに確保します。
(7) work ram 領域を 0x80 バイト確保します。(対応するマイコンの品種により異なります。)
(8) FoUSB のファームウエアプログラムを FirmArea セクションに配置します。
(9) ファームウエアプログラムのサイズを指定します。

- cstartdef.h (必須)

```
#define __STACKSIZE__              0x80    ----------    (1)
#define __ISTACKSIZE__             0x80    ----------    (2)
#define __HEAPSIZE__               0x80    ----------    (3)
#define __STANDARD_IO__            0       ----------    (4)
#define __WATCH_DOG__              0       ----------    (5)
```

(1)　ワークスペース作成ウィザードで入力したスタックサイズに応じて変化します。
(2)　ワークスペース作成ウィザードで入力した割り込みスタックサイズに応じて変化します。
(3)　ワークスペース作成ウィザードで入力したヒープサイズに応じて変化します。
(4)　ワークスペース作成ウィザードで“標準入出力関数を使用する”を選択した場合に、1 が設定されます。
(5)　リセット直後に WATCH DOG 機能を有効にする場合は、1 を設定します。(R8C ファミリのみ)

上記を新規ワークスペース作成後に再度変更を行う場合は、本ファイルを直接変更してください。

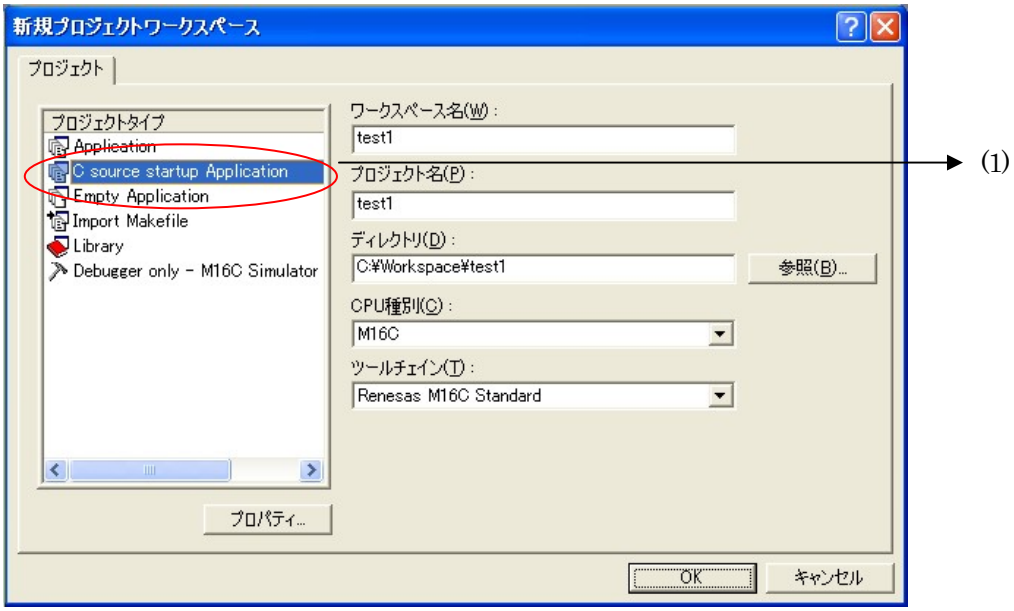
- initsct.h (必須)
  本ファイルの内容は変更しないでください。

- resetprg.h (必須)
  オンチップデバッガをご使用の際は、『L.1.3 オンチップデバッガ選択時の FirmRam_NE セクションと SB レジスタの値に関して』をご参照ください。

- typedefine.h (必須)
  本ファイルの内容は変更しないでください。

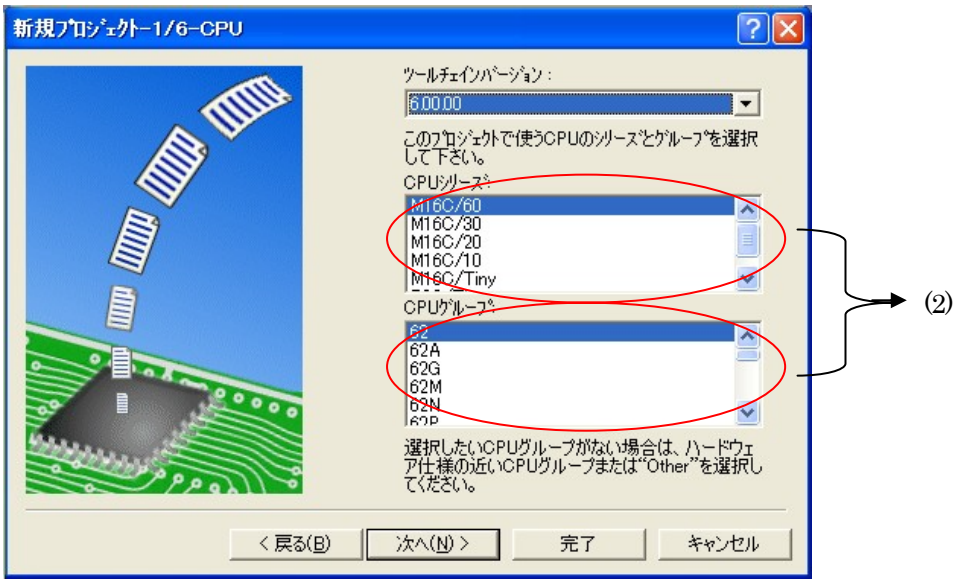
2.3.3　C 言語スタートアップの生成方法

- C 言語スタートアップを使用したプロジェクトの選択



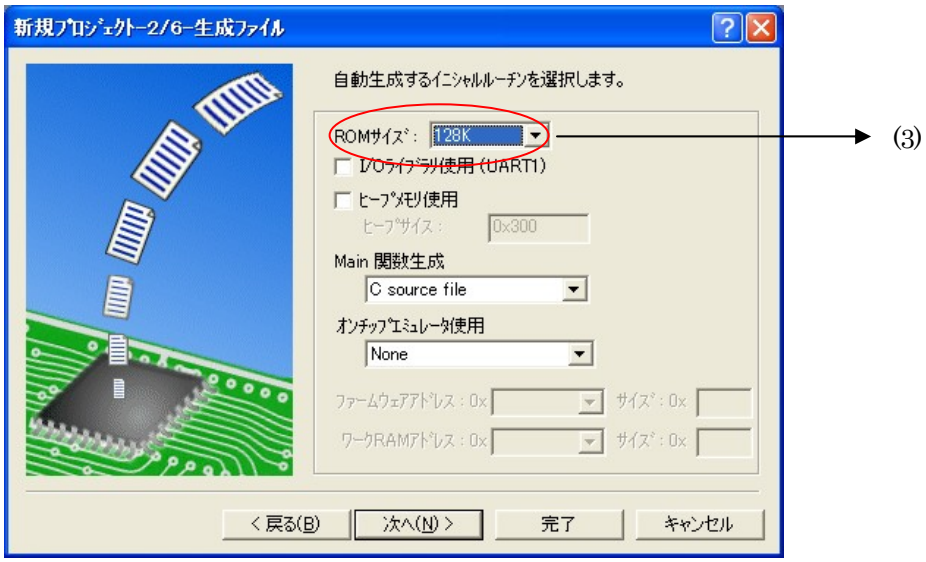
(1)　左窓の C source startup Application を選択します。
※ 複数コンパイラをインストールしている場合で、C source startup Application 選択後 CPU 種別で、他マイコンを選択した場合、C source startup Application へのフォーカスが Application へ移動して、C ソーススタートアップの選択が無効になりますので、再度 C source startup Application を選択してください。

- マイコン品種の選択



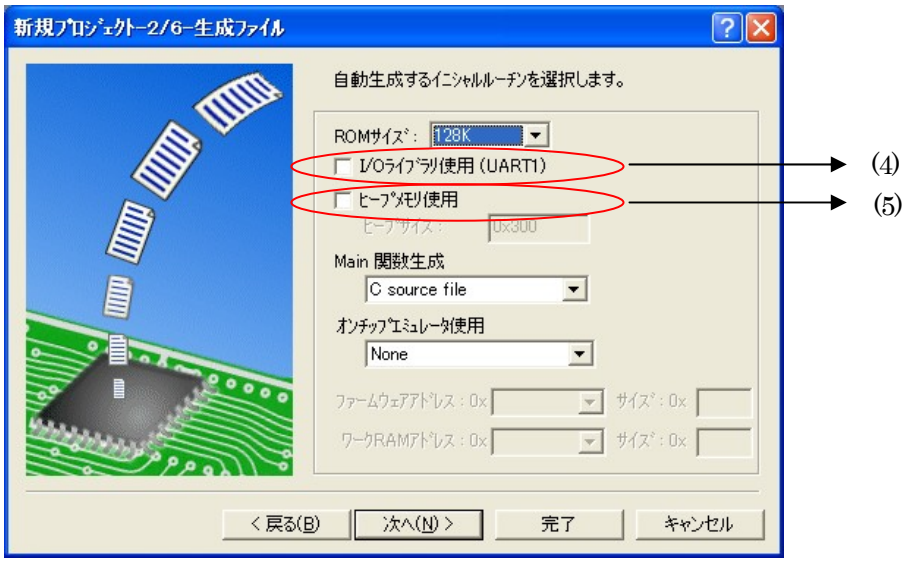
(2)　CPU Series と CPU Group からマイコン品種を選択します。
　この選択により、対応する sfr ヘッダファイルがワークスペースへ登録されます。また、可変ベクタエントリ関数（intprg.c）が登録されます。

- ROM サイズの選択



(3)　(3)で選択する ROM サイズは、オンチップデバッガ選択時の設定に加えて、リンク時の ROM 属性のセクションを ROM サイズに応じて適切に配置するようにします。

- 標準入出力関数ライブラリとメモリ管理関数ライブラリ使用時の設定



(4)　標準入出力関数ライブラリを使用する場合に、チェックします。
　チェックが行われる事により、resetprg.c 中の_init( )呼び出しが有効になります。また、device.c と init.c がプロジェクトに登録されます。

(5)　メモリ管理関数を使用する場合に、チェックします。
　チェックが行われる事により、resetprg.c 中の heap_init()呼び出しが有効になります。また、heapdef.h ,heap.c がプロジェクトに登録されます。

- OnChipDebugger の選択

(6)　OnChipDebugger を使用する場合に選択します。
　選択可能なデバッガは、FoUSB と、E8 になります。ただし、選択するマイコン品種により、いずれか一方、もしくは両方選択できない場合があります。この選択により、firm.c が登録され、(7) に表示のデバッガが占有する領域を変数領域として確保することにより、ユーザプログラムとの重複を回避します。

(7)　Firmware Address と workRamAddress の設定を行います。
FoUSB/E８が占有する、Firmware 用のプログラム領域と work 用の RAM 領域の設定を行います。使用するデバッガでアドレス変更が可能な場合のみ設定変更できます。デバッガ使用時にこれらのアドレスを変更した場合は、デバッガ使用時の設定に合わせて変更してください。変更する際の各アドレス、サイズについては、使用するデバッガのマニュアルを参照ください。

(8)　標準入出力関数ライブラリを選択した状態で OnChipDebugger を選択すると、(UART1)の表示が(UART0)に変わります。これは、標準入出力関数及び OnChipDebugger が共に UART1 を使用するため標準入出力側を UART０へ変更することを意味しています。表示が(UART0)の場合、コンパイルオプション-D__UART0__を設定し、条件コンパイルします。

- スタックサイズの選択

(9)　ユーザスタックの使用有無を選択します。
チェックをしなかった場合は、start 関数内でユーザスタックを使用しない設定に変更します。

(10)　ユーザスタックサイズを設定します。
cstartdef.h 内の define 値を変更します。

(11)　ユーザスタックサイズを設定します。
cstartdef.h 内の define 値を変更します。

プロジェクト作成後、スタックサイズ及び HEAP サイズを変更する場合は、cstartdef.h 内の設定でそれぞれ

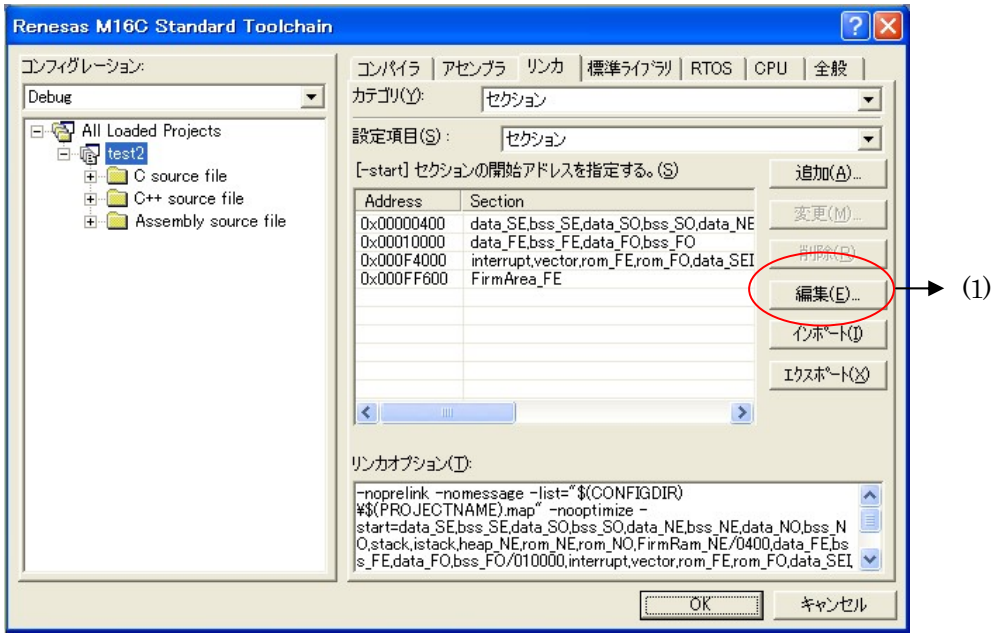
```
#define __STACKSIZE__          0x80
#define __ISTACKSIZE__         0x80
#define __HEAPSIZE__           0x80
```

の値を変更してください。

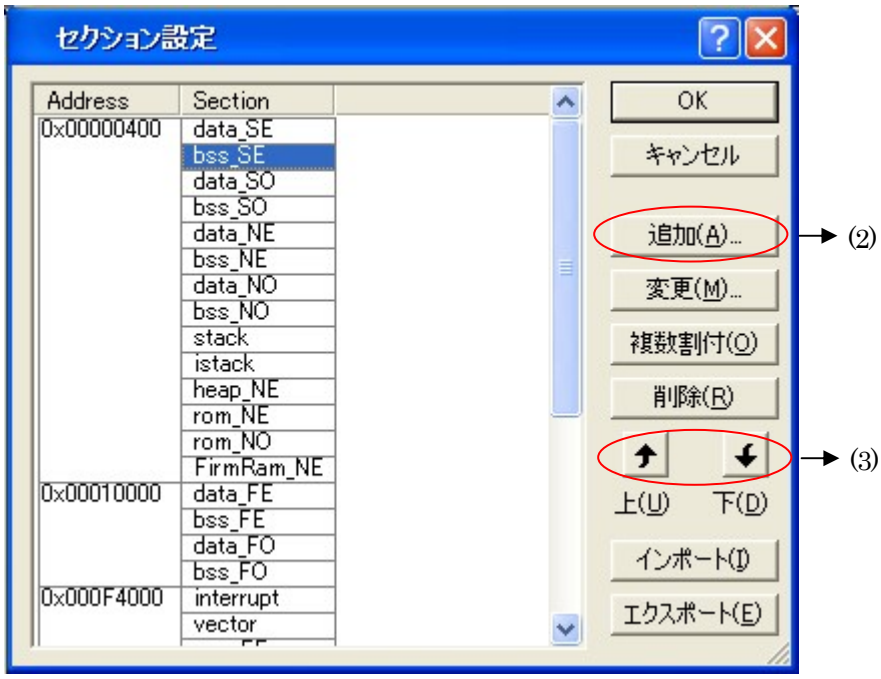
- 登録ファイル一覧

ここで、登録されるファイル一覧が確認できます。

- セクションオーダー



各セクションのリンク順及びリンクアドレスは、[Renesas M16C Standard Toolchain]→[リンク]のカテゴリ: セクションで確認することができます。



#pragma SECTION で新たにセクションを追加した場合などは、(1)の[編集]ボタンを選択して、ウィンドウ「セクション設定」をオープンしてください。フォーカスを Section にした状態で(2)の[追加]ボタンを選択してください。

ウィンドウ「セクションの追加」がオープンしますので、新しいセクション名を入力してください。入力されたセクションが登録されますので、配置するエリアにそのセクションを(3)の UP/DOWN ボタンを使用して移動させてください。

第3章 プログラミング

この章では、本コンパイラを使用してプログラミングを行う上で、注意すべき事項等について説明します。

3.1 注意事項

本資料に記載の製品データ、図、表に示す技術的な内容、プログラムおよびアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサスエレクトロニクス株式会社および株式会社ルネサスソリューションズは、適用可否に対する責任は負いません。

3.1.1 コンパイラのバージョンアップ等についての注意事項

本コンパイラが生成する機械語命令(アセンブリ言語)は、コンパイル時に指定する起動オプション、バージョン変更等により変化します。したがって、起動オプションの変更又はコンパイラのバージョン変更を行った場合は再度お客様が作成したプログラム全体の動作評価を必ず行ってください。

また、割込み処理プログラムと被割込み処理プログラム間、リアルタイム OS 上のタスク間等で、同じ RAM データを参照し内容を変更する場合は、必ず volatile 指定等の排他制御を行ってください。また、ビットフィールド構造体において、メンバ名が異なっている場合においても、同一の RAM 上に確保される場合は、同様に排他制御を行ってください。

3.1.2 マイコンの機種依存部に関する注意事項

SFR 領域のレジスタへの書き込み、または読み出しには特定の命令を使用しなければならないことがあります。この特定の命令は機種ごとに異なりますので、詳しくは各マイコン機種のユーザーズマニュアルを参照してください。

本コンパイラは、SFR領域のレジスタへの書き込み、読み出しには、使用できない命令を生成する場合があります。【図 3.1】のようなC言語記述でSFR領域にアクセスすると、SFR領域で使ってはいけない命令を生成するため、割り込み要求ビットの判定が正常に行われず意図しない動作を行う可能性があります。

SFR 領域のレジスタへの書き込み、読み出しをする場合は、asm 関数を使用してプログラム中に直接命令を記述してください。この場合、コンパイラのバージョン、オプションの有無に関わらず、生成されたコードに問題が無いことを必ず確認してください。

```
#pragma ADDRESS   TA0IC   0055h          /* M16C/62 タイマ A0 割込み制御レジスタ */

struct {
        char    ILVL : 3;
        char    IR : 1;                 /* 割込み要求ビット */
        char    dmy : 4;
} TA0IC;

void    wait_until_IR_is_ON(void)
{
        while(TA0IC.IR == 0)            /* 1 になるまで待つ */
        {
                ;
        }
        TA0IC.IR = 0;                   /* 1 になったら 0 に戻す */
}
```

図3.1 SFR 領域に対する記述例

3.1.3 最適化について

a. 常に行われる最適化

最適化オプションの有無に関わらず、以下のものは最適化されます。

(1) 意味のない変数アクセス

例えば、以下の【図 3.2】に示される変数portは、読み出し結果を使用しないため、読み出し動作が削除されます。

```
extern int  port;

void    func(void)
{
        port;
}
```

図3.2 意味のない変数アクセス例 (最適化される)

この例はportを読み出すだけの操作を行いたいことを意図して記述したものですが、実際には読み出すコードは最適化されて出力されません。最適化を行わないようにするには【図 3.3】に示すようにvolatile修飾子を付加してください。

```
extern int volatile     port;

void    func(void)
{
        port;
}
```

図3.3 意味のない変数アクセス例 (最適化を抑止)

(2) 意味のない比較

```
int         func(char c)
{
            int         i;

            if(c != -1)
                        i = 1;
            else
                        i = 0;
            return i;
}
```

図3.4　意味のない比較

この例の場合、変数 c は char と記述されていますので、本コンパイラでは unsigned char 型として取り扱います。unsigned char 型で表現できる数値の範囲は 0 から 255 までですので、変数 c は-1 の値を持つことはありません。このため、このような論理的に意味の無い文を記述された場合、本コンパイラではアセンブリ言語コードを生成しません。

(3) 実行されることのないプログラム

論理的に実行されることのないプログラムに対するアセンブリ言語コードは、生成されません。

```
void        func(int i)
{
            func2(i);
            return;

            i = 10;                         ← 実行されることがない部分
}
```

図3.5　実行されることのないプログラム

(4) 定数間の演算

定数間の演算はコンパイル時に演算されます。

```
int         func(void)
{
            int         i = 1 + 2;          ← コンパイル時に演算されます

            return i;
}
```

図3.6　定数間の演算

(5) 最適命令の選択

STZ 命令の使用や、乗除算をシフト命令で出力する等の最適命令の選択は、最適化オプションの有無に関わらず、常に行われます。

b. volatile 修飾子について

volatile 修飾子を使用することにより、変数の参照や参照順序、参照回数等に対して最適化の影響が無いようにすることができます。ただし、以下の図に示すような、解釈が曖昧になるような記述は行わないでください。

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

a = b = c;                  /* a = c なのか、a = b なのか? */
```

図3.7 volatile 修飾子の解釈が曖昧になる例

3.1.4 register 変数の使用に関する注意事項

a. register 修飾とコンパイルオプション"-fenable_register(-fER)"について

コンパイルオプション"-fenable_register (-fER) "を指定することにより、特定条件を満たす register 修飾を行った変数を強制的にレジスタに割り当てることができます。

この機能は、最適化に頼らずに生成コードを改良するためのものです。多用すると効率がわるくなることがあるので、必ず生成コードを確認した上で、使用してください。

b. register 修飾と最適化オプションについて

最適化オプションを指定すると最適化の機能の一つとして、変数のレジスタへの割り当てを行います。この割り当ての機能には、register 修飾を行っているか否かは影響しません。

3.2 生成コードの向上のために

3.2.1 コード効率の良いプログラミング方法

a. 整数/変数の取り扱いに関して

(1) 必要でないかぎり符号なしの整数を使用してください。int型、short型、long型は、符号指定子がない場合符号付きとして扱われます。これらのデータ型を持つ整数の演算には、必要でないかぎり符号指定子unsignedを付加してください。[1]

(2) 符合付きの変数の比較には、可能な限り>=、<=を使用しないで、!=、==で条件判断を行ってください。

b. far 型配列に関して

far 型配列の参照は、そのサイズにより機械語レベルでの参照方法が異なります。

(1) サイズが 64K バイト以内の場合
添え字を符号なし 16 ビット整数で計算します。これによりサイズが 64K バイト以下の配列は、効率の良いアクセスができます。

(2) サイズが 64K バイトを越える場合、もしくはサイズが不明の場合
添え字を 32 ビット幅で計算します。

したがって、サイズが 64Kバイトを越えないことが判明している場合、【図 3.8】に示すfar型配列のextern宣言においてサイズを明記するか、もしくはオプション"-fsmall_array(-fSA)"[2]を使用してコンパイルすることにより、コード効率を良くすることができます。

```
extern int far    array[];      ← サイズ不明なので添え字を 32 ビットで計算します
extern int far    array[10];    ← サイズが 64K バイト以内のため効率の良いアクセスを行います
```

図3.8 far 型配列の extern 宣言例

c. プロトタイプ宣言の活用

本コンパイラでは、関数のプロトタイプ宣言を行なうことにより、効率の良い関数呼び出しを行なうことができます。すなわち、本コンパイラでは関数原型宣言を行なわない場合、その関数を呼び出すときに、その関数の引数を【表 3.1】に示す規則によりスタック領域に積んで渡します。

表3.1 引数に関するスタックの使用規則

データ型	スタックに積むときの規則
char 型 signed char 型	int 型に拡張して積む。
float 型	double 型に拡張して積む。
その他の型	型の拡張は行なわずに積む。

このため、関数原型宣言を行なわない場合は、冗長な型拡張を行なう場合があります。関数原型宣言を行なうことにより、これらの冗長な型拡張を抑止し、また、レジスタに引数を割り当てることが可能になるため、効率の良い関数呼び出しを行なうことができます。

[1] char型、ビットフィールド構造体のメンバで符号指定子がない場合、符号なしとして扱われます。
[2] コンパイルオプション"-fsmall_array(-fSA)"は、サイズ不明の配列を 64Kバイト以内と仮定してコード生成を行います。

d. SB レジスタの活用

SBレジスタ[3]を用いたアドレッシングモードを使用することにより、アプリケーションプログラムサイズ(ROM容量)を削減することができます。本コンパイラでは【図 3.9】で示す記述を行なうことにより、SBレジスタを用いたアドレッシングモードを使用する変数を宣言することができます。

```
#pragma SBDATA   val
int              val;
```

図3.9 SB レジスタを用いたアドレッシングモードを使用する変数の宣言例

e. その他

その他の方法として次のような記述の変更を行うことにより、ROM 容量を圧縮できる場合があります。

(1) 一回しか呼ばれない比較的小さな関数を inline 関数にする。
(2) if-else 文を switch 文で置き換える(判定対象の変数が配列、ポインタ、構造体などの単純な変数ではない場合に効果があります)。
(3) ビットの比較を '&&'、'¦¦' ではなく '&'、 '¦' で行う。
(4) char 型の範囲でしか値を返さない関数の、戻り値の型を char 型で宣言する。
(5) 関数呼び出しをまたいで使用する変数をレジスタ変数にしない。
(6) コンパイラ、アセンブラのオプションに-goptimize を指定すると、最適なジャンプ命令を選択します。

3.2.2 スタートアップ処理を高速化する方法

スタートアッププログラム ncrt0.a30 には bss 領域のクリア処理が含まれています。この処理は C 言語の言語仕様として初期化されていない変数は初期値として 0 を持つという規格を満たすための処理です。

例えば、【図 3.10】に示す記述の場合、初期値を記述していませんので、スタートアップ処理時に初期値として 0 を与える処理(bss領域[4]のクリア処理)が必要になります。

```
static int   i;
```

図3.10 初期値を持たない変数の宣言例

アプリケーションによっては初期値を持たない変数を 0 クリアする必要が無いものがあります。この場合はスタートアッププログラム内の bss 領域のクリア処理部をコメントアウトすれば、スタートアップ処理を高速化することができます。

[3] 本コンパイラでは、SBレジスタはリセット後に初期化を行ない、以降は固定で使用することを前提としています。
[4] 初期値を持たないRAM上の外部変数のことを"bss"と呼びます。

```
;===================================================================
; NEAR area initialize.
;-------------------------------------------------------------------
; bss zero clear
;-------------------------------------------------------------------
;   N_BZERO   (topof bss_SE),bss_SE
;   N_BZERO   (topof bss_SO),bss_SO
;   N_BZERO   (topof bss_NE),bss_NE
;   N_BZERO   (topof bss_NO),bss_NO
            :
        (省略)
            :
;===================================================================
; FAR area initialize.
;-------------------------------------------------------------------
; bss zero clear
;-------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
;   BZERO   (topof bss_FE),bss_FE
;   BZERO   (topof bss_FO),bss_FO
.endif
```

図3.11　bss 領域のクリア処理のコメントアウト例

3.3 アセンブリ言語プログラムとの結合方法

3.3.1 C 言語プログラムからアセンブラ関数の呼び出し方法

a. 引数のないアセンブラ関数の呼び出し方法

C/C++言語プログラムからアセンブラ関数を呼び出す場合は、C/C++言語で記述した関数呼び出しと同様にアセンブラ関数名で呼び出します。

アセンブラ関数の先頭ラベル名は名前の最初に_(アンダースコア)を付加する必要があります。C/C++言語プログラムからアセンブラ関数を呼び出すときは、アセンブラ関数の名前(先頭ラベル名)から、アンダースコアを除いた名前を使用します。C言語からアセンブラ関数を呼び出す場合には、【図 3.12】のように必ずアセンブラ関数原型宣言を記述してください。

【図 3.12】にC言語からアセンブラ関数asm_funcを呼び出すときの記述例を示します。

```
extern void  asm_func( void );              ← アセンブラ関数原型宣言

void         main()
{
                 :
             (省略)
                 :
             asm_func();                    ← アセンブラ関数の呼び出し
}
```

図3.12 引数がない場合のアセンブラ関数の呼び出し例 (smp1.c)

```
             .glb        _main
_main:
                 :
             (省略)
                 :
             jsr         _asm_func          ← アセンブラ関数の呼び出し('_'を付加しています)
             rts
```

図3.13 smp1.c のコンパイル結果 (抜粋) (smp1.a30)

C++言語からアセンブラ関数を呼び出す場合には、【図3.14】のように必ずアセンブラ関数原型に「extern "C"」を付けてください。これによりC++固有の名前修飾を行わなくなります。

【図 3.14】にC++言語からアセンブラ関数asm_funcを呼び出すときの記述例を示します。

```
extern "C" {
      extern void asm_func(void);
}

void   main()
{
       (省略)
        :
       asm_func();
}
```

図3.14　C++言語からアセンブラ関数 asm_func を呼び出すときの記述例

b.　アセンブラ関数に対して引数を与える場合

アセンブラ関数に引数を渡す場合、拡張機能の#pragma PARAMETER を使用します。

#pragma PARAMETER は、32bit 汎用レジスタ(R2R0、R3R1)、16bit 汎用レジスタ(R0、R1、R2、R3)、8bit 汎用レジスタ(R0L、R0H、R1L、R1H)、及び、アドレスレジスタ(A0、A1) を介して、アセンブラ関数に引数を渡します。

#pragma PARAMETER でアセンブラ関数を呼び出す手順を以下に示します。

(1)　#pragma PARAMETER でアセンブラ関数の引数リストに使用するレジスタ名を宣言します。

(2)　#pragma PARAMETER 宣言を記述してから、アセンブラ関数のプロトタイプ宣言を行います。(C プログラムとしてコンパイルする場合のみ、#pragma PARAMETER 宣言をする前にプロトタイプ宣言をしてもよいです。)

【図 3.15】に#pragma PARAMETERを使用したアセンブラ関数asm_funcを呼び出すときの記述例を示します。

```
#pragma PARAMETER asm_func(R0, R1)
extern unsigned int asm_func(unsigned int, unsigned int);     ← 引数を R0、R1 レジスタを介して
                                                                アセンブラ関数に渡します
void        main(void)
{
            int         i = 0x02;
            int         j = 0x05;

            asm_func(i, j);
}
```

図3.15　引数がある場合のアセンブラ関数の呼び出し例 (smp2.c)

```
                .SECTION program,CODE,ALIGN
                ._file      'sample.c'
                ._line      5
;## # C_SRC :    {
                .glb        _main
_main:
                enter       #04H
                ._line      6
;## # C_SRC :                int         i = 0x02;
                mov.w       #0002H,-2 [FB]      ;   i
                ._line      7
;## # C_SRC :                int         j = 0x05;
                mov.w       #0005H,-4 FB]       ;   j
                ._line      9
;## # C_SRC :                asm_func(i, j);
                mov.w       -4 [FB],R1  ;   j               ← 引数を R0、R1 レジスタを介して
                mov.w       -2 [FB],R0  ;   i                  アセンブラ関数に渡しています
                jsr         _asm_func
                ._line      10
;## # C_SRC :    }
                exitd
E1:
                .align
                .glb        _asm_func           ← アセンブラ関数の呼び出し('_'を付加しています)
                .END                               #pragma PARAMETER で宣言した関数は常に'_'が付加されます。
```

図3.16　smp2.c のコンパイル結果 (抜粋) (smp2.a30)

c.　#pragma PARAMETER 宣言における引数型及び戻り値型の制限

#pragma PARAMETER 宣言で以下の引数の型は宣言することはできません。

- 構造体型、共用体型の引数
- 64bit 整数型(long long 型)の引数
- 倍精度浮動小数点型(double 型)の引数、もしくは long double 型の引数

また、アセンブラ関数の戻り値として構造体型、共用体型の戻り値は定義できません。

3.3.2 アセンブラ関数の記述方法

a. 呼び出されるアセンブラ関数の記述方法

アセンブラ関数の入り口処理、および出口処理の記述手順を以下に示します。

(1) アセンブラの指示命令".SECTION"でセクション名を指定します。
(2) 関数名ラベルをアセンブラの指示命令".GLB"でグローバル指定します。
(3) 関数名に_(アンダースコア)を付加して、ラベルとして記述します。
(4) 関数内で B 及び U フラグを変更する場合は、フラグレジスタをスタック上に退避してください。
(5) 関数内で B 及び U フラグを変更した場合は、スタックからフラグレジスタを復帰してください。
(6) RTS 命令を記述します。

SB、FB レジスタの内容を書き換える場合は、関数の入口でスタックに退避し、関数の出口でスタックから復帰してください。ただし、SB、FB レジスタの内容を書き換えるには、アセンブラ関数入口から出口のすべての経路で、SB、FB レジスタを書き換えても影響がないことを確認する必要があります。

【図 3.17】にアセンブラ関数の記述例を示します。この例では、セクション名を本コンパイラが出力するセクション名と同じprogramを用いています。

```
                .section    program, align          ← (1)
                .glb        _asm_func, SYM1         ← (2)
_asm_func:                                          ← (3)
                pushc       FLG                     ← (4)
                mov.w       SYM1, R1
                mov.w       SYM1+2,R3
                popc        FLG                     ← (5)
                rts                                 ← (6)
                .END
```

図3.17 アセンブラ関数の記述例

b. アセンブラ関数からの戻り値の返し方

アセンブラ関数からC言語プログラムに値を返す場合、整数型、ポインタ型、浮動小数点型については、レジスタ渡しで戻り値を返すことができます。【表 3.2】に戻り値に関する呼び出し規則を、【図 3.18】に戻り値を返すアセンブラ関数の記述例を示します。

表3.2　戻り値に関する呼び出し規則

戻り値の型	規則
_Bool 型 char 型	R0L レジスタ
int 型 near ポインタ型	R0 レジスタ
float 型 long 型 far ポインタ型	下位 16 ビットは R0 レジスタに、上位 16 ビットは R2 レジスタに格納して返します。
double 型 long double 型	R3、R2、R1、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
long long 型	R3、R1、R2、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
構造体型 共用体型 クラス型	呼び出しを行う直前に、戻り値を格納するための領域を指す far アドレスをスタックに積みます。呼び出された関数はリターンする前にスタックに積まれた far アドレスで指す領域に戻り値を書き込みます。

```
        .section    program
        .glb        _asm_func
_asm_func:
            :
        (省略)
            :
        mov.w       #0001H, R2
        mov.w       #0A000H,R0
        rts
        .END
```

図3.18　long 型の戻り値を返すアセンブラ関数の記述例

c. C/C++言語の変数の参照方法

アセンブラ関数はC/C++言語プログラムとは別のファイルに記述するため、C/C++言語の大域変数のみ参照することができます。

C/C++言語の変数名をアセンブラ関数内で記述するときは、変数名の前に_(アンダースコア)を付加します。また、アセンブリ言語プログラムでは外部参照する変数をアセンブラの指示命令.GLB で外部参照宣言する必要があります。

【図 3.19】にC言語プログラムの大域変数counter をアセンブラ関数asm_func内で参照する例を示します。

```
C 言語プログラム:

unsigned int        counter;          ← C 言語プログラムの大域変数

void        main(void)
{
                :
            (省略)
                :
}


アセンブラ関数:

            .glb        _counter      ← C 言語プログラムの大域変数を外部参照宣言
_asm_func:
                :
            (省略)
                :
            mov.w       _counter, R0  ← 参照
```

図3.19　C 言語の大域変数の参照方法

d. 割り込み処理をアセンブラ関数で記述するときの注意事項

割り込み処理を実行するプログラム(関数)では、出入り口で以下の処理を行う必要があります。

(1)　関数の入口でレジスタ(R0、R1、R2、R3、A0、A1、FB)を一括に退避します。
(2)　関数の出口でレジスタ(R0、R1、R2、R3、A0、A1、FB)を一括に復帰します。
(3)　関数からのリターンにREIT 命令を使用します。

【図 3.20】に割り込み処理のアセンブラ関数の記述例を示します。

```
            .section    program
            .glb        _func
_int_func:
            pushm       R0,R1,R2,R3,A0,A1,FB      ← レジスタの一括退避
            mov.b       #01H, R0L
                :
            (省略)
                :
            popm        R0,R1,R2,R3,A0,A1,FB      ← レジスタの一括復帰
            reit                                  ← C 言語プログラムへリターン
            .END
```

図3.20　割り込み処理のアセンブラ関数の記述例

e. アセンブラから C/C++言語関数を呼び出すときの注意事項

アセンブリ言語プログラムから C/C++言語で記述された関数を呼び出す場合は、以下の点に注意してください。

(1) C/C++言語の関数名に_(アンダースコア)あるいは$(ダラー)を付加したラベル名で呼び出してください。

(2) C/C++言語の関数は、呼び出された時点のレジスタの内容は退避しません。アセンブリ言語プログラムから C/C++言語の関数を呼び出す場合、その前にデータレジスタおよびアドレスレジスタを退避してください。

(3) C++言語の関数は、「extern "C"」を宣言してください。

```
extern "C" void foo( void )
{
        ..............
}
```

3.3.3 アセンブラ関数の記述に関する注意事項

C/C++言語で記述された関数から呼び出すアセンブリ言語の関数(サブルーチン)を記述する場合、以下の点に注意してください。

a. B、U フラグの取り扱いに関する注意事項

アセンブラ関数から C/C++言語プログラムにリターンするときは、必ず B フラグ及び U フラグを呼び出し時と同じ状態にしてください。

b. FB レジスタの取り扱いに関する注意事項

アセンブラ関数の中で FB(フレームベースレジスタ)の値を変更した場合、呼び出し元の C/C++言語プログラムへ正常に復帰できなくなります。システムの設計上やむをえず変更する場合は、関数の先頭でスタックに退避して、リターンするときに復帰させてください。

c. 汎用レジスタ及びアドレスレジスタの取り扱いに関する注意事項

アセンブラ関数の中で汎用レジスタ(R0 、R1、R2、R3)及びアドレスレジスタ(A0、A1)の内容を変更しても問題ありません。

d. アセンブラ関数への引数に関する注意事項

アセンブリ言語で記述した関数に対して引数を渡す場合、#pragma PARAMETER機能を使用してその引数をレジスタを介して渡すことができます。その書式を【図 3.21】に示す(図中のasm_func はアセンブラ関数名です)。

```
#pragma PARAMETER asm_func(R0, R1)
unsigned int near asm_func(unsigned int, unsigned int);          ← アセンブラ関数のプロトタイプ宣言
```

図3.21　アセンブラ関数の記述例

#pragma PARAMETER は、16 ビット汎用レジスタ(R0、R1、R2、R3)、8 ビット汎用レジスタ(R0L、R0H、R1L、R1H)及びアドレスレジスタ(A0、A1)を介してアセンブラ関数に引数を渡します。また、16 ビット汎用レジスタ及びアドレスレジスタを組み合わせて 32 ビットレジスタ(R3R1、R2R0、A1A0)としてアセンブラ関数に引数を渡します。なお、アセンブラ関数のプロトタイプ宣言の前に#pragma PARAMETER 宣言を行って下さい(C プログラムとしてコンパイルしない場合、#pragma PARAMETER 宣言の前にアセンブラ関数のプロトタイプ宣言を行っても効果がありません。)

ただし、#pragma PARAMETER 宣言で以下の引数の型は宣言することはできません。

- 構造体型、共用体型の引数
- 64bit 整数型(long long 型)の引数
- 倍精度浮動小数点型(double 型)の引数、もしくは long double 型の引数

また、アセンブラ関数の戻り値として構造体型、共用体型の戻り値は定義できません。

3.4 その他

3.4.1 NC シリーズコンパイラ間の移植に関する注意事項

本コンパイラは、弊社製 C コンパイラ「NCxx」と言語仕様レベル(拡張機能を含む)で基本的に互換性を有しています。ただし、以下の点について異なりますのでご注意ください。

a. near / far のデフォルトの違い

NCシリーズのnear/farのデフォルトは、以下の【表 3.3】通りとなっています。このため、移植時はnear/far指定の調整を必要とする場合があります。

表3.3 NC シリーズの near / far デフォルト

コンパイラ	RAM データ	ROM データ	プログラム
NC308	near (ただし、ポインタ型は far)	far	far 固定
NC30	near	far	far 固定
NC30 (R8C)	near 固定	near 固定	far 固定
NC30 (R8CE)	near	far	far 固定
NC79	near	near	far
NC77	near	near	far

付録A コマンドオプションリファレンス

付録Aでは、本コンパイラのコンパイルドライバの起動方法と起動オプションの機能を説明します。起動オプションの説明では、本コンパイラから起動できるアセンブラとリンケージエディタの起動オプションを併せて記載しています。

A.1 コンパイルドライバの入力書式

```
% nc30△[起動オプション]△<[アセンブラソースファイル名]△
                        [オブジェクトファイル名]△[C/C++言語ソースファイル名]>

% : プロンプトを示します。
< > : 必須項目を示します。
[ ] : 必要に応じて記述する項目を示します。
△ : スペースを示します。
```

図A.1 コンパイルドライバの入力書式

```
% nc30  -osample  sample.c<RET>

<RET> : リターンキーの入力を示します。
```

図A.2 コンパイルドライバの入力例

A.2 起動オプション

A.2.1 コンパイルドライバの制御に関するオプション

-c

コンパイルドライバの制御

機　　能: オブジェクトファイル (拡張子.obj)を作成し、処理を終了します。

注意事項: 本オプションを選択したときは、アブソリュートファイルは生成されません。

-D *識別子名*

コンパイルドライバの制御

機　　能: プリプロセスコマンドの#define と同じ機能です。
複数の識別子を指定することもできます。

補足説明: -D オプションに複数の識別子 mac1 と mac2 を指定する例を以下に示します。
% nc30 -Dmac1=1 -Dmac2=2 sample.c<RET>
%: プロンプトを示します。
<RET>:リターンキーの入力を示します。

書　　式: nc30△-D 識別子名[=定数]△<C/C++言語ソースファイル名>
※[=定数]は省略できます。

注意事項: 定義できる識別子の数は、使用しているホストマシンの OS のコマンドラインの最大文字数に制限されることがあります。

-dsource -dS

コメントオプション

機　　能: アセンブラソースファイル(拡張子".a30")を生成します(アセンブル後も削除しません)。

補足説明: 本オプションで生成されたアセンブラソースファイルをアセンブルしないでください。

-dsource_in_list -dSL

リストファイルオプション

機　　能: アセンブラリストファイル(拡張子".lst")を生成します。

-E

コンパイルドライバの制御

機　　能:　プリプロセスコマンドのみを処理し結果を標準出力に出力します。

注意事項:　本オプションを選択したときは、アセンブラソースファイル(拡張子.a30)、オブジェクトファイル(拡張子.obj)、アブソリュートファイル(拡張子.abs)等は生成されません。

-I *ディレクトリ名*

コンパイルドライバの制御

機　　能:　プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。

補足説明:　-I オプションに複数のディレクトリ dir1 と dir2 を指定する例を以下に示します。
% nc30 -Idir1 -Idir2 sample.c<RET>
%: プロンプトを示します。
<RET>:リターンキーの入力を示します。

書　　式:　nc30△-I ディレクトリ名△<C/C++言語ソースファイル名>

注意事項:　指定できるディレクトリ名の数は、使用しているホストマシン OS のコマンドライン最大文字数により制限されることがあります。

-P

コンパイルドライバの制御

機　　能:　プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成し処理を終了します。

注意事項:　(1)　本オプションを選択したときは、アセンブラソースファイル(拡張子.a30)、オブジェクトファイル(拡張子.obj)、アブソリュートファイル(拡張子.abs)等は生成されません。
(2)　本オプションにより生成されるファイル(拡張子.i)には、プリプロセッサが生成する #line は含まれません。#line を含む結果を得る場合は、-E オプションを選択し、リダイレクトしてください。

-S

コンパイルドライバの制御

機　　能:　アセンブラソースファイル(拡張子.a30)を作成します。オブジェクトファイルは生成しません。

注意事項:　本オプションを選択したときは、オブジェクトファイル(拡張子.obj)、アブソリュートファイル(拡張子.abs)等は生成されません。
テンプレート関数を static 関数として.a30 ファイル中に出力します。
本オプションで生成されたアセンブラソースファイルをアセンブルすると、C/C++言語レベルのデバッグ情報が失われますので、注意してください。

-silent

コンパイルドライバの制御

機　　能: 起動時のコピーライトメッセージを出力しません。

-U *プリデファインドマクロ名*

コンパイルドライバの制御

機　　能: プリデファインドマクロ定数を未定義にします。

書　　式: nc30△-U プリデファインドマクロ名△<C/C++言語ソースファイル名>

注意事項: NC30, M16C プリデファインドマクロを未定義にすることができます。

-lang

コンパイルドライバの制御

機　　能: ソースファイルの言語を指定します。
-lang=c オプション指定時は、C (C89)言語ソースファイルとしてコンパイルします。
-lang=cpp オプション指定時は、C++言語ソースファイルとしてコンパイルします。
-lang=ecpp オプション指定時は、Embedded C++言語ソースファイルとしてコンパイルします。
本オプション省略時は、拡張子が.cpp,.cc,.cp の時には C++言語ソースファイルとしてコンパイルします。拡張子が.c の時には C(C89)言語ソースファイルとしてコンパイルします。拡張子が.a30 の場合は本オプションの指定にかかわらずアセンブラソースファイルとして扱います。

書　　式: nc30△-lang=c△<C/C++言語ソースファイル名>
nc30△-lang=cpp△<C/C++言語ソースファイル名>
nc30△-lang=ecpp△<C/C++言語ソースファイル名>

注意事項: 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、多重継承、仮想基底クラスをサポートしていません。これらを記述した場合、エラーメッセージを出力します。
EC++ライブラリを使用する場合は、必ず-lang=ecpp オプションを指定してください。

-preinclude

コンパイルドライバの制御

機　　能: 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。
オプション指定フォルダが複数ある場合、左に指定したものから順に検索を行います。

書　　式: nc30△-preinclude=<ファイル名>[,・・・]△<C/C++言語ソースファイル名>

注意事項: 本オプションを複数回指定した場合、指定した全てのファイルが取り込み対象となります。

-exception, -noexception

コンパイルドライバの制御

機　　能: -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。
本オプションの省略時解釈は、-noexception です。

書　　式: nc30△-exception△<C++言語ソースファイル名>
nc30△-noexception△<C++言語ソースファイル名>

注意事項: 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。
- 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 オプションは C++コンパイル時にのみ指定できます。-lang=cpp の指定がなく、かつ入力ファイルの拡張子が.c または .i の場合、-exception オプションは指定できません。指定するとウォーニングになります。

-rtti

コンパイルドライバの制御

機　　能: 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。
本オプションの省略時解釈は、-rtti=off です。

書　　式: nc30△-rtti=on△<C++言語ソースファイル名>
nc30△-rtti=off△<C++言語ソースファイル名>

注意事項: 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(.obj)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(.rel)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。
-rtti=on は、C++コンパイル時にのみ指定できます。-lang=cpp の指定がなく、かつ入力ファイルの拡張子が.c または .i の場合、-rtti=on は指定できません。指定するとウォーニングになります。

A.2.2 出力ファイル指定オプション

-dir *ディレクトリ名*

出力ファイル指定

機　　能: 出力ファイルの出力先ディレクトリ名を指定できます。

書　　式: nc30△-dir ディレクトリ名△<C/C++言語ソースファイル名>

注意事項: デバッグのためのソースファイル情報は、コンパイラを起動したディレクトリ(カレントディレクトリ)を起点として生成されます。
このため、異なるディレクトリに出力ファイルを生成した場合、デバッガ等にコンパイラを起動したディレクトリを通知する必要があります。

-o *ファイル名*

出力ファイル指定

機　　能: optlnk が生成するファイルを指定します。また、ディレクトリ名を含んだパス名も指定できます。ファイルの拡張子は必ず省略してください。
-dir と -o を両方指定し、-o にディレクトリを含む場合は、-dir の指定に関わらず、 -o で指定したパスにファイルを出力します。

書　　式: nc30△-o ファイル名△<C/C++言語ソースファイル名>

A.2.3 バージョン情報及びコマンドライン表示オプション

-v

コマンドプログラム名の表示

機　　能: 内部で実行されるコマンドプログラム名を表示しながらコンパイルを実行します。

注意事項: 本オプションは、小文字のｖを記述します。

-V

バージョン情報の表示

機　　能: コンパイラの内部で実行される各コマンドプログラムのバージョン情報を表示し、処理を終了します。

補足説明: 本オプションはコンパイラが正常にインストールされたか否かを確認するために使用します。コンパイラ内部で実行される各コマンドの正しいバージョン番号はリリースノートに記載しています。
リリースノートに記載されているバージョン番号と、本オプションの表示内容が異なる場合、インストールが正常に行われていない可能性があります。

注意事項: (1) 本オプションは、大文字のＶを記述します。
(2) このオプションを選択した場合、他のオプションはすべて無効になります。

A.2.4 デバッグ用オプション

-g

デバッグ情報の出力

機　　能: デバッグ情報をオブジェクトファイルに出力します。

注意事項: C/C++言語レベルデバッグを行なう場合は、必ず指定してください。本オプションを選択しても、コンパイラの生成コードには影響を与えません。
-finfo オプション指定時は、-g も有効になります。
-fSB_auto(-fSBA)オプション指定時は、-g も有効になります。

-genter

enter 命令の出力

機　　能: 関数呼び出し時に必ず enter 命令を出力します。

注意事項:
(1) デバッガのスタックトレース機能を使用するときには必ず本オプションを選択してください。指定しない場合は、正しい結果が得られません。
(2) 本オプションを選択した場合、必要性の有無にかかわらず関数の入口で enter 命令を使用してスタックフレームを構築するコードを生成します。従いまして、ROM 容量及び使用するスタック容量が増加する可能性があります。

-gno_reg

レジスタ変数に対するデバッグ情報の抑止

機　　能: レジスタ変数に対するデバッグ情報の出力を抑止します。

注意事項: レジスタ変数に対するデバッグ情報が必要でない場合は本オプションを選択して、レジスタ変数に対するデバッグ情報の出力を抑止して下さい。デバッガへのダウンロードの高速化が期待できます。

A.2.5 最適化オプション

主な最適化オプションの効果を【表A.1】に示します

表A.1 最適化オプション効果一覧表

効果	-O	-OR	-OS	-OSA	-OSFA
速度	良	悪	良	良	良
ROM 容量	良	良	悪	良	同（注）
消費スタック	良	悪	同	悪	悪

良: 良く(もしくは同じ)なることを意味します。
悪: 悪く(もしくは同じ)なることを意味します。
同: 変化が無いことを意味します。
（注） スタックフレームを持たない関数が多い場合、コードサイズが増加します。

-O[1-5]

最適化

機　　能:　速度及びROM容量ともに効果のある最適化を行います。本オプションは、-g オプションと同時に指定することができます。数字(レベル)を指定しない場合は、-O3 と同じです。

-O1:　下記のような最適化を実施します
- 変数をレジスタに割り付ける
- 無意味な条件式の削除を行う
- 論理的に実行されないステートメントの削除を行う

-O2:　-O1 と同じです。

-O3:　-O1 の最適化に加え下記の最適化などを行います
- ビット操作をまとめる
- 浮動小数点の定数たたみ込み
- 標準ライブラリ関数のインライン埋め込み

-O4:　-O3 の最適化に加え下記の最適化などを行います
- const 修飾子で宣言した変数の参照を定数に置き換える

-O5:　-O4 の最適化に加え下記の最適化などを行います
- ポインタ及び構造体などのアドレス計算の最適化(-OR 同時指定時)
- ポインタに対する最適化の強化(-OS 同時指定時)

但し以下の条件を満たす場合、正常なコードを出力できない可能性があります。
- 異なるポインタ変数が同時に同じメモリ位置を指す場合。
- それらの変数を同一関数内で使用する場合。

例:

```
int         a = 3;
int         *p = &a;

void        test1(void)
{
            int         b;
            *p = 9;
            a = 10;
            b = *p;                 /* 最適化により"*p"を"9"に置き換えてしまう */
            printf( "b = %d (expect b = 10)¥n",b );
}
```

実行結果:

```
b = 9 (expect =10)
```

-O[1-5]

最適化

注意事項:　SFR 領域のレジスタへの書き込み、読み出しには、ビット操作命令(BTSTC、BTSTS)を、使用することはできません。
本コンパイラでは、最適化オプション(-O5)を使用した場合、アセンブリ言語コードに対して、ビット操作命令(BTSTC、 BTSTS)を生成する場合があります。
以下の例のような記述を行い、最適化オプション(-O5)を使用してコンパイルした場合、割り込み要求ビットの判定が正常に行われず意図しない動作を行います。

最適化オプションを使用してはならない C ソース:

```
#pragma   ADDRESS   TA0IC   0055h          /* M16C/62  タイマ A0 割込み制御レジスタ  */
struct {
            char ILVL : 3;
            char IR : 1;                   /*  割込み要求ビット  */
            char dmy : 4;
} TA0IC;

void        wait_until_IR_is_ON(void)
{
            while (TA0IC.IR == 0)          /* 1  になるまで待つ  */
            {
                        ;
            }
            TA0IC.IR = 0;                  /* 1  になったら  0  に戻す  */
}
```

SFR 領域に対してビット操作命令(BTSTC、BTSTS)が出力されていることが確認されたら、以下のような対策を行った上でコンパイルしてください。いずれの場合も、生成されたコードに問題が無いことを必ず確認してください。

- "-O5"以外の最適化オプションを使用する。
- asm 関数を使用してプログラム中に直接命令を記述する。
- -O5A オプションを追加する。

-OR

最適化

機　　能:　速度は低下する場合がありますが、ROM 容量を重視した最適化を行います。本オプションは、-g オプション、-O オプションと同時に指定することができます。

注意事項:　本オプションを使用した場合、ソース行情報の一部を変更する最適化を行なう可能性があります。このため、デバッグ時に動作が異なって見える場合があります。
ソース行情報を変更したくない場合、-Ono_break_source_debug(-ONBSD)オプションを使用して最適化を抑止してください。

-OS

最適化

機　　能:　ROM 容量は増大する場合がありますが、速度重視の最適化を行います。
本オプションは、-g オプション、-O オプションと同時に指定することができます。

-OR_MAX -ORM

最適化

機　　能: ROM 容量を優先する最適化を行います。

説明詳細:

(1) 以下に示すコンパイルオプションを有効にします。
- -O5
- -OR
- -O5OA
- -goptimize
- -fchar_enumerator (-fCE)
- -fdouble_32 (-fD32)
- -fno_align (-fNA)
- -fno_carry (-fNC)
- -fsmall_array (-fSA)
- -fuse_DIV (-fUD)

(2) 統合開発環境 High-performance Embedded Workshop 上で選択する場合は、「Renesas M16C Standard Toolchain」の「コンパイラ」タブの「サイズとスピード」を有効にし、「ROM サイズを優先する最大限の最適化を行う」を選択します。

注意事項:

(1) ソース行情報の一部を変更する最適化を行なう可能性があります。このため、デバッグ時に動作が異なって見える場合があります。ソース行情報を変更したくない場合は、コンパイルオプション"-Ono_break_source_debug(-ONBSD)"を選択して最適化を抑止してください。

(2) デバッガによっては正しく enum 型を参照できない場合があります。

(3) 関数の定義または宣言時は、原型宣言が必要です。原型宣言がない場合は、不正なコードを生成する場合があります。

(4) double 型に対するデバッグ情報は float 型として扱われます。このため、デバッガ、シミュレータの C ウォッチウィンドウ、グローバルウィンドウ等では、double 型が float 型として表示されます。

(5) malloc 関数等を用いて動的に得たメモリや far 領域に配置した ROM データ等を far 型ポインタ間接でアクセスする場合には、64K バイトの境界をまたがってアクセスしないように注意してください。

(6) コンパイルオプション"-R8C" または"-R8CE"と併用して選択した場合は、コンパイルオプション"-fno_carry(-fNC)"の機能は無効です。

(7) 除算結果がオーバーフローした場合は、ANSI の規定とは異なる動作になります。

(8) 本オプションを指定した場合、-fno_align(-fNA)を付けてライブラリジェネレータで生成した標準ライブラリを使用して下さい。

-OS_MAX -OSM
最適化

機　　能: サイクル数を優先する最適化を行います。

説明詳細:

(1) 以下に示すコンパイルオプションを有効にします。
- -O4
- -OS
- -Oforward_function_to_inline(-OFFTI)
- -goptimize
- -Oloop_unroll=10 (-OLU=10)
- -Ostatic_to_inline (-OSTI)
- -Osp_adjust(-OSA)
- -fchar_enumerator (-fCE)
- -fdouble_32 (-fD32)
- -fno_carry (-fNC)
- -fsmall_array (-fSA)
- -fuse_DIV (-fUD)

(2) 統合開発環境 High-performance Embedded Workshop 上で選択する場合は、「Renesas M16C Standard Toolchain」の「コンパイラ」タブの「サイズとスピード」を有効にし、「速度を重視する最大限の最適化を行う」を選択します。

注意事項:

(1) for 文を展開するため ROM 容量は増加します。
(2) inline 関数扱いになった static 関数の実体の記述に対するアセンブリ言語コードが生成されます。
(3) 関数を強制的に inline 関数扱いする場合は、inline 宣言してください。
(4) デバッガによっては正しく enum 型を参照できない場合があります。
(5) 関数の定義または宣言時は、原型宣言が必要です。原型宣言がない場合は、不正なコードを生成する場合があります。
(6) double 型に対するデバッグ情報は float 型として扱われます。このため、デバッガ、シミュレータの C ウォッチウィンドウ、グローバルウィンドウ等では、double 型が float 型として表示されます。
(7) malloc 関数等を用いて動的に得たメモリや far 領域に配置した ROM データ等を far 型ポインタ間接でアクセスする場合には、64K バイトの境界をまたがってアクセスしないように注意してください。
(8) コンパイルオプション"-R8C"または"-R8CE"と併用して選択した場合は、コンパイルオプション"-fno_carry(-fNC)"の機能は無効です。
(9) 除算結果がオーバーフローした場合は、ANSI の規定とは異なる動作になります。
(10) インライン関数の宣言と、インライン関数の実体定義は、同一ファイル内に記述してください。
(11) インライン関数の引数には、構造体や共用体を使用する事はできません。これらを使用した場合、コンパイルエラーとなります。
(12) インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合、コンパイルエラーとなります。
(13) インライン関数の再帰呼出しをすることはできません。再帰呼出しの記述を行った場合、コンパイルエラーとなります。

-Ocompare_byte_to_word　-OCBTW

最適化

機　　能:　連続した領域のバイト単位の比較をワードで行います。

注意事項:　-O[1-5](もしくは、-OR,-OR_MAX(-ORM),-OS,-OS_MAX(-OSM))オプションを選択したときのみ効果があります。

-Oconst　-OC

最適化

機　　能:　const 修飾子で宣言した、変数の参照を定数に置き換える最適化を行います。-O4 オプション以上の指定時にも有効になります。ただし、変数の領域は確保します。

補足説明:　以下の条件を同時に満たした場合に最適化を行います。

(1)　ビットフィールドおよび、共用体を除く変数
(2)　const 修飾子を指定し、かつ volatile 指定をしていない変数
(3)　同一の C 言語ソースファイル中で初期化を記述している外部変数
(4)　定数または、const 修飾子を指定された変数で初期化している変数

-Oforward_function_to_inline　-OFFTI

最適化

機　　能:　全てのインライン関数に対して、インライン展開を行います。

補足説明:　インライン関数の呼び出しとその実体定義は、インライン関数を呼び出す前に、インライン関数の実体定義を行わなければ成りませんが、本オプションを使用する事により、インライン関数を呼び出した後に、インライン関数の実体定義を行う事ができます。

注意事項:

(1)　インライン関数の宣言と、インライン関数の実体定義は、同一ファイル内に記述してください。
(2)　インライン関数の引数には、構造体や共用体を使用する事はできません。これらを使用した場合、コンパイルエラーとなります。
(3)　インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合、コンパイルエラーとなります。
(4)　インライン関数の再帰呼出しをすることはできません。再帰呼出しの記述を行った場合、コンパイルエラーとなります。
(5)　クラス内定義関数をインライン展開する場合は、本オプションが必要です。

-Oloop_unroll[=*ループ回数*]　-OLU[=*ループ回数*]

ループの展開

機　　能:　ループ文を回さずにループ回数分コードを展開します。"ループ回数"は省略可能、省略時は最大 5 回のループ文が対象となります。

補足説明:　実行回数が明確である for 文 に対してのみ展開したコードを出力します。
for 展開を行う際に対象とする for の回転数の上限を指定します。デフォルトでは、5 回転以下の for 文が対象となります。

注意事項:　for 文を展開するため、ROM 容量は増加します。

-Ono_asmopt　-ONA

アセンブラオプティマイザの抑止

機　　能:　アセンブラオプティマイザ"aopt30"による最適化を抑止します。

-Ono_bit　-ONB

最適化の抑止

機　　能:　ビット操作をまとめる最適化を抑止します。

補足説明:　-O[3～5]、-OR,-OS,-OR_MAX(-ORM)もしくは-OS_MAX(-OSM) オプションを選択した場合は、同じメモリ領域に配置されたビットフィールドに対して連続に定数を代入する操作を 1 つの操作にまとめる最適化を行います。
入出力等のビットフィールドにおいて連続するビット操作に順序がある場合この最適化は望ましくありませんので、本オプションを使用して最適化を抑止してください。

-Ono_break_source_debug　-ONBSD

最適化の抑止

機　　能:　ソース行情報に影響する最適化を抑止します。

補足説明:　-O[3～5]、-OR もしくは-OR_MAX(-ORM) オプション指定には、ソース行情報に影響する最適化を行う可能性があります。本オプションは、ソース行情報に影響する最適化を抑止する場合に使用します。

-Ono_float_const_fold　　-ONFCF

最適化の抑止

機　　能:　浮動小数点の定数畳み込み処理を抑止します。

補足説明:　本コンパイラでは、デフォルトで定数の畳み込み処理を行います。定数の畳み込み処理の例を以下に示します。

最適化前:

```
(val/1000e250)*50.0
```

最適化後:

```
val/20e250
```

この場合に、浮動小数点のダイナミックレンジ全体を使用したアプリケーションでは、計算順序を換えることにより計算結果が異なる場合があります。本オプションは、浮動小数点における定数の畳み込みを抑止し、C ソースに記述した計算順序を保証します。
本機能は、C プログラムとしてコンパイルする場合のみ有効です。

-Ono_logical_or_combine　　-ONLOC

最適化の抑止

機　　能:　論理 OR をまとめる最適化を抑止します。

補足説明:　下記の例のように、-O3 以上、-OR、-OS のいずれかを指定してコンパイルした場合、論理 OR をまとめる最適化を行います。

例:

```
if( a & 0x01 || a & 0x02 || a & 0x04 )
        ↓ (最適化)
if( a & 0x07 )
```

この場合、変数 a に対して最大 3 回の参照が行われますが、最適化後は 1 回の参照になります。しかし、変数 a が、I/O のときのように参照に意味がある場合、正しい動作が行われない可能性があります。この場合は、本オプションを選択して論理 OR をまとめる最適化を抑止してください。
なお、変数に volatile 宣言がされている場合は、論理 OR をまとめる最適化は行われません。

-Ono_stdlib　　-ONS

最適化の抑止

機　　能:　標準ライブラリ関数のインライン埋め込み、ライブラリ関数の変更等の最適化を抑止します。

補足説明:　本オプションは、以下の最適化を抑止します。

- strcpy()、memcpy() 等の標準ライブラリ関数を SMOVF 命令等に置き換える、最適化。
- 引数の near / far に応じたライブラリ関数に変更する最適化。
- -fdouble_32(-fD32)を使用する場合に数学関数ライブラリを変更する最適化。

注意事項:　標準ライブラリ関数と同名の関数をユーザー側で作成する時に、本オプションを選択する必要がある場合があります。

-Osp_adjust　　-OSA

スタック補正コードをまとめる

機　　能:　関数呼び出し後のスタック補正コードをまとめる最適化を行います。

補足説明:　通常は関数呼び出し毎に、関数の引数の領域を解放するために、スタックポインタを補正する処理をします。本オプションを使用することにより、このスタックポインタの補正を関数の呼出し毎ではなく、まとめて行うようにします。

例:

下記の場合、func1()、 func2() それぞれの呼び出し毎にスタックポインタの補正(2 回の補正)が行われるが、本オプションを使用した場合は 1 回の補正となる。

```
long        func1(long, long);
long        func2(long);

void        main( void ) {
            long        i = 1;
            long        j = 2;
            long        k,n;
            k = func1( i, j );
            n = func2( k );
}
```

注意事項:　オプション-Osp_adjust により ROM 容量を削減し、かつ速度を向上することができます。ただし、使用するスタック量が多くなる可能性があります。
-O[1-5], -OR, -OS のうちのいずれかを同時に指定する必要があります。

-Ostack_frame_align -OSFA

スタックフレームのアライメント

機　　能: スタックフレームの、偶数アライメントを行います。

補足説明: 偶数サイズの auto 変数が奇数アドレスに配置された場合は、偶数アドレスに配置された場合よりもメモリアクセスが 1 サイクル多く必要になります。本オプションを指定すると、偶数サイズの auto 変数を偶数アドレスに配置するようにアライメントを行う為、メモリアクセスを高速に行うことができます。

注意事項:

(1) 以下の#pragma で指定した関数はアライメントを行いません。
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma ALMHANDLER
- #pragma CYCHANDLER
- #pragma INTERRUPT[1]

(2) スタートアッププログラムでは、必ずスタックポインタの初期値を偶数アドレスに設定してください。

(3) 本オプションは全てのプログラムに対して適用してください。

(4) 本オプションを指定した場合、本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用して下さい。

[1] 割り込みが発生したタイミングでのスタックポインタの値が偶数である保証がないため割り込み関数に対しては、アライメントを行ないません。このため、割り込み関数から呼ばれる関数に本オプションを指定した場合には、逆に処理速度が遅くなる可能性があります。

-Ostatic_to_inline　-OSTI

static 関数を inline 関数扱いにする

機　　能: static 宣言された関数(static 関数)を、inline 宣言されている関数(inline 関数)として扱い、inline 展開したアセンブリ言語コードを生成します。

補足説明: 以下の条件を満たした場合、static 関数を inline 関数として扱い、inline 展開したアセンブリ言語を生成します。

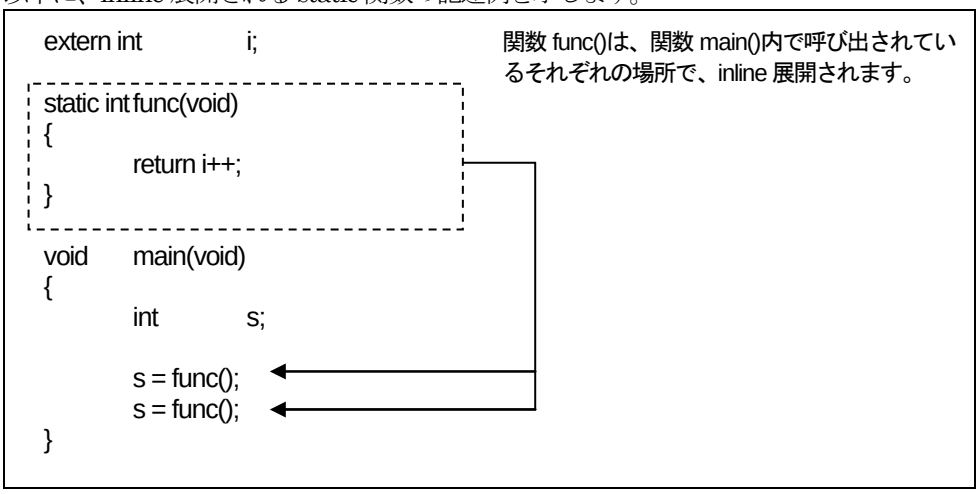
(1) 関数呼び出しの前に、実体が記述されている static 関数を対象とします。
- 関数の呼び出しと、その関数の実体が、同じソースファイル内に記述されていなければなりません。
- "-Oforward_function_to_inline"オプションを選択した場合は、本条件を無視してください。

(2) 対象となる static 関数に対して、プログラム中でアドレス取得を行っていない場合。

(3) 対象となる static 関数を再帰呼び出ししていない場合。

(4) コンパイラのアセンブリ言語コード出力において、フレーム(auto 変数等の確保)の構築が行われない場合。
- 対象となる関数の記述内容、別の最適化オプションとの併用により、フレーム構築の有無の状況は異なります。
- "-Oforward_function_to_inline"オプションを選択した場合は、本条件を無視してください。

以下に、inline 展開される static 関数の記述例を示します。



注意事項:

(1) inline 関数扱いになった static 関数の実体の記述に対するアセンブリ言語コードは、常に生成されます。

(2) 関数を強制的に inline 関数扱いする場合は、inline 宣言してください。

(3) クラス内定義関数をインライン展開する場合は、本オプションが必要です。

-O5OA

最適化の抑止

機　　能: 最適化オプション"-O5"選択時におけるビット操作命令(BTSTC、 BTSTS)を使用したコード生成を抑止します。

注意事項: SFR 領域のレジスタへの書き込み、読み出しは、ビット操作命令(BTSTC、BTSTS)を使用することができません。最適化オプション"-O5"選択時において、SFR 領域のレジスタへの書き込み、読み出しに対してビット操作命令を使用したコードが生成されている場合は、本オプションを選択してください。

-goptimize

機　　能: モジュール間最適化時に使用する付加情報を、出力ファイル内部に生成します。
本オプションを指定したファイルは、リンク時にモジュール間最適化の対象になります。
-fSB_auto と同時に指定することはできません。

A.2.6 生成コード変更オプション

-fansi

生成コードの変更

機　能: C++言語としてコンパイルする場合、以下に示すオプションを有効にします。
- -fnot_reserve_far_and_near: far、near を予約語として扱いません。

C 言語としてコンパイルする場合、以下に示すオプションを有効にします。
- -fnot_reserve_asm: asm を予約語として扱いません。
- -fnot_reserve_far_and_near: far、near を予約語として扱いません。
- -fnot_reserve_inline: inline を予約語として扱いません。
- -fextend_to_int: char 型データを int 型に拡張して演算を行います。

補足説明: 本オプションを選択することにより、ANSI 規格に基づいたコード生成を行います。
asm と inline は標準 C++のキーワードであるため、C++コンパイルの場合、本オプションの選択の有無にかかわらず、asm と inline をキーワードとします。
C++コンパイルの場合、本オプションの選択の有無にかかわらず、汎整数拡張して演算します。

-fchar_enumerator　-fCE

生成コードの変更

機　能: enumerator(列挙子)の型を int 型ではなく unsigned char 型で扱います。

注意事項: 本オプションを選択した場合、デバッガによっては正しく enum 型を参照できない場合があります。

-fconst_not_ROM　-fCNR

生成コードの変更

機　能: const 修飾子で指定した型を ROM データとして扱いません。

補足説明: デフォルトでは、const 指定したデータは ROM 領域に配置されます。

```
int　const　array[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

上記の場合、配列「array」は、ROM 領域に配置されます。本オプションを指示することにより、この「array」を RAM 領域に配置することができます。
通常の用途では、本オプションを使用する必要はありません。

-fdouble_32 -fD32

生成コードの変更

機　　能: double 型を float 型として処理します。

補足説明:
(1) 本オプションを指定した場合は、必ず、関数原型宣言を行なってください。関数原型宣言がない場合は、不正なコードを生成する場合があります。
(2) 本オプションを選択した場合の double 型に対するデバッグ情報は float 型として扱われます。このため、デバッガ、シミュレータの C ウォッチウィンドウ、グローバルウィンドウ等では float 型として表示されます。
(3) 本機能を使用する場合、C++プログラムにおいて float 型と double 型の多重定義はできません。
(4) 本オプションを付けると-Wnon_prototype が同時に有効になります。
(5) 数学関数を呼び出している場合は、単精度用の数学関数に置き換えて呼び出します。

-fenable_register -fER

生成コードの変更

機　　能: register 記憶クラスを指定した変数をレジスタに割り当てます。

補足説明: 「auto 変数をレジスタに割り当てる最適化」を行った場合、必ずしも最適解を得られるとは限りません。本オプションは上記状況下において、プログラム上でレジスタ割り当てを指示する事により効率を高める手段として用意しています。
本オプションを選択することにより、register 指定された以下の変数を強制的にレジスタに割り当てます。

- 整数型変数
- ポインタ変数

注意事項: むやみに register 指定を行うと逆に効率を低下させる場合があります。必ず生成されたアセンブラソースファイルを確認の上、使用してください。

-fextend_to_int　　-fETI

生成コードの変更

機　　能:　char 型又は signed char 型データを int 型に拡張し演算を行います(ANSI 規格で定められた拡張を行います)。

補足説明:　ANSI 規格では char 型データ又は signed char 型データを評価する時に必ず int 型に拡張します。これは char 型の演算、例えば、c1 = c2 ＊ 2 / c3;を行うときに演算の途中で char 型をオーバーフローし、結果が予期せぬ値になるのを防ぐためです。

```
void        main(void)
{
            char        c1:
            char        c2 = 200;
            char        c3 = 2;

            c1 = c2 * 2 / c3;
}
```

この場合「c2 * 2」の演算で char 型をオーバーフローし、正しい結果を求められません。
本オプションを選択することにより、正しい結果を求めることができます。
デフォルトの設定を int 型への拡張を行わないようにしているのは、ROM 効率を少しでも良くするためです。
C++プログラムとしてコンパイルする場合、本オプションの選択の有無にかかわらず、汎整数拡張して演算します。

-ffar_RAM　　-fFRAM

生成コードの変更

機　　能:　RAM データのデフォルト属性を far 属性にします。

補足説明:　RAM データ(変数)は、デフォルトで near 領域に配置されます。near 領域(64K バイトの領域)の外に RAM データを配置する場合に本オプションを使用します。

注意事項:　-R8C または -R8CE と併用できません。

-finfo

生成コードの変更

機　　能:　utl30 に必要なインスペクタ情報をオブジェクトファイル内へ出力します。

注意事項:
(1) asm 関数内でのグローバル変数の使用はチェックされません。このため、utl30 でも asm 関数の使用は無視されます。
(2) -finfo は-g を含みます。
(3) 本オプションを選択しても、コンパイラの生成コードには影響を与えません。
(4) utl30 が生成するヘッダを読み込まないコンパイラソースに対して本オプションを使用しないで下さい。

-fbit -fB

生成コードの変更

機　　能: near 領域に配置した外部変数全てに対して、絶対アドレッシングを用いて 1 ビット操作命令が使用できると仮定して、コード生成を行います。

補足説明: ビット操作を行なう near 外部変数が、M16C シリーズ、R8C ファミリのメモリ空間 0000h～1FFFh の領域にある場合、本オプションを指定することによりコンパイラが生成するコード効率を向上させることができます。
シングルチップ応用でRAMが上記メモリ空間内にある場合に本オプションを指定すると効果的です。万一上記空間外の変数の操作を行おうとしたときは、リンク時にエラーになります。

-fno_carry -fNC

生成コードの変更

機　　能: far 型ポインタ間接でのデータアクセス時のキャリーフラグの加算を抑止します。

補足説明: far 型ポインタ間接での構造体及び 32 ビットデータアクセスを行う場合、64K バイトの境界をまたがって配置されていない仮定で、far 型ポインタ(32 ビットポインタ)の上位 16 ビットへのキャリー加算を行わないコードを生成します。これにより、効率向上が期待できます。

注意事項: malloc 関数等を用いて動的に得たメモリや far 領域に配置した ROM データ等を far 型ポインタ間接でアクセスする場合には、64K バイトの境界をまたがってアクセスしないように注意してください。
-R8C または -R8CE と併用できません。

-fauto_128 -fA1

生成コードの変更

機　　能: 使用するスタックフレームを最大 128 バイトに制限します。
(スタックフレームのデフォルトでの最大値は、255 バイトです。)

-ffar_pointer　-fFP

生成コードの変更

機　　能:　ポインタ型のデフォルト属性を far 属性とします。

補足説明 :
(1) 本コンパイラでのポインタ型変数は、デフォルトで near 属性です。これを far 属性に変更するときに本オプションを使用します。
(2) ポインタ変数の定義の際に、near 修飾子を記述したポインタ変数は、本オプションに関わらず、near 属性として処理します。

例)

```
char near *p;    // near ポインタとして処理します
```

-fnear_ROM　-fNROM

生成コードの変更

機　　能:　ROM データのデフォルト属性を near 属性にします。

補足説明:　ROM データ(const 指定された変数等)は、デフォルトで far 領域に配置されます。本オプションを選択することにより、ROM データを near 領域に配置することができます。
通常の用途では、本オプションを使用する必要はありません。

-fno_align　-fNA

生成コードの変更

機　　能:　関数の先頭アドレスのアライメントを行いません。

補足説明:　以下のように出力アセンブラを変更します。
- 関数シンボルの前にアセンブラ指示命令.align を出力しません。
- コード属性セクションに対するアセンブラ指示命令.section に align を指定しません。

注意事項:　全てのプログラムを本オプションを用いてコンパイルしてください。
本オプションを指定した場合、本オプションを付けてライブラリジェネレータで生成した標準ライブラリを使用して下さい。

-fno_even　-fNE

生成コードの変更

機　　能: データの出力時に、奇数データと偶数データを分離しないで出力します。
即ち、すべてのデータを奇数セクション(data_NO、data_FO、data_NOI、data_FOI、bss_NO、bss_FO、rom_NO、rom_FO)に配置します。

補足説明: デフォルトでは、奇数サイズデータと偶数サイズデータを別のセクションに出力します。

```
char    c;
int     i;
```

上記の場合、変数「c」と変数「i」は別のセクションに出力されます。これは偶数サイズの変数「i」を偶数アドレスに配置するためです。これにより 16 ビットバス幅でアクセスする時に高速なアクセスが期待できます。
本オプションは、8 ビットバス幅でのみ使用する場合で、かつ、セクション数を減らしたいときに使用します。

注意事項: #pragma SECTION を用いてセクション名を変更した場合は、変更された名前のセクションに配置されます。

-fno_switch_table　-fNST

生成コードの変更

機　　能: switch 文に対して、ジャンプテーブルを用いたコードを生成せず、比較を行ってから分岐するコードを生成します。

補足説明: 本オプションを選択しない場合は、コードサイズがより小さくなる場合のみ、ジャンプテーブルを用いたコードを生成します。

注意事項 : 1 関数のコードサイズが 32K バイトを超えるような大きな関数では、switch 文に対してジャンプテーブルを用いたコードを生成すると、リンクエラーが発生する場合があります。この場合、必ず本オプションを指定してください。

-fnot_address_volatile　-fNAV

生成コードの変更

機　　能: #pragma ADDRESS で指定した変数を、volatile で指定された変数として扱いません。

補足説明: I/O 変数を RAM 上に在る変数と同じ最適化を行うと、期待した動作をしない場合があります。これは、I/O 変数に volatile 指定をすることにより避けることができます。
"#pragma ADDRESS"は、通常、I/O 変数に対して使用するため、volatile 指定が無くても、volatile 指定がされているものとして処理されます。
本オプションは、この処理を抑止します

注意事項: 通常の用途では、本オプションを使用する必要はありません。

-fnot_reserve_asm　-fNRA

生成コードの変更

機　　能: asm を予約語として扱いません。

補足説明: 同じ機能の_asm は予約語として扱われます。
C++プログラムとしてコンパイルする場合、本オプションの選択の有無にかかわらず、常に asm を予約語として扱います。

-fnot_reserve_far_and_near　-fNRFAN

生成コードの変更

機　　能: far、near を予約語として扱いません。

補足説明: 同じ機能の_far、_near は予約語として扱われます。

-fnot_reserve_inline　-fNRI

生成コードの変更

機　　能: inline を予約語として扱いません。

補足説明: 同じ機能の_inline は予約語として扱われます。
C++プログラムとしてコンパイルする場合、本オプションの選択の有無にかかわらず、常に inline をキーワードとして扱います。

-fsmall_array　-fSA

生成コードの変更

機　　能: 総サイズが不明の far 型の配列を参照する場合、その総サイズが 64K バイト以内であると仮定し、添字の計算を 16 ビットで行ないます。

補足説明: デフォルトでは far 型配列の要素を参照する場合に、配列のサイズが不明であれば添字を 32 ビットで計算します。これは配列のサイズが 64K バイト以上の場合に対応するためです。

```
extern int far array[];
int          i = array[j];
```

上記の場合、配列「array」の総サイズがコンパイル場合は分からないので、添字「j」を 32 ビットで計算します。本オプションを選択することにより、配列「array」の総サイズを 64K バイト以下と仮定して、添字「j」を 16 ビットで計算します。この結果処理速度の向上、コードサイズの削減が可能となります。
一つの配列のサイズが 64K バイトを超えないのであれば、常に本オプションを使用することを推奨します。

-fswitch_other_section　-fSOS

生成コードの変更

機　　能: switch 文に対するジャンプテーブルをプログラムセクションとは別のセクションに出力します。

補足説明: セクション名は、switch_table です。

注意事項: 本オプションは通常、使用する必要はありません。

-fchange_bank_always　-fCBA

生成コードの変更

機　　能: 毎回バンクを切り換えるコードを出力します。

補足説明: #pragma EXT4MPTR 機能もしくは、_ext4mptr を使用していて、かつ4Mバイト空間へのポインタ変数を複数宣言する場合には、本オプションを指定してください。

注意事項: -R8C または -R8CE と併用できません。

-fauto_over_255　-fAO2

生成コードの変更

機　　能: 1つの関数で確保可能なスタックフレームサイズを、最大64Kバイトに変更します。
(スタックフレームのデフォルトでの最大値は、255バイトです。)

補足説明: (1) #pragma SBDATA 機能と併用することはできません。#pragma SBDATA の記述があるファイルをコンパイルした場合、下記に示すウォーニングを出力し、#pragma SBDATA の記述を無視します。

compile option -fauto_over_255 is specified,#pragma SBDATA was ignored.
===>#pragma SBDATA xxx;

※スタックフレームの構築に SB レジスタを使用するため、#pragma SBDATA を使用することができません。

(2) 以下に示すファイルに対して本オプションを指定してください。

a.255 バイト以上のスタックフレームを必要とする関数(以下、関数Aと呼ぶ)がある場合
=====>関数A が記述されているファイル

b.関数A を処理している間に、割り込み(以下、割り込みAと呼ぶ)が発生し、その割り込みA から#pragma SBDATA 宣言した変数にアクセスしている場合
=====>割り込みA が記述されているファイル

-fsizet_16 -fS16

型定義のビットサイズ変更

機　　能: 型定義 size_t を unsigned long 型から unsinged int 型に変更します。

補足説明: 本オプションを選択した場合、リンクするライブラリは下記のようになります。

a) 統合開発環境(High-performance Embedded Workshop)にてビルドを行う場合、ライブラリジェネレータが自動的にライブラリを生成し、optlnk がそのライブラリをリンクします。

b) コマンドプロンプトにて nc30.exe を起動し、リンクまで行う場合、nc30.exe は下記のライブラリを自動的にリンクします。

- -R8C オプションを同時に使用した場合　r8cs16.lib
- -R8CE オプションを同時に使用した場合　r8ces16.lib
- 上記以外の場合　nc30s16.lib

c) コマンドプロンプトにて optlnk を個別に起動する場合は、ライブラリジェネレータで生成されたライブラリをリンクさせてください。

-fptrdifft_16 -fP16

型定義のビットサイズ変更

機　　能: 型定義 ptrdiff_t を signed long 型から singed int 型に変更します。

補足説明: 本オプションを選択した場合、リンクするライブラリは下記のようになります。

a) 統合開発環境(High-performance Embedded Workshop)にてビルドを行う場合、ライブラリジェネレータが自動的にライブラリを生成し、optlnk がそのライブラリをリンクします。

b) コマンドプロンプトにて nc30.exe を起動し、リンクまで行う場合、nc30.exe は下記のライブラリを自動的にリンクします。

- -R8C オプションを同時に使用した場合　r8cs16.lib
- -R8CE オプションを同時に使用した場合　r8ces16.lib
- 上記以外の場合　nc30s16.lib

c) コマンドプロンプトにて optlnk を個別に起動する場合は、ライブラリジェネレータで生成されたライブラリをリンクさせてください。

-fuse_DIV -fUD

生成コードの変更

機　　能: 除算に対する生成コードを変更します。

補足説明: 除算時に、被除数が 4 バイト値、除数が 2 バイト値で、かつ、演算結果が 2 バイト値の場合や、被除数が 2 バイト値、除数が 1 バイト値で、かつ、演算結果が 1 バイト値の様な演算を行う場合にマイクロコンピュータの"div.w(divu.w)"及び"div.b(divu.b)"命令を生成します。

注意事項:
(1) 本オプションを選択した場合に、除算結果がオーバーフローすると ANSI の規定とは異なる動作になります。
(2) M16C シリーズ、R8C ファミリの div 命令は演算結果がオーバーフローした場合、結果は不定になります。このため"-fuse_DIV(-fUD)"を選択せずにコンパイルを行った場合は、被除数が 4 バイト、除数が 2 バイトで、かつ、結果が 2 バイトのような場合もランタイムライブラリを呼び出します。

-fuse_MUL -fUM

生成コードの変更

機　　能: 乗算に対する生成コードを変更します。

補足説明 : 16 ビット×16 ビットを 32 ビットに格納する場合、上位 16 ビットの結果を得るためには、乗数もしくは被乗数のどちらか一方を 32 ビットでキャストする必要があります。
本オプションを指定すると、キャストせずに 32 ビットの結果を得ることができます。

-R8C

生成コードの変更

機　　能:　R8C ファミリ MCU に対応したコードを生成します。

補足説明 :　本オプションを指定すると、-fnear_ROM(-fNROM)オプションが、常に指定された状態になります。
本オプションを指定すると、キーワード far と_far が無視されます。
本オプションを選択した場合、リンクするライブラリは下記のようになります。

a) 統合開発環境(High-performance Embedded Workshop)にてビルドを行う場合、ライブラリジェネレータが自動的にライブラリを生成し、optlnk がそのライブラリをリンクします。

b) コマンドプロンプトにて nc30.exe を起動し、リンクまで行う場合、nc30.exe は下記のライブラリを自動的にリンクします。

- -fsizet_16 または-fptrdifft_16 オプションを同時に使用した場合　r8cs16.lib
- 上記以外の場合　r8clib.lib

c) コマンドプロンプトにて optlnk を個別に起動する場合は、ライブラリジェネレータで生成されたライブラリをリンクさせてください。

注意事項:　(1) 以下に示すオプションと本オプションを併用することはできません。
これらのオプションを指定した場合は、無視されます。
-ffar_RAM (-fFRAM)
-fno_carry (-fNC)
-fchange_bank_always (-fCBA)
(2) リンクする全てのプログラムに本オプションを付けてください。

-R8CE

生成コードの変更

機　　能:　R8C ファミリ MCU(ROM64K 以上)に対応したコードを生成します。

注意事項:　(1) 以下に示すオプションと本オプションを併用することはできません。
本オプションを指定した場合は、無視されます。
-ffar_RAM (-fFRAM)
-fno_carry (-fNC)
-fchange_bank_always (-fCBA)
(2) ROM 領域が 64K の境界を越える場合に使用します。
(3) 本オプションを選択した場合、リンクするライブラリは下記のようになります。

a) 統合開発環境(High-performance Embedded Workshop)にてビルドを行う場合、ライブラリジェネレータが自動的にライブラリを生成し、optlnk がそのライブラリをリンクします。

b) コマンドプロンプトにて nc30.exe を起動し、リンクまで行う場合、nc30.exe は下記のライブラリを自動的にリンクします。

- -fsizet_16 または-fptrdifft_16 オプションを同時に使用した場合　r8ces16.lib
- 上記以外の場合　r8celib.lib

c) コマンドプロンプトにて optlnk を個別に起動する場合は、ライブラリジェネレータで生成されたライブラリをリンクさせてください。

(4) リンクする全てのプログラムに本オプションを付けてください。

-fSB_auto　　-fSBA

生成コードの変更

機　　能:　関数単位で SB レジスタを切り替えて SB 相対アドレッシングを生成します。

補足説明:　関数単位で外部変数の参照回数を解析し、最適な SB 相対アドレッシングを生成します。

```
int sym;
int a;
int data;
  :
int b;
  :
int func(void){
      a = x;
      sym = xx;          ──────→ _sym のアドレスを SB 相対の基点とする。
      sym = a*b;
      if( sym != 0)
            sym = sub( );
      return sym;
}

int sub( void)
{
       data1 = sym1;     ──────→ _data1 のアドレスを SB 相対の基点とする。
       data2 = data1/2;
       data1 = sub(data2);
          :
```

(1)　SB 相対の基点となったシンボルのアドレスを SB レジスタへ格納します。
(2)　関数の出入り口で SB レジスタの退避/復帰のコードを生成します。
(3)　外部変数のみ有効です。
(4)　-OR, -OS, -OR_MAX(-ORM),-OS_MAX(-OSM)と併用することはできません。
(5)　本オプションを使用したオブジェクトファイルと以下の機能を使用したオブジェクトファイルをリンクしたプログラムの動作を保証しません。
- #pragma SBDATA
- コンパイラオプション-fauto_over_255(-fAO2)

(6)　-goptimize オプションを同時に指定できません。
(7)　-finfo オプションが有効になります。
(8)　utl30 が生成する SBDATA 宣言ヘッダを読み込むコンパイラソースに対して本オプションを使用しないでください。

A.2.7 ライブラリ指定オプション

-l *ライブラリファイル名*

機　　能: optlnk がリンク時に使用するライブラリファイル名を指定します。ファイルの拡張子は省略可能です。

書　　式: nc30△-l ファイル名△<C/C++言語ソースファイル名>

注意事項:
(1) ファイル指定では拡張子を省略することができます。拡張子を省略した場合のファイルの拡張子は".lib"として処理されます。
(2) ファイルの拡張子を指定する場合は、必ず".lib"を指定してください。
(3) ファイルの検索順序は、カレントフォルダ、環境変数 HLNK_DIR 指定フォルダの順になります。
(4) NC30 は環境変数 LIB30 で指定されたディレクトリ内にあるライブラリ "nc30lib.lib" をデフォルトでリンクします（コンパイルオプション "-R8C", "-R8CE", "-fsizet_16", "-fptrdifft_16" 指定時はそれぞれのオプションの解説をご覧ください）。
(5) 本オプションでライブラリを指定した場合、NC30 がデフォルトでリンクするライブラリ(4)の優先順位が最も低くなります。

A.2.8 警告オプション

-Wall

警告オプション

機　　能: 検出可能なウォーニングをすべて表示します。

補足説明:
(1) "-Wlarge_to_small(-WLTS)"、および"-Wno_used_argument(-WNUA)"、"-Wno_used_static_function(-WNUSF)"を使用した場合のウォーニングは除きます。
(2) オプション"-Wnon_prototype(-WNP)"、"-Wunknown_pragma(-WUP)"、"-Wnesting_comment(-WNC)"、"-Wuninitialize_variable(-WUV)"と同等のウォーニングを表示します。
(3) 以下の場合にも警告を表示します。
- if 文、for 文や、&&、¦¦演算子の比較文に代入演算子"＝"を使用した場合。
- 代入演算子 "＝"を間違って "＝＝"と記述した場合。
- 古い形式の関数定義を行った場合。

注意事項: これらの警告は、コンパイラの判断で誤った記述と推測できる範囲で検出しています。このためすべての誤りを警告できるとは限りません。

-Wccom_max_warnings=*ウォーニング回数*　-WCMW=*ウォーニング回数*

警告オプション

機　　能: コンパイラ本体の出力するウォーニングの回数の上限を指定できます。

補足説明: デフォルトでは、ウォーニングの出力には上限がありません。
本オプションは、多量のウォーニング出力により、画面がスクロールするのを調節する場合等に使用します。

注意事項: ウォーニングの出力の上限回数は、0 回以上で指定してください。また、指定回数の省略はできません。0 回を指定するとウォーニング出力を完全に抑止します。
本機能は、C プログラムとしてコンパイルする場合のみ有効です。

-Wlarge_to_small　-WLTS

警告オプション

機　　能:　大きいサイズから、小さいサイズへの暗黙の代入に対して、ウォーニングを出力します。

補足説明:　各型の負数の境界値では、型に収まる数値であっても、ウォーニングを出力する場合があります。
これは言語規約上、負数は単項演算子（-）と整数が結合したものであるためです。
例えば「-32768」は、signed int 型に収まる値ですが、"-"と"32768"に分解すると、"32768"は signed int 型に収まらないため、"signed long 型"になります。従って即値"-32768"は、signed long 型です。
このため、「 int i = -32768; 」のような記述に対して、ウォーニングが出力されます。

注意事項:　本オプションは、多量のウォーニングを出力するため、以下の型変換のみウォーニング出力を抑止しています。

- char 型変数 から char 型変数への代入
- 即値の char 型変数 への代入
- 即値の float 型変数への代入

-Wnesting_comment　-WNC

警告オプション

機　　能:　コメント内に"/*"を記述している場合にウォーニングを発生します。

補足説明:　本オプションを使用することにより、コメントのネストを検出することができます。

-Wno_stop　-WNS

警告オプション

機　　能:　エラーが発生してもコンパイル作業を停止しません。

補足説明:　コンパイラは関数単位でコンパイルします。コンパイル中にエラーが発生すると、デフォルトでは、次の関数のコンパイルを行いません。
また、エラーが原因で別のエラーを引き起こすことがあり、エラーが多いとコンパイルを停止します。
本オプションを使用することにより、可能な限りコンパイルを続けます。

注意事項:　記述によるエラーが原因で System Error が発生する場合があります。その場合は、本オプションを使用している場合であってもコンパイル作業が停止します。
C++プログラムとしてコンパイルする場合、本オプションの選択の有無にかかわらず、100 個のエラーが出力されると停止します。

コンパイラオプション-Wlarge_to_small(-WLTS)に関する注意事項
コンパイラオプション-Wlarge_to_small(-WLTS)を使用する場合、以下の点にご注意下さい。
(1) C++プログラムとしてコンパイルする場合、右辺が定数の場合のみ警告を出力します。
(2) Cプログラムとしてコンパイルする場合、右辺が変数のみの場合、警告を出力しません。

-Wno_used_argument　-WNUA

警告オプション

機　　能:　引数を持つ関数を定義した場合、使用していない引数に対してウォーニングを出力します。

-Wno_used_function　-WNUF

警告オプション

機　　能:　未使用のグローバル関数を、リンク時に表示します。

注意事項:　リンク時に-msg_unused オプションを指定している場合は、本オプションは必要ありません。
リンカオプション-msg_unused と-message を使用している場合は、本オプションは必要ありません。

-Wno_used_static_function　-WNUSF

警告オプション

機　　能:　コード生成が不要な static 関数名を表示します。条件は下記に示すいずれかの場合です。

- static 関数がファイルのどこからも参照されない。
- "-Ostatic_to_inline(-OSTI) "オプションにより、static 関数が inline 化される。

注意事項:　下記に示すように配列の初期化子に関数名を記述した場合は、プログラム動作時に参照されない関数であってもコンパイラは参照されるものとして処理します。
下記の例では、関数 f4 と f5 は参照されませんが、コンパイラはこれらの関数を参照されるものとして処理します。

例:

```
void        (*a[5])(void) = {f1,f2,f3,f4,f5};

            for(i = 0; i < 3; i++) (*a[i])();
```

-Wno_warning_stdlib　-WNWS

警告オプション

機　　能:　"-Wnon_prototype"や、"-Wall"と同時に本オプションを選択すると、「関数原型宣言されていない標準ライブラリに対する警告」を抑止します。

補足説明:　C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、関数原型宣言がなければメッセージを出します。

-Wnon_prototype -WNP

警告オプション

機　　能: 前もってプロトタイプ宣言がされていない関数を使用した場合、または関数の関数原型宣言を行っていない場合にウォーニングを出します。

補足説明: 関数原型宣言を行うことにより、関数引数をレジスタ渡しにすることができます。レジスタ渡しにすることにより、速度向上、コードサイズ削減が期待できます。
また、関数原型宣言を行うことにより、コンパイラが関数の引数を検査するようになります。このため、プログラムの信頼性向上を期待できます。
したがって、本オプションは、常に使用することを推奨します。
C++プログラムとしてコンパイルする場合、本オプションの有無にかかわらず、関数原型宣言がなければメッセージを出します。

-Wstop_at_link -WSAL

警告オプション

機　　能: リンク時に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。エラーメッセージが出力されると、リンク処理を中断します。

-Wstop_at_warning -WSAW

警告オプション

機　　能: コンパイル時にウォーニングが発生した場合、コンパイルを停止します。

-Wundefined_macro -WUM

警告オプション

機　　能: #if の中で未定義のマクロを使用した場合に警告します。

-Wuninitialize_variable -WUV

警告オプション

機　　能: 初期化されていない auto 変数に対してウォーニングを出力します。
本オプションは、"-Wall"指定時にも有効になります。

補足説明: ユーザーアプリケーションにおいて、if 文、for 文などによる条件分岐の中で初期化される場合、コンパイラは初期化されていないと判断します。
そのため本オプションを使用した場合、ウォーニングが出力されます。

-Wunknown_pragma -WUP

警告オプション

機　　能: サポートしていない#pragma を使用した場合、ウォーニングを出します。

補足説明: デフォルトでは、サポートされていない未知の"#pragma"が使用されていてもウォーニングを出しません。
NC シリーズコンパイラのみを使用する場合、本オプションを使用することにより、"#pragma" のスペルミスなどを発見することができます。

注意事項: NC シリーズコンパイラのみを使用する場合は、本オプションを常に用いてコンパイルすることを推奨します。

A.2.9 アセンブル / リンクオプション

-as30 "*オプション*"

アセンブル / リンクオプション

機　　能: アセンブルコマンド as30 のオプションを選択します。
2 個以上のオプションを選択する場合は、"(ダブルクオーテーション)で囲んでください。

書　　式: nc30△-as30△ "オプション 1△オプション 2"△<C 言語ソースファイル名>

注意事項: as30 の-.、-C、-M、-O、-P、-T、-V および-X オプションは指定しないでください。

-lnkcmd=コマンドファイル名

アセンブル / リンクオプション

機　　能: optlnk にコマンドファイルを指定します。optlnk の-subcommand オプションとして渡されます。

書　　式: nc30△ -lnkcmd=<コマンドファイル名>△<C/C++言語ソースファイル名>
コマンドファイルの書式は、optlnk の -subcomand オプションでの説明を参照ください。

A.3 起動オプションに関する注意事項

A.3.1 起動オプションの記述に関する注意事項

nc30 の起動時オプションは、アルファベットの大文字と小文字を区別します。
誤って入力した場合、そのオプションによる機能は取り消されます。

A.3.2 オプションの優先順位

nc30 の起動時オプション中、

- "-c":オブジェクトファイル(拡張子.obj)を作成して処理を終える
- "-S":アセンブラソースファイル(拡張子.a30)を作成して処理を終える

を同時に指定した場合、-S オプションが優先されます。したがって、このときはアセンブラソースファイルのみが生成されます。

付録B 拡張機能リファレンス

NC30 は、M16C シリーズ、R8C ファミリを用いたシステムへの組み込みを容易にするために独自の拡張機能を追加しています。付録 B では、言語仕様に関する機能以外の拡張機能の使用方法を説明します。

本コンパイラは、標準言語仕様のキーワードに加え、次の語句を拡張キーワードにします。

_asm _far _inline _near asm(C++では標準キーワード) far inline(C++では標準キーワード) near _Bool(C のみ) restrict(C のみ) _ext4mptr(C のみ)

表B.1 拡張機能 (1/2)

拡張機能	機能の内容
near/far 修飾子	データをアクセスするアドレッシングモードを指定します。 near.....64K バイト以内の領域(0H～0FFFFH)のアクセス far.....64K バイトを越える領域(全メモリ領域)のアクセス ● 関数は全て far 属性となります。

asm 関数	(1) C/C++言語プログラム中にアセンブリ言語を直接記述できます。関数外でも記述することができます。
	記述例:
	asm(" MOV.W #0, R0");
	(2) 変数名を指定することができます(関数内のみ記述可能)。
	記述例 1:
	asm(" MOV.W R0, $$[FB]",f);
	記述例 2:
	asm(" MOV.W R0, $$",s);
	記述例 3:
	asm(" MOV.W R0, $@",f);
	(3) 最適化を部分的に抑止する方法の一つとしてダミーの asm 関数が記述できます (関数内のみ記述可能) 。
	記述例:
	asm( );

日本語文字	(1) 文字列中に漢字文字を使用することができます。 記述例: L"漢字" (2) 漢字文字の文字定数を使用することができます。 記述例: L'漢' (3) コメント中に漢字文字を記述することができます。 記述例: /* 漢字 */ ● シフト JIS コード及び EUC コードをサポートしています。

表B.2　拡張機能 (2/2)

拡張機能	機能の内容

関数のデフォルト引数宣言	関数の引数にデフォルト値を定義できます。
	記述例 1:
	extern int func(int=1, char=0);
	記述例 2:
	extern int func(int=a, char=0);
	● デフォルト値として変数を記述する時は、関数を宣言するよりも前にデフォルト値として使用する変数の宣言を行ってください。
	● デフォルト値は引数の後ろから順に埋めてください。
	● 本機能はC言語モードでコンパイルする場合の拡張仕様です。C++言語モードのときはC++言語仕様に基づきます。

inline 記憶クラス	inline 記憶クラス指定子により関数をインライン展開することができます。 記述例: inline func(int i); ● 本機能はC言語モードでコンパイルする場合の拡張仕様です。C++言語モードのときはC++言語仕様に基づきます。
#pragma 拡張機能	M16C シリーズ、R8C ファミリ仕様を効率良く活かすための拡張機能を使用できます。
アセンブラマクロ関数	アセンブリ言語の一部を関数として記述することができます。
２進数の整数定数	整数定数で２進数を記述することができます。 0B または 0b に続けて 0 と 1 の数字列を記述します。 記述例: 0b01011100
long long 型	long long 型を扱うことができます。 long long 型の整数定数を記述する場合は、定数値の後ろに LL 又は ll 接尾子を付加します。 記述例: 123456789012LL
_Bool 型	_Bool 型を扱うことができます。 _Bool 型は、０もしくは１を表現します。
C++コメント	C 言語プログラム中で C++コメント（//）を記述することができます。

B.1 near/far 修飾子

M16C シリーズ、R8C ファミリは、0FFFFH 番地を境界としてデータの参照 / 配置等に使用されるアドレッシングモードが変わります。NC30 は、near/far 修飾子によりアドレッシングモードの切り換えを制御できます。

B.1.1 near/far 修飾子の概要

near/far 修飾子は、変数又は関数に対して使用するアドレッシングモードを選択します。

(1) near 修飾子　　000000H～00FFFFH の領域
(2) far 修飾子　　000000H～0FFFFFH の領域

near/far 修飾子は、変数又は関数の宣言時に型指定子に付加して記述します。変数及び関数の宣言時に near/far 修飾子を指定しない場合、NC30 は属性を以下のように解釈します。

(1) 変数の配置　　near 属性
(2) const 修飾された変数の配置　　far 属性
(3) 関数の配置　　far 属性

また、本コンパイラはコンパイルドライバの起動オプションにより、このデフォルトの属性を変更することができます。

B.1.2 変数の宣言書式

near/far修飾子は、文法的にconst、volatile型修飾子と同様の書式で宣言時に記述します。【図B.1】に宣言時の書式を示します。

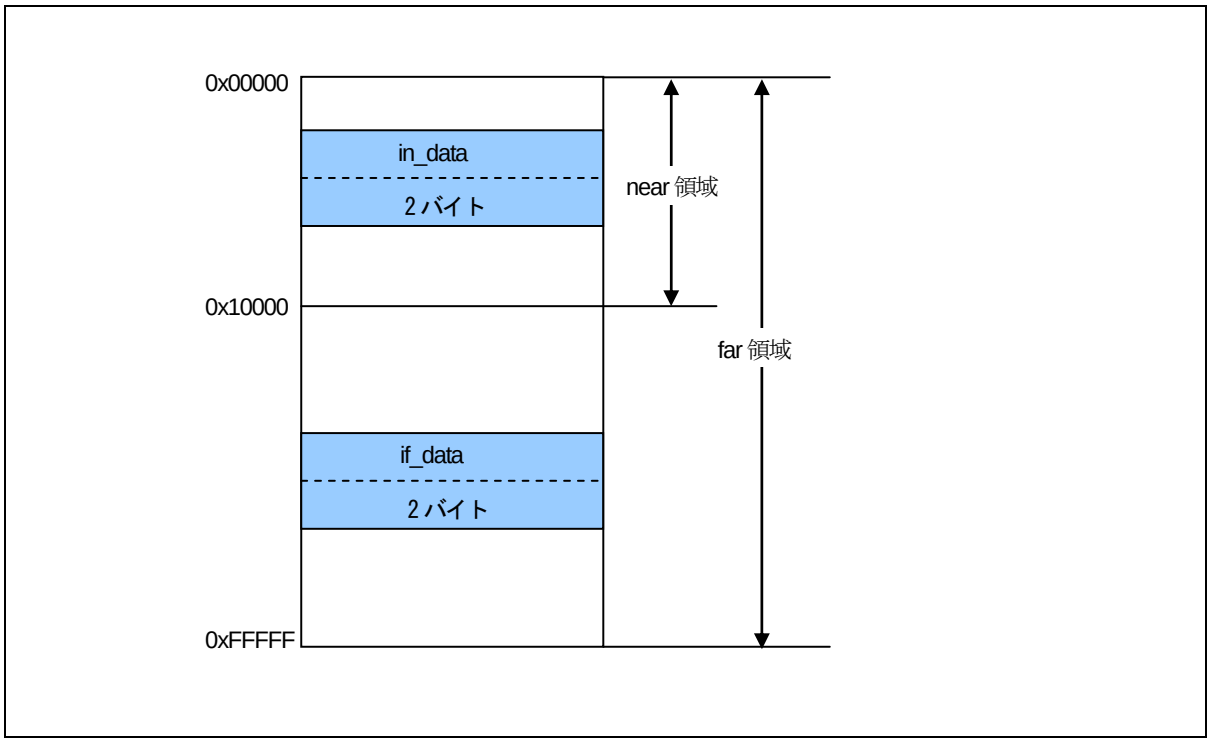
```
型指定子△near 又は far△変数;
```

図B.1　near/far 修飾子を付加した変数の宣言書式

変数の宣言例を【図B.2】に、その変数のメモリ配置図を【図B.3】に示します。

```
int near    in_data;
int far     if_data;

void        func(void)
{
            (以下省略)
                :
```

図B.2　変数の宣言例



図B.3 変数のメモリ配置

B.1.3 ポインタ型変数の宣言書式

ポインタ型変数はデフォルトではnear型(2 バイト)の変数です。ポインタ型の変数の宣言例を【図B.4】に示します。

```
例:
        int         * ptr;
```

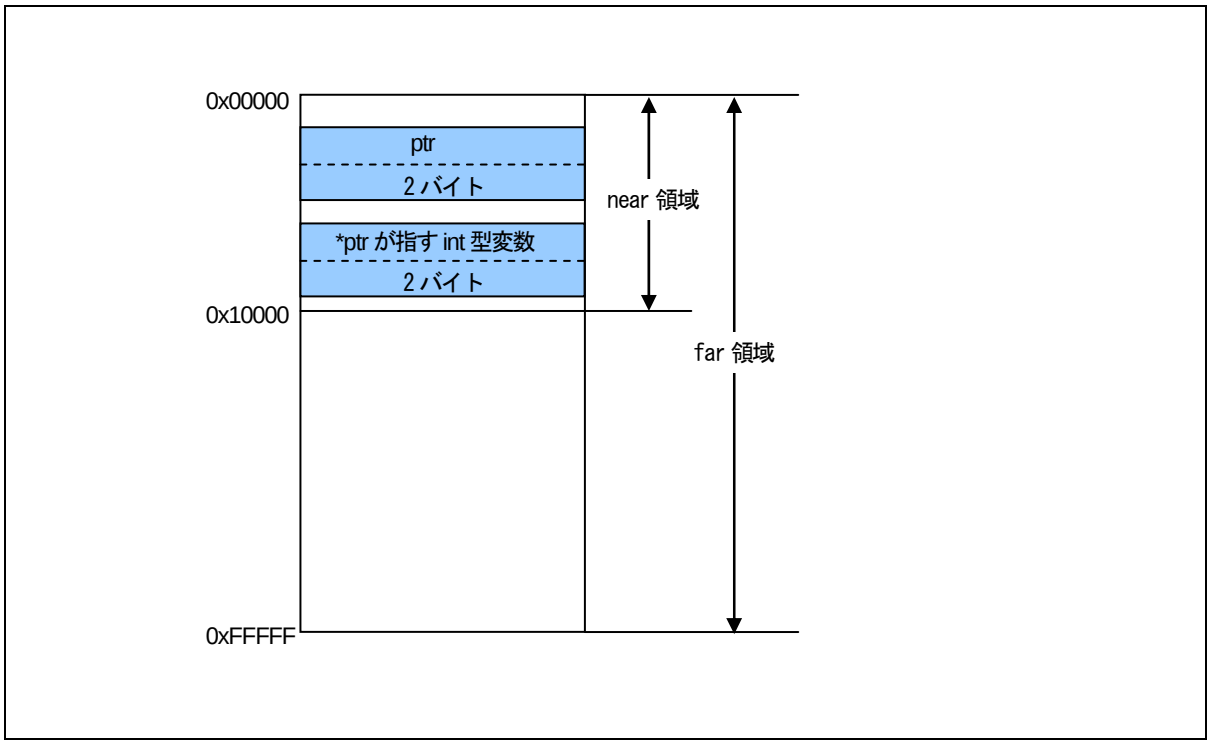
図B.4 ポインタ型変数の宣言例 (1)

変数の配置は near、ポインタ変数の型はnear型となるため、【図B.4】の記述は【図B.5】のように解釈されます。

```
例:
        int         near * near ptr;
```

図B.5 ポインタ型変数の宣言例 (2)

変数ptrは、near領域にあるint型変数を指し示す 2 バイトの変数です。ptr自身はnear領域に配置されます。上記例のメモリ配置を【図B.6】に示します。



図B.6 ポインタ型変数のメモリ配置

明示的にnear/farを指定した場合は、右側に記述した変数/関数を格納するアドレスのサイズを決定します。アドレスを扱うポインタ型の変数の宣言を【図B.7】に示します。

```
例 1:
        int         far *       ptr1;
例 2:
        int         * far       ptr2;
```

図B.7 アドレスを扱うポインタ型変数の宣言例 (3)

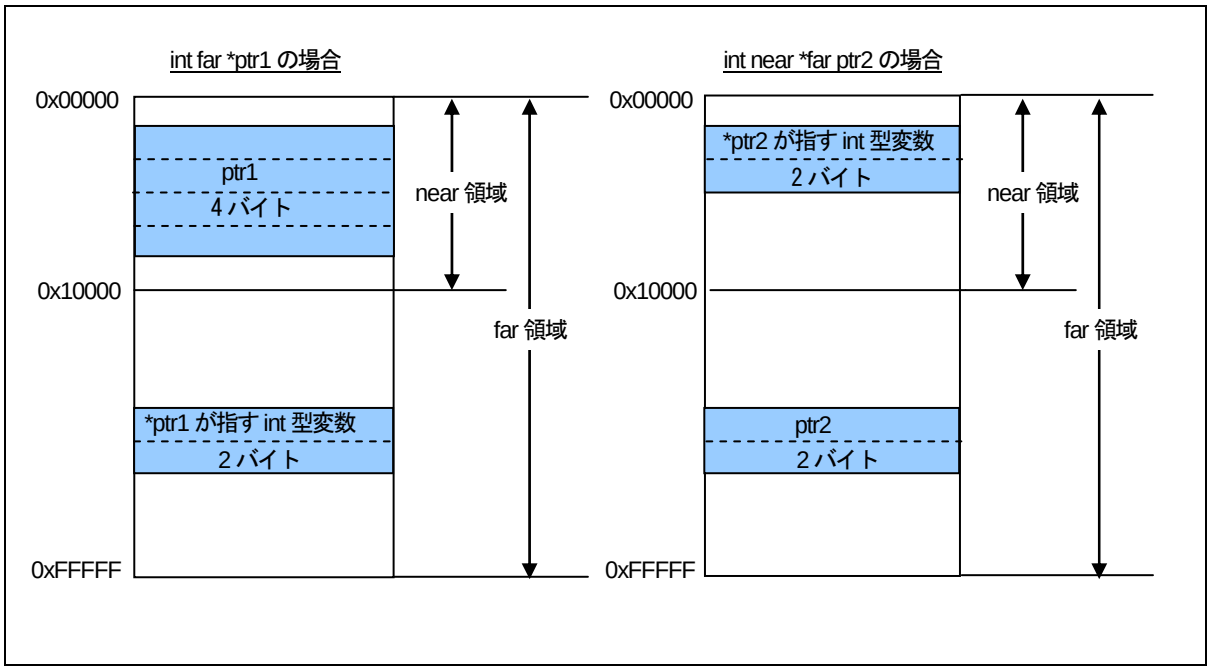
先にも説明したようにnear/farの指定がない場合は、変数の配置を"near"、変数の型を"near"として扱います。したがって、例 1、例 2 はそれぞれ、【図B.8】のように解釈されます。

```
例 1:
        int         far * near  ptr1;
例 2:
        int         near * far  ptr2;
```

図B.8 アドレスを扱うポインタ型変数の宣言例 (4)

例 1 では、変数 ptr1 は far 領域にある int 型変数を指し示す 4 バイト型の変数で、変数自身は near 領域に配置されます。

例 2 では、変数ptr2 はnear領域にあるint型変数を指し示す 2 バイト型の変数で、変数自身はfar領域に配置されます。例 1、例 2 のメモリ配置を【図B.9】に示します。



図B.9　アドレスを扱うポインタ型変数のメモリ配置

B.1.4　関数の宣言

C では、関数の宣言に near 属性を指定するとウォーニングメッセージを出力し、near の宣言を無視します。
C++としてコンパイルする場合、関数への near/far 配置属性はエラーになります。
C++としてコンパイルする場合、near/far 属性による関数の多重定義は const 属性と同様になります。

```
void func(int near * np)  {...}
void func(int far  * fp)  {...}    // near/far 属性ポインタは多重定義できます。
void func(int near n)     {...}
void func(int far  f)     {...}    // near/far 属性配置は多重定義できません。エラーになります。
```

B.1.5　nc30 の起動オプションによる near/far の制御

near/far属性を指定しない場合、NC30 は関数の属性をfar、変数の属性をnearとして扱います。NC30 は、変数(データ)またはポインタのデフォルトnear/far属性を変更するオプションを用意しています(【表B.3】)。

表B.3　nc30 起動オプション

起動オプション	機能
-fnear_ROM(-fNROM)	ROM データのデフォルト属性を near にします。
-ffar_RAM(-fFRAM)	RAM データのデフォルト属性を far にします。
-fconst_not_ROM(-fCNR)	const 修飾子で指定した型を ROM データとして扱いません。
-ffar_pointer(-fFP)	ポインタ型のデフォルト属性を far 属性とします。
-R8C	ROM データのデフォルト属性を near にします。far/_far 修飾子を無視します。

B.1.6 near から far への型変換機能

【図B.10】に示すプログラムの記述において、nearからfarへの型変換が行われます。

```
int         func( int far * );
int         far *f_ptr;
int         near *n_ptr;

void        main(void)
{
            f_ptr = n_ptr;              /* near ポインタを far ポインタに代入 */
              :
            (省略)
              :
            func ( n_ptr );             /* 引数に far ポインタを持つ関数として関数原型宣言した */
                                        /* 関数の呼び出し時に near ポインタの引数を指定 */
}
```

図B.10　near から far への型変換

far に型変換する場合、上位 2 バイトはゼロを設定します。

B.1.7 far から near ポインタへの代入の検査機能

C言語プログラムとしてコンパイルする場合、【図B.11】に示すプログラムの記述に関してアドレスの上位(バンク値)が失われることを示すウォーニングメッセージ(assign far pointer to near pointer, bank value ignored)を出力します。

```
int         func( int near * );
int         far *f_ptr;
int         near *n_ptr;

void        main(void)
{
            n_ptr = f_ptr;              /* far ポインタを near ポインタに代入 */
              :
            (省略)
              :
            func f_ptr                  /* 引数に near ポインタを持つ関数として関数原型宣言した */
                                        /* far ポインタを暗黙的に near にキャスト */
            n_ptr = (near *)f_ptr;      /* far ポインタを明示的に near にキャスト */
}
```

図B.11　far から near への型変換

なお、far ポインタを明示的に near にキャストを行った上で near ポインタに代入した場合もウォーニングメッセージ(far pointer (implicity) casted by near pointer)を出力します。

C++言語プログラムとしてコンパイルする場合、far ポインタから near ポインタへの代入はエラーになります。代入する far ポインタが near 領域を指していることが明らかな場合、キャスト記法または const_cast の演算子により far ポインタを near ポインタにキャストしてエラーを回避してください。

dynamic_cast、static_cast、および reinterpret_cast は near/far を変更できません。

B.1.8 near/far によるクラスの宣言

本コンパイラは、C++コンパイルの場合、クラス宣言に near/far 修飾できます。
near/far 修飾したクラスの this は、RAM データポインタのデフォルト属性によらず、クラス宣言の near/far 修飾子と同じ属性を持ちます。

```
class _far foo {
  public:
     .....
     .....
};
```

図B.12 near/far によるクラス宣言の例

this ポインタの near/far 属性は RAM データポインタのデフォルト属性と同じ near/far 属性です。

この仕様より、以下の制限があります。

- RAM データポインタのデフォルト属性が near の場合、far ポインタから near ポインタへの変換が生じるため、far 属性変数から非静的メンバ関数呼び出しがエラーになることがあります。
- RAM データポインタのデフォルト属性が near 属性のオブジェクトファイルをリンクする場合、heap 領域を near 領域に配置する必要があります。
- 同一のクラスを宣言し、かつ RAM データポインタのデフォルト属性が異なるオブジェクトファイルをリンクしないでください。

なお、クラス宣言における near/far 修飾により、this ポインタの near/far 属性を RAM データポインタのデフォルト属性から独立できます。

B.1.9 テンプレート関数と near/far 宣言

near/far 属性で多重定義できるテンプレート関数は、near 属性と far 属性となります。
テンプレート仮引数を near 修飾する場合、far 修飾したテンプレート実引数でインスタンス化できません。テンプレート仮引数が far 属性、かつテンプレート実引数が near 属性の場合、テンプレート実引数を far 属性に拡張してインスタンスします。

B.1.10　複数の宣言で near/far の確定を行う機能

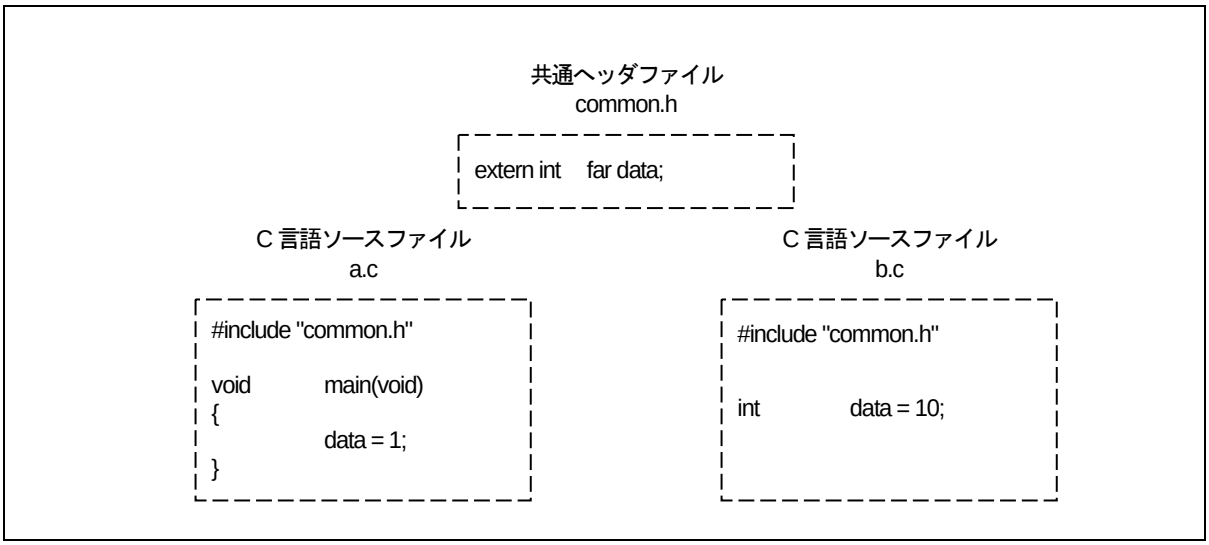
【図B.13】に示すような同一の変数に対して複数の宣言をC言語プログラムとしてコンパイルする場合、変数の型の情報が結合された型として解釈されます。

```
extern int    far idata;
int           idata;
int           idata = 10;

void          func(void)
{
              （以下省略）
                    :


この宣言は、以下の宣言として解釈されます。

extern int    far idata = 10;

void          func(void)
{
              （以下省略）
                    :
```

図B.13　変数の宣言の結合機能

この例に示すように、複数の宣言がある場合、near/far の指定はその内の 1 箇所で行うことで確定することができます。ただし、複数の宣言中、near と far が競合した場合はエラーとなります。

共通のヘッダファイルで near/far の宣言を行うことにより、ソースファイル間の near/far の整合をとることができます。



図B.14　共通ヘッダファイルの宣言例

C++プログラムとしてコンパイルする場合、C プログラムとしてコンパイルする場合許されていた、複数の宣言で near/far を確定する記述がエラーになる場合があります。

```
extern int far fi;
int fi;             // C では、fi の型を int far と解釈します。
                    // C++では、RAM データ配置属性が far の場合、fi の型を int far と解釈します。
                    //          RAM データ配置属性が near の場合、エラーになります。

extern int near ni;
int ni;             // C では、fi の型は int near と解釈します。
                    // C++では、RAM データ配置属性が far の場合、エラーになります。
                    //          RAM データ配置属性が near の場合、ni の型を int near と解釈します。

extern int far * fpi;
int * fpi;          // C では、fpi の型を int far*と解釈します。
                    // C++では、RAM データポインタ属性が far の場合、fpi の型を int far*と解釈します。
                    //          RAM データポインタ属性が near の場合、エラーになります。

extern int near * npi;
int * npi;          // C では、npi の型を int near*と解釈します。
                    // C++では、RAM データポインタ属性が far の場合、エラーになります。
                    //          RAMデータポインタ属性がnearの場合、npiの型をint near*と解釈します。
```

B.1.11 near/far 属性に関する注意事項

a. near/far 修飾子の文法上の注意事項

near/far修飾子は、文法上const修飾子と全く同じに扱われます。したがって、【図B.15】に示す記述はエラーとなります。

```
int     i, far     j;          ← この記述はできません
   ▼
int     i;
int     far  j ;
```

図B.15　変数の宣言例

- C++では、struct, union, class のメンバ変数の配置属性を near/far 修飾できません。
- C++では、mutable 指定された struct, union, class のメンバ変数は、クラスオブジェクトが const 指定されている場合であっても、そのメンバ変数に対する const を除去します。
- C++では、near 修飾型への参照型を持つ変数を far 属性変数で初期化する場合は、エラーになります。
- C++では、struct, union, class のメンバへのポインタが指す先に near/far 修飾できません。指定するとエラーになります。

B.2 asm 関数

本コンパイラでは、C/C++言語ソースプログラム中にアセンブリ言語のルーチン(asm関数)[1]を記述することができます。asm関数では拡張機能として、C/C++言語で記述した変数の参照機能があります。

B.2.1 asm 関数の概要

asm関数は、C/C++言語ソースプログラム中にアセンブリ言語の記述を行うときに使用します。asm関数の書式は、【図B.16】に示すようにasm(" ");のダブルクォーテーションの中にAS30 の言語仕様に準じたアセンブリ言語の命令を記述します。

```
#pragma   ADDRESS   ta0_int   55H
char      ta0_int;

void      func(void)
{
              :
          (省略)
              :
          ta0_int = 0x07;                    ← タイマ A0 割り込みを許可

          asm("       FSET I");              ← 割り込み許可フラグのセット
}
```

図B.16 asm 関数の記述例 (1)

また、【図B.17】に示す記述によりステートメントの前後関係を用いたコンパイラの一部の最適化処理を部分的に抑止することができます。

```
asm( );
```

図B.17 asm 関数の記述例 (2)

本コンパイラで扱う asm 関数は、アセンブリ言語の記述を行うほかに、以下の拡張機能があります。

- C/C++言語プログラムの記憶クラス auto 変数の FB オフセット値を C/C++言語の変数名で指定できます。
- C/C++言語プログラムの記憶クラス register 変数のレジスタ名を C/C++言語の変数名で指定できます。
- C/C++言語プログラムの記憶クラス extern 及び static 変数のシンボル名を C/C++言語の変数名で指定できます。

asm 関数を使用する場合の注意事項として以下の事項があります。

- コンパイラは、asm 関数内で変更するレジスタをチェックしていません。
- レジスタを変更する場合は、asm 関数を使用して、 push 命令、pop 命令を記述し、退避/復帰を行ってください。
- '$'、'_'で始まるシンボルはコンパイラのための予約シンボルです。asm 関数内に'$'、'_'で始まるシンボル定義を記述したプログラムの動作を保証しません。
- asm 関数内に指示命令".section"を記載しないでください。セクション名称を変更する際には、必ず asm 関数外で"#pragma SECTION"を用いてください。

[1] 本ユーザーズマニュアルでは、表現上アセンブリ言語で記述したサブルーチンをアセンブラ関数と表記します。C言語プログラム中にasm()で記述するものはasm関数、もしくはインラインアセンブル記述と表記します。

B.2.2 auto 変数の FB オフセット値の指定

記憶クラス auto および register 変数(引数を含む)は、フレームベースレジスタ(FB)に対するオフセット値で参照/配置されます (最適化等によってレジスタに割り当てられることもあります)。

【図B.18】に示す書式で記述することにより、asm関数内でスタック上に割り当てられたauto変数を使用することができます。

```
asm(" オペコード オペランド ,$$[FB]", 変数名 );
```

図B.18 FB オフセット指定の記述書式

この記述形式で指定できる変数名は 2 つです。変数名として以下の形式をサポートしています。

- 変数名
- 配列名[整数]
- 構造体名.メンバ名(ビットフィールドメンバは除きます)

```
void    func(void)
{
        int     idata;
        int     a[3];
        struct TAG{
                int     i;
                int     k;
        } s;
        :
        asm("   MOV.W   R0, $$[FB]", idata);
                :
        asm("   MOV.W   R0, $$[FB]", a[2]);
                :
        asm("   MOV.W   R0, $$[FB]", s.i);

                (以下省略)
                :
        asm("   MOV.W   $$[FB], $$[FB]", s.i, a[2]);
}
```

図B.19 FB オフセット指定の記述例

auto変数の参照例とコンパイル結果を【図B.20】に示します。

C 言語ソースファイル:

```
void        func(void)
{
            int idata = 1;

            asm("       MOV.W     $$[FB], R0", idata);
            asm("       CMP.W     #00001H ,R0");

            （以下省略）
                :
}
```

アセンブラソースファイル(コンパイル結果):

```
;## # FUNCTION func
;## # FRAME AUTO ( idata) size 2, offset -2
                :
            （省略）

;## # C_SRC : asm("    MOV.W     $$[FB], R0", idata);
;#### ASM START
            MOV.W     -2[FB], R0
            ._line 5
;## # C_SRC : asm(" CMP.W #00001H,R0");
            CMP.W     #00001H ,R0
;#### ASM END

            （以下省略）
                :
```

図B.20　auto 変数の参照例

また、【図B.21】に示す書式で記述することにより、asm関数内でauto変数の 1 ビットのビットフィールドを使用することができます(2 ビット以上のビットフィールドの操作はできません)。

```
asm( "      オペコード    $b[ FB ]" , ビットフィールド名 );
```

図B.21　FB オフセットビット位置指定の記述書式

この記述形式で指定できる変数名は 1 つです。【図B.22】に記述例を示します。

```
void        func(void)
{
            struct TAG{
                        char        bit0:1;
                        char        bit1:1;
                        char        bit2:1;
                        char        bit3:1;
            } s;

            asm("       bset        $b[FB]",s.bit1);
}
```

図B.22　FB オフセットビット位置指定の記述例

auto領域のビットフィールドの参照例とコンパイル結果を【図B.23】に示します。

C 言語ソースファイル:

```
void        func(void)
{
            struct TAG{
                        char        bit0:1;
                        char        bit1:1;
                        char        bit2:1;
                        char        bit3:1;
            } s;
            asm("       bset        $b[FB]",s.bit1);
}
```

アセンブラソースファイル(コンパイル結果):

```
;## # FUNCTION func
;## # FRAME AUTO ( __PAD1) size 1, offset -1
;## # FRAME AUTO ( s) size 1, offset -2
;## # ARG Size(0) Auto Size(2) Context Size(8)
            .section    program,CODE,ALIGN
            ._file      'bit.c'
            ._line 3
            .glb        _func
_func:
            enter       #02H
            ._line 10
;#### ASM START
            bset        1,-2[FB] ; s
;#### ASM END
            ._line 11
            exitd
```

図B.23　auto 領域のビットフィールドの参照例

auto 領域のビットフィールドを参照する場合は、ビット処理で参照可能な範囲(FB レジスタの値を中心に 32 バイト以内の範囲)に配置していることを確認してください。

B.2.3 レジスタ変数のレジスタ名の指定

記憶クラス auto 及び register の変数(引数を含む)は、コンパイラによってレジスタに割り当てられることがあります。

【図B.24】に示す書式で記述することにより、asm関数内でレジスタに割り当てられた変数を使用することができます。[1]

```
asm( "      オペコード   オペランド, $$",  変数名  );
```

図B.24　レジスタ変数の記述書式

この記述形式で指定できる変数名は 2 つです。レジスタ変数の参照例とコンパイル結果を【図B.25】に示します。

C 言語ソースファイル:

```
void        func(void)
{
            register int  i=1;

            asm("       mov.w       $$,A1",i);
}
```

アセンブラソースファイル(コンパイル結果):

```
;## # FUNCTION func
;## # ARG Size(0) Auto Size(0) Context Size(4)
            .section    program,CODE,ALIGN
            ._file      'reg.c'
            ._line 3
;## # C_SRC : {
            .glb        _func
_func:
            ._line 4
;## # C_SRC : register int           i=1;
            mov.w       #0001H,R0 ; i
            ._line 6
;## # C_SRC : asm("    mov.w       $$,A1",i);
;#### ASM START
            mov.w       R0,A1                       ← R0 レジスタ(変数)を A1 レジスタに転送
;#### ASM END
```

図B.25　register 変数の参照例

本コンパイラでは、関数内で使用するレジスタ変数を動的に管理しています。ある位置でレジスタ変数として使用したレジスタが、常に同じレジスタであるとは限りません。そのため、asm 関数内に直接レジスタを記述した場合、コンパイル結果により動作が異なる可能性があります。従いまして、必ずこの機能を用いてレジスタ変数を参照してください。

[1] register修飾子により強制的にレジスタに割り当てるためには、コンパイル時にオプション"-fenable_register(-fER)"を指定してください。

B.2.4 extern 変数及び static 変数のシンボル名の指定

記憶クラス extern 及び static の変数は、シンボルとして参照されます。
【図B.26】に示す書式で記述することにより、asm関数内でextern変数及びstaticの変数を使用することができます。

```
asm("　　オペコード　オペランド $$", 変数名 );
```

図B.26　シンボル名指定の記述書式

この記述形式で２つまで変数を指定できます。変数名として以下の形式をサポートしています。

- 変数名
- 配列名[定数]
- 構造体名.メンバ名(ビットフィールドメンバは除きます)

```
int         idata;
int         a[3];
struct TAG{
            int         i;
            int         k;
} s;

void        func(void)
{
                :
            asm("       MOV.W     R0, $$", idata);
                :
            asm("       MOV.W     R0, $$", a[2]);
                :
            asm("       MOV.W     R0, $$", s.i);

            (以下省略)
                :
}
```

図B.27　シンボル名指定の記述例

extern変数及びstatic変数の参照例を【図B.28】示します。

```
C 言語ソースファイル:
extern int    ext_val;

void          func(void)
{
              static int     s_val;

              asm("          mov.w     #01H,$$",ext_val);
              asm("          mov.w     #01H,$$",s_val);
}

アセンブラソースファイル(コンパイル結果):
_func:
              ._line 7
;## # C_SRC : asm("    mov.w       #01H,$$",ext_val);
;#### ASM START
              mov.w       #01H,_ext_val                    ← extern 変数 ext_val への転送
              ._line 8
;## # C_SRC : asm(" mov.w          #01H,$$",s_val);
              mov.w       #01H,___S0_s_val                 ← 関数内 static 変数 s_val への転送
;#### ASM END
              ._line 9
;## # C_SRC : }
              rts
E1:
              .glb        _ext_val
              .section    bss_NE,DATA
___S0_s_val: ;### C's name is s_val
              .blkb 2
              .END
```

図B.28　extern 変数及び static 変数の参照例

また、【図B.29】に示す書式で記述することにより、asm関数内でextern変数及びstatic変数の１ビットのビットフィールドを使用することができます(2 ビット以上のビットフィールドは記述できません)。

```
asm("    オペコード     $b" , ビットフィールド名 ) ;
```

図B.29　シンボル名指定の記述書式

この記述形式で指定できる変数名は１つです。【図B.30】に記述例を示します。

```
struct TAG{
        char    bit0:1;
        char    bit1:1;
        char    bit2:1;
        char    bit3:1;
} s;

void    func(void)
{
        asm("   bset    $b",s.bit1);
}
```

図B.30　シンボルのビット位置指定の記述例

【図B.30】のCソースファイルのコンパイル結果を【図B.31】に示します。

```
;## # FUNCTION func
;## # ARG Size(0) Auto Size(0) Context Size(4)
        .section    program,CODE,ALIGN
        ._file      'kk.c'
        .align
        ._line 10
;## # C_SRC : {
        .glb        _func
_func:
        ._line 11
;## # C_SRC : asm("   bset    $b",s.bit1);
;#### ASM START
        bset        1,_s                ← 構造体 s のビットフィールド bit0 を参照
;#### ASM END
        ._line 12
;## # C_SRC : }
        rts
E1:
        .align
        .section    bss_NO,DATA
        .glb        _s
_s:
        .blkb 1
        .END
```

図B.31　シンボルに対するビットフィールドの参照例

extern 変数及び static 変数のビットフィールドを参照する場合は、絶対ビット命令アドレッシングの命令で参照可能な範囲(0000H～1FFFH の範囲)に配置していることを確認してください。

B.2.5 記憶クラスに依存しない指定

変数の記憶クラス(auto変数、レジスタ変数[1]、extern変数、static変数)に依存することなく、asm関数内で使用することができます。

【図B.32】に示す書式で記述することにより、変数をasm関数内で使用することができます。[2]

```
asm("　　オペコード　オペランド，$@″，変数名 )；
```

図B.32 変数の記憶クラスに依存しない記述書式

この記述形式で指定できる変数名は2つです。参照例とコンパイル結果を【図B.33】に示します。

C 言語ソースファイル:

```
extern int    e_val;

void          func(void)
{
              int          f_val;                  ← extern 変数
              register int r_val;                  ← auto 変数
              static int   s_val;                  ←レジスタ変数

              asm("        mov.w      #1, $@", e_val);          ← extern 変数の参照
              asm("        mov.w      #2, $@", f_val);          ← auto 変数の参照
              asm("        mov.w      #3, $@", r_val);          ←  レジスタ変数の参照
              asm("        mov.w      #4, $@", s_val);          ← static 変数の参照
              asm("        mov.w      $@, $@", f_val,r_val);
}
```

アセンブラソースファイル(コンパイル結果):

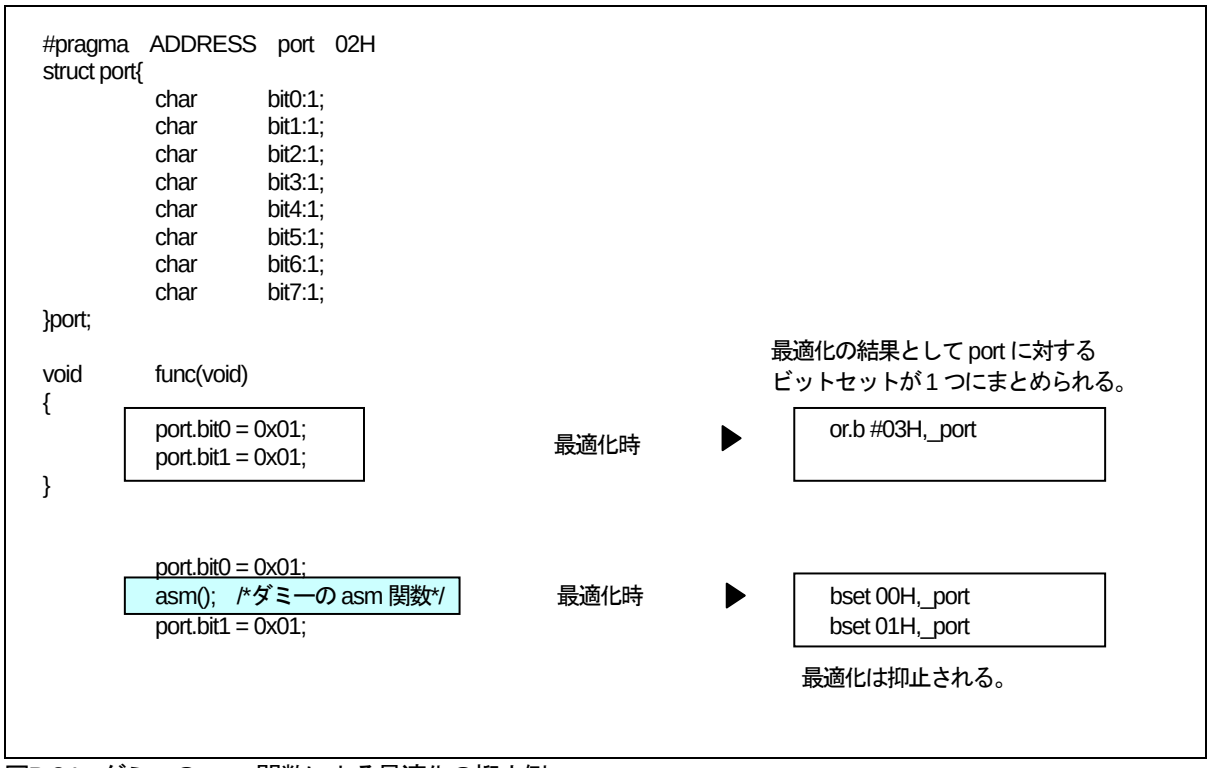
```
              .glb       _func
_func:
              enter      #02H
              pushm      R1
              ._line 9
;## # C_SRC : asm("      mov.w      #1, $@", e_val);
;#### ASM START
              mov.w      #1, _e_val:16                 ← extern 変数の参照
              ._line 10
;## # C_SRC : asm("      mov.w      #2, $@", f_val);
              mov.w      #2, -2[FB]                    ← auto 変数の参照
              ._line 11
;## # C_SRC : asm("      mov.w      #3, $@", r_val);
              mov.w      #3, R1                        ← register 変数の参照
              ._line 12
;## # C_SRC : asm("      mov.w      #4, $@", s_val);
              mov.w      #4, ___S0_s_val:16            ← static 変数の参照
              ._line 13
;## # C_SRC : asm("      mov.w      $@, $@", f_val,r_val);
              mov.w      -2[FB], R1
;#### ASM END
```

図B.33 各記憶クラスの変数の参照例

[1] register修飾子を指定しても、レジスタに割り当てられるとは限りません。
[2] どの記憶クラスに配置されるかは、実際にコンパイルして確認してください。

B.2.6 最適化の部分的な抑止方法

【図B.34】に示すasm関数の記述においてダミーのasm関数を記述することにより、部分的に一部の最適化を抑止することができます。



図B.34 ダミーの asm 関数による最適化の抑止例

B.2.7 asm 関数に関する注意事項

a. asm 関数の拡張機能

asm 関数を用いて以下の処理を行うときは、必ず記述例に示す書式で記述してください。

(1) スタック変数の場合

スタック変数(引数を含む)をフレームベースレジスタ(FB)からのオフセット値を用いて指定しないでください。スタック変数を指定する場合は、【図B.35】に示す書式で記述してください。

```
asm("    MOV.W   #01H,$$[FB]", i);     ← 記憶クラス auto の変数を参照する書式です。
asm("    BSET    $b [FB]", s.bit0);    ← 記憶クラス auto のビットフィールドを参照する書式です。
```

図B.35 asm 関数の記述例 (1)

(2) register 記憶クラスの指定

本コンパイラでは、register記憶クラスを指定することができます。register記憶クラスで指定した変数でかつオプション-fenable_register(-fER)を指定してコンパイルした場合にasm関数内にregister変数を記述する時は、【図B.36】に示す書式で記述してください。

```
asm("       MOV.W     #0,$$", i);              ← レジスタ変数を参照する書式です。
```

図B.36　asm 関数の記述例 (2)

また、オプション-O[1～5], -OR, -OS, -OR_MAX(-ORM), -OS_MAX(-OSM)を指定すると、コード効率の向上の為に register 渡しとなる引数を auto 領域に転送を行わずに register 変数として扱う場合があります。
この場合に、asm 関数内に引数を指定すると変数の FB オフセット値でなくレジスタ名でアセンブリ言語を出力するので、ご注意ください。

(3) asm 関数内に引数を参照する場合

本コンパイラでは、変数(引数およびauto変数を含む)の生存区間についてプログラムフローを解析して処理を行っているため、【図B.37】のようにasm関数内で直接、引数およびauto変数を参照すると、その生存区間の管理が崩れて正しいコード出力する事ができません。
従って、asm 関数の記述で、引数および auto 変数を参照する場合は、必ず asm 関数の「$$,$b,$@」機能を使用して参照してください。

```
void        func( int i,int j)
{
            asm ("      mov.w     2[FB],4[FB]");                    /* j = i; */
}
```

図B.37　正しく参照することができない例

上記の場合コンパイラは、関数 func 内で、"i"、"j" は使用されていないと判断するため、引数を参照するためのフレームを構築するコードを出力しません。このため、引数を正しく参照できなくなります。

(4) asm 関数内でのブランチについて

本コンパイラでは、レジスタの生存区間、変数の生存区間についてプログラムフローを解析して処理を行っているため、asm 関数内にはフローに影響を与えるようなブランチ(条件ブランチ含む)を記述しないようにしてください。

b. レジスタについて

- asm 関数内でレジスタを変更しないでください。変更する場合は、push 命令/pop 命令により、レジスタの退避/復帰を行ってください。
- SBレジスタをスタートアッププログラムによる初期化後固定で使用することを前提としています。万一変更する場合は、連続するasm関数の最後で【図B.38】に示すSBレジスタを元に戻す記述を行なってください。また、変更している間に呼び出す関数や、その間に発生する割り込み処理について十分考慮してください。

```
asm("       .SB     0);
asm("       LDC     #0H, SB");                  ← SB 変更
asm("       MOV.W   R0, _port[SB]");
            :
           (省略)
            :
asm("       .SB     __SB__);
asm("       LDC     #__SB__,SB");               ← SB を元に戻す
```

図B.38 変更したスタティックベースレジスタの復帰方法

- フレームベースレジスタ(FB)は、スタックフレームポインタとして使用しますので、asm 関数内で変更しないでください。

c. ラベルの記述に関する注意事項

本コンパイラが生成するアセンブラソースファイルでは、【図B.39】に示す形式の内部ラベルが出力されます。したがって、asm関数を用いてこの規定と重複する可能性のあるラベルを記述しないでください。

```
アルファベットの大文字 1 文字と数字で構成されるラベル名:

        A1:
        C9830:

_(アンダースコア)で始まる 2 文字以上のラベル名

        __LABEL:
        ___START:
```

図B.39 asm 関数内に記述できないラベル名の形式

B.3 日本語文字サポート

本コンパイラでは、C/C++言語ソースプログラム中に日本語の文字を記述することができます。

B.3.1 日本語文字の概要

日本語の文字は、アルファベット等の 1 バイトで表現される文字と異なり、2 バイトで構成されます。NC30 では、この 2 バイトで構成される文字を文字列、文字定数、コメントに記述することができます。記述できる文字の種類を以下に示します。

- 漢字
- ひら仮名
- 全角のカタ仮名
- 半角のカタ仮名

日本語文字の記述は、以下の漢字コード系のみで使用できます。

- EUC(ただし、1 文字が 3 バイトで構成される外字コードは使用できません)
- シフト JIS

C++におけるワイドキャラクタの文字コードは UCS2 です。

B.3.2 日本語文字を記述するための設定

漢字コードを使用する場合は、以下に示す環境変数を設定してください。
なお、デフォルトでは、NCKIN、NCKOUT 共に SJIS となっています。

- 入力コード系指定環境変数..... NCKIN
- 出力コード系指定環境変数..... NCKOUT

環境変数の設定例を【図B.40】に示します。

```
autoexec.bat ファイルに以下の記述を設定してください。

入出力ともにシフト JIS にする方法
set   NCKIN = SJIS
set   NCKOUT = SJIS

入力を EUC、出力をシフト JIS にする方法
set   NCKIN = EUC
set   NCKOUT = SJIS
```

図B.40 環境変数 NCKIN と NCKOUT の設定例

本コンパイラでは入力漢字コードをプリプロセッサで処理します。プリプロセッサで EUC コードに変換し、その後コンパイラ内の字句解析部終段で、環境変数を基に変換し出力します。

B.3.3 文字列中の日本語文字

文字列に日本語文字を記述するときの書式を【図B.41】に示します。

```
L"漢字文字列"
```

図B.41 文字列中の漢字コード記述書式

日本語を通常の文字列と同様に"漢字文字列"と記述した場合、文字列の操作では char 型へのポインタ型として扱われます。したがって、2 バイト文字として操作することはできません。

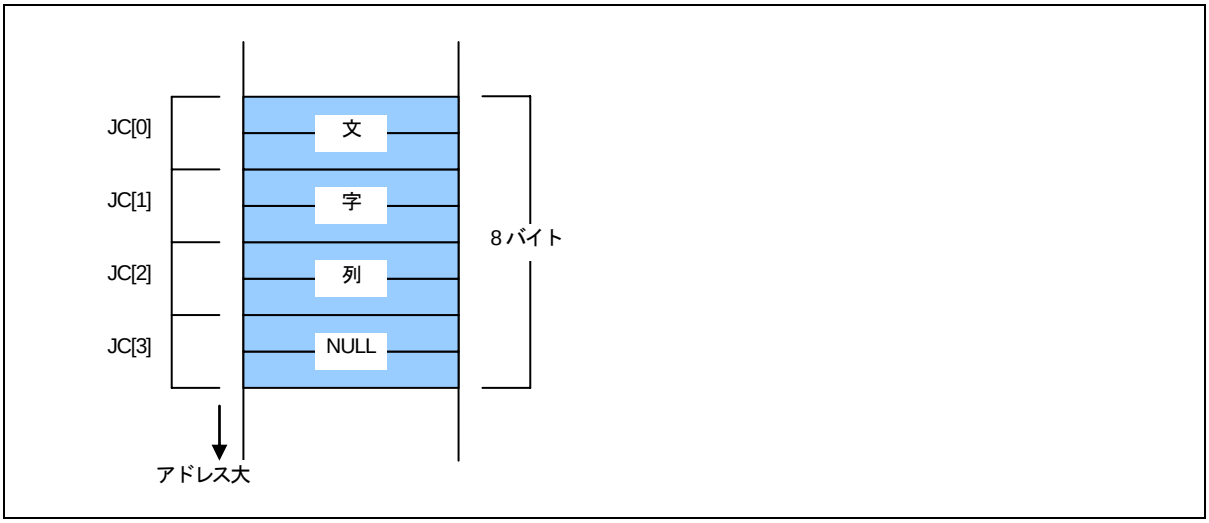
2 バイト文字として扱う場合は、文字列の先頭に"L"を付加して wchar_t 型へのポインタ型として使用します。wchar_t 型は、標準ヘッダファイル stdlib.h の中で unsigned short 型に typedef しています。

【図B.42】に日本語の文字列の記述例を示します。

```
#include <stdlib.h>

void        func(void)
{
            wchar_t     JC[4] = L"文字列";      (1)

            (以下省略)
                :
```

図B.42 日本語文字列の記述例

【図B.42】中の(1)の初期化された文字列のメモリ配置を【図B.43】に示します。



図B.43 wchar_t 型文字列のメモリ配置

B.3.4 文字定数としての日本語文字

文字定数として日本語文字を記述するときの書式を【図 B.44】に示します。

```
L'漢'
```

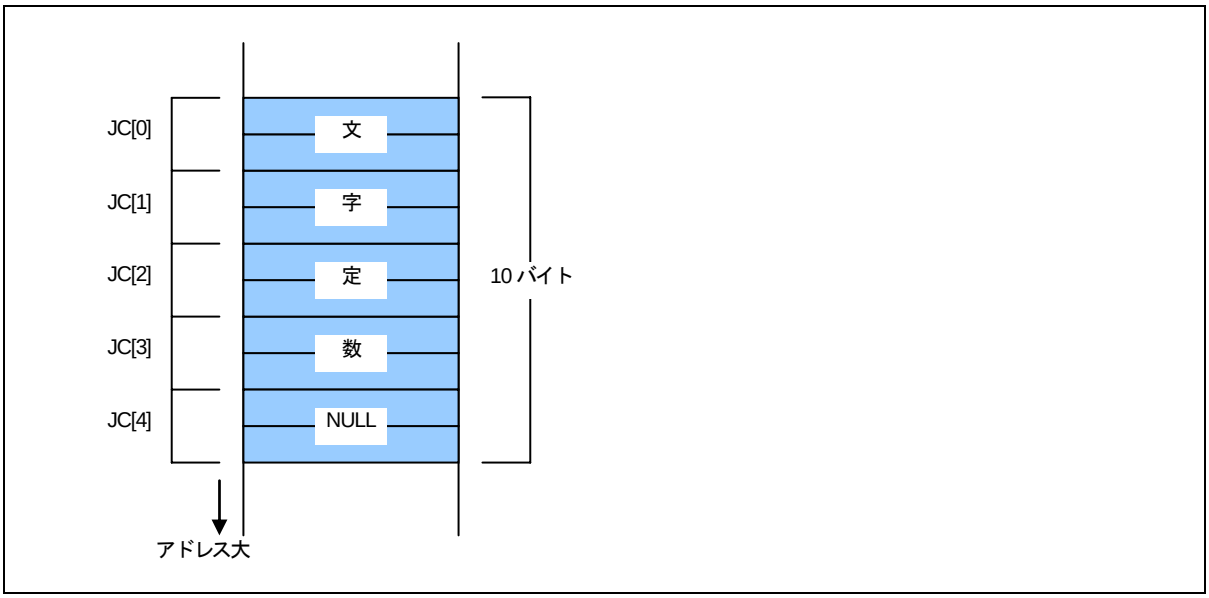
図B.44 文字列中の漢字コード記述書式

文字列と同様に、文字定数の前に"L"を付加した場合は、wchar_t 型として扱われます。文字定数として'文字'のように複数の文字を記述した場合は、最初の文字「文」のみが文字定数として扱われます。
【図B.45】に日本語の文字定数の記述例を示します。

```
#include <stdlib.h>

void        func(void)
{
            wchar_t     JC[5];

            JC[0] = L'文';
            JC[1] = L'字';
            JC[2] = L'定';
            JC[3] = L'数';

            (以下省略)
                :
```

図B.45 日本語文字定数の記述例

【図B.45】中の文字定数を代入した配列のメモリ配置を【図B.46】に示します。



図B.46 配列に代入した wchar_t 型文字定数のメモリ配置

B.4 関数のデフォルト引数宣言

NC30 では、C++の機能と同様に関数の引数にデフォルト値を定義できます。この章では、関数のデフォルト引数宣言機能について説明します。

本機能はC言語モードでコンパイルする場合の拡張仕様です。C++言語モードのときはC++言語仕様に基づきます。

B.4.1 関数のデフォルト引数宣言の概要

NC30 では、関数の関数原型宣言時に、仮引数にデフォルト値を与えることにより暗黙の実引数を使用することができます。この機能を使用することにより、関数呼び出し時に頻繁に使用する値を記述する手間を省くことができます。

B.4.2 関数のデフォルト引数宣言の書式

【図B.47】に関数のデフォルト引数宣言時の書式を示します。

```
記憶クラス指定子△型宣言子△宣言子([仮引数[=デフォルト値あるいは変数],...]);
```

図B.47　関数のデフォルト引数宣言書式

デフォルト引数宣言の例を【図B.48】に、【図B.48】で示すサンプルプログラムのコンパイル結果を【図B.49】に示します。

```
int     func( int i=1 , int j=2 );      ← 関数 func に仮引数のデフォルト値を第 1 引数 : 1、
                                          第 2 引数 : 2 に宣言しています。

void    main(void)
{
        func();                         ← 実引数は第 1 引数 : 1、第 2 引数 : 2 になります。
        func(3);                        ← 実引数は第 1 引数 : 3、第 2 引数 : 2 になります。
        func(3,5);                      ← 実引数は第 1 引数 : 3、第 2 引数 : 5 になります。
}
```

図B.48　関数のデフォルト引数の宣言例 (sample.c)

```
;### C_SRC :          {
                .glb        _main
_main:
                ._line      5
;### C_SRC :                      func();
                mov.w       #0002H,R2                ← 第 2 引数:2
                mov.w       #0001H,R0                ← 第 1 引数:1
                jsr         $func
                ._line      6
;### C_SRC :                      func(3);
                mov.w       #0002H,R2                ← 第 2 引数:2
                mov.w       #0003H,R1                ← 第 1 引数:3
                jsr         $func
                ._line      7
;### C_SRC :                      func(3,5);
                mov.w       #0005H,R2                ← 第 2 引数:5
                mov.w       #0003H,R1                ← 第 1 引数:3
                jsr         $func
                ._line      8
;### C_SRC :          }
                rts
                    :
                （省略）
                    :

NC30 での引数を積む順序は、関数で宣言した引数の後ろからです。
この例では引数をレジスタ渡しで処理しています。
```

図B.49　smp1.c のコンパイル結果 (sample.a30)

関数の引数に変数を記述することができます。

デフォルト引数の変数指定の例を【図B.50】に、【図B.50】で示すサンプルプログラムのコンパイル結果を【図B.51】に示します。

```
int         near sym ;
int         func( int i = sym);          ← デフォルトの引数を変数で指定しています。

void        main(void)
{
            func();                      ← 変数(sym)を引数として関数呼び出しを行ないます。
}
                :
            （省略）
                :
```

図B.50　デフォルト引数の変数指定例 (smp2.c)

```
_main:
        ._line 6
        mov.w   _sym,R1          ← 変数(sym)を引数として関数呼び出しを行ないます。
        jsr     $func
        ._line 7
        rts
```

図B.51 smp2.c のコンパイル結果 (smp2.a30)

B.4.3 関数のデフォルト引数宣言の規定事項

関数のデフォルト引数の宣言を行なう場合は、以下の点に注意してください。

a. 複数の引数にデフォルト値を指定する時

関数の引数が複数ある場合にデフォルト値を指定する場合は、必ず引数の後ろから埋めてください。【図B.52】に不適切な記述の例を示します。

```
void    func1(int i, int j=1, int k=2);     /* 正しい記述です*/
void    func2(int i, int j, int k=2);       /* 正しい記述です*/
void    func3(int i = 0, int j, int k);     /* 誤った記述です*/
void    func4(int i = 0, int j, int k = 1); /* 誤った記述です*/
```

図B.52 関数原型宣言の記述例

b. 変数のデフォルト値を指定する時

デフォルト値として変数を指定する場合は、指定する変数の宣言を行なった後に関数原型宣言を行なってください。関数原型宣言を行なった時点で、宣言していない変数を引数のデフォルト値に指定した場合は、エラーとして処理します。

B.5 inline 関数宣言

C++風に inline 記憶クラスを指定することができます。関数に対して inline 記憶クラスを指定することにより関数をインライン展開することができます。

本機能はC言語モードでコンパイルする場合の拡張仕様です。C++言語モードのときはC++言語仕様に基づきます。

B.5.1 inline 記憶クラスの概要

inline 記憶クラス指定子は、関数に対してインライン展開される関数であることを宣言します。inline 記憶クラス指定した関数は、アセンブリ言語レベルでは直接コードが埋め込まれます。

B.5.2 inline 記憶クラスの宣言書式

inline記憶クラス指定子は、文法的にstatic、extern型記憶クラス指定子と同様の書式で宣言時に記述します。【図B.53】に宣言時の書式を示します。

```
inline△型指定子△関数;
```

図B.53 inline 記憶クラスの宣言書式

関数の宣言例を【図B.54】に、コンパイル結果【図B.55】を示します。

```
inline int    func(int i)                    ← インライン関数の宣言及び定義部
{
              return i++;
}

void          main(void)
{
              int           s;

              s = func(s);                   ← インライン関数呼び出し部
}
```

図B.54 インライン関数のサンプルプログラム (sample.c)

```
                .SECTION  program,CODE,ALIGN
                ._file    'sample.c'
                ._line    7
  ;## # C_SRC :         {
                .glb      _main
  _main:
                enter     #02H
                pushm     R2
                ._line    10
  ;## # C_SRC :                    s = func(s);
                mov.w     -2[FB],R1   ;   s
                ._line    3
  ;## # C_SRC :                    return i++;          ← インライン関数が埋め込まれている
                mov.w     R1,R2
                add.w     #0001H,R1
                ._line    10
  ;## # C_SRC :                    s = func(s);
                mov.w     R1,-2[FB]   ;   s
                ._line    11
  ;## # C_SRC :         }
                popm      R2
                exitd
  E1:
                .align
                .END
```

図B.55　サンプルプログラムのコンパイル結果 (sample.a30)

B.5.3　inline 記憶クラスの規定事項

inline 記憶クラス指定場合は、以下の点に注意してください。

(1)　インライン関数の間接呼び出しについて

インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合は、コンパイルエラーになります。

(2)　インライン関数の再帰呼出しについて

インライン関数の再帰呼出しをすることはできません。再帰呼出しの記述を行った場合、コンパイルエラーになります。

(3)　インライン関数の定義について

関数に対してinline記憶クラスを指定する場合は、呼び出し前に必ず実体定義を行なってください。実体定義は、必ず同一ファイル内に記述してください。本コンパイラでは、【図B.56】の記述はエラーになります。

```
inline void   func(int i);

void          main( void )
{
              func(1);
}
```

エラーメッセージ :
sample.c(5) : C2567 (E) inline function's body is not declared previously
===> func(1);

図B.56 インライン関数の不適切な記述例 (1)

また、ある関数を通常の関数として使用した後に、その関数をインライン関数として定義した時は、本コンパイラではエラーになります（【図B.57】）。

```
int           func(int i);

void          main( void )
{
              func(1);
}

inline int    func(int i)
{
              return i;
}
```

エラーメッセージ:
sample.c(9) : C2565 (E) inline function is called as normal function before
===> {

図B.57 インライン関数の不適切な記述例 (2)

(4) インライン関数のアドレスについて

インライン関数は、アドレスを持ちません。このため、インライン関数に対して&演算子を使用した場合は、エラーになります（【図B.58】）。

```
inline int      func(int i)
{
                return i;
}

void            main(void)
{
                int             (*f)(int);

                f = &func;
}
```

エラーメッセージ:
sample.c(10) : C2555 (E) can't get inline function's address by '&' operator
===> f = &func;

図B.58 インライン関数の不適切な記述例 (3)

(5) static データの宣言

インライン関数内で static データを宣言した場合は、宣言した static データの実体はファイル単位で確保されます。このため、複数のファイルにまたがったインライン関数ではデータのアクセス領域が異なります。インライン関数内で使用する static データは関数外で宣言してください。

本コンパイラでは、インライン関数内でstatic宣言をした場合は、ウォーニングになります。 また、インライン関数内のstatic宣言は推奨しません(【図B.59】)。

```
inline int      func( int j)
{
                static int      i = 0;

                i++;
                return i + j;
}
```

ウォーニングメッセージ:
sample.c(3) : C1636 (W) static variable in inline function
===> static int i = 0;

図B.59 インライン関数の不適切な記述例 (4)

(6) デバッグ情報について

本コンパイラではインライン関数に対する C 言語レベルのデバッグ情報を出力しません。このため、インライン関数のデバッグはアセンブリ言語レベルで行なうことになります。

B.6 #pragma 拡張機能

B.6.1 #pragma 拡張機能の一覧

#pragma に関する拡張機能の内容と規定を一覧表で示します。

a. メモリ配置に関する拡張機能

表B.4 メモリ配置に関する拡張機能 (1/2)

拡張機能	機能の内容
#pragma BIT	絶対ビット命令アドレッシングを使用できる領域にある外部変数（すなわち、00000H 番地から 01FFFH 番地までにある変数）であることを宣言します。 記述形式: #pragma BIT△変数名 記述例: #pragma BIT sym
#pragma SBDATA	SB 相対アドレッシングを使用するデータであることを宣言します。 記述形式: #pragma SBDATA△変数名 記述例: #pragma SBDATA sym
#pragma SECTION	本コンパイラが生成するセクション名を変更します。 記述形式: #pragma SECTION△既定ｾｸｼｮﾝ名△変更ｾｸｼｮﾝ名 記述例: #pragma SECTION bss nonval_data
#pragma STRUCT	(1) 指定したタグを持つ構造体のパックを禁止します。 記述形式: #pragma STRUCT△構造体のタグ名△unpack 記述例: #pragma STRUCT TAG1 unpack (2) 指定したタグを持つ構造体のメンバの並べ替えを行い、偶数サイズのメンバを先に配置します。 記述形式: #pragma STRUCT△構造体のタグ名△arrange 記述例: #pragma STRUCT TAG1 arrange ● 本機能の arrange は、Cプログラムとしてコンパイルする場合のみ有効です。

表B.5 メモリ配置に関する拡張機能 (2/2)

拡張機能	機能の内容
#pragma EXT4MPTR	4M バイト拡張空間をアクセスするポインタであることを宣言します。 記述形式: #pragma EXT4MPTR△ポインタ変数名 記述例: #pragma EXT4MPTR sym
_ext4mptr	4M バイト拡張空間をアクセスするポインタであることを宣言します。 記述形式: _ext4mptr△far△ポインタ変数宣言 記述例: _ext4mptr far int * ptr; ● 本機能は C プログラムとしてコンパイルする場合のみ有効です。

b. 組み込み機器に関する拡張機能

表B.6　組み込み機器に使用するための拡張機能 (1/2)

拡張機能	機能の内容
#pragma ADDRESS	変数を絶対アドレスに割り付けます。 記述形式: #pragma ADDRESS△変数名△絶対アドレス 記述例: #pragma　ADDRESS　port0　2H
#pragma BITADDRESS	変数を指定した絶対アドレスの指定したビット位置に、割り付けます。 記述形式: #pragma BITADDRESS△変数名△ビット位置, 絶対アドレス 記述例: #pragma　BITADDRESS　io　1, 100H
#pragma INTCALL	ソフトウェア割り込み(int 命令)で呼び出す関数を宣言します。 記述形式 1: #pragma INTCALL△INT 番号△アセンブラ関数名 (レジスタ名) 記述例 1: #pragma　INTCALL　25　func(R0, R1) 記述形式 2: #pragma INTCALL△INT 番号△C 言語関数名() 記述例 2: #pragma　INTCALL　25　func() ● レジスタ名が無い場合には括弧が省略できます。
#pragma INTERRUPT	割り込み処理関数を宣言します。この宣言により、関数の出入り口で割り込み処理関数の手続きを行うコードを生成します。 記述形式: #pragma INTERRUPT△[/B¦/E]△割り込み処理関数名 #pragma INTERRUPT△[/B¦/E]△割り込みベクタ番号△割り込み処理関数名 #pragma INTERRUPT△[/B¦/E]△割り込み処理関数名(vect=割り込みベクタ番号) 記述例: #pragma　INTERRUPT　int_func #pragma　INTERRUPT　/B　int_func #pragma　INTERRUPT　10　int_func #pragma　INTERRUPT　/E　10　int_func #pragma　INTERRUPT　int_func (vect=10)

表B.7 組み込み機器に使用するための拡張機能 (2/2)

拡張機能	機能の内容
#pragma PARAMETER	アセンブリ言語で記述された関数を呼び出す際に、その引数をレジスタを介して渡すことを宣言します。 記述形式: #pragma PARAMETER△関数名(レジスタ名) 記述例: #pragma PARAMETER asm_func(R0, R1)
#pragma SPECIAL	スペシャルページサブルーチン呼び出しの関数を宣言します。 記述形式: #pragma SPECIAL△番号△関数名() #pragma SPECIAL△関数名(vect=呼び出し番号) 記述例: #pragma SPECIAL 30 func() #pragma SPECIAL func() (vect=30) ● 関数名の後ろの括弧は省略できます。

c. その他の拡張機能

表B.8　その他の拡張機能

拡張機能	機能の内容
#pragma __ASMMACRO	アセンブラのマクロで定義した関数を宣言します。 記述形式: #pragma __ASMMACRO△関数名（レジスタ名） 記述例: #pragma __ASMMACRO mul(R0,R2)
#pragma ASM #pragma ENDASM	アセンブリ言語で記述を行う領域を指定します。 記述形式: #pragma△ASM #pragma△ENDASM 記述例: #pragma ASM mov.w R0,R1 add.w #02H,R1 #pragma ENDASM
#pragma PAGE	アセンブラリストファイルの改ページの指定を行います。 記述形式: #pragma△PAGE 記述例: #pragma PAGE ● 本機能は、Cプログラムとしてコンパイルする場合のみ有効です。

なお、#pragma で C++言語の多重定義された関数を指定する場合、#pragma 宣言より以下に記述された最も近い関数が対象となります。

#pragma に続けて仕様外の文字列、識別子を記述した場合、その処理指定は無視されます。

また、サポートしていない#pragma を使用した場合、デフォルトではウォーニングを出力しません。-Wunknown_pragma オプションを指定したときのみウォーニングを出力します。

#pragma で変数名、関数名を記述する際に、修飾付き名前を記述することができます。

```
#pragma ADDRESS s1              100H
unsigned short s1;
#pragma ADDRESS N::s2           200H
namespace N {
        unsigned short s2;
}
```

this ポインタをもつ関数は、該当#pragma の対象外となります。

d. Cスタートアップで使用する拡張機能

本 pragma はCスタートアップ専用のため、使用しないでください。

拡張機能	機能の内容
#pragma STACKSIZE	スタックセクション(stack)の出力及び、スタックのトップラベル名を生成します。
#pragma ISTACKSIZE	割り込みスタックセクション (istack)の出力及び、割り込みスタックのトップラベル名を生成します。
#pragma CREG	本 pragma で宣言された内部レジスタにアクセスする場合、専用命令を使用してアクセスするコードを生成します。
#pragma sectaddress	本 pragma で宣言されたセクション名でセクション定義を行います。 同時にアドレス指定された場合は、疑似命令.org を使用したアドレス定義を出力します。
#pragma entry	本pragma で宣言された関数に対してスタックフレームを構築する enter 命令を出力しません。 これは、スタックポインタ初期化前に enter 命令を生成しないようにするためです。
#pragma interrupt/V	本 pragma で宣言された関数に対して割り込みベクタのみを定義します。

B.6.2 メモリ配置に関する拡張機能

本コンパイラは、以下に示すメモリの配置に関する拡張機能を持っています。

#pragma BIT

1ビット操作命令宣言機能

機　能: 絶対ビット命令アドレッシングを使用できる領域にある変数を宣言します。

書　式: #pragma BIT△変数名

解　説: M16C シリーズ,R8C ファミリでは、00000H～01FFFH 番地の領域にある変数に対して ROM 効率の良い絶対ビット命令アドレッシングを使用できます。
#pragma BIT で宣言した変数は絶対ビット命令アドレッシングを使用できる領域にある変数とみなされます。

規　定:
(1) #pragma BIT を変数以外について使用した場合、無効になります。
(2) #pragma BIT で宣言した変数が絶対ビット命令アドレッシングを使用できない場合、アドレスレジスタ間接ビット命令アドレッシング等を使用します。
(3) 本宣言は、変数宣言の前に行ってください。

使 用 例:

```
#pragma BIT bit_data
struct bit_data{
        char bit0:1;
        char bit1:1;
        char bit2:1;
        char bit3:1;
        char bit4:1;
        char bit5:1;
        char bit6:1;
        char bit7:1;
}bit_data;
func( void )
{
        bit_data.bit1 = 0;
                :
            (省略)
                :
```

図B.60 #pragma BIT 宣言の使用例

補　足: 絶対ビット命令アドレッシングを使用する命令は以下の条件で生成します。
(1) -fbit (-fB)オプションが指定され、かつ操作対象が near 型変数である時
(2) #pragma SBDATA で宣言されている変数である時
(3) #pragma ADDRESS で宣言されている変数で、かつその変数が 0000H 番地から 01FFFH 番地の間に配置する時
(4) #pragma BIT で宣言されている変数である時
(5) FB レジスタ相対の 32 バイトに収まっている変数である時

#pragma SBDATA

SB 相対アドレッシング使用変数宣言機能

機　能: SB 相対アドレッシングを使用する変数データであることを宣言します。

書　式: #pragma SBDATA△変数名

解　説: M16C シリーズ,R8C ファミリでは、SB 相対アドレッシングを使用すると効率の良い命令を選択することができます。#pragma SBDATA では、変数のデータ参照時に SB 相対アドレッシングを使用することを宣言します。この機能により ROM 効率の良いコードを生成できます。

規　定:
(1) #pragma SBDATA を宣言した変数は、アセンブラ指示命令.SBSYM で宣言されます。
(2) 変数名以外を指定した場合、無効となります。
(3) 指定した変数が関数内で宣言された static 変数の場合、無効となります。
(4) #pragma SBDATA を宣言した変数は、領域確保の際に SBDATA 属性のセクションに配置されます。
(5) ROM データに対して#pragma SBDATA を宣言した場合、SBDATA 属性のセクションに配置されません。
(6) -fauto_over_255 (-fAO2) オプション指定時は、#pragma SBDATA 宣言は無効になり、警告メッセージ“compile option -fauto_over_255 is specified, #pragma SBDATA was ignored” を出力します。
(7) 本宣言は、変数宣言の前に行ってください。

使 用 例:

```
#pragma   SBDATA   sym_data
struct sym_data{
                char        bit0:1;
                char        bit1:1;
                char        bit2:1;
                char        bit3:1;
                char        bit4:1;
                char        bit5:1;
                char        bit6:1;
                char        bit7:1;
}sym_data;

void            func( void )
{
                sym_data.bit1 = 0;
                    :
                (省略)
                    :
```

図B.61　#pragma SBDATA 宣言の使用例

補　足: NC30 では、SB レジスタはリセット後初期化され、以降は固定として使用することを前提としています。

#pragma SECTION

セクション名変更機能

機　　能:　コンパイラが生成するセクション名を変更します。

書　　式:　#pragma SECTION△既定セクション名△変更セクション名

解　　説:　この宣言を program セクションに対して行った場合は、その#pragma 宣言以降に記述された関数のセクション名を変更します。
なお、この機能を使用してセクション名を変更した場合、セクション名の追記 / 変更、必要があれば該当するセクション領域の初期化等、スタートアッププログラムを変更してください。
#pragma SECTION で変更可能な既定セクションは、program、rom、data、bss の 4 種類のみです。

使 用 例:

C 言語ソースプログラム:

```
#pragma   SECTION    program    pro1          ← program セクション名を pro1 に変更
void      func( void );
              :
          (以下省略)
```

アセンブラソースプログラム:

```
;### FUNCTION func
          .section    pro1                    ← pro1 セクションに配置
          ._file      'smp.c'
          ._line      9
          .glb        _func
_func:
```

data セクションに対して複数回変更する場合の例:

```
#pragma   SECTION   data   data1
int       i =0;                               ← data1_NE セクションに配置

void      func(void)
{
          (省略)
}

#pragma   SECTION   data   data2
int       j =1;                               ← data2_NE セクションに配置

void      sub(void)
{
          (省略)
}
```

図B.62　#pragma SECTION 宣言の使用例

備　　考:　セクションの名称を変更するときには、セクション名の後ろにセクションの配置属性(_NE、_NEI 等)が付加されるのでご注意ください。
既定セクション名として program、data、rom、bss、interrupt のどれでもない文字列を用いた場合は、警告を出してこのプラグマ行を無視します。

#pragma SECTION

セクション名変更機能

注意事項: 本コンパイラの V.3.10 以前では、data 及び rom セクションは bss セクションと同様に、ファイル単位でのみセクション名を変更できました。
このため、V.3.10 以前で作成したプログラムの場合、#pragma SECTION の記述位置に注意が必要です。
また、V.5.43 から bss セクションも data セクションと同様に複数回セクション名を変更することが可能になりました。
そのため、bss セクションについても#pragma SECTION の記述位置に注意が必要です。
文字列データは、最後に宣言された" rom "セクション名で出力されます。
既定セクション名にとして program,data,rom,bss のどれでもない文字列を用いた場合は、警告を出してこのプラグマ行を無視します。

#pragma STRUCT

構造体配列制御機能

機　能:　(1) 構造体のパックを禁止します。
(2) 構造体メンバ配置の並び換えを行います。

書　式:　(1) #pragma STRUCT△構造体のタグ名△unpack
(2) #pragma STRUCT△構造体のタグ名△arrange (C 言語のみ有効です)

解　説:　本コンパイラでは、デフォルトで構造体をパックします。例えば、【図B.63】に示す構造体は奇数サイズですが、構造体の末尾にアライメントのためのパディング(透き間)を入れません。
アライメントが必要な場合は、#pragma STRUCT unpack で構造体を宣言します。
また、構造体のメンバについては、メンバ間にパディングを入れることなく、常にパックします。
メンバ間にパディングを入れないかわりに、#pragma STRUCT arrange でメンバを並び換えることによってアライメントをとることができます。

```
struct s {
    int     i;
    char    c;
    int     j;
};
```

メンバ名	データ型	データサイズ	配置位置 (オフセット)
i	int	16 ビット	0
c	char	8 ビット	2
j	int	16 ビット	3

図B.63　構造体メンバの配置例 (1)

規　定:　(1) 構造体のパックの禁止
本コンパイラでは、拡張機能として構造体のアライメントを制御することができます【図B.63】に示した構造体を#pragma STRUCTでパックを禁止したときの例を【図B.64】に示します。

```
struct s {
    int     i;
    char    c;
    int     j;
};
```

メンバ名	データ型	データサイズ	配置位置 (オフセット)
i	int	16 ビット	0
c	char	8 ビット	2
j	int	16 ビット	3
パディング	(char)	8 ビット	-

図B.64　構造体メンバの配置例 (2)

【図B.64】に示したように、構造体メンバのサイズの合計が奇数バイトのとき、#pragma STRUCTを用いることにより最後のメンバ配置位置の後に、1バイトのパディングが入ります。したがって、#pragma STRUCTでパックを禁止したときの構造体は、すべて偶数バイトのサイズとなります。

#pragma STRUCT

構造体配列制御機能

規　　定:　(2)　メンバ配置の並び換え

本コンパイラでは、拡張機能として構造体の偶数サイズのメンバを先に配置し、奇数サイズのメンバを後に配置することができます。【図B.63】に示した構造体を#pragmaSTRUCTで配置を並び替えたときの配置例を【図B.65】に示します。

```
struct s {
    int     i;
    char    c;
    int     j;
};
```

メンバ名	データ型	データサイズ	配置位置 (オフセット)
i	int	16 ビット	0
j	int	16 ビット	2
c	char	8 ビット	4

図B.65　構造体メンバの配置例 (3)

パックの禁止及びメンバ配置の並び替えのための#pragma STRUCT は、必ず構造体のメンバ定義を行う前に宣言してください。

(3)　構造体タグは、テンプレートクラスを指定できません。

使 用 例:

```
#pragma   STRUCT   TAG   unpack
struct TAG {
            int         i;
            char        c;
} s1;
```

図B.66　#pragma STRUCT 宣言の使用例

備　　考:　本機能の arrange は、Cプログラムとしてコンパイルする場合のみ有効です。

注意事項:　unpack、または arrange の記述がない場合は警告を出力します。この場合、本#pragma 指定は無効です。

#pragma EXT4MPTR

4M バイト拡張空間ポインタ宣言機能

機　　能:　4M バイト拡張空間をアクセスするポインタであることを宣言します。

書　　式:　#pragma EXT4MPTR△ポインタ変数名

解　　説:　M16C/62 グループの一部の製品が持つ、拡張モード 2（4M バイト拡張モード）に対応した機能です。
4M バイト空間をアクセスするポインタ変数を宣言します。これによりコンパイラは、4M バイト空間をアクセスするために、バンク切り替えのコードを生成します。このバンク切り替えのコードは、関数単位で、最初にそのポインタが使用された所で生成されます。従って、連続した操作である場合においても 1 回だけ、バンク設定が行われます。
複数のポインタ変数を使用する場合等では、アクセス毎に毎回バンク設定を行う「-fchange_bank_always(- fCBA)オプション」を使用してください。

規　　定 :
(1) オプション-fchange_bank_always(- fCBA)を指定しない場合は、1 翻訳単位に” #pragma EXT4MPTR” を 2 回以上書くと、エラーメッセージ” multiple #pragma EXT4MPTR's pointer” を出力します。
(2) “#pragma EXT4MPTR” の後に空白類文字しか無い、あるいはポインタ変数名として不正な文字(変数名 1234 等はありえない)がある場合は、警告” #pragma EXT4MPTR format error,ignored” を出してこの行を無視します。
(3) 本機能は C プログラムとしてコンパイルする場合のみ有効です。

使 用 例:

C 言語ソースプログラム:

```
#pragma EXT4MPTR pointer
struct tagh{
        int bitmap;
        char code;
} far *pointer;
void main(void)
{
        int data;
        data = pointer->bitmap;
}
```

アセンブリ言語プログラム(抜粋):

```
mov.w   _pointer,A0
mov.w   _pointer+2,A1
.glb    __BankSelect
mov.b   A1,__BankSelect      ←バンク切り替え
bclr    3,A1
bset    2,A1
lde.w   [A1A0],-2[FB]
```

図B.67　#pragma EXT4MPTR 宣言の使用例

注意事項:
(1) 本機能を使用する前に、マイコンおよびシステム（ハードウエア）が、4M バイト拡張モードを使用可能かどうか確認行ってください。
(2) -R8C および-R8CE オプションを使用した場合、本宣言は無視されます。
(3) 本宣言は、変数宣言の前に行ってください。

_ext4mptr

4M バイト拡張空間ポインタ宣言機能

機　　能:　4M バイト拡張空間をアクセスするポインタであることを宣言します。

書　　式:　_ext4mptr far△ポインタ変数宣言

解　　説:　M16C/62 グループの一部の製品が持つ、拡張モード 2（4M バイト拡張モード）に対応した機能です。
4M バイト空間をアクセスするポインタ変数を宣言します。これによりコンパイラは、4M バイト空間をアクセスするために、バンク切り替えのコードを生成します。このバンク切り替えのコードは、関数単位で、最初にそのポインタが使用された所で生成されます。従って、連続した操作である場合においても 1 回だけ、バンク設定が行われます。
複数のポインタ変数を使用する場合等では、アクセス毎に毎回バンク設定を行う「-fchange_bank_always(- fCBA)オプション」を使用してください。

規　　定 :　本機能は C プログラムとしてコンパイルする場合のみ有効です。

使 用 例:

C 言語ソースプログラム:

```
struct tagh{
        int bitmap;
        char code;
};
struct tagh _ext4mptr far *pointer;
main()
{
        int data;
        data = pointer->bitmap;
}
        mov.w _pointer,A0
        mov.w _pointer+2,A1
        mov.w A1,__BankSelect     ← バンク切り替え
        bclr 3,A1
        bset 2,A1
        lde.w [A1A0],-2[FB]
```

図B.68　_ext4mptr 宣言の使用例

注意事項:
(1)　本機能を使用する前に、マイコンおよびシステム（ハードウエア）が、4M バイト拡張モードを使用可能かどうか確認行ってください。
(2)　-R8C オプション及び-R8CE オプションを使用した場合、本宣言は無視されます。

B.6.3 組み込み機器に関する拡張機能の使用方法

本コンパイラは、以下に示す組み込みに関する拡張機能を持っています。

#pragma ADDRESS

入出力変数の絶対アドレス割り付け機能

機　能: 変数を絶対アドレスに割り付けます。

書　式: #pragma ADDRESS△変数名△絶対アドレス

解　説: この宣言により指定された絶対アドレスは、文字列としてアセンブラソースファイルに展開され、アセンブラ指示命令.EQU を用いて定義されます。したがいまして、数値の記述形式はアセンブラに依存します。アセンブラの数値表現を以下に示します。

- 2 進数数値の最後に'B'又は'b'を付けます。
- 8 進数数値の最後に'O'又は'o'を付けます。
- 10 進数整数のみで記述します。
- 16 進数数値の最後に'H'又は'h'を付けます。先頭が英文字(A～F)で始まる場合は先頭に 0 を付けます。

規　定:

(1) #pragma ADDRESS により指定された変数に対する extern、static 等の記憶クラスはすべて無効になります。
(2) #pragma ADDRESS により指定された変数は、関数外で定義された変数に対してのみ有効になります。
(3) 本宣言は変数宣言の前に行ってください。ただし、C言語をコンパイルする場合、本宣言の前に宣言した変数に対しても有効になります。
(4) 変数名以外を指定した場合、無効となります。
(5) C言語をコンパイルする場合、#pragma ADDRESS 宣言の二重定義はエラーとなりませんが、後に宣言したアドレスが有効になります。
(6) 初期化式を記述した場合はウォーニングとなり、記述した初期化式は無効となります。
(7) #pragma ADDRESS は通常、I/O 変数に対して使用するため、volatile 指定が無くても、volatile 指定がされているものとして処理されます。
(8) #pragma ADDRESS 宣言で宣言した変数は、外部参照できません。
(9) オプション-fnot_address_volatile(-fNAV)を指定すると、#pragma ADDRESS 指定した変数を volatile が指定されているとして扱いません。
(10) “#pragma ADDRESS”に続く文字列が空白類文字だけの場合、あるいは変数名として不正な文字列（123 など）が書かれている場合、アドレス部分に空白類文字だけがある場合は、警告” #pragma ADDRESS format error,ignored”を出力して、この行を無視します。
(11) アドレス部分の文字列に、8 ビット目が 1 である文字を含んでいると、警告” Kanji in #pragma ADDRESS”を出力します。

使 用 例:

```
#pragma  ADDRESS  port  24H        #pragma  ADDRESS  io  24H
int          io;

void         func(void)
{
             io = 10;
}
```

図B.69　#pragma ADDRESS 宣言

#pragma BITADDRESS

入出力変数のビット位置指定付き絶対アドレス割り付け機能

機　能:　指定した絶対アドレスの指定したビット位置に、変数を割り付けます。

書　式:　#pragma BITADDRESS△変数名△ビット位置, 絶対アドレス

解　説:　この宣言により指定した絶対アドレスは、文字列としてアセンブラソースファイルに展開され、アセンブラ指示命令.BITEQU を用いて定義されます。したがいまして、数値の記述形式はアセンブラに依存します。アセンブラの数値表現を以下に示します。また、ビット位置の記述可能範囲も以下に示します。

(1) ビット位置
- 0～65535 の範囲で 10 進数のみ。

(2) アドレス
- 2 進数数値の最後に'B'又は'b'を付けます。
- 8 進数数値の最後に'O'又は'o'を付けます。
- 10 進数整数のみで記述します。
- 16 進数数値の最後に'H'又は'h'を付けます。先頭が英文字(A～F)で始まる場合は先頭に 0 を付けます。

規　定:
(1) _Bool 型または bool 型の変数のみ、変数名に指定することができます。_Bool 型と bool 型以外の変数を指定した場合は、エラーとなります。
(2) #pragma BITADDRESS により指定された変数に対する extern、static 等の記憶クラスはすべて無効になります。
(3) #pragma BITADDRESS により指定された変数は、関数外で定義された変数に対してのみ有効になります。
(4) 本宣言は変数宣言の前に行ってください。ただし、C 言語をコンパイルする場合、本宣言の前に宣言した変数に対しても有効になります。
(5) 変数名以外を指定した場合、無効となります。
(6) #pragma BITADDRESS 宣言の二重定義はエラーとなります。
(7) 初期化式を記述した場合、エラーとなります。
(8) #pragma BITADDRESS は通常、I/O 変数に対して使用するため、volatile 指定が無くても、volatile 指定がされているものとして処理されます。
(9) オプション-fnot_address_volatile(-fNAV)を指定すると、#pragma ADDRESS 指定した変数を volatile が指定されているとして扱いません。
(10) “#pragma BITADDRESS”に続く文字列が空白類文字だけの場合、あるいは変数名として不正な文字列（123 など）が書かれている場合、アドレス部分に空白類文字だけがある場合は、警告” #pragma BITADDRESS format error,ignored”を出力して、この行を無視します。
(11) アドレス部分の文字列に、8 ビット目が 1 である文字を含んでいると、警告” Kanji in #pragma ADDRESS”を出力して、この行を無視します。
(12) 本宣言は、変数宣言の前に行ってください。

使 用 例:

```
#pragma   BITADDRESS   io   1, 100H
_Bool     io;

void      func(void)
{
          io = 1;
}
```

図B.70　#pragma BITADDRESS 宣言

#pragma INTCALL

ソフトウェア割り込み関数宣言機能

機　能:　ソフトウェア割り込み(int 命令)で呼び出す関数を宣言します。

書　式:　(1) #pragma INTCALL△INT 番号△アセンブラ関数名(レジスタ名,レジスタ名, ...)
(2) #pragma INTCALL△INT 番号△C 言語関数名( )

解　説:　指定した INT 番号によって int 命令を発行して、ソフトウェア割り込みにより関数の呼び出しを行ないます。

規　定:

- アセンブラ関数を宣言する場合
  - (1) #pragma INTCALL 宣言は、アセンブラ関数の関数原型宣言の前に行ってください。C 言語としてコンパイルする際には、関数原型宣言後に#pragma INTCALL があってもかまいません。
  - (2) 関数原型宣言では、以下の項目を守ってください。
    - (1) プロトタイプ宣言の引数の数と#pragma INTCALL 宣言の引数の数が一致している必要があります。
    - (2) アセンブラ関数の引数に以下の型は宣言できません。
      - 構造体型
      - 共用体型
      - double 型
      - long double 型
      - long long 型
    - (3) アセンブラ関数のリターン値の型として以下の関数は宣言できません。
      - 構造体、共用体を返す関数
  - (3) 呼び出し場合は、引数として以下のレジスタを使用することができます。
    - float 型、long 型 (32 ビットレジスタ)
      R2R0, R3R1
    - far * {far ポインタ}(32 ビットレジスタ)
      R2R0, R3R1, A1A0
    - int 型、near * {near ポインタ}(16 ビットレジスタ)
      A0, A1, R0, R1, R2, R3
    - char 型、_Bool 型 、bool 型(8 ビットレジスタ)
      R0L, R0H, R1L, R1H
    - レジスタ名を記述する際、大文字、小文字の区別はしません。
  - (4) INT 番号は、10 進数でのみ記述可能です。
- C/C++言語で実体を記述した関数を宣言する場合
  - (1) #pragma INTCALL 宣言は、関数原型宣言の前に行ってください。C 言語としてコンパイルする際には、関数原型宣言後に#pragma INTCALL があってもかまいません。
  - (2) プロトタイプ宣言では、以下の項目を守ってください。
    - (1) 関数原型宣言では、関数の呼び出し規則において、全ての引数がレジスタ渡しとなる関数のみ宣言できます。
    - (2) 関数の返却値の型が構造体型または共用体型の関数を宣言できません。
  - (3) レジスタ名が無い場合には括弧が省略できます。C 言語関数名の後ろの括弧は省略できます。
  - (4) INT 番号は、10 進数でのみ記述可能です。
  - (5) INT 番号は、0 から 63 が指定できます。それ以外はエラーとなり、メッセージ“Invalid #pragma INTCALL interrupt number”を出力します。

#pragma INTCALL

ソフトウェア割り込み関数宣言機能

使 用 例:

```
#pragma INTCALL 25 asm_func(R2R0, R1)
int asm_func(unsigned long, unsigned int);          アセンブラ関数の関数原型宣言

void        main(void)
{
            int         i;
            long        l;

            i = 0x7FFD;
            l = 0x007F;

            asm_func( l, i );                       ← アセンブラ関数の呼び出し
}
```

図B.71　#pragma INTCALL 宣言の使用例 ((1)アセンブラ関数を宣言する場合)

```
#pragma INTCALL 25 c_func();                ← 引数のレジスタ名を指定しないでください。
int c_func(unsigned int, unsigned int);     ← C 言語関数の関数原型宣言

void        main(void)
{
            int         i, j;

            i = 0x7FFD;
            j = 0x007F;

            c_func( i, j );                 ← C 言語関数の呼び出し
}
```

図B.72　#pragma INTCALL 宣言の使用例 ((2)C 言語で実体を記述した関数を宣言する場合)

備　　考:　付属のスタートアップファイルをご使用になる場合は、vector セクションの内容を変更して下さい。変更方法の詳細は、「第 2 章 スタートアッププログラムの準備」を参照してください。

#pragma INTERRUPT

割り込み関数の記述機能

機　能:　割り込み処理関数の宣言を行います。

書　式:
(1) #pragma INTERRUPT△[/B¦/E]△割り込み処理関数名
(2) #pragma INTERRUPT△[/B¦/E]△割り込みベクタ番号△割り込み処理関数名
(3) #pragma INTERRUPT△[/B¦/E]△割り込み処理関数名(vect=割り込みベクタ番号)

解　説:
(1) 割り込み処理関数を上記の書式で宣言することにより、関数の出入り口で以下の割り込みのための処理を行うコードを生成します。
- 入り口処理では、マイコンのすべてのレジスタ(SB レジスタを除く)をスタックに退避します。
- 出口処理では、退避したレジスタを復帰させて、REIT 命令でリターンします。

(2) 宣言時に以下のスイッチを指定できます。
- [/B]
  関数呼び出し時にレジスタをスタックに退避する代わりに裏レジスタへ切り換えることができます。これにより、高速な割り込み処理を実現することができます。
  裏レジスタを使用する場合、割り込みのネストにより裏レジスタが変更されないように注意してください。
- [/E]
  割り込みに入った直後に多重割り込みを許可します。これにより、割り込みの応答性が向上します。

(3) 宣言時に割り込みベクタ番号を指定できます。
(4) 割り込みベクタ番号を記述する場合、コンパイラは可変ベクタテーブルを自動生成します。
- 可変ベクタテーブルは、#pragma INTERRUPT 宣言したファイルごとのオブジェクトファイルに生成されます。
- 生成された可変ベクタテーブルは、optlnk が生成するマップファイルで確認することができます。

なお、自動生成された可変ベクタテーブルは、セクション vector を生成します。プログラムにセクション vector がある場合、リンクエラーが発生しますので、ご注意下さい。

(5) 本宣言は、プロトタイプ宣言の前に行ってください。

#pragma INTERRUPT

割り込み関数の記述機能

規　　定:

(1) 引数を持つ割り込み処理関数を記述した場合、コンパイル時にウォーニングを出力します。
(2) 戻り値を返す割り込み処理関数を記述した場合、コンパイル時にウォーニングを出力します。関数の戻り値は、必ず void 型であるように宣言してください。
(3) #pragma INTERRUPT 宣言以降に関数の実体を定義した関数のみ有効となります。
(4) 関数名以外を指定した場合、何の効果も及ぼしません。
(5) #pragma INTERRUPT 宣言の二重定義はエラーとなりません。
(6) スイッチ/E と/B を同時に指定すると、警告メッセージ” #pragma INTERRUPT conflict,ignored”を出力してこの行を無視します。
(7) 同じ割り込み処理関数に、異なる割り込みベクタ番号を記述した場合は、後に宣言されたベクタ番号が有効になります。

```
#pragma   INTTERUPT   intr(vect=10)
#pragma   INTTERUPT   intr(vect=20)          /* 割り込みベクタ番号 20 が有効 */
```

図B.73　異なる割り込みベクタ番号の記述例

(8) #pragma INTTERUPT 宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にウォーニングとなります。
- #pragma ALMHANDLER
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma CYCHANDLER
- #pragma TASK

(9) “#pragma INTERRUPT”の後に空白類文字だけが続く場合、あるいは割り込み処理関数名が不正な文字列（123 など）の場合は警告” #pragma INTERRUPT format error,ignored”を出力して、この行を無視します。
(10) ベクタ番号として、0～63 でない値を書いた場合は、警告” Invalid, #pragma INTERRUPT vector number”を出してこの行を無視します。
(11) 本宣言は、プロトタイプ宣言の前に行ってください。

使 用 例:

```
extern int    int_counter;

#pragma   INTERRUPT   /B   i_func

void          i_func(void)
{
              int_counter += 1;
}
```

図B.74　#pragma INTERRUPT 宣言の使用例

備　　考:

割り込みベクタ番号を記述しない#pragma INTERRUPT を使用する場合、セクション vector の定義が必要です。変更方法の詳細は、「第 2 章 スタートアッププログラムの準備」を参照してください。

#pragma PARAMETER

レジスタ渡しのアセンブラ関数宣言機能

機　能:　引数をレジスタに格納して渡すアセンブラ関数を宣言します。

書　式:　#pragma PARAMETER△アセンブラ関数名(レジスタ名, レジスタ名, ..)

解　説:　アセンブラ関数を呼び出すときに、その引数を以下のレジスタに格納して渡すことを宣言します。

- float 型、long 型 (32 ビットレジスタ)
  R2R0, R3R1
- far * {far ポインタ}(32 ビットレジスタ)
  R2R0, R3R1, A1A0
- int 型、near * {near ポインタ}(16 ビットレジスタ)
  A0, A1, R0, R1, R2, R3
- char 型、_Bool 型 (8 ビットレジスタ)
  R0L, R0H, R1L, R1H
- レジスタ名を記述する際、大文字、小文字の区別はしません。
- long long 型(64bit 整数型)、double 型、及び構造体型、共用体型は、宣言できません。

規　定:

(1) #pragma PARAMETER 宣言は、アセンブラ関数の関数原型宣言の前に行ってください。C 言語としてコンパイルする際には、関数原型宣言後に#pragma PARAMETER があってもかまいません。

(2) 関数原型宣言では、以下の項目を守ってください。

  (1) 関数原型宣言の引数の数と#pragma PARAMETER 宣言の引数の数が一致している必要があります。

  (2) アセンブラ関数の引数の型として、以下の型は宣言できません。
    - 構造体型
    - 共用体型
    - double 型
    - long double 型
    - long long 型

  (3) アセンブラ関数のリターン値の型として以下の関数は宣言できません。
    - 構造体、共用体を返す関数

(3) #pragma PARAMETER で指定した関数の出力シンボルは、常に_ (アンダースコア) が付加されます。

使 用 例:

```
#pragma PARAMETER asm_func(R0, R1)
int asm_func(unsigned int, unsigned int);                    アセンブラ関数の関数原型宣言
void        main(void)
{
            int             i, j;

            i = 0x7FFD;
            j = 0x007F;

            asm_func( i, j );                                ← アセンブラ関数の呼び出し
}
```

図B.75　#pragma PARAMETER 宣言の使用例

#pragma SPECIAL

スペシャルページサブルーチン呼び出し関数宣言機能

機　能:　スペシャルページサブルーチン呼び出し(JSRS 命令)の関数を宣言します

書　式:
(1) #pragma SPECIAL△呼び出し番号△関数名()
(2) #pragma SPECIAL△関数名(vect=呼び出し番号)

解　説:
(1) #pragma SPECIAL で宣言した関数は、スペシャルページベクタテーブルの要素に設定した番地に 0F0000H を加算したアドレスに配置されたものとして、スペシャルページサブルーチン呼び出しが行われます。
(2) リンク時にスペシャルページベクタテーブルを自動生成することができます。

必ず0F0000Hから 0FFFFFHの領域に配置してください。

規　定:
(1) #pragma SPECIAL で宣言した関数は、 program_S セクションに配置されます。program_S セクションは、~~必ず 0F000H から 0FFFFH の領域に配置してください。~~
(2) 呼び出し番号は、18 から 255 までです。また 10 進数のみ指定可能です。
(3) 同じ関数に、異なる呼び出し番号を記述した場合は、エラーになります。

```
#pragma   SPECIAL   func(vect=20)
#pragma   SPECIAL   func(vect=30)          /* 呼び出し番号 30 が有効 */
```

図B.76　異なる呼出し番号の記述例

(4) 関数が定義されているファイル、関数の呼び出しを行っているファイルが別のファイルの場合、その両方のファイルに本宣言を行ってください。
(5) 書式（1）での関数名の後ろの括弧は省略できます。
(6) (vect=スペシャルページ番号)における vect は全て小文字です。
(7) 本宣言は、関数のプロトタイプ宣言の前に行ってください。C 言語としてコンパイルする際には、関数原型宣言後に#pragma SPECIAL があってもかまいません。

使 用 例:

```
#pragma   SPECIAL   20   func()
void          func(unsigned int, unsigned int);

void          main(void)
{
              int i, j;

              i = 0x7FFD;
              j = 0x007F;

              func( i, j );              ← スペシャルページサブルーチン呼び出し
}
```

図B.77　#pragma SPECIAL 宣言の使用例

注意事項:　#pragma SPECIAL で指定する関数が既に他の#pragma で指定されている場合はコンパイルエラーになります。

B.6.4 その他の拡張機能の使用方法

本コンパイラは、前述以外に以下の拡張機能を持っています。

#pragma __ASMMACRO

アセンブラマクロ関数宣言機能

機　能: アセンブラのマクロで定義した関数を宣言します。

書　式: #pragma __ASMMACRO 関数名(レジスタ名,...)

規　定:

(1) 本宣言は、関数の関数原型宣言の前に行ってください。C 言語としてコンパイルする際には、関数原型宣言後に#pragma __ASMMACRO があってもかまいません。アセンブラマクロ関数は、必ず static 宣言してください。static 宣言が無い場合、アセンブルエラーになります。

(2) 引数のない関数は宣言できません。引数はレジスタ渡しになります。引数の型と合致するレジスタを指定してください(#pragma PARAMETER に準じます)。

(3) 宣言した関数名の先頭にアンダスコア(_)を付加したマクロ名で、アセンブラマクロを定義してください。

(4) 戻り値の返し方は、関数呼出し規則に従い、以下のようになります。集合体(構造体・共用体)を戻り値とすることはできません。

_Bool 型、char 型:	R0L	float 型:	R2R0
int、short 型:	R0	double 型:	R3R2R1R0
long 型:	R2R0	long long 型:	R3R1R2R0

(5) アセンブラマクロ内で内容が変更されるレジスタは、アセンブラマクロの先頭で退避して、復帰直前に復帰してください(戻り値の格納レジスタの退避・復帰は不要です)。

(6) 関数呼び出しよりも後に#pragma __ASMMACRO を宣言した場合は、エラーメッセージ” #pragma __ASMMACRO must be declared before use”を出力します。

(7) 関数で無い識別子に#pragma __ASMMACRO を宣言した場合は、警告メッセージ” #pragma __ASMMACRO not function,ignored”を出力して、このプラグマを無視します。

(8) 関数宣言が関数原型形式で無い場合は、警告メッセージ”#pragma __ASMMACRO's function must be prototyped,ignored”を出力して、このプラグマを無視します。

使 用 例:

```
#pragma __ASMMACRO mul( R0, R2 )
static long mul( int, int );                    /* 必ず static にしてください */

asm("   _mul     .macro¥n"
    "   mul.w   R2,R0¥n"
    "   .endm");

long        l;

void        test_func( void )
{
            l = mul( 2, 3 );
}
```

図B.78 #pragma __ASMMACRO 記述例

#pragma ASM～#pragma ENDASM

インラインアセンブラ指定機能

機　能:　アセンブリ言語で記述を行なう領域を指定します。

書　式:
#pragma ASM
アセンブリ言語記述
#pragma ENDASM

解　説:　#pragma ASM と#pragma ENDASM の間の領域を直接、アセンブラソースファイルにそのまま出力します。
#pragma ASM を記述する際は、必ず#pragma ENDASM を組み合わせて記述してください。#pragma ASM と対になる#pragma ENDASM がない場合、本コンパイラは処理を中断します。

規　定:
(1) アセンブリ言語記述において、レジスタの内容を変更する記述をしないでください。レジスタの内容を変更する記述をする場合は、push 命令・pop 命令を使用して、レジスタ内容の退避・復帰を行ってください。
(2) "#pragma ASM～#pragma ENDASM"内では、引数および auto 変数を参照しないでください。
(3) "#pragma ASM～#pragma ENDASM"内でフローに影響を与えるようなブランチ(条件ブランチ含む)を記述しないようにしてください。
(4) '$'、'_'で始まるシンボルはコンパイラのための予約シンボルです。"#pragma asm-endasm"内に'$'、'_'で始まるシンボル定義を記述したプログラムの動作を保証しません。
(5) "#pragma asm-endasm"内に指示命令".section"を記述しないでください。セクション名称を変更する場合には、必ず"#pragma asm-endasm"外で #pragma SECTION を用いるようにしてください。
(6) “#pragma ASM”の後に、空白類文字でない文字を書くと、警告を出して”#pragma ASM”の行を無視します。その結果として、”#pragma ASM”の次行を C ソースとして解釈してしまいます。
(7) “pragma ASM”の記述があり、”#pragma ENDASM”の記述が無いままファイルの読みこみが EOF に到達した場合、フェータルエラー” no #pragma ENDASM”を表示します。
(8) アセンブリ言語記述の 1 行が、改行コードを含めて 1024 文字を超える場合は、警告”#pragma ASM line too long,then cut”を出力して、1024 文字目以降から改行までを無視します。
(9) アセンブリ言語記述の行にコメント開始文字”;”が含まれている場合は、その行全体を環境変数 NCKOUT の設定に合わせて出力漢字コードを変換します。但し、オプション-E または-P を指定したときのみ、環境変数 NCKIN に合わせます。

#pragma ASM～#pragma ENDASM

インラインアセンブラ指定機能

使 用 例:

```
void        func(void)
{
            int         i, j;

            for(i=0; i < 10;i++){
                        func2();
}

#pragma   ASM
            FCLR        I
LOOP1:
            MOV.W       #0FFH,R0
                :
            (省略)
                :
            FSET        I
#pragma   ENDASM
}
```

この領域をアセンブラソースファイルにそのまま出力します。

図B.79　#pragma ASM(ENDASM)記述例

補　　足: #pragma ASM～#pragma ENDASM までに記述されたアセンブリ言語プログラムは、C 言語プリプロセッサの処理対象となります。

#pragma PAGE

.PAGE 指示命令出力機能

機　能: アセンブラリストファイルの改ページの指定を行います。

書　式: #pragma PAGE

解　説: ソースファイル中に#pragma PAGE を記述した場合、コンパイラが出力するアセンブラソースファイルに、アセンブラ指示命令.PAGE を出力します。アセンブラによりアセンブラリストファイルを出力する場合に、改ページ指定を行うことができます。

規　定:
(1) アセンブラ指示命令.PAGE のヘッダに指定する文字列の指定はできません。
(2) auto 変数の宣言中に#pragma PAGE を記述できません。
(3) 本機能は、Cプログラムとしてコンパイルする場合のみ有効です。

使 用 例:

```
void        func(void)
{
            int         i, j;

            for(i=0; i < 10;i++){
                        func2();
            }
#pragma PAGE
            i++;
}
```

図B.80　#pragma PAGE 記述例

B.7 アセンブラマクロ関数

B.7.1 アセンブラマクロ関数の概要

本コンパイラでは、アセンブリ言語の一部を C 言語の関数として記述することができます。

この機能を使用することにより、特定のアセンブラの命令を直接的に C 言語のプログラム上に記述できるので、プログラムのチューンアップが行いやすくなります。

B.7.2 アセンブラマクロ関数の記述例

アセンブラマクロ関数は【図B.81】のように、C言語プログラム中にC言語の関数と同じ書式で記述することができます。

アセンブラマクロ関数機能を使用する場合は、必ず asmmacro.h をインクルードしてください。

```
#include   <asmmacro.h>                  /* アセンブラマクロ関数の定義ファイルをインクルード */
long       l;
char       a[20];
char       b[20];

void       func(void)
{
           l = rmpa_b(0,19,a,b);         /* アセンブラマクロ関数(rmpa 命令) */
}
```

図B.81 アセンブラマクロ関数の記述例

B.7.3 アセンブラマクロ関数で記述可能な命令

アセンブラマクロ関数で記述可能なアセンブリ言語と、アセンブラマクロ関数としての機能と書式を示します。

ABS

機　能: val の絶対値を返します。

書　式:
```
#include <asmmacro.h>

static signed char abs_b( signed char val );
/* 8 ビットでの演算の場合 */

static signed int abs_w( signed int val );
/*16 ビットでの演算の場合 */
```

DADC

機　能: val1 と val2 のキャリー付き 10 進加算の結果を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char dadc_b(unsigned char val1,unsigned    char val2);
/* 8 ビットでの演算の場合 */

static unsigned int    dadc_w(unsigned int val1, unsigned int val2);
/* 16 ビットでの演算の場合 */
```

DADD

機　能: val1 と val2 のキャリーなし 10 進加算の結果を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char dadd_b(unsigned char val1, unsigned char val2);
/* 8 ビットでの演算の場合 */

static unsigned int    dadd_w(unsigned int val1, unsigned int val2);
/* 16 ビットでの演算の場合 */
```

DIV

機　　能: val1 と val2 の符号付き除算を行い商を返します。

書　　式:
```
#include <asmmacro.h>

static signed char div_b( signed char val1, signed int val2 );
/* 符号付きの 16 ビット/ 8 ビットの演算の場合 */

static signed int div_w( signed int val1, signed long val2 );
/* 符号付きの 32 ビット/ 16 ビットの演算の場合 */
```

DIVU

機　　能: val1 と val2 の符号なし除算を行い商を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char divu_b( unsugned char val1, unsigned int val2 );
/* 符号なしの 16 ビット/ 8 ビットの演算の場合 */

static unsigned int divu_w( unsigned int val1, unsigned long val2 );
/* 符号なしの 32 ビット/ 16 ビットの演算の場合 */
```

DIVX

機　　能: val1 と val2 の符号なし除算を行い商を返します。剰余の符号は、除数の符号と同一です。

書　　式:
```
#include <asmmacro.h>

static signed char divx_b( unsugned char val1, signed int val2 );
/* 符号なしの 16 ビット/ 8 ビットの演算の場合 */

static signed int divx_w( signed int val1, signed long val2 );
/* 符号なしの 32 ビット/ 16 ビットの演算の場合 */
```

MOD,MODU

機　　能: val1 と val2 の除算を行ない余りを返します。

書　　式:
```
#include <asmmacro.h>

static signed char mod_b(signed char val1,signed int val2);
/* 符号付きの 16 ビット/ 8 ビット の演算の場合 */

static signed int mod_w(signed int val1,signed long val2);
/* 符号付きの 32 ビット/ 16 ビットの演算の場合 */

static unsigned char modu_b(unsigned char val1,unsigned int val2);
/* 符号なしの 16 ビット/ 8 ビット の演算の場合 */

static unsigned int modu_w(unsigned int val1,unsigned long val2);
/* 符号なしの 32 ビット/ 16 ビット の演算の場合 */
```

NOT

機　　能: val を反転した値を返します。

書　　式:
```
#include <asmmacro.h>

static signed char not_b(signed char val);
/*8bit での演算の場合*/

static signedd int not_w(signed int val);
/*16bit での演算の場合*/
```

NEG

機　　能: val の 2 の補数を返します。

書　　式:
```
#include <asmmacro.h>

static signed char neg_b(signed char val);
/*8bit での演算の場合*/

static signed int neg_w(signed int val);
/*16bit での演算の場合*/
```

DSBB

機　　能: val2 から val1 のボロー付き 10 進減算の結果を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char dsbb_b(unsigned char val1, unsigned char val2);
/* 8 ビットでの演算の場合 */

static unsigned int   dsbb_w(unsigned int val1, unsigned int val2);
/* 16 ビットでの演算の場合 */
```

DSUB

機　　能: val2 から val1 のボローなし 10 進減算の結果を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char dsub_b(unsigned char val1, unsigned char val2);
/* 8 ビットでの演算の場合 */

static unsigned int   dsub_w(unsigned int val1, unsigned int val2);
/* 16 ビットでの演算の場合 */
```

MOVdir

機　　能: val1 から val2 への 4 ビット転送を行ないます。

書　　式:
```
#include <asmmacro.h>

static unsigned char movll( unsigned char val1, unsigned char val2 );
/*  下位 4 ビットから下位 4 ビットへの転送 */

static unsigned char movlh( unsigned char val1, unsigned char val2 );
/*  下位 4 ビットから上位 4 ビットへの転送 */

static unsigned char movhl( unsigned char val1, unsigned char val2 );
/*  上位 4 ビットから下位 4 ビットへの転送 */

static unsigned char movhh( unsigned char val1, unsigned char val2 );
/*  上位 4 ビットから上位 4 ビットへの転送 */
```

RMPA

機　能: 初期値：init、回数：count、乗数の格納されている先頭アドレスをそれぞれ p1、p2 として積和演算を行い、結果を返します。

書　式:
```
#include <asmmacro.h>

static int rmpa_b(signed int init, unsigned int count, signed char _near *p1, signed char _near *p2);
/* 8 ビット での演算の場合 */

static long rmpa_w(signed long init, unsigned int count, signed int _near *p1, signed int _near *p2);
/* 16 ビットでの演算の場合 */
```

SMOVF

機　能: p1 で示される転送元番地から、p2 で示される転送先番地に count 回数分、アドレスの加算方向へストリング転送を行います。戻り値はありません。

書　式:
```
#include <asmmacro.h>

static void smovf_b(unsigned char _near *p1, unsigned _near char *p2, unsigned int count);
/* 8 ビットでの演算の場合 */

static void smovf_w(unsigned int _near *p1, unsigned _near int *p2, unsigned int count);
/* 16 ビットでの演算の場合 */
```

SHA

機　能: val を count 回数分、算術シフトした値を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char sha_b( signed char count, unsigned char val );
/* 8 ビットでの演算の場合 */

static unsigned int sha_w( signed char count, unsigned int val );
/* 16 ビットでの演算の場合 */

static unsigned long sha_l( signed char count, unsigned long val );
/* 32 ビットでの演算の場合 */
```

SHL

機　能: val を count 回数分、論理シフトした値を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char shl_b( signed char count, unsigned char val );
/* 8 ビットでの演算の場合 */

static unsigned int shl_w( signed char count, unsigned int val );
/* 16 ビットでの演算の場合 */

static unsigned long shl_l( signed char count, unsigned long val );
/* 32 ビットでの演算の場合 */
```

SMOVB

機　能: p1 で示される転送元番地から、p2 で示される転送先番地に count 回数分、アドレスの減算方向へストリング転送を行います。戻り値はありません。

書　式:
```
#include <asmmacro.h>

static void smovb_b(unsigned char _near *p1, unsigned char _near *p2, unsigned int count);
/* 8 ビットでの演算の場合 */

static void smovb_w(unsigned int _near *p1, unsigned int _near *p2, unsigned int count);
/* 16 ビットでの演算の場合 */
```

SSTR

機　能: val をストアするデータ、p を転送するアドレス、count を転送回数としてストリングストアを行います。戻り値はありません。

書　式:
```
#include <asmmacro.h>

static void sstr_b(unsigned char val, unsigned char _near *p, unsigned int count);
/* 8 ビットでの演算の場合 */

static void sstr_w(unsigned int val, unsigned int _near *p, unsigned int count);
/* 16 ビットでの演算の場合 */
```

ROLC

機　　能: val を C フラグを含めて、1 ビット左へ回転した値を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char rolc_b( unsigned char val );
/* 8 ビットでの演算の場合 */

static unsigned int rolc_w( unsigned int val );
/* 16 ビットでの演算の場合 */
```

RORC

機　　能: val を C フラグを含めて、1 ビット右へ回転した値を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char rorc_b( unsigned char val );
/* 8 ビットでの演算の場合 */

static unsigned int rorc_w( unsigned int val );
/* 16 ビットでの演算の場合 */
```

ROT

機　　能: val を count 回数分、回転した値を返します。

書　　式:
```
#include <asmmacro.h>

static unsigned char rot_b( signed char count, unsigned char val );
/* 8 ビットでの演算の場合 */

static unsigned int rot_w( signed char count, unsigned int val );
/* 16 ビットでの演算の場合 */
```

付録C 翻訳限界

コンパイラの翻訳限界を表 C.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C.1　コンパイラの翻訳限界 (1/2)

項目	仕様
ソースファイルでの 1 行の文字数	改行コードを含む 512 バイト(文字)
ソースファイルでの行数	最大 65535 行
指定可能なファイル数	制限なし（メモリ容量依存）
ファイル名の長さ	OS に依存します
起動オプション-D で指定可能なマクロ名の総数	制限なし（メモリ容量依存）
起動オプション-I で指定可能なディレクトリ数	最大 256
起動オプション-as30 で引き渡し可能なパラメータ数	制限なし（メモリ容量依存）
複文、繰り返し制御構造、及び選択制御構造に対するネスト数	制限なし（メモリ容量依存）
条件コンパイルにおけるネスト数	制限なし（メモリ容量依存）
宣言中の基本型を修飾するポインタ、配列、及び関数宣言子の数	制限なし（メモリ容量依存）
関数定義の数	制限なし（メモリ容量依存）
1 つのブロック中におけるブロック有効範囲を持つ識別子の数	制限なし（メモリ容量依存）
1 つのソースファイル中で同時に定義され得るマクロ識別子の数	制限なし（メモリ容量依存）
マクロ名の置き換えの数	制限なし（メモリ容量依存）
入力プログラムにおける論理ソース行の数	制限なし（メモリ容量依存）
#include ファイルに対するネスト数	最大 40
1 つの switch 文中における case 名札の数 (switch 文のネストがない場合)	制限なし（メモリ容量依存）
#if、#elif 文で指定できる演算子、被演算子の合計数	制限なし（メモリ容量依存）
関数 1 個あたりで確保可能なスタックフレーム容量(バイト数)	最大 64K バイト
#pragma ADDRESS で定義可能な変数の数	制限なし（メモリ容量依存）
括弧のネスト数	制限なし（メモリ容量依存）
初期化式付きの変数定義を行う場合の定義可能な初期値の数	制限なし（メモリ容量依存）
修飾宣言子のネスト数	YACC スタックに依存
宣言子の括弧によるネスト数	YACC スタックに依存
演算子の括弧によるネスト数	YACC スタックに依存
1 つの内部識別子又はマクロ名で意味をもつ文字数	~~制限なし（メモリ容量依存）~~ 最大200
1 つの外部識別子で意味をもつ文字数	~~制限なし（メモリ容量依存）~~ 最大200
1 ソースファイル中の外部識別子の数	制限なし（メモリ容量依存）
1 ブロックでブロックスコープを持つ識別子	制限なし（メモリ容量依存）
1 ソースファイル中のマクロ数	制限なし（メモリ容量依存）
1 つの関数、関数呼び出しのパラメータ数	制限なし（メモリ容量依存）
1 つのマクロ定義、マクロ呼び出しのパラメータ数	最大 31
結合後の文字列リテラル内の文字数	制限なし（メモリ容量依存）
1 つのオブジェクトサイズ(バイト数)	制限なし（メモリ容量依存）

表C.2 コンパイラの翻訳限界 (2/2)

項目	仕様
1つの構造体／共用体のメンバ数	制限なし（メモリ容量依存）
1つの列挙中の列挙定数の数	制限なし（メモリ容量依存）
1つの struct 宣言リスト中の構造体/共用体のネスト数	制限なし（メモリ容量依存）
1つの文字列の文字数	OS に依存します
1ファイルの行数	制限なし（メモリ容量依存）
識別子の最大長	200 文字

付録D C/C++言語実装仕様

本コンパイラが扱うデータの内部構造/配置、演算時等における符号拡張規則と、関数の呼び出し、および関数からの戻り値に関する規則を説明します。

D.1 言語仕様

a. キーワード

本コンパイラは、以下のものをキーワードとして解釈します。

C/C++プログラム共通キーワード：

_asm	_far	_near	asm	auto
break	case	char	const	continue
default	do	double	else	enum
extern	far	float	for	goto
if	inline	int	long	near
register	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	_inline

C プログラム専用キーワード：

_Bool	restrict	_ext4mptr

C++プログラム専用キーワード：

bool	catch	class	const_cast	delete
dynamic_cast	explicit	false	friend	mutable
namespace	new	operator	private	protected
public	reinterpret_cast	static_cast	template	this
throw	true	try	typeid	typename
using	virtual	wchar_t	and	and_eq
bitand	bitor	compl	not	not_eq
or	or_eq	xor	xor_eq	

C++プログラムでは inline をキーワードとして扱います。C++プログラムとしてコンパイルする際、コンパイルオプション-fnot_reserve_inline は無効です。

b. 整数定数

整数定数は、10 進数のほか、8 進数、16 進数、2 進数を指定することができます。各進数の書式を【表D.1】に示します。

表D.1　整数定数の記述法

進数	記述法	構成	記述例
10 進数	0(ゼロ)以外の数字で始まる	0123456789	15
8 進数	0(ゼロ)で始まる	01234567	017
16 進数	0X 又は 0x で始まる	0123456789abcdefABCDEF	0XF 又は 0xf
2 進数	0B 又は 0b で始まる	01	0B1 又は 0b1

2 進数では'_'は無視されます。'_'を視覚的な区切りとして使用できます。
例)　char port = 0b_0_111_1_011;　　/* 0b01111011 と同じ値です */

整数定数の型は、その値の大小により以下の順で決定されます。

- 8 進数、16 進数、2 進数
  signed int 型→unsigned int 型→signed long 型→unsigned long 型→signed long long 型→unsigned long long 型
- 10 進数（C 言語の場合）
  signed int 型→signed long 型→signed long long 型
  10 進数（C++言語の場合）
  signed int 型→signed long 型→unsigned long 型→signed long long 型→unsigned long long 型

また、接尾子 U 又は u、L 又は l、LL 又は ll を付加した場合、以下のように扱われます。

(1)　符号無し定数
符号無し定数は、定数値の後に U 又は u を付加して記述します。型は値により以下の順で決定されます。
unsigned int 型→unsigned long 型→unsigned long long 型

(2)　long 型定数
long 型定数は、定数値の後に L 又は l を付加して記述します。型は値により以下の順で決定されます。

- 8 進数、16 進数、2 進数
  signed long 型→unsigned long 型→signed long long 型→unsigned long long 型
- 10 進数（C 言語の場合）
  signed long 型→signed long long 型
  10 進数（C++言語の場合）
  signed long 型→unsigned long 型→signed long long 型→unsigned long long 型

(3)　long long 型定数
long long 型定数は、定数値の後に LL 又は ll を付加して記述します。型は値により以下の順で決定されます。

- 8 進数、16 進数、2 進数
  signed long long 型→unsigned long long 型
- 10 進数（C 言語の場合）
  signed long long 型
  10 進数（C++言語の場合）
  signed long long 型→unsigned long long 型

D.2 データの内部表現

D.2.1 整数型

【表D.2】に整数型のデータが使用するバイト数を示します。

表D.2 整数型のデータサイズ

型	符号の有無	ビットサイズ	表現できる数値
_Bool	なし	8	0、1
char unsigned char	なし	8	0～255
signed char	有り	8	-128～127
int short signed int signed short	有り	16	-32768～32767
unsigned int unsigned short	なし	16	0～65535
long signed long	有り	32	-2147483648～2147483647
unsigned long	なし	32	0～4294967295
long long signed long long	有り	64	-9223372036854775808～ 9223372036854775807
unsigned long long	なし	64	0～18446744073709551615
bool	なし	8	false, true
wchar_t	なし	16	0～65535

- _Bool 型の符号指定はできません。
- char 型は、符号指定がない場合、unsigned char 型と解釈します。
- int 型、short 型は、符号指定がない場合 signed int 型、signed short 型と解釈します。
- long 型は、符号指定がない場合 signed long 型と解釈します。
- long long 型は、符号指定がない場合 signed long long 型と解釈します。
- 構造体のビットフィールドメンバで符号指定がない型は、符号なしと解釈します。
- long long 型のビットフィールドは使用できません。
- _Bool 型と bool 型はビット 0 のみを使用しています。上位 7 ビットは不定です。

D.2.2 浮動小数点型

【表D.3】に浮動小数点型のデータが使用するバイト数を示します。

表D.3 浮動小数点型のデータサイズ

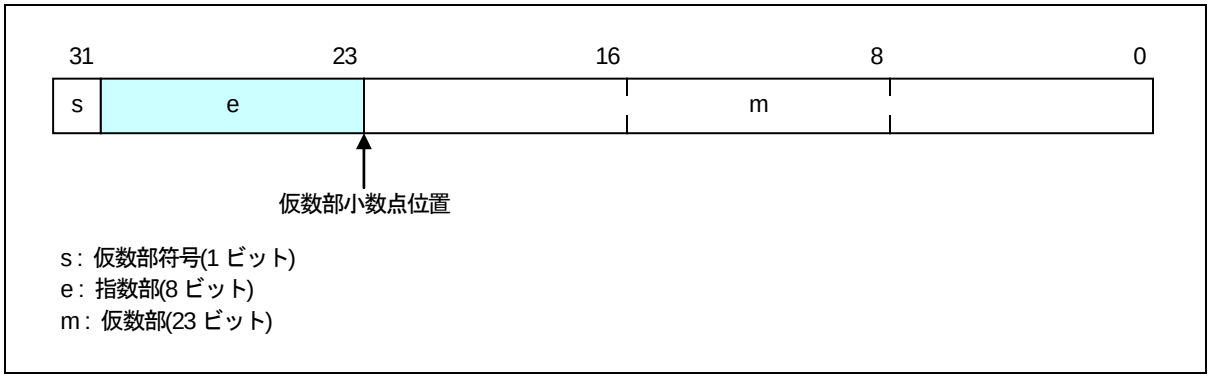
型	符号の有無	ビットサイズ	表現できる数値
float	有り	32	1.17549435e-38F～3.40282347e+38F
double long double	有り	64	2.2250738585072014e-308 ~1.7976931348623157e+308

コンパイルオプション-fdouble_32(-fD32)を使用する場合、型 double は型 float と同じ型とみなされます。

本コンパイラの浮動小数点フォーマットは、IEEE(The Institute of Electrical and Electronics Engineers)規格の形式に準拠しています。以下に、単精度/倍精度の浮動小数点フォーマットを示します。

(1) 単精度浮動小数点データフォーマット

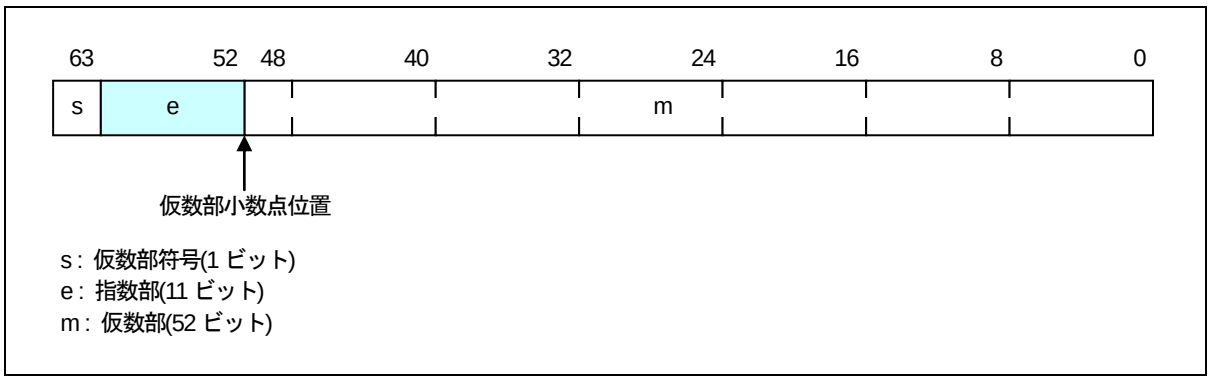
【図D.1】に示すデータ形式で 2 進数の浮動小数点(float)データを表現します。



図D.1 単精度浮動小数点データフォーマット

(2) 倍精度浮動小数点データフォーマット

【図D.2】に示すデータ形式で 2 進数の浮動小数点(double、long double)データを表現します。



図D.2 倍精度浮動小数点データフォーマット

D.2.3 列挙型

列挙型は、C 言語の場合は unsigned int 型、C++言語の場合は int 型と同じ内部表現となります。特に指定しない場合、メンバの出現順に 0、1、2 ..... の整数値が与えられます。列挙子（列挙型メンバ）の型は、C/C++共通で int 型となります。

また、コンパイルオプション"-fchar_enumerator(-fCE)"を使用することにより、列挙型と列挙子の型は unsigned char 型と同じ内部表現にできます。

D.2.4 ポインタ型

【表D.4】にポインタ型のデータが使用するバイト数を示します。

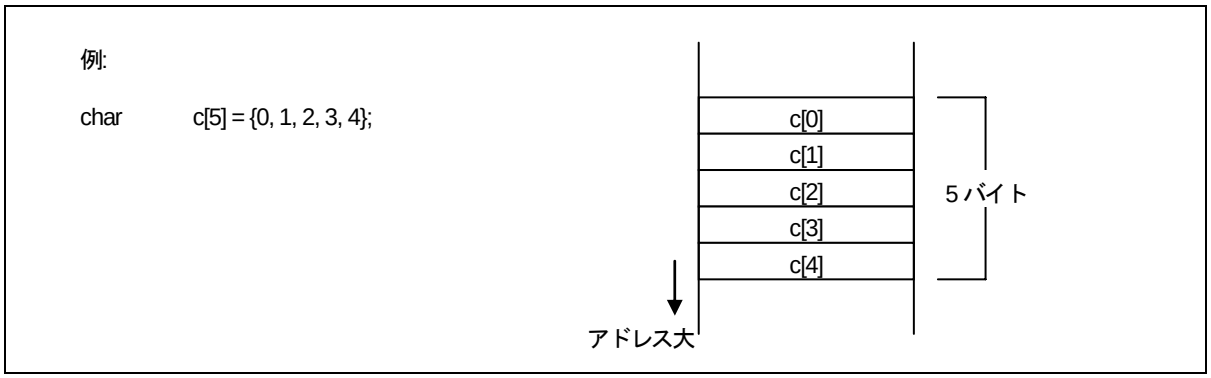
表D.4 ポインタ型のデータサイズ

型	符号の有無	ビットサイズ	表現できる数値
near ポインタ	なし	16	0～0xFFFF
far ポインタ	なし	32	0～0xFFFFFFFF

far ポインタは、32 ビット長の下位 20 ビットを有効ビットとして使用します。

D.2.5 配列型

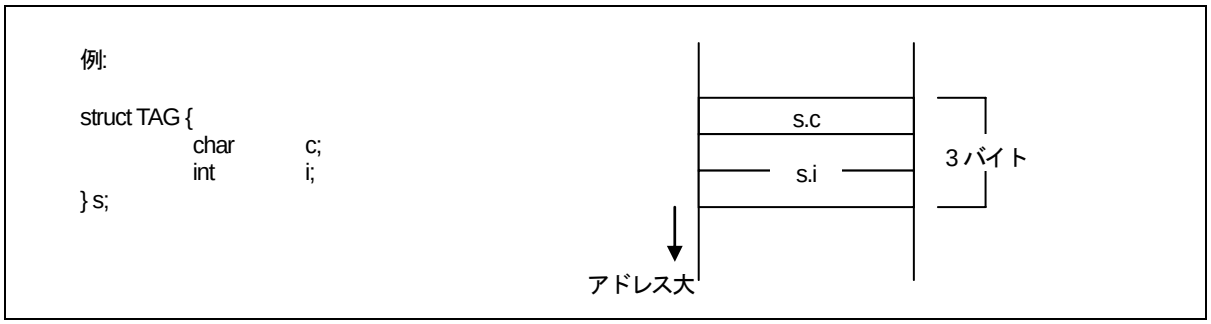
配列型は、要素のサイズ(バイト数)と要素数との積で表す領域に連続して配置されます(要素の出現順にメモリに配置されます)。【図D.3】に配置例を示します。



図D.3 配列の配置例

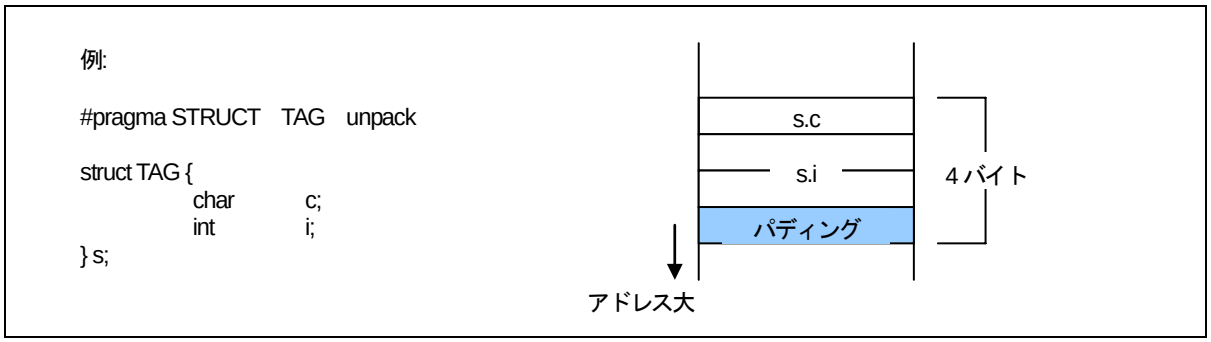
D.2.6 構造体型

構造体型は、メンバのデータを出現順に連続して配置します。【図D.4】に配置例を示します。



図D.4　構造体の配置例 (1)

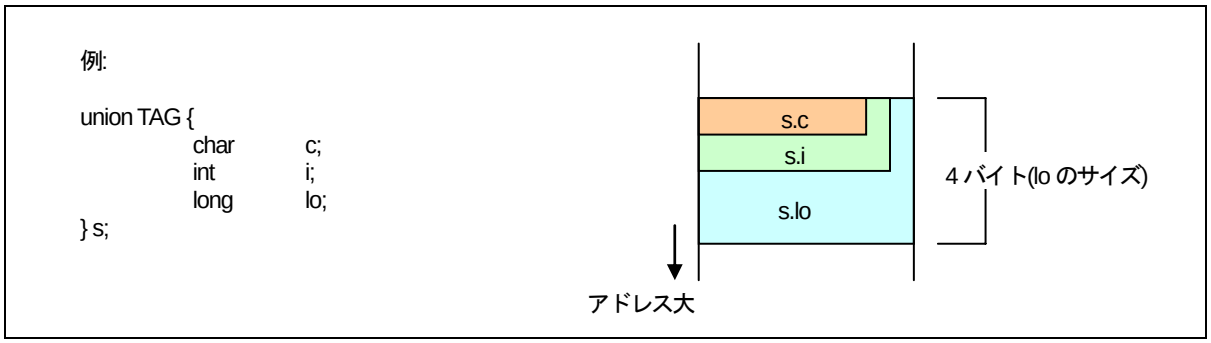
構造体は通常の場合、ワードアライメントを行いません。複数の構造体のメンバは連続して配置されます。ワードアライメントを行う場合は、拡張機能の#pragma STRUCTを使用します。#pragma STRUCTを使用することにより、メンバのサイズの合計が奇数バイトであるときに１バイトのパディングを付加します。【図D.5】に配置例を示します。



図D.5　構造体の配置例 (2)

D.2.7 共用体型

共用体型は、メンバの中で最大のデータサイズの領域をとります。【図D.6】に配置例を示します。



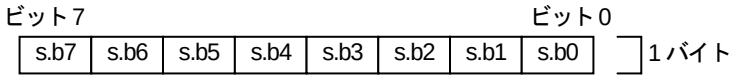
図D.6　共用体の配置例

D.2.8 ビットフィールド型

ビットフィールド型は、最下位のビットから配置されます。【図D.7】に配置例を示します。

例:

```
struct BTAG {
        char    b0 : 1;
        char    b1 : 1;
        char    b2 : 1;
        char    b3 : 1;
        char    b4 : 1;
        char    b5 : 1;
        char    b6 : 1;
        char    b7 : 1;
} s;
```

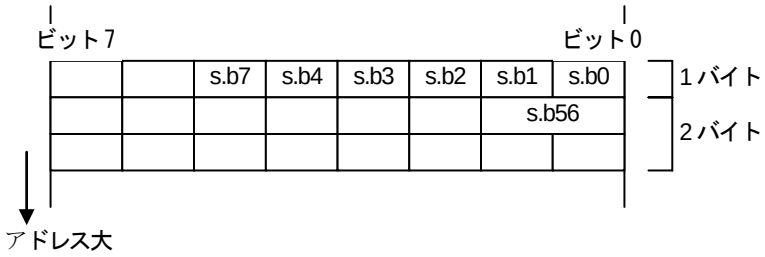


図D.7 ビットフィールドの配置例 (1)

ビットフィールドのメンバ中で、データ型が異なるものは次のアドレスに配置されます。この場合、同じデータ型のメンバは同じデータ型が配置されるアドレス上に最下位アドレスから連続して配置されます。

例:

```
struct BTAG {
        char    b0 : 1;
        char    b1 : 1;
        char    b2 : 1;
        char    b3 : 1;
        char    b4 : 1;
        int     b56 :2;
        char    b7 : 1;
} s;
```



図D.8 ビットフィールドの配置例 (2)

注意：

(1) ビットフィールドのメンバの型は、符号指定が無い場合 unsigned 型とみなします。

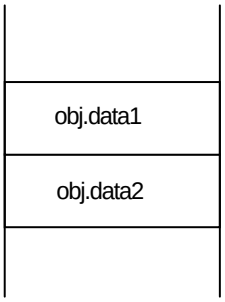
(2) long long 型のビットフィールドは宣言できません。

D.2.9 クラス型 (C++言語)

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振ります。

例:

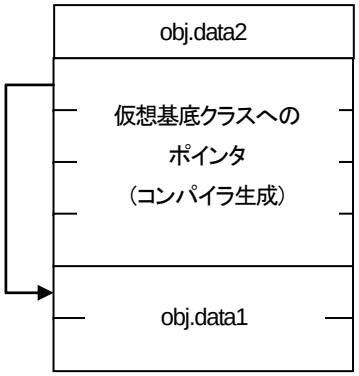
```
class A {
    char data1;
    char data2;
public:
    A();
    int get() { return data1; }
} obj;
```



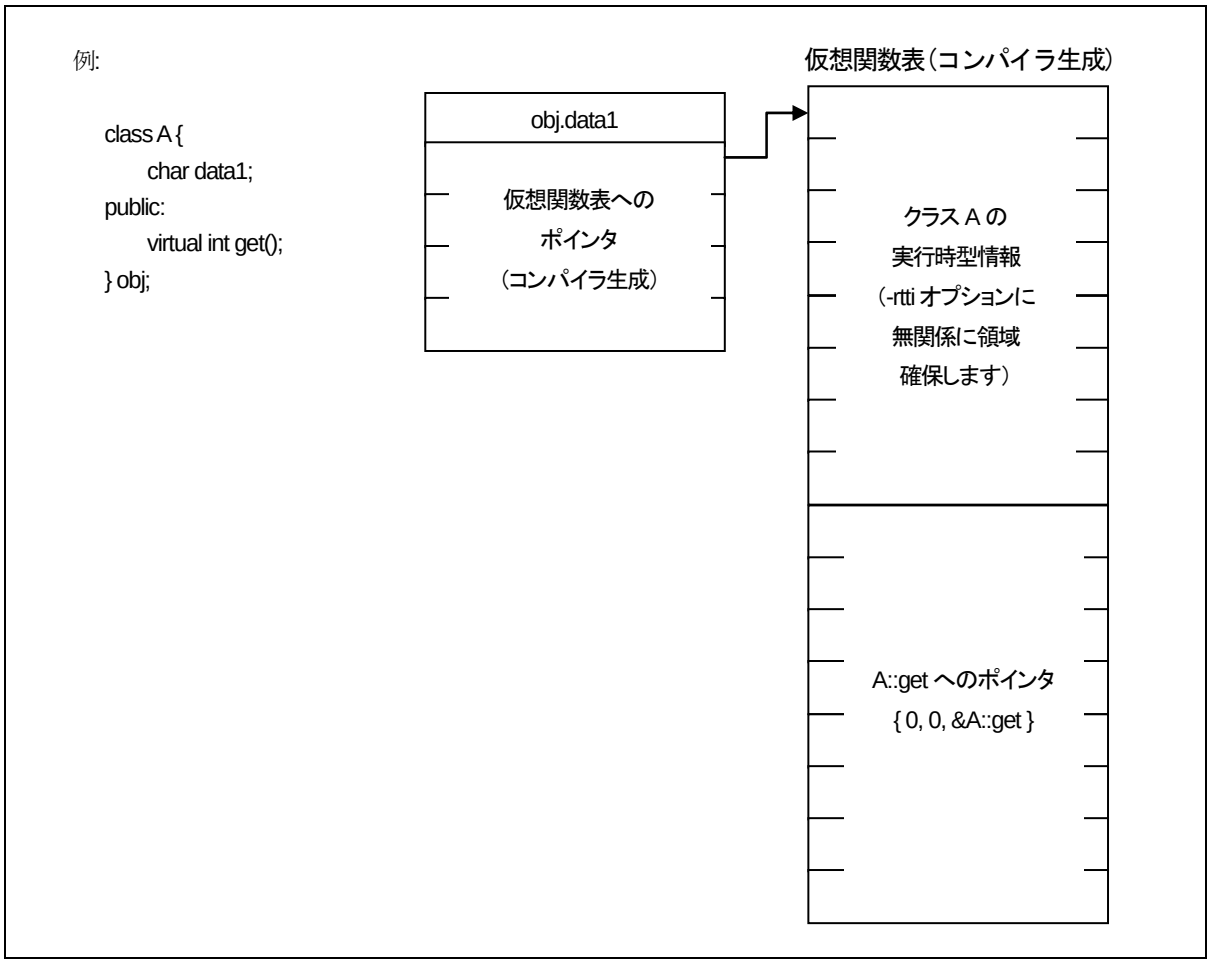
クラスに仮想基底クラスがある場合、仮想基底クラスへのポインタを割り付けます。
仮想基底クラスへのポインタのサイズは、コンパイルオプション-R8C 指定時は 2 バイト、コンパイルオプション-R8C 非指定時は 4 バイトです。

例:

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



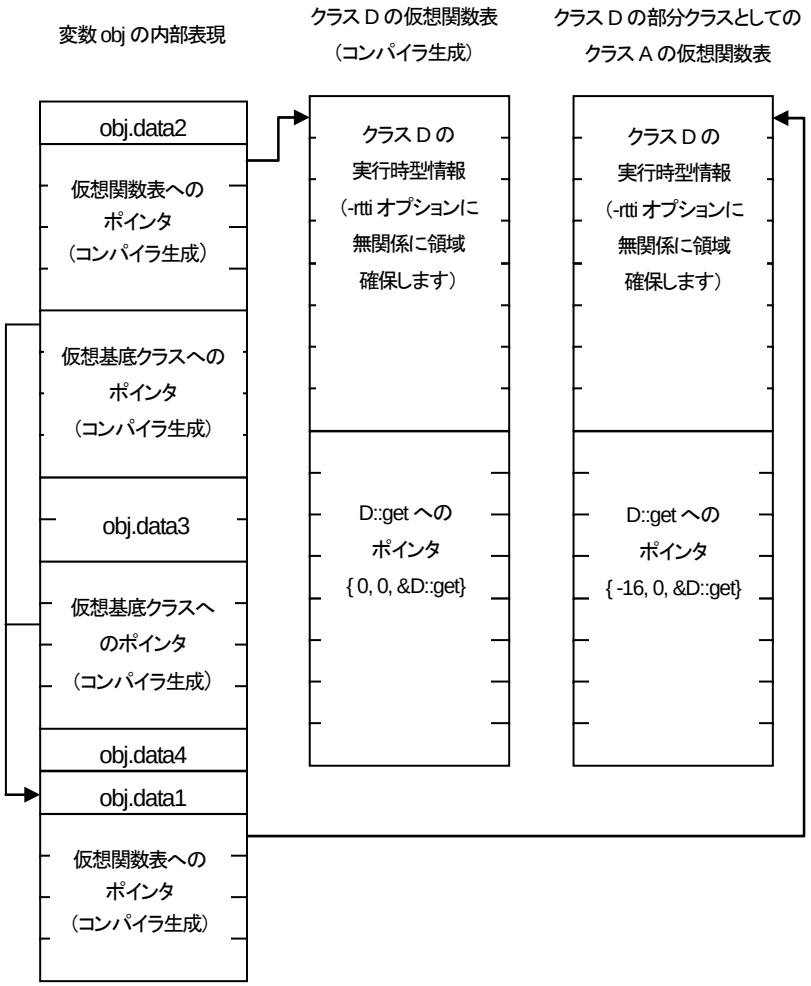
クラスに仮想関数がある場合、コンパイラは仮想関数表を生成し、仮想関数表へのポインタを割り付けます。
仮想関数表へのポインタのサイズは、コンパイルオプション-R8C 指定時は 2 バイト、コンパイルオプション-R8C 非指定時は 4 バイトです。


例:
class A {
char data1;
public:
virtual int get();
} obj;
obj.data1
仮想関数表への
ポインタ
（コンパイラ生成）
仮想関数表（コンパイラ生成）
クラス A の
実行時型情報
（-rtti オプションに
無関係に領域
確保します）
A::get へのポインタ
{ 0, 0, &A::get }

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:

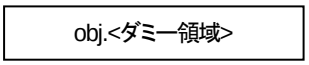
```
class A {
        char data1;
public:
        virtual short get();
};
class B : virtual A {
        char data2;
public:
        short get();
};
class C : virtual A {
        short data3;
};
class D : virtual A, B, C {
        char data4;
public:
        short get();
} obj;
```



空クラスの場合、１バイトのダミー領域を割りつけます。

例:

```
class A {
public:
        int fun();
} 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、ダミー領域は割付けません。

例:

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

obj.data1

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

obj.<ダミー領域>

D.2.10　参照型およびメンバへのポインタ型

【表D.5】に参照型およびメンバへのポインタ型のデータが使用するバイト数を示します。

表D.5　参照型およびメンバへのポインタ型のデータサイズ

型	符号の有無	ビットサイズ	表現できる数値
near 参照	なし	16	
far 参照	なし	32	
データメンバへのポインタ	なし	16	0～0xFFFF
関数メンバへのポインタ	なし	64	-

D.3 符号拡張規則

標準の言語仕様では char 型、signed char 型または unsigned char 型のデータは演算時等において int 型に符号拡張して処理を行う規則を記しています。

本コンパイラでは、デフォルトでコード効率と実行速度を重視したコードを生成するために、char 型、signed char 型または unsigned char 型を int 型に拡張しません。この仕様は、コンパイルオプション"-fansi"又は"-fextend_to_int(-fETI)"を使用することにより無効となり、標準の C 言語と同様の拡張を行います。

コンパイルオプション"-fansi"又は"-fextend_to_int(-fETI)"を使用せず、【図 D.9】のように演算結果を char 型に代入するような演算を記述する場合は、char 型、signed char 型または unsigned char 型で表現できる最小値及び最大値 が演算途中でオーバフローしないように注意してください。

C++言語モードとしてコンパイルする場合、常に char 型、signed char 型または unsigned char 型を int 型に型変換しますので、注意してください。

このプログラムでは、C言語モードでは変数 i に 0x24 が代入され、C++言語モードでは変数 i に 0x124 が代入されます。

例:

```
{
    int i;
    char a = 0x8f;
    char b = 0x95;
    i = a + b;
 }
```

D.4 関数呼び出し規則

D.4.1 戻り値に関する規則

関数から戻り値を返す場合、戻り値の型が整数型、ポインタ型、浮動小数点型の場合は、レジスタ渡しになります。【表D.6】に戻り値に関する呼び出し規則を示します。

表D.6 戻り値に関する呼び出し規則

戻り値の型	規則
char 型 signed char 型 unsigned char 型 _Bool 型 bool 型 列挙型[1]	R0L レジスタに格納して返します。
signed short 型 unsigned short 型 signed int 型 unsigned int 型 near ポインタ型 near 参照型 wchar_t 型 列挙型[2] データメンバへのポインタ型	R0 レジスタに格納して返します。
float 型 double型[3] signed long 型 unsigned long 型 far ポインタ型 far 参照型	下位 16 ビットは R0 レジスタに、上位 16 ビットは R2 レジスタに格納して返します。
double型[4] long double 型	R3、R2、R1、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
signed long long 型 unsigned long long 型	R3、R1、R2、R0 レジスタの順に、上位から 16 ビット区切りで格納して返します。
構造体型 共用体型 クラス型 関数メンバへのポインタ型	呼び出しを行う直前に、戻り値を格納するための領域を指す far ポインタをスタックに退避します。呼び出された関数はリターンする前にスタックに退避された far ポインタで指す領域に戻り値を書き込みます。

[1] -fchar_enumerator(-fCE),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれか指定ありの場合に限ります。
[2] -fchar_enumerator(-fCE),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれも指定なしの場合に限ります。
[3] -fdouble_32(-fD32),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれか指定ありの場合に限ります。
[4] -fdouble_32(-fD32),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれも指定なしの場合に限ります。

D.4.2 引数渡しに関する規則

本コンパイラは、関数への引数渡しの方法として、レジスタ渡しとスタック渡しの 2 通りがあります。

(1) 引数のレジスタ渡し

以下に示す条件を満たす場合、【表D.7】中の「使用するレジスタ」を用いて引数を渡します。

- 関数の関数原型宣言[1]を行ない、関数呼びだし時に引数の型が確定している。
- 関数原型宣言に可変引数"..."を使用していない。
- 関数の引数の型として、【表D.7】の引数と引数の型が一致している。

表D.7 レジスタ引数渡しの規則 (NC30)

引数	引数の型	使用するレジスタ
第 1 引数	char 型 signed char 型 unsigned char 型 _Bool 型 bool 型 列挙型[2]	R1L レジスタ
	signed short 型 unsigned short 型 signed int 型 unsigned int 型 near ポインタ型 near 参照型 wchar_t 型 列挙型[3] データメンバへのポインタ型	R1 レジスタ
第 2 引数	signed short 型 unsigned short 型 signed int 型 unsigned int 型 near ポインタ型 near 参照型 wchar_t 型 列挙型[4] データメンバへのポインタ型	R2 レジスタ

[1]本コンパイラでは、関数原型宣言を行なった時のみ、レジスタ渡しを適応します。K&R形式の記述を行なった場合は、すべての引数をスタック渡しで行ないます。また、C言語の言語仕様上、関数に対して関数原型宣言を行なう記述形式とK&R形式の記述を混在すると、引数が関数に正しく渡されない場合があることに注意してください。
上記理由により、プロトタイプ宣言を行なう記述形式に統一して C 言語ソースファイルを記述することを推奨しています。
[2] -fchar_enumerator(-fCE),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれか指定ありの場合に限ります。
[3] -fchar_enumerator(-fCE),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれも指定なしの場合に限ります。
[4] -fchar_enumerator(-fCE),-OR_MAX(-ORM),-OS_MAX(-OSM)のいずれも指定なしの場合に限ります。

(2) 引数のスタック渡し

レジスタ渡しの条件を満たさない引数は、すべてスタック渡しになります。

D.4.3 関数のアセンブリ言語シンボルへの変換規則

C言語ソースファイルでの関数定義時の関数名は、アセンブラソースファイルでの関数の先頭ラベルとして使用します。

アセンブラソースファイルでの関数の先頭ラベルは、C言語ソースファイルでの関数名の先頭に_(アンダースコア)あるいは$(ダラー)を付加した文字列です。付加文字列と文字列が付加される条件を【表D.8】に示します。

表D.8 関数に文字列の付加される条件

付加文字列	条件
$(ダラー)	1つでも引数がレジスタ渡しとなる関数
_(アンダースコア)	上記条件以外の関数

【図D.9】に示すプログラムは、関数の引数がレジスタ引数を持つものと、関数の引数をスタック渡しのみで扱う例です。

```
int         func_proto( int , int , int);                          ← (1)

int         func_proto(int i, int j, int k)                        (2)
{
            return i + j + k;
}

int         func_no_proto( i, j, k)                                (3)
int         i;
int         j;
int         k;
{
            return i + j + k;
}

void        main(void)                                             (4)
{
            int         sum;

            sum = func_proto(1,2,3);                    ← (5)
            sum = func_no_proto(1,2,3);                 ← (6)
}
```

(1) 関数 func_proto の関数原型宣言です。
(2) 関数 func_proto の実体です(関数原型宣言を行なっています)。
(3) 関数 func_no_proto の実体です(K&R 形式の記述です)。
(4) 関数 main の実体です。
(5) 関数 func_proto を呼んでいます。
(6) 関数 func_no_proto を呼んでいます。

図D.9　関数呼び出しのサンプルプログラム(sample.c)

上記サンプルプログラムのコンパイル結果について、関数func_protoの定義((2) の部分)を【図D.10】に、関数func_no_protoの定義((3) の部分)を【図D.11】に、関数func_protoと関数func_no_protoの呼び出し((4) の部分)を【図D.12】に示します。

C++言語プログラムの関数名をアセンブリプログラムから参照する場合は、C 結合を使用します。

C++言語プログラムの関数をC結合で宣言（extern "C"を用いて宣言）すると、その関数はC言語プログラムと同じ規則で参照することができます。ただし、C結合で宣言した関数は多重定義できません。

```
;## #        FUNCTION func_proto
;## #        FRAME    AUTO      (         j)  size  2,    offset -4
;## #        FRAME    AUTO      (         i)  size  2,    offset -2
;## #  FRAME    ARG (         k)  size    2,      offset 5                    ← (7)
;## #  REGISTER ARG     (        i)    size    2,    REGISTER R1             ← (9)
;## #  REGISTER ARG     (        j)    size    2,    REGISTER R2             ← (8)
;## #        ARG Size(2)            Auto Size(4)            Context Size(5)

             .SECTION  program,CODE,ALIGN
             ._file    'sample.c'
             ._line    4
;## # C_SRC :         {
             .glb      $func_proto
$func_proto:                                                                 ← (10)
             enter     #04H
             mov.w     R1,-2[FB]   ;  i  i
             mov.w     R2,-4[FB]   ;  j  j
             ._line    5
;## # C_SRC :                      return i + j + k;
             mov.w     -2[FB],R0   ;  i
             add.w     -4[FB],R0   ;  j
             add.w     5[FB],R0    ;  k
             exitd
E1:


(7)  第３引数 k をスタック渡しにしています。
(8)  第２引数 j をレジスタ渡しにしています。
(9)  第１引数 i をレジスタ渡しにしています。
(10) 関数 func_proto の先頭アドレスです。
```

図D.10　サンプルプログラム(sample.c)のコンパイル結果 (1)

【図D.10】では、関数func_protoは、関数原型宣言を行なっているため第１、第２引数をレジスタ渡しになります。第３引数はレジスタ渡しの対象とはならないため、スタック渡しになります。

また、関数の引数がレジスタ渡しになるため、関数の先頭アドレスのシンボル名は、C 言語ソースファイルに記述した" func_proto" の前に$（ダラー）を付加した"$func_proto"になります。

```
;## #        FUNCTION func_no_proto
;## #  FRAME    ARG (        i)  size    2,     offset 5                    (11)
;## #  FRAME    ARG (        j)  size    2,     offset 7
;## #  FRAME    ARG (        k)  size    2,     offset 9
;## #        ARG Size(6)          Auto Size(0)          Context Size(5)

                ._line       12
;## # C_SRC :   {
                .glb         _func_no_proto                          ← (12)
_func_no_proto:
                enter        #00H
                ._line       13
;## # C_SRC :                 return i + j + k;
                mov.w        5[FB],R0    ;  i
                add.w        7[FB],R0    ;  j
                add.w        9[FB],R0    ;  k
                exitd
E2:


(11) すべての引数をスタック渡しにしている。
(12) 関数 func_no_proto の先頭アドレスです。
```

図D.11 サンプルプログラム(sample.c)のコンパイル結果 (2)

【図D.11】では、関数func_no_protoはK&R形式の記述を行なっているため、すべての引数がスタック渡しになります。

また、関数の引数にレジスタ渡しを含まないため、関数の先頭アドレスのシンボル名は、C 言語ソースファイルに記述した"func_no_proto"の前に_(アンダースコア)を付加した"_func_no_proto"になります。

```
;### #        FUNCTION main
;### #        FRAME    AUTO      (       sum) size   2,     offset -2
;### #        ARG Size(0)          Auto Size(2)            Context Size(5)

                ._line      17
;### # C_SRC :          {
                .glb        _main
_main:
                enter       #02H
                ._line      20
;### # C_SRC :                    sum = func_proto(1,2,3);
                push.w      #0003H                                          (13)
                mov.w       #0002H,R2
                mov.w       #0001H,R1
                jsr         $func_proto
                add.b       #02H,SP
                mov.w       R0,-2[FB]  ; sum
                ._line      21
;### # C_SRC :                    sum = func_no_proto(1,2,3);
                push.w      #0003H                                          (14)
                push.w      #0002H
                push.w      #0001H
                jsr         _func_no_proto
                add.b       #06H,SP
                mov.w       R0,-2[FB]  ; sum
                ._line      22
;### # C_SRC :          }
                exitd
E3:
                .align
                .END
```

図D.12　サンプルプログラム(sample.c)のコンパイル結果 (3)

【図D.12】において、(13)の部分はfunc_protoの呼び出しを、(14)の部分はfunc_no_protoの呼び出しを行っています。

D.4.4 関数間のインターフェース

【図D.13】に示すプログラムにおいて、スタックフレームの構築及び解放の処理を【図D.16】～【図D.18】に示します。なお、【図D.14】と【図D.15】は、【図D.13】のプログラムをコンパイルした結果、出力されたアセンブリ言語プログラムです。

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

void        main(void)
{
            int         i = 0x1234;             ← func への引数
            int         j = 0x5678;             ← func への引数
            int         k = 0x9abc;             ← func への引数

            k = func( i, j ,k);
}

int         func( int x,int y,int z )
{
            int         sum;

            sum = 0;
            sum = x + y + z;
            return sum;                         ← main への戻り値
}
```

図D.13 C 言語サンプルプログラム

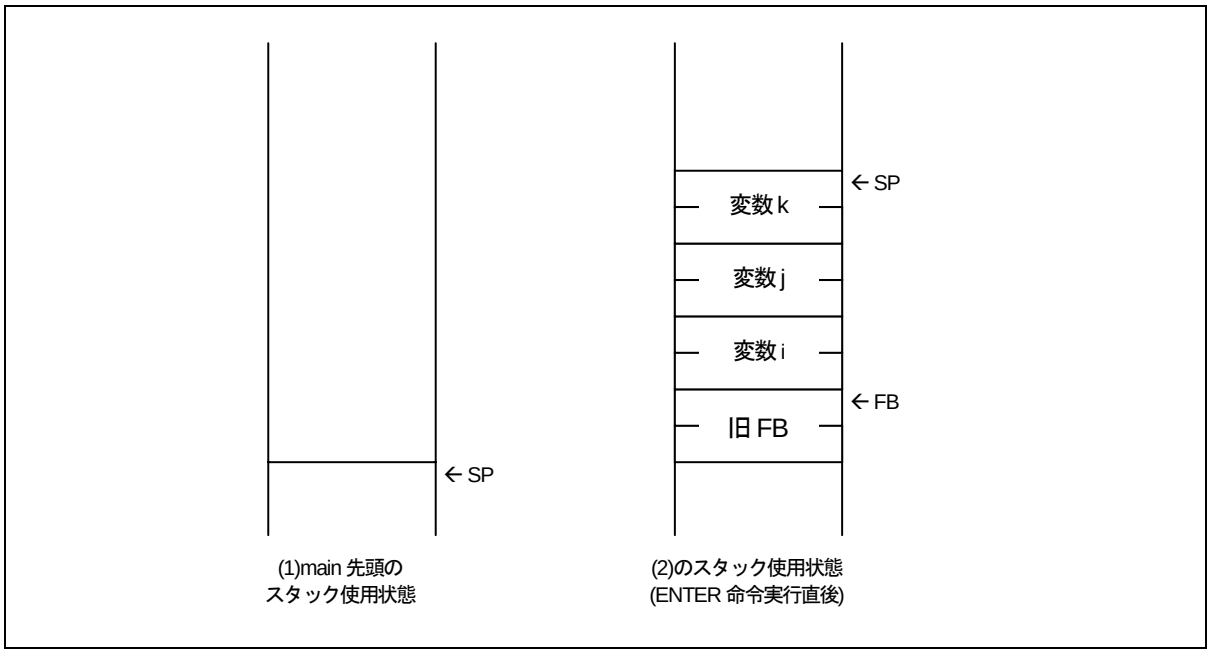
```
;## #       FUNCTION main
;## #       FRAME   AUTO    (       k)  size   2,    offset -6
;## #       FRAME   AUTO    (       j)  size   2,    offset -4
;## #       FRAME   AUTO    (       i)  size   2,    offset -2
;## #       ARG Size(0)             Auto Size(6)            Context Size(5)


            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            ._line    4
;## # C_SRC :        {
            .glb      _main
_main:                                                          ← (1)
            enter     #06H                                      ← (2)
            ._line    5
;## # C_SRC :                int             i = 0x1234;
            mov.w     #1234H,-2[FB]         ;   i
            ._line    6
;## # C_SRC :                int             j = 0x5678;
            mov.w     #5678H,-4[FB]         ;   j
            ._line    7
;## # C_SRC :                int             k = 0x9abc;
            mov.w     #9abcH,-6[FB]         ;   k
            ._line    9
;## # C_SRC :                k = func( i, j ,k);
            push.w    -6[FB]      ;   k                         ← (3)
            mov.w     -4[FB],R2   ;   j                         ← (4)
            mov.w     -2[FB],R1   ;   i                         ← (5)
            jsr       $func                                     ← (6)
            add.b     #02H,SP                                   ← (10)
            mov.w     R0,-6 [FB]  ;   k                         ← (11)
            ._line    10
;## # C_SRC :        }
            exitd
E1:
```

図D.14　アセンブリ言語サンプルプログラム (1)

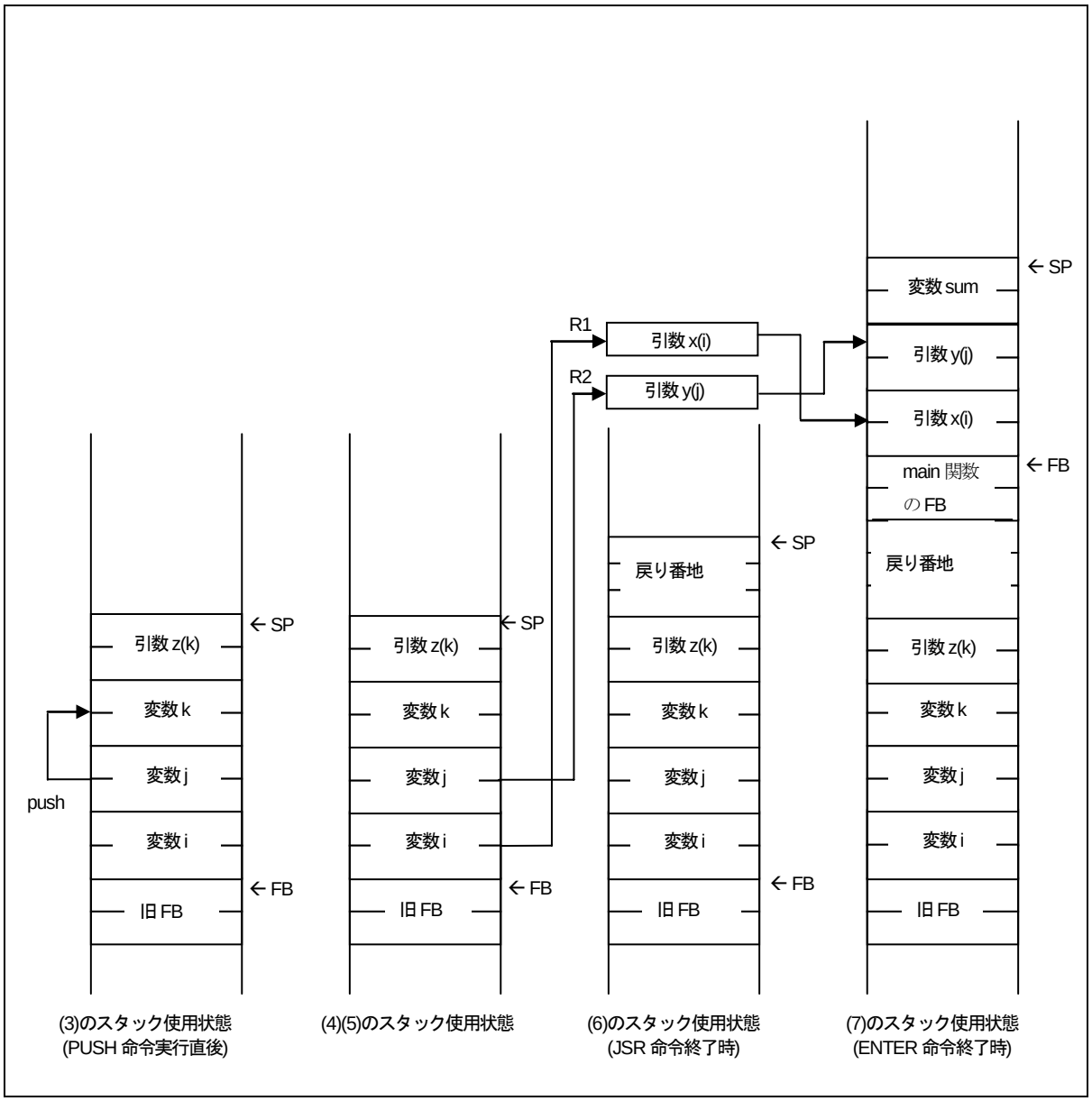
```
;## #        FUNCTION func
;## #        FRAME    AUTO      (       sum) size   2,     offset -6
;## #        FRAME    AUTO      (         y) size   2,     offset -4
;## #        FRAME    AUTO      (         x) size   2,     offset -2
;## #   FRAME    ARG (         z)  size    2,      offset 5
;## #   REGISTER ARG      (         x)    size    2,    REGISTER R1
;## #   REGISTER ARG      (         y)    size    2,    REGISTER R2
;## #        ARG Size(2)            Auto Size(6)            Context Size(5)

             ._line      13
;## # C_SRC :          {
             .glb        $func
$func:
             enter       #06H                                          ← (7)
             mov.w       R1,-2[FB]  ;  x  x
             mov.w       R2,-4[FB]  ;  y  y
             ._line      16
;## # C_SRC :          sum = 0;
            mov.w        #0000H,-6[FB] ;  sum
             ._line      17
;## # C_SRC :                  sum = x + y + z ;
             mov.w       -2[FB],R0  ;  x
             add.w       -4[FB],R0  ;  y
             add.w       5[FB],R0   ;  z
             mov.w       R0,-6[FB]  ;  sum
             ._line      18
;## # C_SRC :                  return sum;
             mov.w       -6[FB],R0  ;  sum                             ← (8)
             exitd                                                     ← (9)
E2:
             .align
             .END
```

図D.15　アセンブリ言語サンプルプログラム (2)

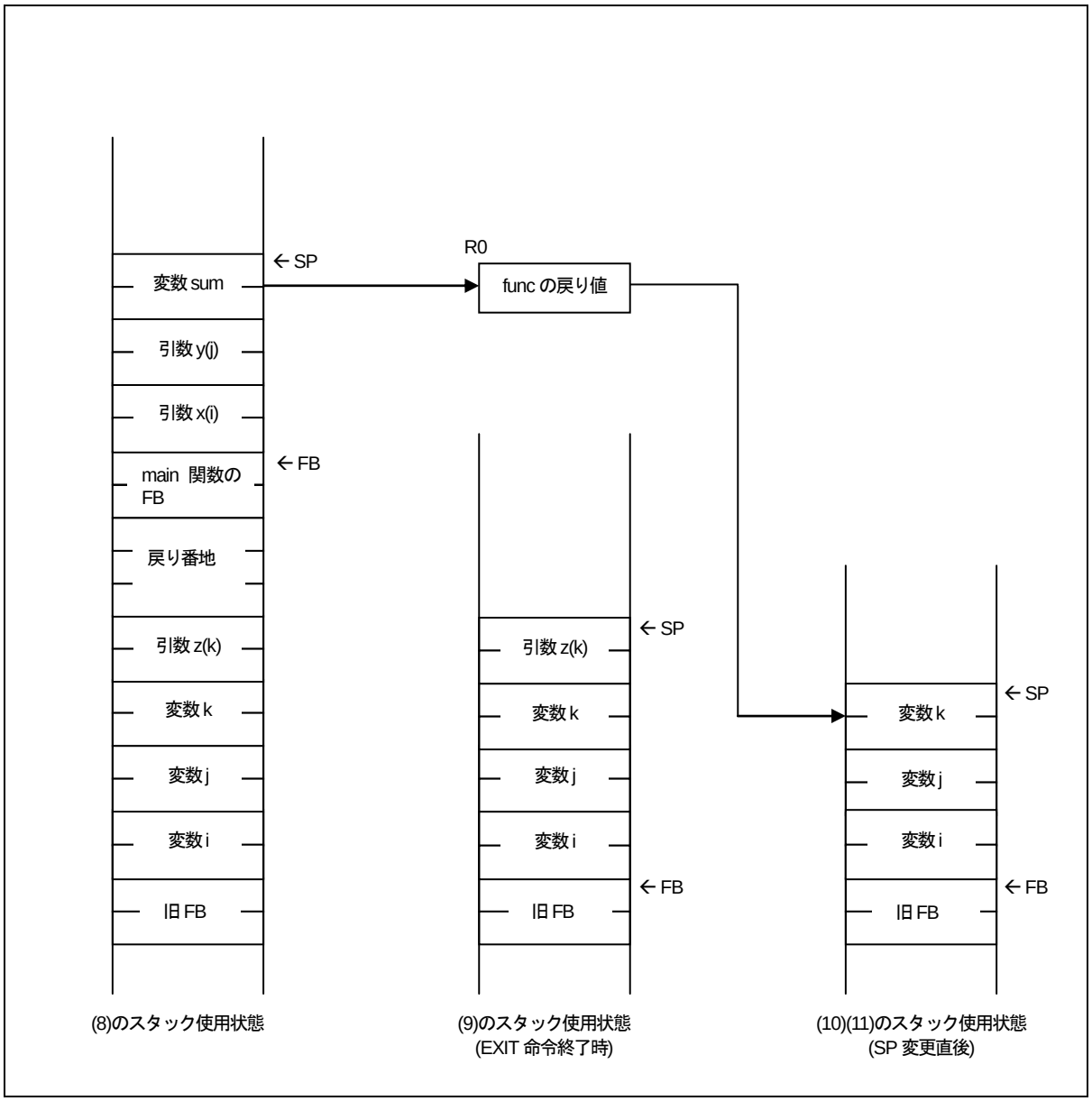
【図D.14】中における (1) → (2) の処理(関数mainの入り口処理)を【図D.16】に、(3) →(4) → (5) → (6) → (7) の処理(関数funcの呼び出し及び関数funcで使用するスタックフレームの構築処理)を【図D.17】に、【図D.15】中における(8) → (9) → (10) → (11)の処理(関数funcから関数mainへの戻り処理)を【図D.18】に、各々のスタック及びレジスタの遷移を示します。



図D.16　関数 main の入り口処理



図D.17 関数 func の呼び出し及び、入り口処理



図D.18 関数 func の出口処理

D.5 auto 変数の領域確保

記憶クラスautoの変数は、マイコンのスタック上に配置されます。【図D.19】に示すようなC言語ソースプログラムでは、記憶クラスautoの変数が有効となる領域が互いに重ならない場合、1 つの領域のみ確保を行い複数の変数でその領域を共有します。

```
void        func(void)
{
            int         i, j, k;

        for( i=0 ; i<=0 ; i++ ){
                    処理                                    i の有効範囲
        }
                    :
                    (省略)
                    :
        for( j=0xFF ; j<=0 ; j-- ){
                    処理                                    j の有効範囲
                    }
                    :
                    (省略)
                    :
        for( k=0 ; k<=0 ; k++ ){
                    処理                                    k の有効範囲
        }
}
```

図D.19 C 言語ソースプログラム例

この例では、3 つのauto変数i、j、kは有効となる範囲が重ならないため、同じ 2 バイトの領域(FBからのオフセット位置)を共有します。【図D.19】をコンパイルして生成されたアセンブラソースファイルを【図D.20】に示します。

```
;### FUNCTION        func
;###          FRAME AUTO          (          k)          size 2,          offset -2          ← (1)
;###          FRAME AUTO          (          j)          size 2,          offset -2          ← (2)
;###          FRAME AUTO          (          i)          size 2,          offset -2          ← (3)
             .section program
             ._file         'test1.c'
             ._line         3
             .glb           _func
_func:
             enter          #02H
                  :
             (以下省略)
```

※3 つの auto 変数は(1)～(3)に示すように、FB オフセット-2 の領域を共有しています。

図D.20 アセンブラソースプログラム例

D.6 レジスタの退避

関数を呼び出す場合のレジスタの退避規則を以下に示します。

(1) 関数の呼び出し側で退避するレジスタ
- 呼び出し側で使用するレジスタ。

(2) 呼び出された関数の入口処理で退避するレジスタ
- 退避しません

D.7 プリプロセッサ仕様

D.7.1 インクルードファイルの読み込み方法

書式 1: #include△<ファイル名>
書式 2: #include△"ファイル名"

書式 1 の場合、起動オプション-I で指定されたディレクトリのファイルを取り込みます。
ファイルが見つからない場合は、以下のディレクトリを検索します。
- 環境変数 INC30 により設定された標準ディレクトリ

書式 2 の場合、コンパイル対象ファイルのあるディレクトリからファイルを取り込みます。ファイルが見つからない場合は、以下のディレクトリを順番に検索します。
- 起動オプション-I で指定されたディレクトリ
- 環境変数 INC30 により設定された標準ディレクトリ

D.7.2 プリデファインドマクロ

定義済みのプリデファインマクロは以下のとおりです。

__DATE__	コンパイルの日付が定義されます。
__FILE__	ソースファイルの名前が定義されます。
__LINE__	ソースファイルの行番号が定義されます。
__TIME__	コンパイル時刻が定義されます。
__STDC__	オプション-fansi 指定時に 1 が定義されます。
__RENESAS__	常に 1 が定義されます。
__RENESAS_VERSION__	コンパイラのバージョン番号が定義されます。
NC30	常に空白が定義されます。
M16C	常に空白が定義されます。
__R8C__	オプション-R8C, -R8CE 指定時に空白が定義されます。
__cplusplus	C++言語をコンパイル時に 1 が定義されます。

D.7.3 #assert

定数式の結果が 0 (ゼロ) の場合に、以下のウォーニングを出力します。コンパイルはそのまま続行されます。
sample.c(1) : C6696 (W) Assertion warning

D.8 C++言語コンパイルの注意事項

D.8.1 const 修飾した変数に関する注意事項

C++コンパイルの場合、const 指定された変数を rom セクションに配置するとは限りません。動的初期化を伴うものは、bss セクションに配置します。

```
const int a = func();          // 変数 a を bss セクションに配置します。

const struct S {
        int a;
        S() {}
} b;                           // 変数 b を bss セクションに配置します。
```

D.8.2 new/delete 演算子関数の注意事項

new/delete 演算子関数は、new/delete 演算子で呼び出します。本コンパイラでは、new 演算子関数の返却値と delete 演算子の第 1 引数の型を"void _far *"とします。

```
struct S
{
    static void* operator new(size_t);
    static void  operator delete(void*);
};

void _far * alloc_int_S()
{
    void _far *(*pf)(size_t) = S::operator new;    // new 演算子関数の返却値の型は暗黙に
                                                   // far ポインタなので、
                                                   // RAM データポインタが near 属性の場合、
                                                   // far 修飾が必要。
    return (*pf)(sizeof(int));
}

void dealloc_int_S(void _far * ptr)
{
    void (*pf)(void _far *) = S::operator delete;  // delete 演算子関数の第 1 引数の型は暗黙に
                                                   // far ポインタなので、
                                                   // RAM データポインタが near 属性の場合、
                                                   // far 修飾が必要。
    (*pf)(ptr);
}
```

D.8.3 char 型に関する注意事項

C++言語でコンパイルする際は、char 型と unsigned char 型は別の型として扱います。したがって、C プログラムでコンパイルできていた次のソースがエラーになるので注意してください。

```
extern unsigned char port; //  宣言
                    char port; //  定義
```

ソースプログラムの可搬性を向上するため、C プログラムでも、文字を表現する型の場合は char 型を、1 バイト長整数を表現する型の場合は、signed char 型または unsigned char 型を使用することを推奨します。

D.8.4 複数の宣言で near/far を確定する記述に関する注意事項

C++プログラムとしてコンパイルする場合、C プログラムとしてコンパイルする場合許されていた、複数の宣言で near/far を確定する記述がエラーになる場合があります。

```
extern int far fi;
int fi;                 // C では、fi の型を int far と解釈する。
                        // C++では、RAM データ配置属性が far の場合、fi の型を int far と解釈する。
                        //          RAM データ配置属性が near の場合エラー。

extern int near ni;
int ni;                  // C では、fi の型は int near と解釈する。
                        // C++では、RAM データ配置属性が far の場合、エラー。
                        //          RAM データ配置属性が near の場合、ni の型を int near と解釈する。

extern int far * fpi;
int * fpi;              // C では、fpi の型を int far*と解釈する。
                        // C++では、RAM データポインタ属性が far の場合、
                        //          fpi の型を int far*と解釈する。
                        //          RAM データポインタ属性が near の場合、エラー。

extern int near * npi;
int * npi;              // C では、npi の型を int near*と解釈する。
                        // C++では、RAM データポインタ属性が far の場合、エラー。
                        //          RAM データポインタ属性が near の場合、
                        //          npi の型を int near*と解釈する。
```

D.8.5 メンバ配置属性の near/far に関する注意事項

メンバ配置属性の変数に対する near/far の宣言は、C プログラムでは無視されます。C++プログラムではエラーになります。

```
struct Tag {
    int near mem1;   // C 言語の場合は near を無視し、C++言語の場合はエラーになります。
    int far  mem2;   // C 言語の場合は far を無視し、C++言語の場合はエラーになります。
};
```

D.8.6 インライン関数に関する注意事項

C++コンパイルの場合、キーワード inline とクラス定義内に定義するメンバ関数は static 関数と同等に扱われ、インライン展開をしません。これらの関数をインライン展開をする場合、コンパイラオプション -Ostatic_to_inline(-OSTI)と-Oforward_function_to_inline(-OFFTI)を使用して下さい。なお、大域名前空間に関数を定義し、キーワード_inline を使用する場合、C コンパイルの場合と同様にインライン展開します。

D.8.7 参照型を持つ変数の配置属性の near/far に関する注意事項

次のような参照型を持つ変数の配置属性に対して、near/far 修飾子で指定しないでください。

```
int   near   ni ;
int   near   & rni = ni ;              // OK
int   near   & near rni = ni ;         // NG
```

付録E C/C++ライブラリ

E.1 標準ヘッダファイルの機能と、その仕様の詳細

標準ライブラリを使用する場合、その関数の宣言を行っているヘッダファイルをインクルードする必要があります。標準ヘッダファイルの機能との仕様の詳細を説明します。

E.1.1 標準ヘッダファイルの概要

本コンパイラは、【表E.1】に示す標準ヘッダファイルを用意しています。

表E.1 標準ヘッダファイル一覧表

ヘッダファイル名	内容
assert.h	プログラムの診断情報の出力
ctype.h	文字判定関数のマクロ宣言
errno.h	エラー番号の定義
float.h	浮動小数点数の内部表現に関する各種制限値の定義
limits.h	コンパイラの内部処理に関する各種制限値の定義
locale.h	関数/マクロの地域化
math.h	数値計算
mathf.h	数値計算(float 用)
setjmp.h	分岐関数、分岐関数で使用する構造体の定義
signal.h	非同期割り込みを処理するための定義/宣言
stdarg.h	可変個の実引数を持つ関数の宣言と定義
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の定義
stdio.h	(1) FILE 構造体の定義 (2) ストリーム名の定義 (3) 入出力関数の関数原型宣言
stdlib.h	メモリ管理関数、終了関数の関数原型宣言
string.h	文字列操作関数、メモリ操作関数のプ関数原型宣言
time.h	現在の暦時間を得る

E.1.2 標準ヘッダファイルリファレンス

本コンパイラが用意している標準ヘッダファイルの詳細仕様を説明します。ヘッダファイルは、アルファベット順に掲載しています。

ヘッダファイルの内部で宣言している本コンパイラの標準関数と、データ型の数値表現における制限値を定義しているマクロを、対応するヘッダファイルと共に説明します。

assert.h

機　　能:　関数マクロ assert を定義しています。

ctype.h

機　　能:　文字操作関数を宣言、及びマクロを定義しています。文字操作関数を以下に示します。

関数名	機能
isalnum	英数字の判定
isalpha	英字の判定
iscntrl	コントロール文字の判定
isdigit	数字の判定
isgraph	英数字、ブランク以外の文字判定
islower	英小文字の判定
isprint	ブランク文字を含む印字可能文字の判定
ispunct	区切り文字の判定
isspace	ブランク、タブ、改行の判定
isupper	英大文字の判定
isxdigit	16 進数字の判定
tolower	大文字から小文字への変換
toupper	小文字から大文字への変換

errno.h

機　　能:　エラー番号を定義しています。

float.h

機　　能: 浮動小数点数の内部表現に関する各種制限値を定義しています。以下に、浮動小数点数の制限値を定義したマクロを示します。
本コンパイラでは、long double は double として扱います。long double の各制限値は double と同じに定義しています。

マクロ名	内容	定義値
DBL_DIG	double 型の 10 進精度の最大桁数	15
DBL_EPSILON	1.0+DBL_EPSILON が 1.0 と異なると判断できる正の最小値	2.2204460492503131e-16
DBL_MANT_DIG	double 型の浮動小数点数値を基数に合わせて表現したときの仮数部の最大桁数	53
DBL_MAX	double 型の変数が値として持つことができる最大値	1.7976931348623157e+308
DBL_MAX_10_EXP	double 型の浮動小数点数値として表現できる 10 のべき剰の最大値	308
DBL_MAX_EXP	double 型の浮動小数点数値として表現できる基数のべき剰の最大値	1024
DBL_MIN	double 型の変数が値として持つことができる最小値	2.2250738585072014e-308
DBL_MIN_10_EXP	double 型の浮動小数点数値として表現できる 10 のべき剰の最小値	-307
DBL_MIN_EXP	double 型の浮動小数点数値として表現できる基数のべき剰の最小値	-1021
FLT_DIG	float 型の 10 進精度の最大桁数	6
FLT_EPSILON	1.0+FLT_EPSILON が 1.0 と異なると判断できる正の最小値	1.19209290e-07F
FLT_MANT_DIG	float 型の浮動小数点数値を基数に合わせて表現したときの仮数部の最大桁数	24
FLT_MAX	float 型の変数が値として持つことができる最大値	3.40282347e+38F
FLT_MAX_10_EXP	float 型の浮動小数点数値として表現できる 10 のべき剰の最大値	38
FLT_MAX_EXP	float 型の浮動小数点数値として表現できる基数のべき剰の最大値	128
FLT_MIN	float 型の変数が値として持つことができる最小値	1.17549435e-38F
FLT_MIN_10_EXP	float 型の浮動小数点数値として表現できる 10 のべき剰の最小値	-37
FLT_MIN_EXP	float 型の浮動小数点数値として表現できる基数のべき剰の最小値	-125
FLT_RADIX	浮動小数の指数の基数	2
FLT_ROUNDS	浮動小数点数のまるめ方法	1(四捨五入)

備考

- コンパイラオプション-fdouble_32(-fD32)、-OR_MAX(-ORM)または-OS_MAX(-OSM)を使用する場合、DBL_XXX マクロは FLT_XXX マクロの定義と同じ値を定義します。
- long double 型マクロ LDBL_XXX も定義しています。LDBL_XXX の DBL_XXX と同じ定義です。ただし、浮動小数点定数には接尾子 L をつけて定義します。

limits.h

機　　能: コンパイラの内部処理に関する各種制限値を定義しています。以下に、各種制限値を定義したマクロを示します。

マクロ名	内容	定義値
MB_LEN_MAX	マルチバイト文字型のバイト数の最大値	1
CHAR_BIT	char 型のビット数	8
CHAR_MAX	char 型の変数が値として持つことができる最大値	255
CHAR_MIN	char 型の変数が値として持つことができる最小値	0
SCHAR_MAX	signed char 型の変数が値として持つことができる最大値	127
SCHAR_MIN	signed char 型の変数が値として持つことができる最小値	-128
INT_MAX	int 型の変数が値として持つことができる最大値	32767
INT_MIN	int 型の変数が値として持つことができる最小値	-32768
SHRT_MAX	short int 型の変数が値として持つことができる最大値	32767
SHRT_MIN	short int 型の変数が値として持つことができる最小値	-32768
LONG_MAX	long 型の変数が値として持つことができる最大値	2147483647
LONG_MIN	long 型の変数が値として持つことができる最小値	-2147483648
LLONG_MAX	signed long long int 型の変数が値として持つことができる最大値	9223372036854775807
LLONG_MIN	signed long long int 型の変数が値として持つことができる最小値	-9223372036854775808
UCHAR_MAX	unsigned char 型の変数が値として持つことができる最大値	255
UINT_MAX	unsigned int 型の変数が値として持つことができる最大値	65535
USHRT_MAX	unsigned short int 型の変数が値として持つことができる最大値	65535
ULONG_MAX	unsigned long int 型の変数が値として持つことができる最大値	4294967295
ULLONG_MAX	unsigned long long int 型の変数が値として持つことができる最大値	18446744073709551615

locale.h

機　　能: プログラムの地域化を操作するマクロと関数を定義/宣言しています。関数原型を宣言している関数を以下に示します。

関数名	機能
localeconv	構造体 lconv を初期化
setlocale	プログラムのロケール情報の設定と検索

math.h (mathf.h)

機　　能: 数学関数の関数原型を宣言しています。関数原型を宣言している関数を以下に示します。

関数名	機能
acos	逆コサインを計算
asin	逆サインを計算
atan	逆タンジェントを計算
atan2	逆タンジェントを計算
ceil	整数繰り上げ値を計算
cos	コサインを計算
cosh	双曲線コサインを計算
exp	指数関数を計算
fabs	倍精度浮動小数の絶対値を計算
floor	整数繰り下げ値を計算
fmod	剰余計算
frexp	浮動小数を仮数部と指数部に分割
ldexp	浮動小数の巾を計算
log	自然対数を計算
log10	常用対数を計算
modf	実数を仮数部と指数部に分割
pow	巾乗計算
sin	サインを計算
sinh	双曲線サインを計算
sqrt	数値の平方根を計算
tan	タンジェントを計算
tanh	双曲線タンジェントを計算

setjmp.h

機　　能: 分岐関数のプ関数原型宣言と、その関数で使用する構造体を定義しています。関数原型を宣言している関数を以下に示します。

関数名	機能
longjmp	大域ジャンプ
setjmp	大域ジャンプのためのスタック環境の設定

signal.h

機　　能: 非同期割り込みを処理するためのマクロと関数を定義/宣言しています。

stdarg.h

機　　能: 可変長の引数リストを処理するためのマクロと関数を定義しています。

stddef.h

機　　能: 各標準ヘッダファイルで共通に使用するマクロ名を定義しています。

stdio.h

機　　能: FILE 構造体/ストリーム名の定義と、入出力関数の関数原型を宣言しています。関数原型を宣言している関数を以下に示します。

種別	関数名	機能
初期化	clearerr	エラー状態指示子を初期化(クリア)
入力	fgetc	一文字入力
	getc	一文字入力
	getchar	stdin からの一文字入力
	fgets	一行入力
	gets	stdin からの一行入力
	fread	指定データ数入力
	scanf	stdin からの書式付き入力
	fscanf	書式付き入力
	sscanf	文字列からの書式付きデータ入力
出力	fputc	一文字出力
	putc	一文字出力
	putchar	stdout への一文字出力
	fputs	一行出力
	puts	stdout への一行出力
	fwrite	指定データ数出力
	perror	stdout へのエラーメッセージ出力
	printf	stdout への書式付き出力
	fflush	出力バッファのストリームをフラッシュ
	fprintf	書式付き出力
	sprintf	書式付き文字列設定
	vfprintf	ストリームへの書式付き出力
	vprintf	stdout への書式付き出力
	vsprintf	バッファへのへの書式付き出力
返還	ungetc	一文字入力の返還
判定	ferror	入出力エラーの判定
	feof	EOF(End Of File)の判定

stdlib.h

機　　能:　メモリ管理関数、終了関数の関数原型を宣言しています。関数原型を宣言している関数を以下に示します。

関数名	機能
abort	プログラムの実行を終了
abs	int 型整数の絶対値を計算
atof	文字列を double 型浮動小数に変換
atoi	文字列を int 型に変換
atol	文字列を long 型に変換
bsearch	配列内のバイナリサーチを行う
calloc	メモリの確保と 0(ゼロ)による初期化
div	int 型整数の除算と剰余
free	メモリの解放
labs	long 型整数の絶対値を計算
ldiv	long 型整数の除算と剰余
malloc	メモリの確保
mblen	マルチバイト文字列の長さを計算
mbstowcs	マルチバイト文字列をワイド文字列に変換
mbtowc	マルチバイト文字をワイド文字に変換
qsort	配列をソート
realloc	確保済み領域の大きさを変更
strtod	文字列を double 型に変換
strtol	文字列を long 型に変換
strtoul	文字列を unsigned long 型に変換
wcstombs	ワイド文字列をマルチバイト文字列に変換
wctomb	ワイド文字をマルチバイト文字に変換

string.h

機　　能: 文字列操作関数とメモリ操作関数の関数原型を宣言しています。関数原型を宣言している関数を以下に示します。

種別	関数名	機能
複写	strcpy	文字列の複写
	strncpy	文字列の複写(n 文字の複写)
連結	strcat	文字列の連結
	strncat	文字列の連結(n 文字の連結)
比較	strcmp	文字列の比較
	strcoll	文字列の比較(ロケール情報を使用)
	stricmp	文字列の比較(すべての英字は英大文字として扱う)
	strncmp	文字列の比較(n 文字の比較)
	strnicmp	文字列の比較(n 文字の比較、英字は英大文字として扱う)
検索	strchr	文字列の先頭より指定文字を検索
	strcspn	文字列より指定以外の文字列の長さを計算
	strpbrk	文字列より指定文字の検索
	strrchr	文字列の末尾より指定文字を検索
	strspn	文字列より指定文字列の長さを計算
	strstr	文字列より指定文字列の検索
	strtok	文字列より文字列を切り出す
長さ	strlen	文字列中の文字数を計算
変換	strerror	エラー番号を文字列に変換
	strxfrm	文字列を変換(ロケール情報を使用)
初期化	bzero	メモリ領域の初期化(ゼロクリア)
複写	bcopy	メモリ領域の複写
	memcpy	メモリ領域の複写(n 文字の複写)
	memset	メモリ領域の設定
比較	memcmp	メモリ量域の比較(n バイトの比較)
	memicmp	メモリ領域の比較(英文字は大文字として扱う)
検索	memchr	メモリ領域より文字を検索

time.h

機　　能: 現在の暦時間を表現するための関数の宣言と型を定義しています。

E.2 標準関数リファレンス

本コンパイラの標準関数ライブラリの機能と詳細の仕様を説明します。

E.2.1 標準関数ライブラリの概要

本コンパイラは標準関数ライブラリを用意しています。各関数は機能的に以下の種類に分類されます。

(1) 文字列操作関数
文字列のコピー、比較等を行う関数です。

(2) 文字判定関数
アルファベット、10 進文字等の判定、及び大文字から小文字へ、小文字から大文字へ変換する関数です。

(3) 入出力関数
文字、文字列の入出力を行う関数です。中には、書式変換付きの入出力、及び文字列操作を行う関数も含まれています。

(4) メモリ管理関数
動的メモリ領域の確保、及び解放を行う関数です。

(5) メモリ操作関数
メモリ領域の複写、設定、及び比較を行う関数です。

(6) 実行制御関数
プログラムの実行を終了する関数と、現在実行中の関数から別の関数へジャンプするための関数です。

(7) 数学関数
sin、cos 等の演算を行う関数です。
- これらの関数は時間を要します。このため、監視タイマ使用時は十分注意してください。

(8) 整数算術関数
整数値に対しての演算を行う関数です。

(9) 文字列数値変換関数
文字列を数値に変換する関数です。

(10) 多バイト文字/多バイト文字列操作関数
多バイト文字/多バイト文字列を扱う関数です。

(11) 地域化関数
ロケールに関する関数です。

E.2.2 標準関数ライブラリ機能別一覧

a. 文字列操作関数

文字列操作関数の一覧を【表 E.2】に示します。

表E.2 文字列操作関数

種別	関数名	機能	ﾘｴﾝﾄﾗﾝﾄ性
複写	strcpy	文字列の複写	○
	strncpy	文字列の複写(n 文字の複写)	○
連結	strcat	文字列の連結	○
	strncat	文字列の連結(n 文字の連結)	○
比較	strcmp	文字列の比較	○
	strcoll	文字列の比較(ロケール情報を使用)	○
	stricmp	文字列の比較(すべての英字は英大文字として扱う)	○
	strncmp	文字列の比較(n 文字の比較)	○
	strnicmp	文字列の比較(n 文字の比較、英字は英大文字として扱う)	○
検索	strchr	文字列の先頭より指定文字を検索	○
	strcspn	文字列より指定以外の文字列の長さを計算	○
	strpbrk	文字列より指定文字の検索	○
	strrchr	文字列の末尾より指定文字を検索	○
	strspn	文字列より指定文字列の長さを計算	○
	strstr	文字列より指定文字列の検索	○
	strtok	文字列より文字列を切り出す	×
長さ	strlen	文字列中の文字数を計算	○
変換	strerror	エラー番号を文字列に変換	×
	strxfrm	文字列の複写（n 文字の複写、ロケール情報を使用）	○

b. 文字操作関数

文字操作関数の一覧を【表E.3】に示します。

表E.3 文字操作関数

関数名	機能	ﾘｴﾝﾄﾗﾝﾄ性
isalnum	英数字の判定	○
isalpha	英字の判定	○
iscntrl	コントロール文字の判定	○
isdigit	数字の判定	○
isgraph	英数字、ブランク以外の文字判定	○
islower	英小文字の判定	○
isprint	ブランク文字を含む印字可能文字の判定	○
ispunct	区切り文字の判定	○
isspace	ブランク、タブ、改行の判定	○
isupper	英大文字の判定	○
isxdigit	16 進数字の判定	○
tolower	大文字から小文字への変換	○
toupper	小文字から大文字への変換	○

c. 入出力関数

入出力関数の一覧を【表E.4】に示します。

表E.4 入出力関数

種別	関数名	機能	ﾘｴﾝﾄﾗﾝﾄ性
初期化	clearerror	エラー状態指示子を初期化(クリア)	×
入力	fgetc	一文字入力	×
	getc	一文字入力	×
	getchar	stdin からの一文字入力	×
	fgets	一行入力	×
	gets	stdin からの一行入力	×
	fread	指定データ数入力	×
	scanf	stdin からの書式付き入力	×
	fscanf	書式付き入力	×
	sscanf	文字からの書式付きデータ入力	×
出力	fputc	一文字出力	×
	putc	一文字出力	×
	putchar	stdout への一文字出力	×
	fputs	一行出力	×
	puts	stdout への一行出力	×
	fwrite	指定データ数出力	×
	perror	stdout へのエラーメッセージ出力	×
	printf	stdout への書式付き出力	×
	fflush	出力バッファのストリームをフラッシュ	×
	fprintf	書式付き出力	×
	sprintf	書式付き文字列設定	×
	vfprintf	ストリームへの書式付き出力	×
	vprintf	stdout への書式付き出力	×
	vsprintf	バッファへの書式付き出力	×
返還	ungetc	一文字入力の返還	×
判定	ferror	入出力エラーの判定	×
	feof	EOF(End Of File)の判定	×

d. メモリ管理関数

メモリ管理関数の一覧を【表E.5】に示します。

表E.5 メモリ管理関数

関数名	機能	リエントラント性
calloc	メモリの確保と 0(ゼロ)による初期化	×
free	メモリの解放	×
malloc	メモリの確保	×
realloc	確保済み領域の大きさを変更	×

e. メモリ操作関数

メモリ操作関数の一覧を【表E.6】に示します。

表E.6 メモリ操作関数

種別	関数名	機能	リエントラント性
初期化	bzero	メモリ領域の初期化(ゼロクリア)	○
複写	bcopy	メモリ領域の複写	○
	memcpy	メモリ領域の複写(n 文字の複写)	○
	memset	メモリ領域の設定	○
比較	memcmp	メモリ量域の比較(n バイトの比較)	○
	memicmp	メモリ領域の比較(英文字は大文字として扱う)	○
移動	memmove	文字列の領域を移動	○
検索	memchr	メモリ領域より文字を検索	○

f. 実行制御関数

実行制御関数の一覧を【表E.7】に示します。

表E.7 実行制御関数

関数名	機能	リエントラント性
abort	プログラムの実行を終了	○
longjmp	大域ジャンプ	○
setjmp	大域ジャンプのためのスタック環境を設定	○

g. 数学関数

数学関数の一覧表を【表E.8】に示します。

表E.8 数学関数

関数名	機能	リエントラント性
acos	逆コサインを計算	×
asin	逆サインを計算	×
atan	逆タンジェントを計算	○
atan2	逆タンジェントを計算	×
ceil	整数繰り上げ値を計算	○
cos	コサインを計算	○
cosh	双曲線コサインを計算	○
exp	指数関数を計算	○
fabs	倍精度浮動小数の絶対値を計算	○
floor	整数繰り下げ値を計算	○
fmod	剰余計算	○
frexp	浮動小数を仮数部と指数部に分割	○
labs	long 型整数の絶対値を計算	○
ldexp	浮動小数の巾を計算	○
log	自然対数を計算	×
log10	常用対数を計算	×
modf	実数を仮数部と指数部に分割	○
pow	巾乗計算	×
sin	サインを計算	○
sinh	双曲線サインを計算	○
sqrt	数値の平方根を計算	×
tan	タンジェントを計算	○
tanh	双曲線タンジェントを計算	○

h. 整数算術関数

整数算術関数の一覧表を【表E.9】に示します。

表E.9 整数算術関数

関数名	機能	リエントラント性
abs	整数の絶対値を計算	○
bsearch	配列内のバイナリサーチを行う	○
div	int 型整数の除算と剰余	○
labs	long 型整数の絶対値を計算	○
ldiv	long 型整数の除算と剰余	○
qsort	配列をソート	×
rand	疑似乱数を発生	○
srand	疑似乱数発生関数 rand にシード（種）を与える	○

i. 文字列数値変換関数

文字列数値変換関数の一覧表を【表E.10】に示します。

表E.10 文字列数値変換関数

関数名	機能	リエントラント性
atof	文字列を double 型に変換	○
atoi	文字列を int 型に変換	○
atol	文字列を long 型に変換	○
strtod	文字列を double 型に変換	○
strtol	文字列を long 型に変換	○
strtou	文字列を unsigned long 型に変換	○

j. 多バイト文字/多バイト文字列操作関数

多バイト文字/多バイト文字列操作関数の一覧表を【表E.11】に示します。

表E.11 多バイト文字/多バイト文字列操作関数

関数名	機能	リエントラント性
mblen	マルチバイト文字列の長さを計算	○
mbstowcs	マルチバイト文字列をワイド文字列に変換	○
mbtowc	マルチバイト文字をワイド文字に変換	○
wcstombs	ワイド文字列をマルチバイト文字列に変換	○
wctomb	ワイド文字をマルチバイト文字に変換	○

k. 地域化関数

地域化関数の一覧表を【表E.12】に示します。

表E.12 地域化関数

関数名	機能	リエントラント性
localeconv	構造体 lconv を初期化	○
setlocale	プログラムのロケール情報の設定と検索	○

E.2.3 標準関数リファレンス

以降に本コンパイラが提供する標準関数の詳細仕様を説明します。関数は、アルファベット順に掲載しています。なお、[書式]に記述している標準ヘッダファイル(拡張子.h)は、その関数を使用するときに必ずインクルードしてください。

A

abort

実行制御関数

機　　能: プログラムを異常終了します。

書　　式: #include <stdlib.h>

void abort(void);

実現方法: 関数

引　　数: 引数はありません。

戻 り 値: 戻り値はありません。

解　　説: プログラムを異常終了します。

注　　意: 実際には、abort プログラム内部で無限ループとなります。

abs

整数算術関数

機　　能: int 型整数の絶対値を計算します

書　　式: #include <stdlib.h>

int abs(n);

実現方法: 関数

引　　数: int n; 実数

戻 り 値: int 型整数 n の絶対値(0 からの距離)を返します。

acos

数学関数

機　　能: 逆コサインを計算します。

書　　式: #include <math.h>

double acos(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値:
- x の値が-1.0 から 1.0 の範囲外の場合はエラーとして 0 を返します。
- それ以外の場合は、0 からπラジアンの範囲の値を返します。

asin

数学関数

機　　能: 逆サインを計算します。

書　　式: #include <math.h>

double asin(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値:
- x の値が-1.0 から 1.0 の範囲外の場合はエラーとして 0 を返します。
- それ以外の場合は、-π/2 からπ/2 ラジアンの範囲の値を返します。

atan

数学関数

機　　能: 逆タンジェントを計算します。

書　　式: #include <math.h>

double atan(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: -π/2 からπ/2 ラジアンの範囲の実数を返します。

atan2

数学関数

機　　能: "x"と"y"の商に対する逆タンジェントを計算します。

書　　式: #include <math.h>

double atan2(x , y);

実現方法: 関数

引　　数: double x; 実数
double y; 実数

戻 り 値: -πからπラジアンの範囲の値を返します。

atof

文字列数値変換関数

機　　能: 文字列を浮動小数に変換します。

書　　式: #include <stdlib.h>

double atof(s);

実現方法: 関数

引　　数: const char _far *s; 変換文字列へのポインタ

戻 り 値: 文字列を倍精度浮動小数に変換した値を返します。

atoi

文字列数値変換関数

機　　能: 文字列を int 型整数に変換します。

書　　式: #include <stdlib.h>

int atoi(s);

実現方法: 関数

引　　数: const char _far *s; 変換文字列へのポインタ

戻 り 値: 文字列を int 型整数に変換した値を返します。

atol

文字列数値変換関数

機　　能: 文字列を long 型整数に変換します。

書　　式: #include <stdlib.h>

long atol(s);

実現方法: 関数

引　　数: const char _far *s; 変換文字列へのポインタ

戻 り 値: 文字列を long 型整数に変換した値を返します。

B

bcopy

メモリ操作関数

機　　能: メモリ領域の複写を行います。

書　　式:
```
#include <string.h>

void bcopy( src, dtop, size );
```

実現方法: 関数

引　　数:

char _far *src;	複写元のメモリ領域の先頭アドレス
char _far *dtop;	複写先のメモリ領域の先頭アドレス
unsigned long size;	複写するバイト数

戻 り 値: dtop で示される領域に、src で示される領域の先頭から size で指定されたバイト数分の内容を複写します。

bsearch

整数算術変換関数

機　　能: 要素を配列から検索します

書　　式:
```
#include <stdlib.h>

void *bsearch( key, base, nelem, size, cmp );
```

実現方法: 関数

引　　数:

const void _far *key;	検索キー
const void _far *base;	配列開始アドレス
size_t nelem;	要素数
size_t size;	各要素の大きさ
int cmp();	比較関数

戻 り 値:
- 検索キーに等しい配列要素へのポインタを返します。
- 一致する要素がない場合は NULL ポインタを返します。

解　　説: 昇順にソート済みの配列内から指定された項目を検索します。

bzero

メモリ操作関数

機　　能: メモリ領域の初期化(ゼロクリア)を行います。

書　　式:
```
#include <string.h>

void bzero( top, size );
```

実現方法: 関数

引　　数:

char _far ＊top;	ゼロクリアするメモリ領域の先頭アドレス
unsigned long size;	ゼロクリアするバイト数

戻 り 値: 戻り値はありません。

解　　説: top で示される領域の先頭アドレスから size で示されるバイト数分の内容を 0 で初期化します。

C

calloc

メモリ管理関数

機　　能: メモリの割り当てとゼロクリアを行います。

書　　式:
```
#include <stdlib.h>

void _far * calloc( n, size );
```

実現方法: 関数

引　　数:

size_t n;	要素の数
size_t size;	要素の大きさをバイト数で表した値

戻 り 値: 指定した大きさの領域が確保できなかった場合、戻り値として NULL を返します。

解　　説:
- 指定されたメモリを割り当てた後、ゼロクリアを行います。
- メモリ領域の大きさは 2 つの引数の積になります。

規　　則: メモリの確保規則については、malloc と同様です。

ceil

数学関数

機　　能: 整数繰り上げ値を返します。

書　　式:
```
#include <math.h>

double ceil( x );
```

実現方法: 関数

引　　数:

double x;	実数

戻 り 値: x より大きい整数の中から最小の整数値を double 型で返します。

clearerr

入出力関数

機　　能: ストリームのエラー状態指示子をクリアします。

書　　式: #include <stdio.h>

void clearerr(stream);

実現方法: 関数

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値: 戻り値はありません。

解　　説: エラー状態指示子とファイル終端状態指示子を正常値にリセットします。

cos

数学関数

機　　能: コサインを計算します。

書　　式: #include <math.h>

double cos(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする引数"x"のコサインを計算します。

cosh

数学関数

機　　能: 双曲線コサインを計算します。

書　　式: #include <math.h>

double cosh(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の双曲線コサインを計算します。

D

div

剰余算関数

機　　能: int 型整数の除算を行います。

書　　式: #include <stdlib.h>

div_t div(number, denom);

実現方法: 関数

引　　数:

int number;	被除数
int denom;	除数

戻 り 値: "number"を"denom"で割った商と剰余を返します。

解　　説:

- "number"を"denom"で割った商と剰余を div_t の構造体で返します。
- div_t は stdlib.h 内で定義しています。この構造体は、int quot、int rem というメンバーから構成されます。

E

exp

数学関数

機　　能:　指数関数を計算します。

書　　式:　#include <math.h>

double exp(x);

実現方法:　関数

引　　数:　double x;　実数

戻 り 値:　"x"の指数関数の計算結果を返します。

F

fabs

数学関数

機　　能: 倍精度浮動小数の絶対値を計算します。

書　　式: #include <math.h>

double fabs(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: 倍精度浮動小数の絶対値を返します。

feof

入出力関数

機　　能: ストリームのファイル状態指示子(EOF)を調べます。

書　　式: #include <stdio.h>

int feof(stream);

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- ストリームが EOF 場合、真(0 以外)を返します。
- それ以外の場合、NULL(0)を返します。

解　　説:
- ストリームが EOF まで読み込んだかどうか判定します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

ferror

入出力関数

機　　能: ストリームのエラー状態を調べます。

書　　式:
```
#include <stdio.h>

int ferror( stream );
```

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- ストリームがエラーの場合、真(0 以外)を返します。
- それ以外の場合、NULL(0)を返します。

解　　説:
- ストリームがエラーかどうか判定します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

fflush

入出力関数

機　　能: 出力バッファをフラッシュします。

書　　式:
```
#include <stdio.h>

int fflush( stream );
```

実現方法: 関数

引　　数: FILE _far *stream; ストリームのポインタ

戻 り 値: 常に 0 を返します。

fgetc

入出力関数

機　　能: ストリームから 1 文字を入力します。

書　　式:
```
#include <stdio.h>

int fgetc( stream );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終りの場合、EOF を返します。

解　　説:
- ストリームから 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

fgets

入出力関数

機　　能: ストリームから文字列を入力します。

書　　式:
```
#include <stdio.h>

char _far * fgets( buffer, n, stream );
```

実現方法: 関数

引　　数:

char _far *buffer;	格納先のポインタ
int n;	最大文字数
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に入力できた場合、格納先のポインタ(引数で与えたポインタと同じ)を返します。
- エラー又はストリームの終わりの場合、NULL ポインタを返します。

解　　説:
- 指定したストリームから文字列を入力し、バッファ(buffer)に格納します。入力を終了するのは、次の 3 つの場合です。
  (1) 改行文字('\n')を入力した場合
  (2) n-1 個の文字を入力した場合
  (3) ストリームの終わりまで入力した場合
- 入力した文字列の最後には、ヌル文字('\0')を付加します。
- 改行文字('\n')は、そのまま格納します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

floor

数学関数

機　　能: 整数の繰り下げ値を計算します。

書　　式: #include <math.h>

double floor(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: 整数の繰り下げ値を double 型で返します

fmod

数学関数

機　　能: 剰余計算を行います。

書　　式: #include <math.h>

double fmod(x ,y);

実現方法: 関数

引　　数: double x; 被除数
double y; 除数

戻 り 値: "x"を"y"で割ったときの剰余を返します。

fprintf

入出力関数

機　　能: ストリームへの書式付き出力を行います。

書　　式:
```
#include <stdio.h>

int fprintf( stream, format, argument... );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列のポインタ

戻 り 値:
- 出力した文字数を返します。
- ハードウェアに起因するエラーの場合、EOF を返します。

解　　説:
- format の指定に従って argument を文字列に変換し、ストリームへ出力します。
- format の指定方法は printf と同様です。

fputc

入出力関数

機　　能: ストリームに 1 文字を出力します。

書　　式:
```
#include <stdio.h>

int fputc( c, stream );
```

実現方法: 関数

引　　数:

int c;	出力する文字
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に出力できた場合、出力した文字を返します。
- エラーの場合、EOF を返します。

解　　説: ストリームに 1 文字を出力します。

fputs

入出力関数

機　　能: ストリームへ文字列を出力します。

書　　式:
```
#include <stdio.h>

int fputs ( str, stream );
```

実現方法: 関数

引　　数:

const char _far *str;	出力する文字列のポインタ
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に出力できた場合、0 を返します。
- エラーの場合、0 以外(EOF)を返します。

解　　説: ストリームへ文字列を出力します。

fread

入出力関数

機　　能: ストリームから固定長データを入力します。

書　　式:
```
#include <stdio.h>

size_t fread( buffer, size, count, stream );
```

実現方法: 関数

引　　数:

void _far *buffer;	格納先のポインタ
size_t size;	データ 1 項目のバイト数
size_t count;	最大データ項目数
FILE _far *stream;	ストリームのポインタ

戻 り 値: 入力したデータ項目数を返します。

解　　説:
- ストリームから size のデータ長を持つデータを count の項目数だけ入力し、バッファ(buffer)に格納します。
- count 分のデータを入力する前にストリームの終わりになった場合、それまでに入力したデータ項目数を返します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

free

メモリ管理関数

機　　能: メモリの解放を行います。

書　　式: #include <stdlib.h>

void free(cp);

実現方法: 関数

引　　数: void _far *cp; 解放するメモリ領域へのポインタ

戻 り 値: 戻り値はありません。

解　　説:
- 以前に関数 malloc、calloc によって割り当てられたメモリ領域の解放を行います。
- NULL を引数にした場合は処理を行いません。

frexp

数学関数

機　　能: 浮動小数を仮数部と指数部に分割します。

書　　式: #include <math.h>

double frexp(x,prexp);

実現方法: 関数

引　　数: double x; 浮動小数
int _far *prexp; 2 を底とする指数を格納する領域へのポインタ

戻 り 値: "x"の仮数部を返します。

fscanf

入出力関数

機　　能: ストリームからの書式付き入力を行います。

書　　式:
```
#include <stdio.h>

int fscanf( stream, format, argument... );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列のポインタ

戻 り 値:
- 各引数 argument に格納したデータ数を返します。
- ストリームからデータとして EOF を入力した場合、EOF を返します。

解　　説:
- format の指定に従ってストリームからの入力文字を変換し、各引数 argument が示す引数に格納します。
- argument は各引数のポインタでなければいけません。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。
- format の指定方法は、scanf と同様です。

fwrite

入出力関数

機　　能: ストリームへ固定長データを出力します。

書　　式:
```
#include <stdio.h>

size_t fwrite( buffer, size, count, stream );
```

実現方法: 関数

引　　数:

const void _far *buffer;	出力データのポインタ
size_t size;	データ 1 項目のバイト数
size_t count;	最大データ項目数
FILE _far *stream;	ストリームのポインタ

戻 り 値: 出力したデータ項目数を返します。

解　　説:
- ストリームへ size のデータ長を持つデータを count の項目数だけ出力します。
- count 分のデータを出力する前にエラーになった場合は、それまでに出力したデータ項目数を返します。

G

getc

入出力関数

機　　能: ストリームから 1 文字を入力します。

書　　式: #include <stdio.h>

int getc(stream);

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終わりの場合、EOF を返します。

解　　説:
- ストリームから 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

getchar

入出力関数

機　　能: stdin から 1 文字を入力します。

書　　式: #include <stdio.h>

int getchar(void);

実現方法: マクロ

引　　数: 引数はありません。

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終わりの場合、EOF を返します。

解　　説:
- ストリーム(stdin)から 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

gets

入出力関数

機　　能: stdin から文字列を入力します。

書　　式: #include <stdio.h>

char _far * gets(buffer);

実現方法: 関数

引　　数: char _far *buffer; 格納先のポインタ

戻 り 値:
- 正常に入力できた場合、格納先のポインタ(引数で与えたポインタと同じ)を返します。
- エラー又はストリームの終わりの場合、NULL ポインタを返します。

解　　説:
- stdin から文字列を 1 行入力し、バッファ(buffer)に格納します。
- 行末の改行文字('\n')は、ヌル文字('\0')に置き換えます。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

I

isalnum

文字操作関数

機　　能: 英字(A～Z、a～z)、数字(0～9)を判定します。

書　　式: #include <ctype.h>

int isalnum(c);

実現方法: マクロ

引　　数: int c; 判定する文

戻 り 値:
- 英字、又は数字の場合、0 以外を返します。
- 英字、又は数字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isalpha

文字操作関数

機　　能: 英字(A～Z、a～z)を判定します。

書　　式: #include <ctype.h>

int isalpha(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 英字の場合、0 以外を返します。
- 英字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

iscntrl

文字操作関数

機　　能: コントロール文字(0x00～0x1f、0x7f)を判定します。

書　　式: #include <ctype.h>

int iscntrl(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- コントロール文字の場合、0 以外を返します。
- コントロール文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isdigit

文字操作関数

機　　能: 数字(0～9)を判定します。

書　　式: #include <ctype.h>

int isdigit(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 数字の場合、0 以外を返します。
- 数字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isgraph

文字操作関数

機　　能: ブランク以外の文字(0x21～0x7e)を印字文字判定します。

書　　式: #include <ctype.h>

int isgraph(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 印字可能な場合、0 以外を返します。
- 印字不可の場合、0 を返します。

解　　説: 引数の文字を判定します。

islower

文字操作関数

機　　能: 英小文字(a～z)を判定します。

書　　式: #include <ctype.h>

int islower(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 英小文字の場合、0 以外を返します。
- 英小文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isprint

文字操作関数

機　　能: ブランク文字を含む文字(0x20～0x7e)の印字文字を判定します。

書　　式: #include <ctype.h>

int isprint(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 印字可能な場合、0 以外を返します。
- 印字不可の場合、0 を返します。

解　　説: 引数の文字を判定します

ispunct

機　　能: 区切り文字を判定します

書　　式: #include <ctype.h>

int ispunct(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 区切り文字の場合、0 以外を返します。
- 区切り文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isspace

文字操作関数

機　　能: ブランク、タブ、改行を判定します。

書　　式: #include <ctype.h>

int isspace(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- ブランク、タブ、改行の場合、0 以外を返します。
- ブランク、タブ、改行でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isupper

文字操作関数

機　　能: 文字操作関数

書　　式: #include <ctype.h>

int isupper(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 英大文字の場合、0 以外を返します。
- 英大文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isxdigit

文字操作関数

機　　能: 16 進文字(0～9、A～F、a～f)を判定します。

書　　式: #include <ctype.h>

int isxdigit(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 16 進文字の場合、0 以外を返します。
- 16 進文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

L

labs

整数算術関数

機　　能: long 型整数の絶対値を計算します。

書　　式: #include <stdlib.h>

long labs(n);

実現方法: 関数

引　　数: long n; long 型整数

戻 り 値: long 型整数の絶対値(0 からの距離)を返します。

ldexp

数学関数

機　　能: 浮動小数の巾を計算します。

書　　式: #include <math.h>

double ldexp(x,exp);

実現方法: 関数

引　　数: double x; 実数
int exp; 巾乗数

戻 り 値: "x"*(2 の"exp"乗)を返します。

ldiv

整数算術関数

機　　能: long 型整数の除算を行います。

書　　式: #include <stdlib.h>

ldiv_t ldiv(number, denom);

実現方法: 関数

引　　数: long number; 被除数
long denom; 除数

戻 り 値: "number"を"denom"で割った商と剰余を返します。

解　　説:
- "number"を"denom"で割った商と剰余を ldiv_t の構造体で返します。
- ldiv_t は stdlib.h 内で定義しています。この構造体は、long quot,long rem というメンバーから構成されます。

localeconv

地域化関数

機　　能: 構造体 lconv を初期化します。

書　　式: #include <locale.h>

struct lconv _far *localeconv(void);

実現方法: 関数

引　　数: 引数はありません。

戻 り 値: 初期化された構造体 lconv へのポインタを返します。

log

数学関数

機　　能: 自然対数を計算します。

書　　式: #include <math.h>

double log(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の自然対数を返します。

解　　説: 関数 exp の逆関数です。

log10

数学関数

機　　能: 常用対数を計算します。

書　　式: #include <math.h>

double log10(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の常用対数を返します。

longjmp

実行制御関数

機　　能: 関数呼び出し時の環境の回復を行います。

書　　式: #include <setjmp.h>

void longjmp(env, val);

実現方法: 関数

引　　数: jmp_buf env; 環境を回復する領域へのポインタ
int val; setjmp の結果として返す値

戻 り 値: 戻り値はありません。

解　　説:
- "env"で示される領域から環境を回復します。
- プログラムの制御は、以前に setjmp 関数を呼んだ次の文に移ります。
- "val"で指定した値は、setjmp 関数の結果として返します。ただし、"val"が"0"の場合"1"に変換されます。

M

malloc

メモリ管理関数

機　　能: malloc メモリの割り当てを行います。

書　　式: #include <stdlib.h>

void _far * malloc(nbytes);

実現方法: 関数

引　　数: size_t nbytes; 割り当てるメモリの大きさバイト数

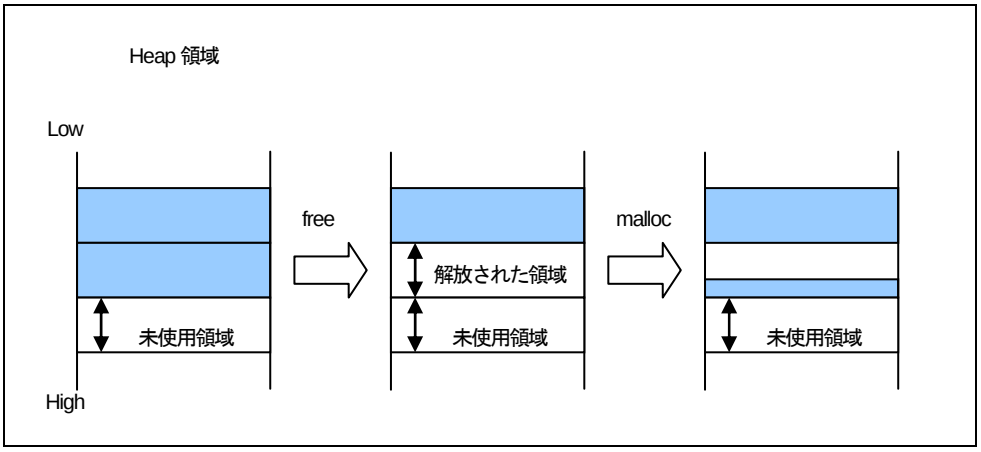
戻 り 値: 指定した大きさの領域が確保できなかった場合は戻り値として NULL を返します。

解　　説: 動的なメモリ領域を割り当てます。

規　　則: malloc を行う場合、以下の(1)～(2)の順にチェックを行い、適合する箇所でメモリを確保します。

(1) free で解放された領域がある場合

- 確保するサイズが free で解放された領域より小さい場合、free で作成された連続空き領域の上位番地から下位番地方向へ領域が確保されます。

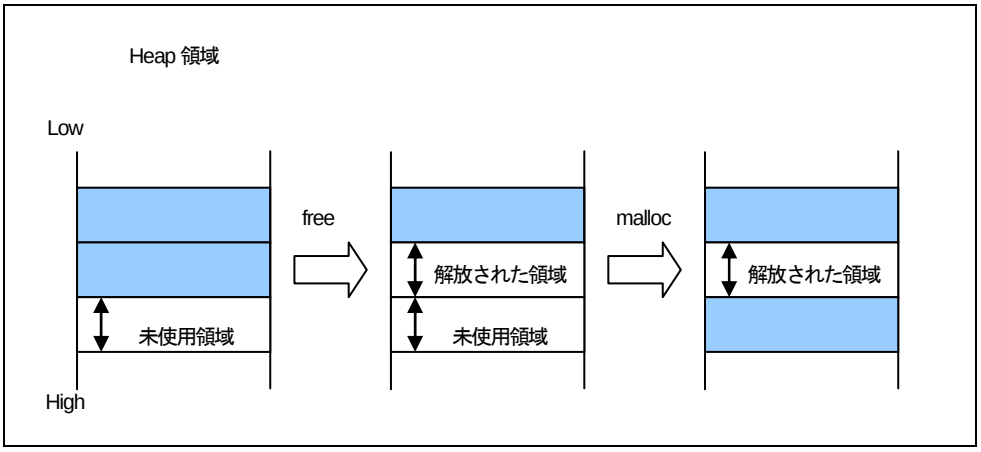


malloc

メモリ管理関数

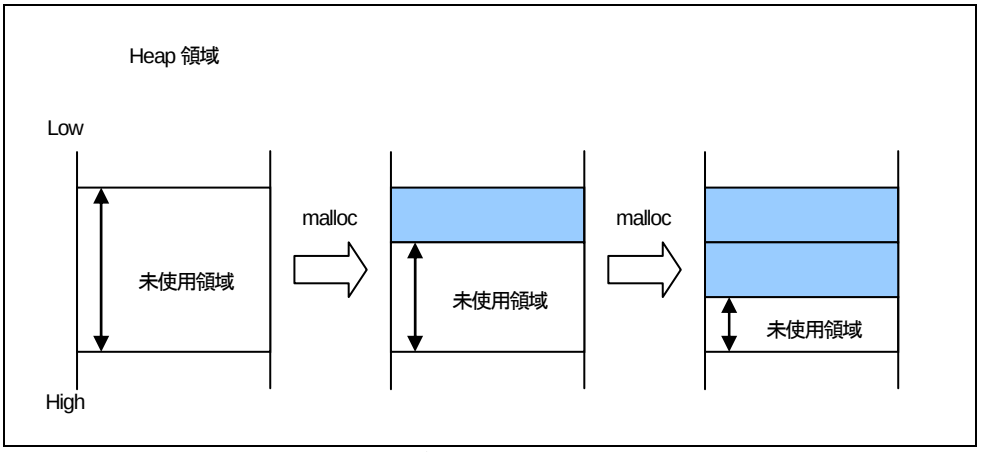
規　　則:

- 確保するサイズが free で解放された領域よりも大きい場合、未使用領域の下位番地から上位番地方向に領域が確保されます。



(2) free で解放された領域がない場合

- 確保可能な未使用領域が存在する場合、未使用領域の下位番地から上位番地方向に領域が確保されます。



- 確保可能な未使用領域が存在しない場合、メモリは確保せずに、NULL を返します。

注意事項:　ガーベージコレクションは行っていません。したがって、小さな未使用領域が多く存在していても、指定した領域サイズ以上の未使用領域が無ければメモリを確保できず、NULL を返します。

mblen

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字列の長さを計算します。

書　　式:
```
#include <stdlib.h>

int mblen ( s,n );
```

実現方法: 関数

引　　数:

const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	検索バイト数

戻 り 値:
- s が正しいマルチバイト文字列をなしている場合はそのバイト数を返します。
- s が正しいマルチバイト文字列をなしていない場合は-1 を返します。

解　　説: s が NULL 文字を指している場合は 0 を返します。

mbstowcs

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字列をワイド文字列に変換します。

書　　式:
```
#include <stdlib.h>

size_t mbstowcs( wcs,s,n );
```

実現方法: 関数

引　　数:

wchar_t _far *wcs;	変換ワイド文字列格納領域へのポインタ
const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	格納ワイド文字数

戻 り 値:
- 変換したマルチバイト文字列の文字数を返します。
- 正しいマルチバイト文字列をなしてない場合は-1 を返します。

mbtowc

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字をワイド文字に変換します。

書　　式:
```
#include <stdlib.h>

int mbtowc( wcs,s,n );
```

実現方法: 関数

引　　数:

wchar_t _far *wcs;	変換ワイド文字列格納領域へのポインタ
const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	検索バイト文字数

戻 り 値:
- s が正しいマルチバイト文字をなしている場合は変換したワイド文字数を返します。
- s が正しいマルチバイト文字をなしていない場合は-1 を返します。
- s が NULL 文字の場合は 0 を返します。

memchr

メモリ操作関数

機　　能: メモリ領域より文字の検索を行います。

書　　式:
```
#include <string.h>

void _far * memchr( s, c, n );
```

実現方法: 関数

引　　数:

const void _far *s;	検索先のメモリ領域へのポインタ
int c;	検索する文字
size_t n;	検索するメモリ領域の大きさ

戻 り 値:
- 見つかった場合、指定された文字"c"の位置(ポインタ)を返します。
- メモリ領域中に文字"c"が見つからない場合は NULL を返します。

解　　説:
- "s"で示されるアドレスから始まる"n"で指定された大きさのメモリ領域から、"c"で示される文字を検索します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memcmp

メモリ操作関数

機　　能: 2 つのメモリ領域の(n バイト)比較を行います。

書　　式:
```
#include <string.h>

int memcmp( s1, s2, n );
```

実現方法: 関数

引　　数:

const void _far *s1;	比較する 1 番目のメモリ領域へのポインタ
const void _far *s2;	比較する 2 番目のメモリ領域へのポインタ
size_t n;	比較するバイト数

戻 り 値:
- 戻り値==0　2 番目のメモリ領域は等しい
- 戻り値>0　1 番目のメモリ領域(s1)の方が大きい
- 戻り値<0　2 番目のメモリ領域(s2)の方が大きい

解　　説:
- 2 つのメモリ領域を n バイト分、バイト単位に比較します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memcpy

メモリ操作関数

機　　能: メモリ領域の(n バイト)複写を行います。

書　　式:
```
#include <string.h>

void _far * memcpy( s1, s2, n );
```

実現方法: 関数

引　　数:

void _far *s1;	複写先のメモリ領域へのポインタ
const void _far *s2;	複写元のメモリ領域へのポインタ
size_t n;	複写するバイト数

戻 り 値: 複写先のメモリ領域へのポインタを返します。

解　　説:
- "s1"で示される領域に、"n"で指定されたバイト数分"s2"で示されるメモリ領域より複写します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memicmp

メモリ操作関数

機　　能: 2 つのメモリ領域の(n バイト)比較を行います(英字はすべて大文字として扱います)。

書　　式:
```
#include <string.h>

int memicmp( s1, s2, n);
```

実現方法: 関数

引　　数:

char _far *s1;	比較する 1 番目のメモリ領域へのポインタ
char _far *s2;	比較する 2 番目のメモリ領域へのポインタ
size_t n;	比較するバイト数

戻 り 値:
- 戻り値= =0　2 番目のメモリ領域は等しい
- 戻り値>0　1 番目のメモリ領域(s1)の方が大きい
- 戻り値<0　2 番目のメモリ領域(s2)の方が大きい

解　　説:
- 2 つのメモリ領域を、n バイト分バイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memmove

メモリ操作関数

機　　能: メモリ領域を移動します。

書　　式:
```
#include <string.h>

void _far * memmove( s1, s2, n );
```

実現方法: 関数

引　　数:

void _far *s1;	移動先へのポインタ
const void _far *s2;	移動元へのポインタ
size_t n;	移動バイト数

戻 り 値: 移動先へのポインタを返します。

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memset

メモリ操作関数

機　　能: メモリ領域を設定します。

書　　式:
```
#include <string.h>

void _far * memset( s, c, n );
```

実現方法: 関数

引　　数:

void _far *s;	設定先のメモリ領域への先頭ポインタ
int c;	設定するデータ
size_t n;	設定するバイト数

戻 り 値: 設定先のメモリ領域への先頭ポインタを返します。

解　　説:
- "s"で示される領域に、"n"で指定されたバイト数分"c"で指定されたデータを設定します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

modf

数学関数

機　　能: 実数を整数部と小数部に分割します。

書　　式:
```
#include <math.h>

double modf ( val,pd );
```

実現方法: 関数

引　　数:

double val;	実数
double *pd;	整数部格納領域へのポインタ

戻 り 値: 実数の小数部を返します。

P

perror

入出力関数

機　　能: エラーメッセージを stderr に出力します。

書　　式: #include <stdio.h>

void perror(s);

実現方法: 関数

引　　数: const char _far *s; メッセージの前につく文字列へのポインタ

戻 り 値: 戻り値はありません。

pow

数学関数

機　　能: 巾乗計算を行います。

書　　式: #include <math.h>

double pow(x,y);

実現方法: 関数

引　　数: double x; 被乗数
double y; 巾乗数

戻 り 値: "x"の"y"乗を返します。

printf

入出力関数

機　　能:　stdout への書式付き出力を行います。

書　　式:　#include <stdio.h>

int printf(format, argument...);

実現方法:　関数

引　　数:　const char _far *format;　　書式指定文字列のポインタ

format で与えられる文字列の%以降の指定は、次の意味を持ちます。[]内は省略可能です。

> **書　式**：%[フラグ][最小フィールド幅][精度][修飾文字]変換指定記号
> **書式例**：%-05.8ld

戻 り 値:
- 出力した文字数を返します。
- ハードウェアに起因するエラーの場合、EOF を返します。

解　　説:
- format の指定に従って argument を文字列に変換し、stdout へ出力します。
- argument にポインタを与える場合は、far 型ポインタである必要があります。

(1)　変換指定記号

- d, i
  引数の整数を符号付き 10 進数に変換します。
- u
  引数の整数を符号なし 10 進数に変換します。
- o
  引数の整数を符号なし 8 進数に変換します。
- x
  引数の整数を符号なし 16 進数に変換します。0AH～0FH に小文字の"abcdef"を使用します。
- X
  引数の整数を符号なし 16 進数に変換します。0AH～0FH に大文字の"ABCDEF"を使用します。
- c
  引数の文字を 1 文字(ASCII 文字)で出力します。
- s
  引数の文字列 far 型ポインタ(char *)以降(ヌル文字'\0'又は精度数まで)を文字列に変換します。なお、wchar_t 型の文字列は扱うことができません。
- p
  引数のポインタ(すべての型に対応)を 24 ビットアドレスで出力します。
- n
  n 変換は引数の整数ポインタにそれまでに出力した文字数を格納します。引数は変換されません。
- e
  double の引数を指数形式に変換します。形式は[-]d.dddddde±dd です。
- E
  指数表示の e の代りに E が使用される以外は、e と同じ書式です。

printf

入出力関数

解　　説:

- f
  double の引数を[-]d.dddddd 形式に変換します。
- g
  double の引数を e 又は f により指定される形式に変換します。通常は f 型の変換を行います。指数部が-4 以下、又は精度が指数値以下の場合は、e 型に変換されます。
- G
  指数表示の e の代りに E が使用される以外は、g と同じ書式です。
- －
  変換の結果を最小フィールド幅内で左寄せします。デフォルトは右寄せです。
- ＋
  符号付きの変換結果に＋又は－を付けます。デフォルトは、負の数のみ－を付けます。
- ブランク' '
  デフォルトで符号付きの変換結果が符号なしになった場合、頭にブランク' ' を付けます。
- #
  o 変換では、頭に 0 を付けます。
  x、X 変換では 0 以外の時、頭に 0x、0X を付けます。
  e、E、f 変換では、常に小数点を付けます。
  g、G 変換では、常に小数点を付け、小数点以下の 0 も切り捨てずに出力します。

(2) 最小フィールド幅

- 正の 10 進整数で最小のフィールド幅を指定します。
- 変換結果の文字数が指定したフィールド数より少ない場合 左に埋め文字を挿入してフィールド幅を合わせます。
- デフォルトの埋め文字はブランク' 'です。頭に 0 を付けた整数でフィールド幅を指定した場合は、'0'を埋め文字とします。
- －フラグを指定した場合は、変換結果を左寄せし右に埋め文字を挿入します。この場合、埋め文字は常にブランク' 'です。
- 最小フィールド幅にアスタリスク(*)を指定した場合、引数の整数がフィールド幅を指定します。引数の値が負の場合、－フラグの後に正のフィールド幅を指定したことになります。

(3) 精度

'.'の後に正の整数で指定し、'.'のみの場合はゼロを指定したことになります。
機能、及びデフォルト値は変換タイプにより異なります。
精度の指定がない浮動小数点型データの出力は、精度 6 で処理されます。ただし、精度が 0 のときは小数点は出力されません。

- d、i、o、u、x、X 変換の場合
  (1) 変換結果の桁数が指定した桁数より小さい場合、頭に'0'を挿入して桁数を合わせます。
  (2) この指定が最小フィールド幅より大きい場合、こちらの指定が優先されます。
  (3) この指定が最小フィールド幅より小さい場合、最小桁数の処理を行った後、フィールド幅の処理を行います。
  (4) デフォルト値は 1 です。
  (5) ゼロを最小桁数 0 で変換した場合、何も出力しません。

printf

入出力関数

解　説:

- s 変換の場合
  (1) 最大文字数を表します。
  (2) 変換結果が指定した文字数を越える場合⇒後が切り捨てられます。
  (3) デフォルトには文字数制限がありません。
  (4) 精度の指定にアスタリスク（＊）を指定した場合⇒引数の整数が精度を指定します。
  (5) 引数の値が負の場合⇒精度の指定は無効になります。
- e、E、f 変換の場合
  小数点の後に n 個(n は精度)の数字を出力します。
- g、G 変換の場合
  n 個(n は精度)以上の有効数字を出力しません。

(4) 修飾子

- l の場合、d、i、o、u、x、X、n 変換を long int 又は unsigned long int の引数に対して行います。l を d、i、o、u、x、X、n 変換以外で指定した場合、指定が無視されます。
- ll の場合、d、i、o、u、x、X、n 変換を long long 又は unsigned long long の引数に対して行います。ll を d、i、o、u、x、X、n 変換以外で指定した場合、指定が無視されます。
- h の場合、d、i、o、u、x、X 変換を short int 又は unsigned short int の引数に対して行います。h を d、i、o、u、x、X、n 変換以外で指定した場合、指定が無視されます。
- L の場合、e、E、f、g、G 変換を double の引数に対して行います。

注　意:

統合開発環境(High-performance Embedded Workshop)で、R8C(ROM64KB 未満) の新規プロジェクトを作成した場合、ROM 容量を少なくするために、浮動小数点変換(%e,%E,%f,%g,%G)を使えない仕様です。
また標準で添付している r8clib.lib,r8cs16.lib ライブラリも同様です。
必要な場合には、ライブラリジェネレータで-nofloat を指定せずにライブラリを生成してご使用ください。

putc

入出力関数

機　　能: ストリームに 1 文字を出力します。

書　　式:
```
#include <stdio.h>

int putc( c, stream );
```

実現方法: マクロ

引　　数:

int c;	出力する文字
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に出力できた場合、出力した文字を返します。
- エラーの場合、EOF を返します。

解　　説: ストリームに 1 文字を出力します。

putchar

入出力関数

機　　能: stdout に 1 文字を出力します。

書　　式:
```
#include <stdio.h>

int putchar( c );
```

実現方法: マクロ

引　　数:

int c;	出力する文字

戻 り 値:
- 正常に出力できた場合、出力した文字を返します。
- エラーの場合、EOF を返します。

解　　説: stdout に 1 文字を出力します。

puts

入出力関数

機　　能: stdout へ文字列を出力します。

書　　式:
```
#include <stdio.h>

int puts( str );
```

実現方法: 関数

引　　数: char _far *str; 出力する文字列のポインタ

戻 り 値:
- 正常に出力できた場合、0 を返します。
- エラーの場合、-1(EOF)を返します。

解　　説:
- stdout へ文字列を出力します。
- 文字列最後のヌル文字('\0')は、改行文字('\n')に置き換えて出力します。

Q

qsort

整数算術関数

機　　能: 配列をソートします。

書　　式: #include <stdlib.h>

void qsort(base,nelen,size,cmp(e1,e2));

実現方法: 関数

引　　数:

void _far *base;	配列開始アドレス
size_t nelen;	要素数
size_t size;	各要素の大きさ
int cmp( );	要素を比較する関数

戻 り 値: 戻り値はありません

解　　説: 配列をソートします。

R

rand

整数算術関数

機　　能: 疑似乱数を発生します。

書　　式:
```
#include <stdlib.h>

int rand( void );
```

実現方法: 関数

引　　数: 引数はありません。

戻 り 値:
- srand で指定した seed 乱数の系列を返します。
- 発生させる乱数は 0～RAND_MAX 間の値をとります。

realloc

メモリ管理関数

機　　能: 確保済み領域の大きさを変更します。

書　　式:
```
#include <stdlib.h>

void _far * realloc( cp, n bytes );
```

実現方法: 関数

引　　数:

void _far *cp;	変更前のメモリ領域へのポインタ
size_t nbytes;	変更するメモリ領域の大きさ(バイト数)

戻 り 値:
- 大きさが変更された領域のポインタを返します。
- 指定した大きさの領域が確保できなかった場合は NULL を返します。

解　　説:
- 関数 malloc や calloc ですでに確保済みの領域の大きさを変更します。
- 引数"cp"にすでに確保済みのポインタを指定し、引数"n bytes"で変更する大きさを指定します。

S

scanf

入出力関数

機　　能:　stdin からの書式付き入力を行います。

書　　式:
```
#include <stdio.h>
#include <ctype.h>

int scanf( format, argument... );
```

実現方法:　関数

引　　数:　const char _far *format;　　書式指定文字列のポインタ

戻 り 値:　format で与える文字列の%以降の指定は、次の意味を持ちます。[ ]内は省略可能です。

> **書式 :**　%[＊][最大フィールド幅][修飾子]変換指定記号
> **書式例 :**　%*5ld

戻 り 値:
- 各引数 argument に格納したデータ数を返します。
- stdin からデータとして EOF を入力した場合、EOF を返します。

解　　説:
- format の指定に従って stdin からの入力文字を変換し、各引数 argument が示す変数に格納します。
- argument は各変数の far 型ポインタでなければなりません。
- c、[ ]以外の変換において、先頭で入力したスペース文字は無視されます。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

(1)　変換指定記号

- d
  符号付きの 10 進数を変換します。対応する引数は整数へのポインタでなければいけません。
- i
  符号付きの 10 進数、8 進数、16 進数の入力を変換します。8 進数は先頭が 0、16 進数は先頭が 0x、0X で始まります。対応する引数は整数へのポインタでなければいけません。
- u
  符号なし 10 進数を変換します。対応する引数は、符号なし整数へのポインタでなければいけません。
- o
  符号付き 8 進数を変換します。対応する引数は整数へのポインタでなければいけません。
- x、X
  符号付き 16 進数を変換します。0AH～0FH には大文字、小文字が使用できます。対応する引数は整数へのポインタでなければいけません。

scanf

入出力関数

解　　説:

- s
ヌル文字'\0'までの文字列を格納します。対応する引数はヌル文字'\0'を含む文字列を格納するのに、充分な大きさを持つ文字配列を指すポインタでなければいけません。最大フィールド幅に達して入力を中止した場合は、それまでの文字にヌル文字を付加した文字列を格納します。
- c
文字を格納します。スペース文字は読み飛ばさずにそのまま格納します。
最大フィールド幅で 2 以上を指定した場合は、複数の文字を格納します。ただし、この場合はヌル文字'\0'は付加しません。
対応する引数は文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。
- p
引数のポインタを出力します。
- [ ]
[ ]内の文字(複数が指定可能)を入力している間、入力文字を格納します。[ ]内の文字以外を入力した場合、格納を中止します。
また[の次に ^ (サーカムフレックス)を指定した場合は、 ^ と]の間で指定した文字以外が入力許可文字となります。
指定した文字を入力した場合、格納を中止します。
対応する引数は自動的に付加するヌル文字'\0'を含む文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。
- n
書式変換で既に読み込まれた文字数を格納します。対応する引数は整数へのポインタでなければいけません。
- e、E、f、g、G
浮動小数点の形式に変換します。修飾子 l を指定したとき、対応する引数は double へのポインタとなります。デフォルトは float へのポインタです。

(2) *(データ格納の抑止指定)

- *が指定されている場合、変換したデータを引数に格納することを抑止します。

(3) 最大フィールド幅

- 正の 10 進整数で最大入力文字数を指定します。1 つの書式変換において、この文字数よりも多くの文字を読み込むことはありません。
- 指定した文字数分の文字を読み込む以前に、スペース文字(関数 isspace()で真となる文字)、又は要求する書式以外の文字を入力した場合は、その時点で読み込みを中止します。

(4) 修飾子

- l の場合、d、i、o、u、x、X、n の変換結果を long int 又は unsigned long int として格納します。また、e、E、f、g、G の変換結果を double として格納します。l を d、i、o、u、x、X、n、e、E、f、g、G 変換以外で指定した場合、指定が無視されます。
- ll の場合、d、i、o、u、x、X、n の変換結果を long long 又は unsigned long long として格納します。ll を d、i、o、u、x、X、n 変換以外で指定した場合、指定が無視されます。
- h の場合、d、i、o、u、x、X の変換結果を short int 又は unsigned short int として格納します。h を d、i、o、u、x、X 変換以外で指定した場合、指定が無視されます。
- L の場合、e、E、f、g、G の変換結果を float として格納します。

setjmp

実行制御関数

機　　能: 関数呼び出し時の環境の退避を行います。

書　　式: #include <setjmp.h>

int setjmp(env);

実現方法: 関数

引　　数: jmp_buf _far env; 環境を退避する領域へのポインタ

戻 り 値: longjmp の引数で与えられた数値を返します。

解　　説: "env"で示される領域に環境の退避を行います。

setlocale

地域化関数

機　　能: プログラムのロケール情報の設定と検索を行います。

書　　式: #include <locale.h>

char _far *setlocale(category,locale);

実現方法: 関数

引　　数: int category; ロケール情報、検索部情報
const char _far *locale; ロケール情報文字列へのポインタ

戻 り 値:
- ロケール情報文字列へのポインタを返します。
- 設定、検索できない場合は NULL を返します。

sin

数学関数

機　　能: サインを計算します。

書　　式: #include <math.h>

double sin(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする"x"のサインを返します。

sinh

数学関数

機　　能: 双曲線サインを計算します。

書　　式: #include <math.h>

double sinh(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の双曲線サインを返します。

sprintf

入出力関数

機　　能: 文字列への書式付き出力を行います。

書　　式: #include <stdio.h>

int sprintf(pointer, format, argument...);

実現方法: 関数

引　　数: char _far *pointer; 格納先のポインタ
const char _far *format; 書式指定文字列のポインタ

戻 り 値: 出力した文字数を返します。

解　　説:
- format の指定に従って argument を文字列に変換し、pointer 以降に格納します。
- format の指定方法は、printf と同様です。

sqrt

数学関数数

機　　能: 数値の平方根を計算します。

書　　式: #include <math.h>

double sqrt(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: 平方根値を返します。

srand

整数算術関数

機　　能: 疑似乱数発生ルーチンにシードを与えます。

書　　式: #include <stdlib.h>

void srand(seed);

実現方法: 関数

引　　数: unsigned int seed; 乱数の系列値

戻 り 値: 戻り値はありません。

解　　説: 引数"seed"を用いて、rand によって与えられる疑似乱数系列を初期化します。

sscanf

入出力関数

機　　能: 文字列からの書式付き入力を行います。

書　　式: #include <stdio.h>

int sscanf(string, format, argument...);

実現方法: 関数

引　　数: const char _far *string; 入力文字列のポインタ
const char _far *format; 書式指定文字列のポインタ

戻 り 値:
- 各引数 argument に格納したデータ数を返します。
- ヌル文字('\0')をデータとして入力した場合、EOF を返します。

解　　説:
- format の指定に従って入力文字を変換し、各引数 argument が示す変数に格納します。
- argument は各変数の far 型ポインタでなければいけません。
- format の指定方法は、scanf と同様です。

strcat

文字列操作関数

機　　能: 文字列の連結を行います。

書　　式:
```
#include <string.h>

char _far * strcat( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	連結先の文字列へのポインタ
const char _far *s2;	連結する文字列へのポインタ

戻 り 値: 連結された文字列領域(s1)へのポインタを返します。

解　　説:
- "s1"で示される文字列の最後に、"s2"で示される文字列を連結します。
- 連結された文字列は、NULL で終了します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strchr

文字列操作関数

機　　能: 文字列より文字の検索を行います。

書　　式:
```
#include <string.h>

char _far * strchr( s, c );
```

実現方法: 関数

引　　数:

const char _far *s;	検索先の文字列へのポインタ
int c;	検索する文字

戻 り 値:
- 文字列"s"中で最初に現れた文字"c"の位置を返します。
- 文字列"s"が文字"c"を含まない場合は NULL を返します。

解　　説:
- "s"で示される領域の先頭より、"c"で示される文字を検索します。
- ?\0'も検索対象とすることができます。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strcmp

文字列操作関数

機　　能: 2 つの文字列の比較を行います。

書　　式:
```
#include <string.h>

int strcmp( s1, s2 );
```

実現方法: マクロ、関数

引　　数:

const char _far *s1;	比較する 1 番目の文字列へのポインタ
const char _far *s2;	比較する 2 番目の文字列へのポインタ

戻 り 値:

- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　1 番目の文字列(s1)の方が大きい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説:

- 通常は、マクロが使用されます。ライブラリ関数を使用したい場合は、#include<string.h> の記述の後に、#undef strcmp と記述してください。
- NULL で終了している 2 つの文字列をバイト単位に比較します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strcoll

文字列操作関数

機　　能: ロケール情報を使用して 2 つの文字列を比較します。

書　　式:
```
#include <string.h>

int strcoll( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far *s1;	比較する 1 番目の文字列へのポインタ
const char _far *s2;	比較する 2 番目の文字列へのポインタ

戻 り 値:

- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　1 番目の文字列(s1)の方が大きい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strcpy

文字列操作関数

機　　能: 文字列の複写を行います。

書　　式:
```
#include <string.h>

char _far * strcpy( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	複写先の文字列へのポインタ
const char _far *s2;	複写元の文字列へのポインタ

戻 り 値: 複写先の文字列へのポインタを返します。

解　　説:
- s1"で示される領域に"s2"で示される(NULL で終了している)文字列を複写します。
- 複写先の文字列は、NULL で終了します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化により関数のインライン展開を行う可能性があります。

strcspn

文字列操作関数

機　　能: 指定外文字列の長さを求めます。

書　　式:
```
#include <string.h>

size_t strcspn( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far *s1;	検索先の文字列へのポインタ
const char _far *s2;	検索する文字列へのポインタ

戻 り 値: 指定外文字列の長さを返します。

解　　説:
- "s1"で示される領域の先頭より、"s2"で示される文字列に含まる文字以外で構成される最初の部分の文字列の長さを求めます。
- '\0'は検索対象とすることができません。

stricmp

文字列操作関数

機　　能: 2 つの文字列の比較を行います(英字はすべて大文字として扱います)。

書　　式: #include <string.h>

int stricmp(s1, s2);

実現方法: 関数

引　　数: char _far *s1; 比較する 1 番目の文字列へのポインタ
char _far *s2; 比較する 2 番目の文字列へのポインタ

戻 り 値:
- 戻り値= =0 2 つの文字列は等しい
- 戻り値>0 1 番目の文字列(s1)の方が大きい
- 戻り値<0 2 番目の文字列(s2)の方が大きい

解　　説: NULL で終了している 2 つの文字列をバイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。

strerror

文字列操作関数

機　　能: エラー番号を文字列に変換します。

書　　式: #include <string.h>

char _far * strerror(errcode);

実現方法: 関数

引　　数: int errcode; エラーコード

戻 り 値: エラーコードに対するメッセージ文字列へのポインタを返します。

解　　説: stderr は静的な配列に対してポインタを返します。

strlen

文字列操作関数

機　　能: 文字列の長さを求めます。

書　　式: #include <string.h>

size_t strlen(s);

実現方法: 関数

引　　数: const char _far *s; 長さを求める文字列へのポインタ

戻 り 値: 文字列の長さを返します。

解　　説: "s"で示される文字列(NULL まで)の長さを求めます。

strncat

文字列操作関数

機　　能: 文字列の(n 文字)連結を行います。

書　　式: #include <string.h>

char _far * strncat(s1, s2, n);

実現方法: 関数

引　　数:

char _far *s1;	連結先の文字列へのポインタ
const char _far *s2;	連結する文字列へのポインタ
size_t n;	連結する文字数

戻 り 値: 連結された文字列領域へのポインタを返します。

解　　説:

- "s1"で示される文字列の最後に、"n"で指定された文字数の文字を"s2"で示される文字列より連結します。
- 連結された文字列は、NULL で終了します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strncmp

文字列操作関数

機　　能: 2 つの文字列の(n 文字)比較を行います。

書　　式:
```
#include <string.h>

int strncmp( s1, s2, n );
```

実現方法: 関数

引　　数:

const char _far *s1;	比較する 1 番目の文字列へのポインタ
const char _far *s2;	比較する 2 番目の文字列へのポインタ
size_t n;	比較する文字数

戻 り 値:
- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　1 番目の文字列(s1)の方が大きい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説:
- NULL で終了している 2 つの文字列を"n"文字分バイト単位に比較します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strncpy

文字列操作関数

機　　能: 文字列の複写を行います。

書　　式:
```
#include <string.h>

char _far * strncpy( s1, s2, n );
```

実現方法: 関数

引　　数:

char _far *s1;	複写先の文字列へのポインタ
const char _far *s2;	複写元の文字列へのポインタ
size_t n;	複写する文字数

戻 り 値: 複写先の文字列へのポインタを返します。

解　　説:
- "s1"で示される領域に、"n"で指定された文字数の文字を、"s2"で示される文字列より複写します。ただし、この時"n"で指定された文字数より多い部分は複写されず、その後に'\0'を付加することも行いません。逆に、"s2"の示す文字列の長さが"n"文字より短い場合は、複写された文字列の後に"n"文字になるまで'\0'が付加されます。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strnicmp

文字列操作関数

機　　能: 2 つの文字列の(n 文字)比較を行います(英字はすべて大文字として扱います)。

書　　式:
```
#include <string.h>

int strnicmp( s1, s2, n );
```

実現方法: 関数

引　　数:

char _far *s1;	比較する 1 番目の文字列へのポインタ
char _far *s2;	比較する 2 番目の文字列へのポインタ
size_t n;	比較する文字数

戻 り 値:
- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　2 つの文字列は等しい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説:
- NULL で終了している 2 つの文字列を"n"文字分バイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strpbrk

文字列操作関数

機　　能: 文字列より指定文字の検索を行います。

書　　式:
```
#include <string.h>

char _far * strpbrk( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far *s1;	検索先の文字列へのポインタ
const char _far *s2;	検索する文字の文字列へのポインタ

戻 り 値:
- 見つかった場合、見つかった位置(ポインタ)を返します。
- 見つからない場合、NULL を返します。

解　　説:
- "s1"で示される領域より、"s2"で示される文字列中の文字が含まれているか検索します。
- '\0'は検索対象とすることができません。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strrchr

文字列操作関数

機　　能: 文字列より文字の検索を行います。

書　　式:
```
#include <string.h>

char _far * strrchr( s, c );
```

実現方法: 関数

引　　数:

const char _far *s;	検索先の文字列へのポインタ
int c;	検索する文字

戻 り 値:
- 文字列"s"中で最後に現れた文字"c"の位置を返します。
- 文字列"s"が文字"c"を含まない場合は NULL を返します。

解　　説:
- "s"で示される領域の末尾より、"c"で示される文字を検索します。
- '\0'も検索対象とすることができます。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strspn

文字列操作関数

機　　能: 指定文字列の長さを求めます。

書　　式:
```
#include <string.h>

size_t strspn( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far ＊s1;	検索先の文字列へのポインタ
const char _far ＊s2;	検索する文字列へのポインタ

戻 り 値:
- 指定文字列の長さを返します。

解　　説:
- " s1"で示される領域の先頭より、"s2"で示される文字列に含まる文字のみで構成される最初の部分の文字列の長さを求めます。
- '¥0'は検索対象とすることができません。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strstr

文字列操作関数

機　　能: 指定文字列の検索を行います。

書　　式:
```
#include <string.h>

char _far * strstr( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far *s1;	検索先の文字列へのポインタ
const char _far *s2;	検索する文字の文字列へのポインタ

戻 り 値:
- 見つかった場合、見つかった位置(ポインタ)を返します。
- 見つからない場合、NULL を返します。

解　　説:
- "s1"で示される領域の先頭より、"s2"で示される文字列が最初に現れた位置(ポインタ)を返します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtod

文字列操作関数

機　　能: 文字列を倍精度浮動小数に変換します。

書　　式:
```
#include <stdlib.h>

double strtod( s,endptr );
```

実現方法: 関数

引　　数:

const char _far *s;	変換文字列
char _far * _far *endptr;	変換されなかった残りの文字列へのポインタ

戻 り 値:

● 戻り値 == 0L	数を構成しません。
● 戻り値 != 0L	構成した数を double 型で返します。

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtok

文字列操作関数

機　　能: 文字列の切り出しを行います。

書　　式:
```
#include <string.h>

char _far * strtok( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	切り出し先の文字列へのポインタ
const char _far *s2;	区切り文字へのポインタ

戻 り 値:
- 見つかった場合、切り出したトークンへのポインタを返します。
- 見つからない場合、NULL を返します。

解　　説:
- 最初の呼び出しでは、最初の字句の先頭文字へのポインタを返します。このとき返される文字の最後には、NULL 文字が書き込まれます。その後の呼び出し(" s 1 "が NULL の場合)では、順次、次の字句が返されます。"s1"に字句がなくなると NULL を返します。
- オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtol

文字列数値変換関数

機　　能: 文字列を long 型数値に変換します。

書　　式:
```
#include <stdlib.h>

long strtol( s,endptr,base );
```

実現方法: 関数

引　　数:

const char _far *s;	変換文字列
char _far * _far *endptr;	変換されなかった残りの文字列へのポインタ
int base;	読み込む数値の基数(0～36)。0 の場合は整数定数の形式を読み込みます。

戻 り 値:

● 戻り値 == 0L	数を構成しません。
● 戻り値 != 0L	構成した数を long 型で返します。

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtoul

文字列数値変換関数

機　　能: 文字列を unsigned long 型数値に変換します。

書　　式:
```
#include <stdlib.h>

unsigned long strtoul( s,endptr,base );
```

実現方法: 関数

引　　数:

const char _far *s;	変換文字列
char _far * _far *endptr;	変換されなかった残りの文字列へのポインタ
int base;	読み込む数値の基数(0～36)。0 の場合は整数定数の形式を読み込みます。

戻 り 値:
- 戻り値 == 0L　数を構成しません。
- 戻り値 != 0L　構成した数を unsigned long 型で返します。

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strxfrm

文字列操作関数

機　　能: ロケール情報を使用して文字列を変換します。

書　　式:
```
#include <string.h>

size_t strxfrm( s1,s2,n );
```

実現方法: 関数

引　　数:

char _far *s1;	変換結果文字列格納領域へのポインタ
const char _far *s2;	変換元文字列へのポインタ
size_t n;	変換バイト数

戻 り 値: 変換されたバイト数を返します。

解　　説: オプション-O[3～5], -OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

T

tan

数学関数

機　　能: タンジェントを計算します。

書　　式: #include <math.h>

double tan(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする"x"のタンジェントを返します。

tanh

数学関数

機　　能: 双曲線タンジェントを計算します。

書　　式: #include <math.h>

double tanh(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする"x"の双曲線タンジェントを返します。

tolower

文字操作関数

機　　能: 英大文字を英小文字に変換します。

書　　式: #include <ctype.h>

int tolower(c);

実現方法: マクロ

引　　数: int c; 変換する文字

戻 り 値:
- 引数が英大文字の場合、英小文字を返します。
- それ以外は渡した引数を返します。

解　　説: 引数の英大文字を英小文字に変換します。

toupper

文字操作関数

機　　能: 英小文字を英大文字に変換します。

書　　式: int toupper(c);

実現方法: マクロ

引　　数: int c; 変換する文字

戻 り 値:
- 引数が英小文字の場合、大文字を返します。
- それ以外は渡した引数を返します。

解　　説: 引数の英小文字を英大文字に変換します。

U

ungetc

入出力関数

機　　能: ストリームへ 1 文字戻します。

書　　式:
```
#include <stdio.h>

int ungetc( c, stream );
```

実現方法: マクロ

引　　数:

int c;	戻す文字
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常な場合は、戻した 1 文字を返します。
- ストリームが書き込みモード、エラー、EOF の場合、又は EOF を戻そうとした場合、EOF を返します。

解　　説:
- ストリームに 1 文字を戻します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

V

vfprintf

入出力関数

機　　能: フォーマットを指定してテキストを指定ストリームに書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( stream, format, ap );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列へのポインタ
va_list ap;	引数リストの先頭へのポインタ

戻 り 値: 出力した文字数を返します。

解　　説:
- フォーマットを指定してテキストを指定ストリームに書き込みます。
- 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

vprintf

入出力関数

機　　能: フォーマットを指定してテキストを stdout に書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vprintf( format, ap );
```

実現方法: 関数

引　　数:

const char _far *format;	書式指定文字列へのポインタ
valist ap;	引数リストの先頭へのポインタ

戻 り 値: 出力した文字数を返します

解　　説:
- フォーマットを指定してテキストを stdout に書き込みます。
- 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

vsprintf

入出力関数

機　　能: フォーマットを指定してテキストを指定バッファに書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( s, format, ap );
```

実現方法: 関数

引　　数:

char _far *s;	格納先へのポインタ
const char _far *format;	書式指定文字列へのポインタ
va_list ap;	引数リストの先頭へのポインタ

戻 り 値: 出力した文字数を返します。

解　　説: 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

W

wcstombs

多バイト文字/多バイト文字列操作関数

機　　能: ワイド文字列をマルチバイト文字列に変換します。

書　　式:
```
#include <stdlib.h>

size_t _wcstombs( s, wcs, n );
```

実現方法: 関数

引　　数:

char _far *s;	変換マルチバイト文字列格納領域へのポインタ
const wchar_t _far *wcs;	ワイド文字列先頭へのポインタ
size_t n;	格納マルチバイト文字数

戻 り 値:
- 正しく変換された場合は格納したマルチバイト文字数を返します。
- そうでない場合は-1 を返します。

wctomb

多バイト文字/多バイト文字列操作関数

機　　能: ワイド文字をマルチバイト文字に変換します。

書　　式:
```
#include <stdlib.h>

int wctomb( s,wchar );
```

実現方法: 関数

引　　数:

char _far *s;	変換マルチバイト文字格納領域へのポインタ
wchar_t wchar;	ワイド文字マルチバイト文字に含めたバイト数を返します。

戻 り 値:
- マルチバイト文字に含めたバイト数を返します。
- 対応するマルチバイト文字が存在しない場合は-1 を返します。
- ワイド文字が 0 の場合は 0 を返します。

E.2.4 標準関数ライブラリの使用に関する注意事項

a. 標準ヘッダファイルに関する注意事項

標準関数ライブラリ中の関数を使用する時は、必ず指定された標準ヘッダファイルをインクルードしてください。インクルードしない場合、引数及び戻り値の整合がとれませんのでプログラムが正常に動作しないことがあります。

b. 標準ライブラリ関数の最適化に関する注意事項

最適化オプション-O[3～5]、-OR, -OR_MAX(-ORM), -OS, -OS_MAX(-OSM)のいずれかを指定した場合、標準関数に関する最適化を行ないます。この最適化は-Ono_stdlib を指定することにより、抑止することができます。ユーザー関数として、標準ライブラリ関数と同名の関数を使用する場合は、この最適化を抑止してください。

(1) 関数のインライン埋め込み

関数strcpy及びmemcpyについて、【表E.13】の条件を満たす時、関数のインライン埋め込みを行ないます。

表E.13 標準ライブラリ関数に対する最適化条件

関数名	最適化条件	記述例
strcpy	第 1 引数 : near ポインタ 第 2 引数 : 文字列定数	strcpy(str, "sample");
memcpy	第 1 引数 : near ポインタ 第 2 引数 : far ポインタ 第 3 引数 : 定数	memcpy(str ,"sample", 6); memcpy(str , fp, 6);

E.3 標準入出力関数ライブラリのカスタマイズ方法

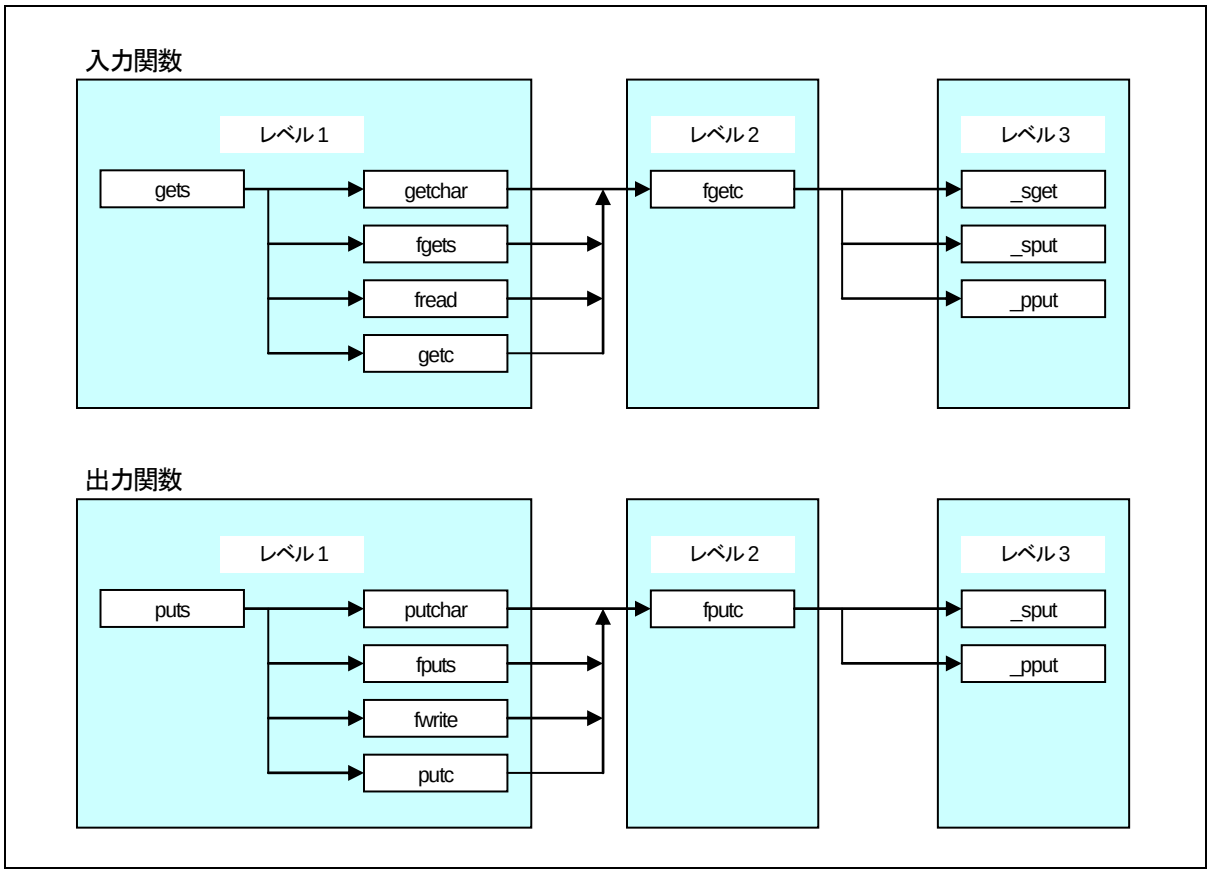
入出力関数として scanf、printf 等の高機能な関数ライブラリを用意しています。これらの関数は一般的に高水準入出力関数と呼ばれます。高水準入出力関数は、ハードウェア環境に依存した低水準入出力関数の組み合わせで実現しています。

アプリケーションプログラムにおいて、組み込むシステムのハードウェアによって入出力関数を変更する場合があります。このような場合、製品に添付しています標準関数ライブラリのソースファイルを変更することで、対処できます。

E.3.1 入出力関数の構成

入出力関数は、【図E.1】に示すようにレベル 1 の関数から下位の関数(レベル 2→レベル 3)を呼び出すことで機能を実現しています。例えば、fgetsはレベル 2 のfgetcを呼び出し、更にfgetcはレベル 3 の関数を呼び出します。

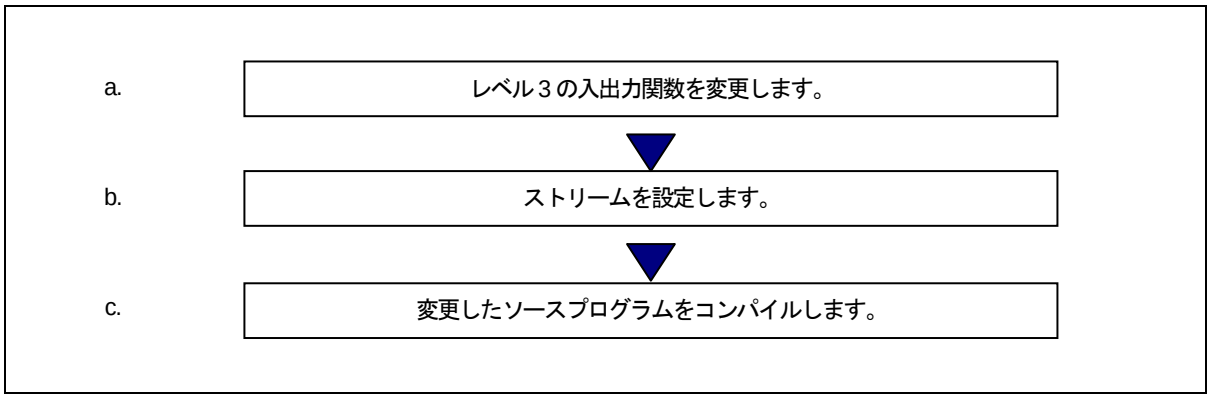
マイコンのハードウェア(入出力ポート)に依存しているのは最下位のレベル 3 の関数のみです。入出力関数をアプリケーションプログラムで使用する場合、必要に応じてレベル 3 の関数のソースファイルを書き換えることで、システムに順応した関数に変更できます。



図E.1 入出力関数の呼び出し関係図

E.3.2 入出力関数の変更手順

入出力関数を組み込むシステムに合わせて変更する方法の概略を【図E.2】に示します。



図E.2 入出力関数の変更手順例

a. レベル 3 の入出力関数の変更方法

レベル 3 の入出力関数は、M16C シリーズ、R8C ファミリの入出力ポートに対して 1 バイトの入出力を行う関数です。レベル 3 の入出力関数は、シリアル通信回路(UART)に対して入出力を行う_sget、_sput と、セントロニクス仕様の通信回路に対して入出力を行う_pput があります。

(1) 回路の諸設定

- プロセッサ動作モード : マイクロプロセッサモード
- クロック発信周波数 : 20MHz
- 外部バス幅 : 16 ビット

(2) シリアル通信の初期設定

- UART1 を使用
- ボーレート : 9600bps
- データ長 : 8 ビット
- パリティ : なし
- ストップビット : 2 ビット

※これらのシリアル通信の初期設定は_init 関数中で設定されています。

レベル 3 の入出力関数は、C言語で記述されたソースファイルdevice.cに記述しています。レベル 3 の関数の仕様を【表E.14】に示します。

表E.14 レベル 3 の関数の仕様

入力関数	引数	戻り値(int 型)
_sget	なし	正常に入力できた場合は入力した文字を返します。 エラーの場合は EOF を返します。

出力関数	引数(int 型)	戻り値(int 型)
_sput _pput	出力する文字	正常に出力できた場合は 1 を返します。 エラーの場合は EOF を返します。

シリアル通信は、M16C シリーズ、R8C ファミリが持つ 2 本の UART の内 UART1 に設定しています。device.c では、条件コンパイルコマンドで UART0 を選択できるように記述しています。選択方法は、

- UART0 を使用する場合
  #define __UART0__1 を device.c ファイルの先頭で記述するか、あるいは
- UART0 を使用する場合
  -D__UART0__をコンパイル時に指定します。

2 本の UART を使用する場合は、以下の手順で変更します。

(1) device.c ファイルの先頭に記述している条件コンパイルの記述を削除します。
(2) #pragma ADDRESS で定義されている UART0 の特殊レジスタ名を UART1 と異なった変数に書き換えます。
(3) レベル 3 の関数_sget、_sput を UART0 用にそれぞれ複製し、_sget0、_sput0 等の異なった関数名に書き換えます。
(4) speed 関数も UART0 用に複製し、speed0 等の異なった関数名に書き換えます。

以上で device.c ファイルの変更は終了します。

次に、入出力関数の初期設定を行っている_init 関数を変更し、ストリームの設定を変更します。このストリームの設定方法は、次節で説明します。

b. ストリームの設定

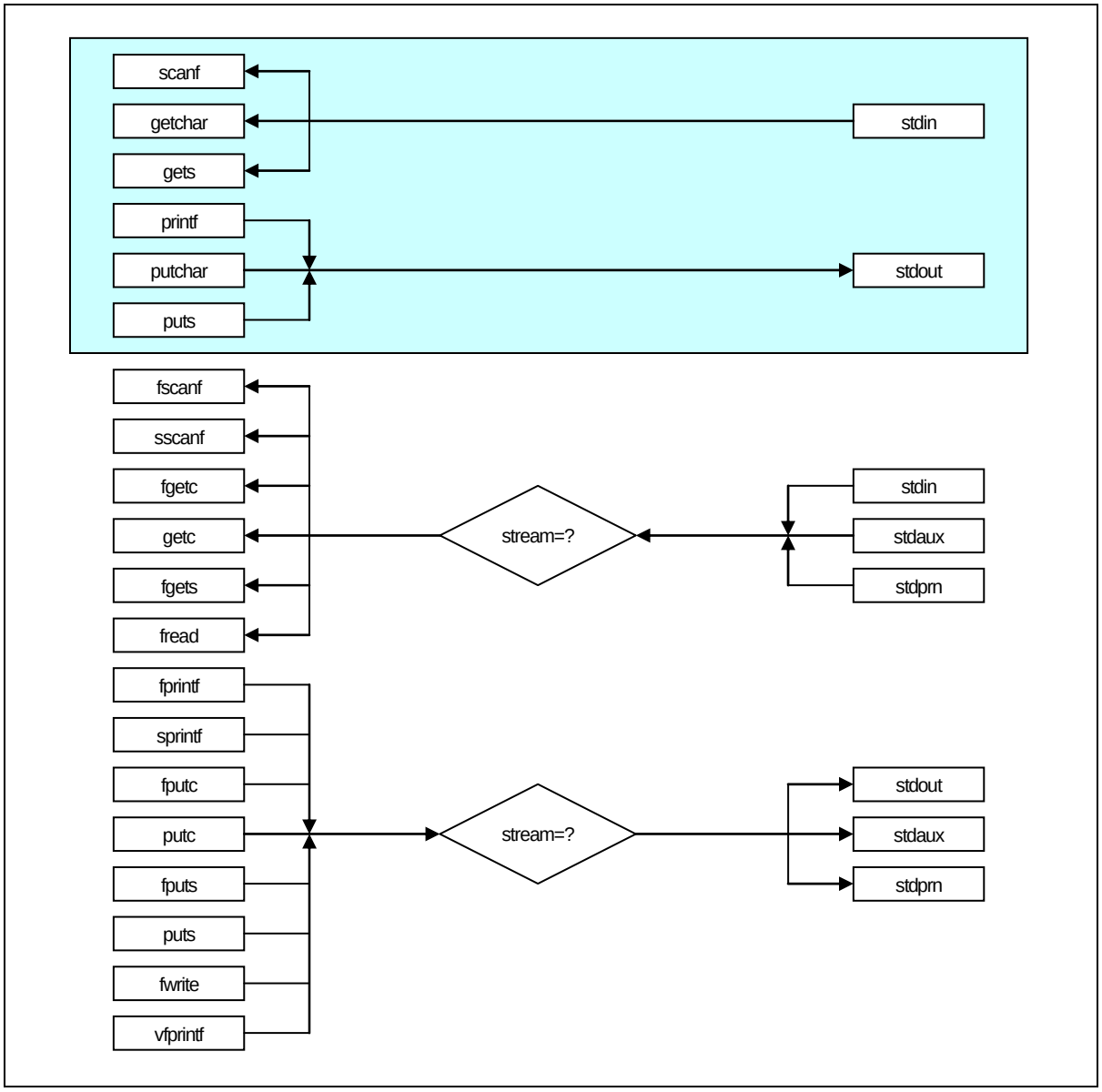
本コンパイラの標準ライブラリは stdin、stdout、stderr、stdaux、stdprn の 5 種類のストリーム情報を外部構造体として持っています。この外部構造体は標準ヘッダファイル stdio.h 内で定義されており、各ストリームのモード情報(入力ストリームか出力ストリームを表すフラグ)やステータス情報(エラー又は EOF を表すフラグ)等を管理しています。

表E.15 ストリーム情報一覧表

ストリーム情報	名称
stdin	標準入力
stdout	標準出力
stderr	標準エラー出力（stderr は stdout に定義されています。）
stdaux	標準補助入出力
stdprn	標準プリンタ出力

本コンパイラの標準ライブラリ関数中、【図E.3】の網掛部に示す関数の対応するストリームは標準入力(stdin)又は標準出力(stdout)に固定しています。これらの関数に関しては、ストリームを変更することはできません。なお、stderrはstdoutに#defineで定義されています。

ストリームは、fgetc、fputc 等の引数としてストリームへのポインタを指定できる関数のみ変更することができます。



図E.3 関数とストリームの相互関係図

【図E.4】にstdio.h内のストリーム定義部を示します。

```
/***************************************************************************
*
* standard I/O header file
                :
            (省略)
                :
typedef struct _iobuf {
            char        _buff;                          /* Store buffer for ungetc */          ← (1)
            int         _cnt;                           /* Strings number in _buff(1 or 0) */  ← (2)
            int         _flag;                          /* Flag */                             ← (3)
            int         _mod;                           /* Mode */                             ← (4)
            int         (*_func_in)(void);      /* Pointer to one byte input function */       ← (5)
            int         (*_func_out)(int);      /* Pointer to one byte output function */      ← (6)
} FILE;
#define     _IOBUF_DEF
                :
            (省略)
                :
extern FILE _iob[];
#define     stdin       (&_iob[0])  /* Fundamental input */
#define     stdout      (&_iob[1])  /* Fundamental output */
#define     stdaux      (&_iob[2])  /* Fundamental auxiliary input output */
#define     stdprn      (&_iob[3])  /* Fundamental printer output */

#define     stderr      stdout                  /* NC no-support */

/*********************************************************
*
*********************************************************/
#define     _IOREAD  1          /* Read only flag */
#define     _IOWRT   2          /* Write only flag */
#define     _IOEOF   4          /* End of file flag */
#define     _IOERR   8          /* Error flag */
#define     _IORW    16         /* Read and write flag */
#define     _NFILE   4          /* Stream number */
#define     _TEXT    1          /* Text mode flag */
#define     _BIN     2          /* Binary mode flag */

            (以降省略)
                :
```

図E.4　stdio.h 内のストリーム定義部 (stdio.h)

【図E.4】に示すファイル構造体の要素を以下に説明します。説中の(1)～(6)は、【図E.4】中の(1)～(6)に対応しています。

(1)　char _buff

関数 scanf、fscanf では入力の際に 1 文字分の先読みを行っています。
先読みした文字が不要な場合は関数 ungetc を呼び出して、先読みした文字をこの変数に格納します。
入力関数はこの変数にデータが存在する場合はこのデータを入力データとします。

(2)　int _cnt

_buff のデータ数を格納します(0 もしくは 1)。

(3) int _flag

読み込み専用フラグ(_IOREAD)、書き込み専用フラグ(_IOWRT)、読み書き両用フラグ(_IORW)、エンドオブファイルフラグ(_IOEOF)、エラーフラグ(_IOERR)のフラグを格納します。

- _IOREAD、_IOWRT、_IORW

  ストリームの動作モードを指定するフラグです。これらのフラグは、ストリームの初期化部分で設定します。

- _IOEOF、_IOERR

  入出力関数内で EOF、エラーの発生に応じて設定されます。

(4) int _mod

テキストモード(_TEXT)、バイナリモード(_BIN)を表すフラグを格納します。

- テキストモード

  入出力データのエコーバック、文字変換を行います。エコーバック、文字変換の詳細については、関数 fgetc、fputc のソースプログラム(fgetc.c、fputc.c)を参照ください。

- バイナリモード

  入出力データを無変換で扱います。これらのフラグはストリームの初期化部分で設定します。

(5) int (*_func_in)(void)

ストリームが読み込みモード(_IOREAD)又は読み書き両用モード(_IORW)の場合、レベル 3 入力関数のポインタを格納します。それ以外の場合は NULL ポインタを格納します。

この情報をもとに、レベル 2 入力関数はレベル 3 入力関数を間接呼び出しで呼び出します。

(6) int (*_func_out)(void)

ストリームが書き込みモード(_IOWRT)の場合、レベル 3 出力関数のポインタを格納します。また、ストリームが入力可能な場合(_IOREAD 又は_IORW)で、かつテキストモードの場合、エコーバックするためのレベル 3 出力関数のポインタを格納します。それ以外の場合は、NULL ポインタを格納します。

この情報をもとに、レベル 2 入力関数はレベル 3 入力関数を間接呼び出しで呼び出します。

ストリームの初期化では char _buff 以外のすべての要素に値を設定してください。

関数_init でストリームの初期化を行っています。関数_init は、スタートアッププログラム ncrt0.a30 または resetprg.c から呼び出されています。

【図E.5】に_init関数のソースプログラムを示します。

```
#include <stdio.h>

FILE _iob[4];

void _init( void );

void _init( void )
{
    stdin->_cnt = 0;
    stdout->_cnt = 0;
    stdaux->_cnt = 0;
    stdprn->_cnt = 0;
    stdin->_flag = _IOREAD;
    stdout->_flag = _IOWRT;
    stdaux->_flag = _IORW;
    stdprn->_flag = _IOWRT;

    stdin->_mod = _TEXT;
    stdout->_mod = _TEXT;
    stdaux->_mod = _BIN;
    stdprn->_mod = _TEXT;

    stdin->_func_in = _sget;
    stdout->_func_in = NULL;
    stdaux->_func_in = _sget;
    stdprn->_func_in = NULL;

    stdin->_func_out = _sput;
    stdout->_func_out = _sput;
    stdaux->_func_out = _sput;
    stdprn->_func_out = _pput;

#ifdef __UART0__
    speed(_96, _B8, _PN, _S2);
#else /* UART1 : default */
    speed(_96, _B8, _PN, _S2);
#endif
}
```

図E.5　init 関数のソースファイル (init.c)

M16C シリーズ、R8C ファミリの 2 本の UART を使用するシステムでは、_init 関数を以下の手順で変更します。前節では、device.c ソースファイル中で UART0 用の関数を仮に_sget0、_sput0、speed0 と設定しました。

(1) UART0 用のストリームには、標準補助入出力(stdaux)を使用します。
(2) 標準補助入出力に対するフラグ(_flag)、モード(_mod)をシステムに合わせて設定します。
(3) 標準補助入出力に対するレベル 3 関数のポインタを設定します。
(4) speed 関数に対する条件コンパイルコマンドを削除し、UART0 用の speed0 関数に書き換えます。

以上の設定で、2 本のUARTが使用できます。ただし、標準入出力のストリームを使用する関数はUART0 が使用する標準補助入出力に対して使用することができません。したがって、関数の引数にストリームを記述できる関数で使用してください。【図E.6】に_init関数の変更例を示します。

```
void _init ( void )
{
                :
        (省略)
                :
    stdaux->_flag = _IORW;              ← (2) 読み書き両用モードに設定
                :
        (省略)
                :
    stdaux->_mod = _TEXT;               ← (2) テキストモードに設定
                :
        (省略)
                :
    stdaux->_func_in = _sget0;          ← (3) UART0 用のレベル 3 の入力関数を指定
                :
        (省略)
                :
    stdaux->_func_out = _sput0;         ← (3) UART0 用のレベル 3 の出力関数を指定
                :
        (省略)
                :
    speed(_96, _B8, _PN, _S2);          ← (4) UART0 用の speed 関数を指定
}
```

図E.6 init 関数の変更例

c. 変更したソースプログラムの組み込み

変更した関数のソースファイルのオブジェクトファイルをリンク時に指定してください。この場合、リンク時に指定した関数が有効となり、ライブラリファイル内の同一名の関数は組み込まれません。

例を【図E.7】に、示します。

```
% nc30  -c  -g  -osample  ncrt0.a30  device.obj  init.obj  sample.c<RET>

※この例は、device.c と init.c を変更したときの記述方法です。
```

図E.7 変更したソースプログラムを直接リンクする方法

E.4 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E.16 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E.16 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。

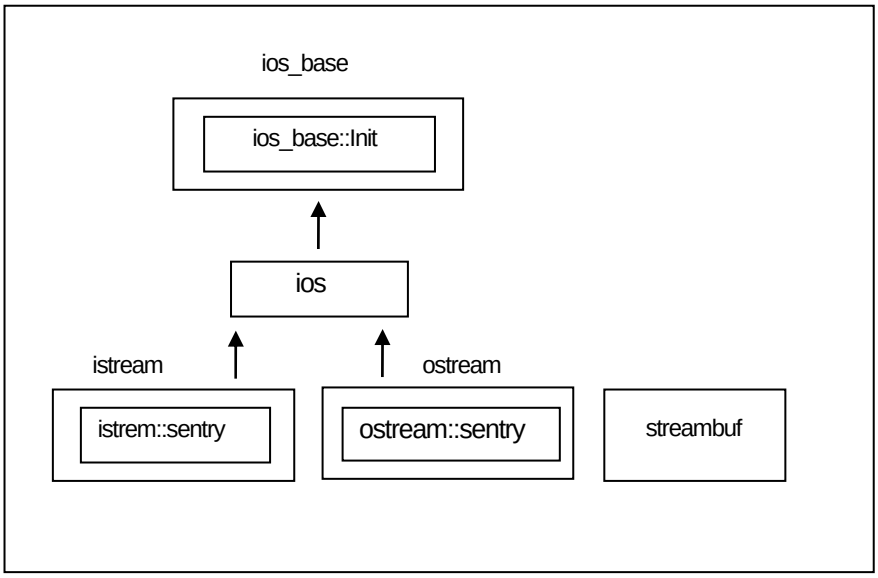
- <iostream>

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。



ファイルに対するストリーム操作が必要な場合は、ファイルに対するバッファ操作クラスを実装する必要があります。その作成にあたっては、mystrbuf が参考になります。

現状の mystrbuf は、un-bufferd 実装ですので、以下のインターフェースを適切に実装する必要があります。

open(), close(), setvbuf(), seek(), ftell()

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	long 型で定義された型です。
	int_type	long 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。
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 _far &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 _far & 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 _far * sbptr)	
	void init(streambuf _far * sbptr)	初期設定を行います。
	virtual ~ios()	デストラクタです。
	operator void _far *() 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 _far * tie() const	ostream オブジェクトへのポインタ(tiestr)を参照します。
	ostream _far * tie(ostream _far * tstrptr)	tstrptr を ostream オブジェクトへのポインタ(tiestr)に設定します。
	streambuf _far * rdbuf() const	streambuf オブジェクトへのポインタ(sb)を参照します。
	streambuf _far * rdbuf(streambuf _far * sbptr)	sbptr を streambuf オブジェクトへのポインタ(sb)に設定します。
	ios _far & copyfmt (const ios _far & 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 _far * 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 _far * 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 _far *() 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 _far * 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 _far * ios::tie(ostream _far * 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 _far * 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 _far * ios::rdbuf(streambuf _far * 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 _far & copyfmt (const ios _far & 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 _far & showbase(ios_base _far & str)	基数表示接頭辞モードに設定します。
	ios_base _far & noshowbase(ios_base _far & str)	基数表示接頭辞モードをクリアします。
	ios_base _far & showpoint (ios_base _far & str)	小数点生成モードに設定します。
	ios_base _far & noshowpoint (ios_base _far & str)	小数点生成モードをクリアします。
	ios_base _far & showpos(ios_base _far & str)	+記号生成モードに設定します。
	ios_base _far & noshowpos(ios_base _far & str)	+記号生成モードをクリアします。
	ios_base _far & skipws(ios_base _far & str)	空白読み飛ばしモードに設定します。
	ios_base _far & noskipws(ios_base _far & str)	空白読み飛ばしモードをクリアします。
	ios_base _far & uppercase(ios_base _far & str)	大文字変換モードに設定します。
	ios_base _far & nouppercase(ios_base _far & str)	大文字変換モードをクリアします。
	ios_base _far & internal(ios_base _far & str)	内部補充モードに設定します。
	ios_base _far & left(ios_base _far & str)	左側補充モードに設定します。
	ios_base _far & right(ios_base _far & str)	右側補充モードに設定します。
	ios_base _far & dec(ios_base _far & str)	10 進モードに設定します。
	ios_base _far & hex(ios_base _far & str)	16 進モードに設定します。
	ios_base _far & oct(ios_base _far & str)	8 進モードに設定します。
	ios_base _far & fixed(ios_base _far & str)	固定小数点モードに設定します。
	ios_base _far & scientific(ios_base _far & str)	科学表記法モードに設定します。
	ios_base _far & boolalpha (ios_base _far & str)	bool 型の値の出力を true または false にします。戻り値は str です。
	ios_base _far & noboolalpha (ios_base _far & str)	bool 型の値の出力を 1 または 0 にします。戻り値は str です。

ios_base _far & showbase(ios_base _far & 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 _far & noshowbase(ios_base _far & 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 _far & showpoint(ios_base _far & 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 _far & noshowpoint(ios_base _far & 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 _far & showpos(ios_base _far & 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 _far & noshowpos(ios_base _far & 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 _far & skipws(ios_base _far & 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 _far & noskipws(ios_base _far & 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 _far & uppercase(ios_base _far & 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 _far & nouppercase(ios_base _far & 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 _far & internal(ios_base _far & 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 _far & left(ios_base _far & 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 _far & right(ios_base _far & 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 _far & dec(ios_base _far & 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 _far & hex(ios_base _far & 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 _far & oct(ios_base _far & 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 _far & fixed(ios_base _far & 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 _far & scientific(ios_base _far & 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。

ios_base _far & boolalpha (ios_base _far & str)

bool 型の値の出力を true または false にします。

戻り値は str です。

ios_base _far & noboolalpha (ios_base _far & str)

bool 型の値の出力を 1 または 0 にします。

戻り値は 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 _far * _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 _far * _far & _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 _far * _far & _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 _far * _far * begptr, char _far * _far * curptr, long _far * cntptr, ong _far * lenptr, char _far * 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 _far * pubsetbuf(char _far * s, streamsize n)	ストリーム入出力用のバッファを確保します。この関数では setbuf(s,n)[*1] を呼び出します。
	pos_type pubseekoff(off_type off, ios _base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out)	way で指定された方法で入出力ストリームの読み書き位置を移動させます。この関数では seekoff(off,way,which)[*1] を呼び出します。
	pos_type pubseekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out)	ストリームの先頭から現在の位置までのオフセットを求めます。この関数では seekpos(sp,which)[*1] を呼び出します。
	int pubsync()	出力ストリームをフラッシュします。この関数では sync()[*1] を呼び出します。
	streamsize in_avail()	入力ストリームの最後尾から現在位置までのオフセットを求めます。

種別	定義名	説明
関数	int_type snextc()	次の一文字を読み込みます。
	int_type sbumpc()	一文字読み込みポインタを次に設定します。
	int_type sgetc()	一文字読み込みます。
	int sgetn(char _far * 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 _far * s, streamsize n)	s の指す n 個の文字を挿入します。
	char _far * eback() const	入力ストリームの先頭ポインタを求めます。
	char _far * gptr() const	入力ストリームの次ポインタを求めます。
	char _far * 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 _far * gbeg, char _far * gnext, char _far * gend)	入力ストリームの各ポインタを代入します。
	char _far * pbase() const	出力ストリームの先頭ポインタを求めます。
	char _far * pptr() const	出力ストリームの次ポインタを求めます。
	char _far * 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 _far * pbeg, char _far * pend)	出力ストリームの各ポインタを設定します。
	virtual streambuf _far * setbuf(char _far * 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 _far * 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 _far * 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 _far * streambuf::pubsetbuf(char _far * 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 _far * 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 _far * 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 _far * 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 _far * 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 _far * 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 _far * gbeg, char _far * gnext, char _far * 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 _far * 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 _far * 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 _far * 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 _far * pbeg, char _far * 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 _far * streambuf::setbuf(char _far * 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 _far * 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 _far * 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 _far & 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 _far & 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 _far * str, unsigned int dig, int mode)	str を dig が示す基数で変換します。
	istream(streambuf _far * sb)	コンストラクタです。
	virtual ~istream()	デストラクタです。
	istream _far & operator>>(bool _far & n)	抽出した文字を n に格納します。
	istream _far & operator>>(short _far & n)	
	istream _far & operator>>(unsigned short _far & n)	
	istream _far & operator>>(int _far & n)	
	istream _far & operator>>(unsigned int _far & n)	
	istream _far & operator>>(long _far & n)	
	istream _far & operator>>(unsigned long _far & n)	
	istream _far & operator>>(long long _far & n)	
	istream _far & operator>>(unsigned long long _far & n)	
	istream _far & operator>>(float _far & n)	
	istream _far & operator>>(double _far & n)	
	istream _far & operator>>(long double _far & n)	
	istream _far & operator>>(void _far * _far & p)	void を指すポインタに変換して p に格納します。
	istream _far & operator>>(streambuf _far * 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 _far & get(char _far & c)	文字を抽出し c に格納します。
	istream _far & get(signed char _far & c)	
	istream _far & get(unsigned char _far & c)	
	istream _far & get(char _far * 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 _far & get(signed char _far * s, streamsize n)	
	istream _far & get(unsigned char _far * s, streamsize n)	
	istream _far & get(char _far * 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 _far & get(signed char _far * s, streamsize n, char delim)	
	istream _far & get(unsigned char _far * s, streamsize n, char delim)	

種別	定義名	説明
関数	istream _far & get(streambuf _far & sb)	文字列を抽出し、sb の指す記憶領域に格納します。
	istream _far & get(streambuf _far & sb, char delim)	文字列を抽出し、sb の指す記憶領域に格納します。途中で文字'delim'を検出したら、入力を終了します。
	istream _far & getline(char _far * 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 _far & getline(signed char _far * s, streamsize n)	
	istream _far & getline(unsigned char _far * s, streamsize n)	
	istream _far & getline(char _far * 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 _far & getline(signed char _far * s, streamsize n, char delim)	
	istream _far & getline(unsigned char _far * s, streamsize n, char delim)	
	istream _far & 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 _far & read(char _far * 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 _far & read(signed char _far * s, streamsize n)	
	istream _far & read(unsigned char _far * s, streamsize n)	
	streamsize readsome(char _far * 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 _far * s, streamsize n)	
	streamsize readsome(unsigned char _far * s, streamsize n)	
	istream _far & putback(char c)	文字を入力ストリームに戻します。
	istream _far & 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 _far & 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 _far & 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 _far * 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 _far * 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 _far & istream::operator>>(bool _far & n)
istream _far & istream::operator>>(short _far & n)
istream _far & istream::operator>>(unsigned short _far & n)
istream _far & istream::operator>>(int _far & n)
istream _far & istream::operator>>(unsigned int _far & n)
istream _far & istream::operator>>(long _far & n)
istream _far & istream::operator>>(unsigned long _far & n)
istream _far & istream::operator>>(long long _far & n)
istream _far & istream::operator>>(unsigned long long _far & n)
istream _far & istream::operator>>(float _far & n)
istream _far & istream::operator>>(double _far & n)
istream _far & istream::operator>>(long double _far & 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 _far & istream::operator>>(void _far * _far & p)

抽出した文字を void _far *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 _far & istream::operator>>(streambuf _far * 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 _far & istream::get(char _far & c)
istream _far & istream::get(signed char _far & c)
istream _far & istream::get(unsigned char _far & 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 _far & istream::get(char _far * s, streamsize n)
istream _far & istream::get(signed char _far * s, streamsize n)
istream _far & istream::get(unsigned char _far * 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 _far & istream::get(char _far * s, streamsize n, char delim)
istream _far & istream::get(signed char _far * s, streamsize n, char delim)
istream _far & istream::get(unsigned char _far * 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 _far & istream::get(streambuf _far & 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 _far & istream::get(streambuf _far & 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 _far & istream::getline(char _far * s, streamsize n)
istream _far & istream::getline(signed char _far * s, streamsize n)
istream _far & istream::getline(unsigned char _far * 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 _far & istream::getline(char _far * s, streamsize n, char delim)
istream _far & istream::getline(signed char _far * s, streamsize n, char delim)
istream _far & istream::getline(unsigned char _far * 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 _far & 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 _far & istream::read(char _far * s, streamsize n)
istream _far & istream::read(signed char _far * s, streamsize n)
istream _far & istream::read(unsigned char _far * 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 _far * s, streamsize n)
streamsize istream::readsome(signed char _far * s, streamsize n)
streamsize istream::readsome(unsigned char _far * 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & ws(istream _far & is)	**空白類を読み飛ばします。**

istream _far & ws(istream _far & 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 _far & operator>>(istream _far & in, char _far * s)	**文字列を抽出し、s の指す記憶領域に格納します。**
	istream _far & operator>>(istream _far & in, signed char _far * s)	
	istream _far & operator>>(istream _far & in, unsigned char _far * s)	
	istream _far & operator>>(istream _far & in, char _far & c)	**文字を抽出し、c に格納します。**
	istream _far & operator>>(istream _far & in, singed char _far & c)	
	istream _far & operator>>(istream _far & in, unsigned char _far & c)	

istream _far & operator>>(istream _far & in, char _far * s)
istream _far & operator>>(istream _far & in, signed char _far * s)
istream _far & operator>>(istream _far & in, unsigned char _far * 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 _far & operator>>(istream _far & in, char _far & c)
istream _far & operator>>(istream _far & in, singed char _far & c)
istream _far & operator>>(istream _far & in, unsigned char _far & 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 _far & 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 _far & 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 _far * sbptr)	コンストラクタです。
	virtual ~ostream()	デストラクタです。
	ostream _far & operator<<(bool n)	n を出力ストリームに挿入します。
	ostream _far & operator<<(short n)	
	ostream _far & operator<<(unsigned short n)	
	ostream _far & operator<<(int n)	
	ostream _far & operator<<(unsigned int n)	
	ostream _far & operator<<(long n)	
	ostream _far & operator<<(unsigned long n)	
	ostream _far & operator<<(long long n)	
	ostream _far & operator<<(unsigned long long n)	
	ostream _far & operator<<(float n)	
	ostream _far & operator<<(double n)	
	ostream _far & operator<<(long double n)	
	ostream _far & operator<<(void _far * n)	
	ostream _far & operator<<(streambuf _far * sbptr)	sbptr の出力列を出力ストリームに挿入します。
	ostream _far & put(char c)	文字 c を出力ストリームに挿入します。
	ostream _far & write(const char _far * 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 _far & write(const signed char _far * s, streamsize n)	
	ostream _far & write(const unsigned char _far * s, streamsize n)	
	ostream _far & 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 _far & 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 _far & 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 _far * 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 _far & ostream::operator<<(bool n)
ostream _far & ostream::operator<<(short n)
ostream _far & ostream::operator<<(unsigned short n)
ostream _far & ostream::operator<<(int n)
ostream _far & ostream::operator<<(unsigned int n)
ostream _far & ostream::operator<<(long n)
ostream _far & ostream::operator<<(unsigned long n)
ostream _far & ostream::operator<<(long long n)
ostream _far & ostream::operator<<(unsigned long long n)
ostream _far & ostream::operator<<(float n)
ostream _far & ostream::operator<<(double n)
ostream _far & ostream::operator<<(long double n)
ostream _far & ostream::operator<<(void _far * 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 _far & ostream::operator<<(streambuf _far * 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 _far & 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 _far & ostream::write(const char _far * s, streamsize n)
ostream _far & ostream::write(const signed char _far * s, streamsize n)
ostream _far & ostream::write(const unsigned char _far * 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 _far & 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 _far & 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 _far & 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 _far & endl(ostream _far & os)	改行を挿入し、出力ストリームをフラッシュします。
	ostream _far & ends(ostream _far & os)	ヌルコードを挿入します。
	ostream _far & flush(ostream _far & os)	出力ストリームをフラッシュします。

ostream _far & endl(ostream _far & 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 _far & ends(ostream _far & 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 _far & flush(ostream _far & 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 _far & operator<<(ostream _far & os, char s)	s を出力ストリームに挿入します。
	ostream _far & operator<<(ostream _far & os, signed char s)	
	ostream _far & operator<<(ostream _far & os, unsigned char s)	
	ostream _far & operator<<(ostream _far & os, const char _far * s)	
	ostream _far & operator<<(ostream _far & os, const singed char _far * s)	
	ostream _far & operator<<(ostream _far & os, const unsigned char _far * s)	

ostream _far & operator<<(ostream _far & os, char s)
ostream _far & operator<<(ostream _far & os, signed char s)
ostream _far & operator<<(ostream _far & os, unsigned char s)
ostream _far & operator<<(ostream _far & os, const char _far * s)
ostream _far & operator<<(ostream _far & os, const singed char _far * s)
ostream _far & operator<<(ostream _far & os, const unsigned char _far * 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。

(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 _far * operator new(size_t size)	size 分の領域を確保します。
	void _far * operator new[ ](size_t size)	size 分の配列領域を確保します。
	void _far * operator new(size_t size, void _far * 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 _far * operator new[ ](size_t size, void _far * 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 _far * ptr)	領域を解放します。
	void operator delete[ ](void _far * 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 _far * 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 _far * 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 _far * operator new(size_t size, void _far * 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 _far * operator new[ ](size_t size, void _far * 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 _far * 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 _far * 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 _far & 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 _far & 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 _far & operator+=(float rhs)	rhs を実数部に加算し、和を*this に格納します。
	float_complex _far & operator-=(float rhs)	rhs を実数部から減算し、差を*this に格納します。
	float_complex _far & operator*=(float rhs)	rhs を乗算し、積を*this に格納します。
	float_complex _far & operator/=(float rhs)	rhs で除算し、商を*this に格納します。
	float_complex _far & operator=(const float_complex _far & rhs)	rhs をコピーします。
	float_complex _far & operator+=(const float_complex _far & rhs)	rhs を加算し、和を*this に格納します。
	float_complex _far & operator-=(const float_complex _far & rhs)	rhs を減算し、差を*this に格納します。
	float_complex _far & operator*=(const float_complex _far & rhs)	rhs を乗算し、積を*this に格納します。
	float_complex _far & operator/=(const float_complex _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & float_complex::operator=(const float_complex _far & rhs)
rhs をコピーします。
リターン値は*this です。

float_complex _far & float_complex::operator+=(const float_complex _far & 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 _far & float_complex::operator-=(const float_complex _far & 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 _far & float_complex::operator*=(const float_complex _far & 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 _far & float_complex::operator/=(const float_complex _far & 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 _far & lhs)	lhs の単項+演算を行います。
	float_complex operator+(const float_complex _far & lhs, const float_complex _far & rhs)	lhs と rhs を加算し、和を lhs に格納します。
	float_complex operator+(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator+(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator-(const float_complex _far & lhs)	lhs の単項−演算を行います。
	float_complex operator-(const float_complex _far & lhs, const float_complex _far & rhs)	lhs から rhs を減算し、差を lhs に格納します。
	float_complex operator-(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator-(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator*(const float_complex _far & lhs, const float_complex _far & rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	float_complex operator*(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator*(const float _far & lhs, const float_complex _far & rhs)	
	float_complex operator/(const float_complex _far & lhs, const float_complex _far & rhs)	lhs を rhs で除算し、商を lhs に格納します。
	float_complex operator/(const float_complex _far & lhs, const float _far & rhs)	
	float_complex operator/(const float _far & lhs, const float_complex _far & rhs)	

種別	定義名	説明
関数	bool operator==(const float_complex _far & lhs, const float_complex _far & 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 _far & lhs, const float _far & rhs)	
	bool operator==(const float _far & lhs, const float_complex _far & rhs)	
	bool operator!=(const float_complex _far & lhs, const float_complex _far & 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 _far & lhs, const float _far & rhs)	
	bool operator!=(const float _far & lhs, const float_complex _far & rhs)	
	istream _far & operator>>(istream _far & is, float_complex _far & 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 _far & operator<<(ostream _far & os, float_complex _far & 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 _far & x)	実数部を求めます。
	float imag(const float_complex _far & x)	虚数部を求めます。
	float abs(const float_complex _far & x)	絶対値を求めます。
	float arg(const float_complex _far & x)	位相角度を求めます。
	float norm(const float_complex _far & x)	２乗の絶対値を求めます。
	float_complex conj(const float_complex _far & x)	共役複素数を求めます。
	float_complex polar(const float _far & rho, const float _far & 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 _far & x)	複素余弦を求めます。
	float_complex cosh(const float_complex _far & x)	複素双曲余弦を求めます。
	float_complex exp(const float_complex _far & x)	指数関数を求めます。
	float_complex log(const float_complex _far & x)	自然対数を求めます。
	float_complex log10(const float_complex _far & x)	常用対数を求めます。

種別	定義名	説明
関数	float_complex pow(const float_complex _far & x, int y)	x の y 乗を求めます。
	float_complex pow(const float_complex _far & x, const float _far & y)	
	float_complex pow(const float_complex _far & x, const float_complex _far & y)	
	float_complex pow(const float _far & x, const float_complex _far & y)	
	float_complex sin(const float_complex _far & x)	複素正弦を求めます。
	float_complex sinh(const float_complex _far & x)	複素双曲正弦を求めます。
	float_complex sqrt(const float_complex _far & 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 _far & x)	複素正接を求めます。
	float_complex tanh(const float_complex _far & x)	複素双曲正接を求めます。

float_complex operator+(const float_complex _far & 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 _far & lhs, const float_complex _far & rhs)
float_complex operator+(const float_complex _far & lhs, const float _far & rhs)
float_complex operator+(const float _far & lhs, const float_complex _far & 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 _far & 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 _far & lhs, const float_complex _far & rhs)
float_complex operator-(const float_complex _far & lhs, const float _far & rhs)
float_complex operator-(const float _far & lhs, const float_complex _far & 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 _far & lhs, const float_complex _far & rhs)
float_complex operator*(const float_complex _far & lhs, const float _far & rhs)
float_complex operator*(const float _far & lhs, const float_complex _far & 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 _far & lhs, const float_complex _far & rhs)
float_complex operator/(const float_complex _far & lhs, const float _far & rhs)
float_complex operator/(const float _far & lhs, const float_complex _far & 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 _far & lhs, const float_complex _far & rhs)
bool operator==(const float_complex _far & lhs, const float _far & rhs)
bool operator==(const float _far & lhs, const float_complex _far & 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 _far & lhs, const float_complex _far & rhs)
bool operator!=(const float_complex _far & lhs, const float _far & rhs)
bool operator!=(const float _far & lhs, const float_complex _far & 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 _far & operator>>(istream _far & is, float_complex _far & 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 _far & operator<<(ostream _far & os, const float_complex _far & 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 _far & x)
実数部を求めます。
リターン値は x.real()です。

float imag(const float_complex _far & x)
虚数部を求めます。
リターン値は x.imag()です。

float abs(const float_complex _far & 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 _far & 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 _far & 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 _far & 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 _far & rho, const float _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & x, int y)
float_complex pow(const float_complex _far & x, const float _far & y)
float_complex pow(const float_complex _far & x, const float_complex _far & y)
float_complex pow(const float _far & x, const float_complex _far & 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 _far & x,const float_complex _far & y)の場合 : exp(y*logf(x))

上記以外 : exp(y*log(x))

float_complex sin(const float_complex _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far &)	
	double real() const	実数部を求めます。
	double imag() const	虚数部を求めます。
	double_complex _far & 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 _far & operator+=(double rhs)	rhs を実数部に加算し、和を*this に格納します。
	double_complex _far & operator-=(double rhs)	rhs を実数部から減算し、差を*this に格納します。
	double_complex _far & operator*=(double rhs)	rhs を乗算し、積を*this に格納します。
	double_complex _far & operator/=(double rhs)	rhs で除算し、商を*this に格納します。
	double_complex _far & operator=(const double_complex _far & rhs)	rhs をコピーします。
	double_complex _far & operator+=(const double_complex _far & rhs)	rhs を加算し、和を*this に格納します。
	double_complex _far & operator-=(const double_complex _far & rhs)	rhs を減算し、差を*this に格納します。
	double_complex _far & operator*=(const double_complex _far & rhs)	rhs を乗算し、積を*this に格納します。
	double_complex _far & operator/=(const double_complex _far & 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 _far &)

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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & double_complex::operator=(const double_complex _far & rhs)

rhs をコピーします。

リターン値は*this です。

double_complex _far & double_complex::operator+=(const double_complex _far & 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 _far & double_complex::operator-=(const double_complex _far & 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 _far & double_complex::operator*=(const double_complex _far & 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 _far & double_complex::operator/=(const double_complex _far & 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 _far & lhs)	lhs の単項+演算を行います。
	double_complex operator+(const double_complex _far & lhs, const double_complex _far & rhs)	lhs と rhs を加算し、和を lhs に格納します。
	double_complex operator+(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator+(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator-(const double_complex _far & lhs)	lhs の単項−演算を行います。
	double_complex operator-(const double_complex _far & lhs, const double_complex _far & rhs)	lhs から rhs を減算し、差を lhs に格納します。
	double_complex operator-(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator-(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator*(const double_complex _far & lhs, const double_complex _far & rhs)	lhs と rhs を乗算し、積を lhs に格納します。
	double_complex operator*(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator*(const double _far & lhs, const double_complex _far & rhs)	
	double_complex operator/(const double_complex _far & lhs, const double_complex _far & rhs)	lhs を rhs で除算し、商を lhs に格納します。
	double_complex operator/(const double_complex _far & lhs, const double _far & rhs)	
	double_complex operator/(const double _far & lhs, const double_complex _far & rhs)	

種別	定義名	説明
関数	bool operator==(const double_complex _far & lhs, const double_complex _far & 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 _far & lhs, const double _far & rhs)	
	bool operator==(const double _far & lhs, const double_complex _far & rhs)	
	bool operator!=(const double_complex _far & lhs, const double_complex _far & 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 _far & lhs, const double _far & rhs)	
	bool operator!=(const double _far & lhs, const double_complex _far & rhs)	
	istream _far & operator>>(istream _far & is, double_complex _far & 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 _far & operator<<(ostream _far & os, const double_complex _far & 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 _far & x)	実数部を求めます。
	double imag(const double_complex _far & x)	虚数部を求めます。
	double abs(const double _complex _far & x)	絶対値を求めます。
	double arg(const double_complex _far & x)	位相角度を求めます。
	double norm(const double_complex _far & x)	２乗の絶対値を求めます。
	double_complex conj(const double_complex _far & x)	共役複素数を求めます。
	double_complex polar(const double _far & rho, const double _far & 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 _far & x)	複素余弦を求めます。
	double_complex cosh(const double_complex _far & x)	複素双曲余弦を求めます。
	double_complex exp(const double_complex _far &)	指数関数を求めます。
	double_complex log(const double_complex _far & x)	自然対数を求めます。
	double_complex log10(const double_complex _far & x)	常用対数を求めます。

種別	定義名	説明
関数	double_complex pow(const double_complex _far & x, int y)	x の y 乗を求めます。
	double_complex pow(const double_complex _far & x, const double _far & y)	
	double_complex pow(const double_complex _far & x, const double_complex _far & y)	
	double_complex pow(const double _far & x, const double_complex _far & y)	
	double_complex sin(const double_complex _far & x)	複素正弦を求めます。
	double_complex sinh(const double_complex _far & x)	複素双曲正弦を求めます。
	double_complex sqrt(const double_complex _far & 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 _far & x)	複素正接を求めます。
	double_complex tanh(const double_complex _far & x)	複素双曲正接を求めます。

double_complex operator+(const double_complex _far & 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 _far & lhs, const double_complex _far & rhs)
double_complex operator+(const double_complex _far & lhs, const double _far & rhs)
double_complex operator+(const double _far & lhs, const double_complex _far & 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 _far & 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 _far & lhs, const double_complex _far & rhs)
double_complex operator-(const double_complex _far & lhs, const double _far & rhs)
double_complex operator-(const double _far & lhs, const double_complex _far & 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 _far & lhs, const double_complex _far & rhs)
double_complex operator*(const double_complex _far & lhs, const double _far & rhs)
double_complex operator*(const double _far & lhs, const double_complex _far & 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 _far & lhs, const double_complex _far & rhs)
double_complex operator/(const double_complex _far & lhs, const double _far & rhs)
double_complex operator/(const double _far & lhs, const double_complex _far & 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 _far & lhs, const double_complex _far & rhs)
bool operator==(const double_complex _far & lhs, const double _far & rhs)
bool operator==(const double _far & lhs, const double_complex _far & 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 _far & lhs, const double_complex _far & rhs)
bool operator!=(const double_complex _far & lhs, const double _far & rhs)
bool operator!=(const double _far & lhs, const double_complex _far & 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 _far & operator>>(istream _far & is, double_complex _far & 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 _far & operator<<(ostream _far & os, const double_complex _far & 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 _far & x)
実数部を求めます。
リターン値は x.real()です。

double imag(const double_complex _far & x)
虚数部を求めます。
リターン値は x.imag()です。

double abs(const double_complex _far & 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 _far & 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 _far & 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 _far & 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 _far & rho, const double _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & x, int y)
double_complex pow(const double_complex _far & x, const double _far & y)
double_complex pow(const double_complex _far & x, const double_complex _far & y)
double_complex pow(const double _far & x, const double_complex _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far & 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 _far *型です。
	const_iterator	const char _far *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 _far & str, size_t pos = 0, size_t n = npos)	
	string::string(const char _far * str, size_t n)	
	string::string(const char _far * str)	
	string::string(size_t n, char c)	
	~string()	デストラクタです。
	string _far & operator=(const string _far & str)	str を代入します。
	string _far & operator=(const char _far * str)	
	string _far & 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 _far & operator[](size_t pos) const	s_ptr[pos]を参照します。
	char _far & operator[](size_t pos)	
	const char _far & at(size_t pos) const	
	char _far & at(size_t pos)	
	string _far & operator+=(const string _far & str)	str の文字列を追加します。
	string _far & operator+=(const char _far * str)	
	string _far & operator+=(char c)	c の文字を追加します。
	string _far & append(const string _far & str)	str の文字列を追加します。
	string _far & append(const char _far * str)	
	string _far & append(const string _far & 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 _far & append (const char _far * str, size_t n)	文字列 str の n 文字分を追加します。
	string _far & append(size_t n, char c)	n 個の文字 c を追加します。
	string _far & assign(const string _far & str)	str の文字列を代入します。
	string _far & assign(const char _far * str)	
	string _far & assign(const string _far & 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 _far & assign(const char _far * str, size_t n)	文字列 str の n 文字分を代入します。
	string _far & assign(size_t n, char c)	n 個の文字 c を代入します。
	string _far & insert(size_t pos1, const string _far & str)	位置 pos1 に str の文字列を挿入します。
	string _far & insert(size_t pos1, const string _far & 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 _far & insert(size_t pos, const char _far * str, size_t n)	pos の位置に文字列 str を n 文字分挿入します。
	string _far & insert(size_t pos, const char _far * str)	pos の位置に文字列 str を挿入します。
	string _far & 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 _far & 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。

種別	定義名	説明
関数	string _far & replace(size_t pos1, size_t n1, const string _far & 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 _far & replace(size_t pos1, size_t n1, const char _far * str)	
	string _far & replace(size_t pos1, size_t n1, const string _far & 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 _far & replace(size_t pos, size_t n1, const char _far * 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 _far & 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 _far & replace(iterator i1, iterator i2, const string _far & 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 _far & replace(iterator i1, iterator i2, const char _far * str)	
	string _far & replace(iterator i1, iterator i2, const char _far * 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 _far & 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 _far * 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 _far & str)	str の文字列と交換します。
	const char _far * 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 _far * data() const	

種別	定義名	説明
関数	size_t find(const string _far & 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 _far * str,, size_t pos = 0) const	
	size_t find(const char _far * 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 _far & 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 _far * str,, size_t pos = npos) const	
	size_t rfind(const char _far * 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 _far & 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 _far * str , size_t pos = 0) const	
	size_t find_first_of(const char _far * 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 _far & 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 _far * str , size_t pos = npos) const	
	size_t find_last_of(const char _far * 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。

種別	定義名	説明
関数	size_t find_first_not_of(const string _far & 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 _far * str , size_t pos = 0) const	
	size_t find_first_not_of(const char _far * 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 _far & 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 _far * str , size_t pos = npos) const	
	size_t find_last_not_of(const char _far * 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 _far & 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 _far & 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 _far & 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 _far * str) const	str と比較します。
	int compare(size_t pos1, size_t n1, const char _far * 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 _far & 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 _far * 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 _far * str)

以下に設定します。

s_ptr =

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 _far & string::operator=(const string _far & str)

str のデータを代入します。

リターン値は*this です。

string _far & string::operator=(const char _far * 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 _far & 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 _far & string::operator[](size_t pos) const
char _far & string::operator[](size_t pos)
const char _far & string::at(size_t pos) const
char _far & 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 _far & string::operator+=(const string _far & 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 _far & string::operator+=(const char _far * 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 _far & 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 _far & string::append(const string _far & str)
string _far & string::append(const char _far * 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 _far & string::append(const string _far & 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 _far & string::append(const char _far * 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 _far & 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 _far & string::assign(const string _far & str)
string _far & string::assign(const char _far * str)

str の文字列を代入します。
リターン値は*this です。

string _far & string::assign(const string _far & 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 _far & string::assign(const char _far * 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 _far & 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 _far & string::insert(size_t pos1, const string _far & 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 _far & string::insert(size_t pos1, const string _far & 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 _far & string::insert(size_t pos, const char _far * 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 _far & string::insert(size_t pos, const char _far * 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 _far & 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 _far & 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 _far & string::replace(size_t pos1, size_t n1, const string _far & str)

string _far & string::replace(size_t pos1, size_t n1, const char _far * 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 _far & string::replace(size_t pos1, size_t n1, const string _far & 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 _far & string::replace(size_t pos, size_t n1, const char _far * 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 _far & 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 _far & string::replace(iterator i1, iterator i2, const string _far & str)

string _far & string::replace(iterator i1, iterator i2, const char _far * 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 _far & string::replace(iterator i1, iterator i2, const char _far * 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 _far & 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* _far 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 _far & str)

str の文字列と交換します。

const char _far * string::c_str() const

const char _far * 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 _far & str, size_t pos = 0) const
size_t string::find(const char _far * 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 _far * 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 _far & str, size_t pos = npos) const
size_t string::rfind(const char _far * 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 _far * 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 _far & str, size_t pos = 0) const
size_t string::find_first_of(const char _far * 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 _far * 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 _far & str, size_t pos = npos) const
size_t string::find_last_of(const char _far * 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 _far * 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 _far & str, size_t pos = 0) const
size_t string::find_first_not_of(const char _far * 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 _far * 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 _far & str, size_t pos = npos) const
size_t string::find_last_not_of(const char _far * 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 _far * 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 _far & 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 _far & 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 _far & 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 _far * 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 _far * 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 _far & lhs, const string _far & 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 _far * lhs, const string _far & rhs)	
	string operator+(char lhs, const string _far & rhs)	
	string operator+(const string _far & lhs, const char _far * rhs)	
	string operator+(const string _far & lhs, char rhs)	
	bool operator==(const string _far & lhs, const string _far & rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator==(const char _far * lhs, const string _far & rhs)	
	bool operator==(const string _far & lhs, const char _far * rhs)	
	bool operator!=(const string _far & lhs, const string _far & rhs)	lhs の文字列と rhs の文字列を比較します。
	bool operator!=(const char _far * lhs, const string _far & rhs)	
	bool operator!=(const string _far & lhs, const char _far * rhs)	
	bool operator<(const string _far & lhs, const string _far & rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<(const char _far * lhs, const string _far & rhs)	
	bool operator<(const string _far & lhs, const char _far * rhs)	
	bool operator>(const string _far & lhs, const string _far & rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>(const char _far * lhs, const string _far & rhs)	
	bool operator>(const string _far & lhs, const char _far * rhs)	
	bool operator<=(const string _far & lhs, const string _far & rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator<=(const char _far * lhs, const string _far & rhs)	
	bool operator<=(const string _far & lhs, const char _far * rhs)	
	bool operator>=(const string _far & lhs, const string _far & rhs)	lhs の文字列長と rhs の文字列長を比較します。
	bool operator>=(const char _far * lhs, const string _far & rhs)	
	bool operator>=(const string _far & lhs, const char _far * rhs)	
	void swap(string _far & lhs, string _far & rhs)	lhs の文字列と rhs の文字列を交換します。
	istream _far & operator>>(istream _far & is, string _far & str)	文字列を str に抽出します。
	ostream _far & operator<<(ostream _far & os, const string _far & str)	文字列を挿入します。
	istream _far & getline(istream _far & is, string _far & 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 _far & getline(istream _far & is, string _far & 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 _far & lhs, const string _far & rhs)
string operator+(const char _far * lhs, const string _far & rhs)
string operator+(char lhs, const string _far & rhs)
string operator+(const string _far & lhs, const char _far * rhs)
string operator+(const string _far & 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 _far & lhs, const string _far & rhs)
bool operator==(const char _far * lhs, const string _far & rhs)
bool operator==(const string _far & lhs, const char _far * 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 _far & lhs, const string _far & rhs)
bool operator!=(const char _far * lhs, const string _far & rhs)
bool operator!=(const string _far & lhs, const char _far * 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 _far & lhs, const string _far & rhs)
bool operator<(const char _far * lhs, const string _far & rhs)
bool operator<(const string _far & lhs, const char _far * 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 _far & lhs, const string _far & rhs)
bool operator>(const char _far * lhs, const string _far & rhs)
bool operator>(const string _far & lhs, const char _far * 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 _far & lhs, const string _far & rhs)
bool operator<=(const char _far * lhs, const string _far & rhs)
bool operator<=(const string _far & lhs, const char _far * 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 _far & lhs, const string _far & rhs)
bool operator>=(const char _far * lhs, const string _far & rhs)
bool operator>=(const string _far & lhs, const char _far * 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 _far & lhs, string _far & rhs)

lhs の文字列と rhs の文字列を交換します。

istream& operator>>(istream _far & is, string _far & str)

文字列を str に抽出します。
リターン値は is です。

ostream& operator<<(ostream _far & os, const string _far & str)

文字列を挿入します。
リターン値は os です。

istream& getline(istream _far & is, string _far & 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 _far & is, string _far & 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。

付録F コンパイラのエラーメッセージ

F.1 エラー形式とエラーレベル

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
　　対策内容
　　エラー内容（その 2）
　　対策内容（その 2）

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**	**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。**

F.2 メッセージ一覧

C1001 (W) Ignore option '-?'
　　使用できないコンパイルオプションオプション-?を使用しています。
　　正しいコンパイルオプション指定してください。

C1002 (W) Ignore option 'オプション' is no effect when compiling C++
　　C++ コンパイルにおいて効果のないオプションが指定されています。
　　指定されたオプションは、C++ コンパイル時には外してください。

C1003 (W) Ignore option 'オプション' is no effect when compiling C
　　C コンパイルにおいて効果のないオプションが指定されています。
　　指定されたオプションは、C コンパイル時には外してください。

C1004 (W) Nothing to compile, assemble or link
　　コンパイル、アセンブル、またはリンク対象の入力ファイルが指定されていません。
　　コマンドラインに、コンパイル、アセンブル、またはリンク対象の入力ファイルを指定してください。

C1005 (W) Can't specified 'オプションA' with 'オプションB' option. 'オプションA' was ignored.
指定されたオプションAは オプションB と同時に指定できないオプションです。
指定されたオプションAを オプションB と同時に指定しないようにしてください。

C1511 (W) #pragma プラグマ名 & HANDLER both specified
1つの関数に#pragma プラグマ名と#pragma HANDLERの両方を指定しています。
#pragma プラグマ名と#pragma HANDLERは、排他的に指定してください。

C1511 (W) #pragma プラグマ名 & INTERRUPT both specified
1つの関数に#pragmaプラグマ名と#pragma INTERRUPTの両方を指定しています。
#pragmaプラグマ名と#pragma INTERRUPTは、排他的に指定してください。

C1511 (W) #pragma プラグマ名 & TASK both specified
1つの関数に#pragmaプラグマ名と#pragmaTASKの両方を指定しています。
#pragma プラグマ名と#pragma TASKは、排他的に指定してください。

C1512 (W) #pragma プラグマ名 format error
#pragma プラグマ名 の記述に誤りがあります。
マニュアルの文法に従って記述してください。

C1512 (W) #pragma プラグマ名 format error, ignored
#pragma プラグマ名 の記述に誤りがあります。この行を無視します。
マニュアルの文法に従って記述してください。

C1513 (W) #pragma JSRA illegal location, ignored
関数のスコープ内に#pragma JSRAを記述しています。
#pragma JSRAは、関数外に記述してください。

C1513 (W) #pragma JSRW illegal location, ignored
関数のスコープ内に#pragma JSRWを記述しています。
#pragma JSRWは、関数外に記述してください。

C1514 (W) #pragma プラグマ名 not function, ignored
#pragma プラグマ名 において関数ではない名前を記述しています。
#pragmaの作用対象には関数名を書いてください。

C1515 (W) #pragma プラグマ名's function must be pre-declared, ignored
#pragma プラグマ名 で指定した関数が宣言されていません。
#pragma プラグマ名 で指定する関数は、予め関数原型宣言を行ってください。

C1516 (W) #pragma プラグマ名's function must be prototyped,ignored
#pragma プラグマ名 で指定した関数が、関数原型宣言されていません。
#pragma プラグマ名 で指定する関数は、予め関数原型宣言を行ってください。

C1517 (W) #pragma プラグマ名's function return type invalid,ignored
#pragma プラグマ名 で指定された関数の返却値の型が不当です。
返却値の型は、struct、union、double 型以外を指定してください。

C1518 (W) #pragma プラグマ名 variable initialized, initialization ignored
#pragma プラグマ名 で指定した変数を初期化しています。初期化を無視します。
#pragma プラグマ名 か、初期化式のどちらかを削除してください。

C1527 (W) #pragma プラグマ名 variable must be far pointer for 変数名, ignored
#pragma プラグマ名 宣言された変数は、far ポインタである必要があります。#pragma 宣言を無視します。
#pragma を有効にする場合は、変数を far ポインタとして宣言してください。

C1528 (W) #pragma プラグマ名 variable must be unsigned int for 変数名,ignored
#pragma プラグマ名 宣言された変数は、unsigned int 型である必要があります。#pragma 宣言を無視します。
#pragma を有効にする場合は、変数を unsigned int 型として宣言してください。

C1529 (W) #pragma プラグマ名, register conflict
#pragma プラグマ名 宣言 で同一レジスタを複数回使用しています。
１つのレジスタを使用するのは１回だけにしてください。

C1530 (W) #pragma プラグマ名, unknown register name used
#pragma プラグマ名 宣言 においてレジスタ指定の文字列が間違っています。
マニュアルの文法に従って記述してください。

C1531 (W) #pragma プラグマ名 variable must be pre-declared,ignored
#pragma プラグマ名 宣言された変数は、型宣言がされている必要があります。
#pragma よりも先に変数を宣言してください。

C1532 (W) #pragma ASM line too long, then cut
#pragma ASM で記述できる一行の文字数 1024 バイトを越えています。
1024 バイト以下で記述してください。

C1533 (W) #pragma directive conflict
１つの関数に対して、同時指定不可能な#pragma を複数個指定しています。
同時指定不可能な#pragma を宣言から外してください。

C1534 (W) #pragma for non-function type can not use for function
関数には指定できない#pragma を、関数に指定しています。
#pragma を削除してください。

C1536 (W) #pragma PARAMETER function's address used
#pragma PARAMETER で指定された関数のアドレスを参照しています。
参照しないでください。

C1537 (W) #pragma SECTADDRESS's attribute format error, ignored
#pragma SECTADDRESS のセクション属性文字列が間違っています。
正しいセクション属性名を書いてください

C1538 (W) #pragma プラグマ名 unknown switch, ignored
#pragma プラグマ名で不正なスイッチを記述しています。
正しい switch を指定してください。

C1538 (W) #pragma unknown switch, ignored
#pragma に対して不正なスイッチを指定しています。#pragma 宣言を無視します。
正しいスイッチを指定してください。

C1539 (W) #pragma 'プラグマ名' is aleady setted to '値'
既に同じ'プラグマ名'で、'値'設定済みです。
1つの変数・関数に対して、同じプラグマで異なる値を複数回設定しないでください。

C1541 (W) invalid #pragma プラグマ名
#pragma EQU の記述に誤りがあります。この行を無視します。
マニュアルの文法に従って記述してください。

C1542 (W) invalid #pragma SECTION, unknown section base name
#pragma SECTION においてセクション名に誤りがあります。指定できるセクション名は data、bss、program、rom です。この行を無視します。
マニュアルの文法に従って記述してください。

C1543 (W) Kanji in #pragma ADDRESS
#pragma ADDRESS の記述に漢字コードが含まれています。この行を無視します。
この宣言では漢字コードを使わないでください。

C1543 (W) Kanji in #pragma BITADDRESS
#pragma BITADDRESS の記述に漢字コードが含まれています。この行を無視します。
この宣言では漢字コードを使わないでください。

C1544 (W) this return type can not use for #pragma プラグマ名,#pragma is ignored
この型を返す関数には'プラグマ名'を指定できません。
#pragma を指定しないか、関数の型を変えてください。

C1545 (W) this variable's type is not match for 'レジスタ名', #pragma 'プラグマ名' is ignored
引数の型とレジスタサイズが一致しません。
引数の型とレジスタサイズを一致させてください。

C1546 (W) unknown pragma pragma-指示 used
サポートされていない #pragma を記述しています。
#pragma の内容を確認してください。このウォーニングは コンパイルオプション-Wunknown_pragma (-WUP)、および -Wall 指定時のみ表示されます。

C1547 (W) OS version specifier conflict with another #pragma
#pragma で RTOS のバージョンを混在させることはできません。
RTOS のバージョンは一貫させてください。

C1548 (W) cannot use SPECIAL PAGE number 値,#pragma is ignored
この値はスペシャルページで使用可能な範囲から外れています。
スペシャルページで使用可能な値を設定してください。

C1549 (W) function "関数名" in #pragma is not declared
#pragma 指定されている関数が宣言されていません。
関数を宣言するか、#pragma を削除してください。

C1550 (W) #pragma DMAC variable must be unsigned long for 変数, ignored
DMAC 指定する'変数'は unsigned long 型で無くてはなりません。
変数の型とレジスタ名を合わせてください。

C1551 (W) #pragma DMAC variable must be far pointer to object for 変数, ignored
DMAC 指定する'変数'はオブジェクト型を指す far ポインタで無くてはなりません。
変数の型とレジスタ名を合わせてください。

C1571 (W) constant variable assignment
const 型修飾子で指定した変数に対して代入しています。
変数宣言から const を外すか、代入を止めてください。

C1573 (W) octal constant is out of range
8 進数の定数に 8 進数で使用できない文字が含まれています。
8 進数の定数は 0 から 7 で記述してください。

C1574 (W) integer constant is out of range
整数定数の値が unsigned long long で表現できる値を超えました。
定数の値を unsigned long long で表現できる値で記述してください。

C1575 (W) multi-character character constant
2 文字以上の文字定数を使用しています。
2 文字以上のときはワイド文字(L'xx')を使用してください。

C1576 (W) hex character is out of range
文字定数において１６進数拡張表記が長すぎます。また、¥の後に 16 進数以外の文字が入っています。
１６進数拡張表記を短くしてください。

C1577 (W) too big octal character
文字定数、文字列中の8進数の定数が、限界値(10進数で255)を超えました。
255以下の値で記述してください。

C1591 (W) assign far pointer to near pointer, bank value ignored
farポインタをnearポインタに代入しています。farポインタの下位2バイトのみを使用します。
データの型、 near / far を確認してください。

C1592 (W) assignment from const pointer to non-const pointer
constポインタから非constポインタへの代入により、const性が失われます。
記述を確認してください。記述が正しい場合は、このウォーニングは無視してください。

C1593 (W) assignment from volatile pointer to non-volatile pointer
volatileポインタから非volatileポインタへの代入により、voiatile性が失われます。
記述を確認してください。記述が正しい場合は、このウォーニングは無視してください。

C1594 (W) far pointer (implicitly) casted by near pointer
farポインタが、nearポインタに変換されました。
データの型、near/farを確認してください。

C1595 (W) incompatible pointer types
ポインタの示すオブジェクトの型が異なります。
ポインタの示すオブジェクトの型を合わせてください。

C1596 (W) mismatch function pointer assignment
レジスタ引数の関数のアドレスを、レジスタ引数でない(関数原型されていない)関数のポインタ変数へ代入しています。
関数のポインタ変数の宣言を関数原型形式にしてください。

C1597 (W) RESTRICT qualifier can set only pointer type
RESTRICT修飾子がポインタ以外で宣言されています。
ポインタのみに宣言してください。

C1598 (W) near pointer not supported, near qualifier ignored
nearポインタは使えません。
near修飾子を削除してください。

C1599 (W) _ext4mptr qualifier can set only pointer type
ポインタで無い型に_ext4mptr修飾子をつけています。
_ext4mptrを使用する場合はポインタを用いてください。

C1600 (W) invalid '%s' operand
この型に対する演算は、言語規格では許されていません。
言語規格に従って記述してください。

C1611 (W) assignment in comparison statement

比較式を書く場所に代入文を記述しています。

" = = " と記述するところを誤って " = " と記述している可能性があります。故意にそう記述したものかを確認してください。

C1612 (W) meaningless statement

文が "= = "で終わっています。

" = " と記述するところを誤って " = =" と記述している可能性があります。故意にそう記述したものかを確認してください。

C1613 (W) can't get size of function

sizeof 演算子のオペランドに関数名を記述しています。

C1614 (W) can't get size of function, unit size 1 assumed

関数へのポインタを++,--しています。増分、減分の値を 1 として処理を継続します。

関数へのポインタを++、--しないでください。

C1617 (W) cyclic or alarm handler function has argument

#pragma CYCHANDLER 又は ALMHANDLER で指定した関数が、引数を使用しています。

#pragma CYCHANDLER 又は ALMHANDLER で指定した関数は、引数を使用できません。引数を削除してください。

C1618 (W) function 関数名 has no-used argument (変数名)

関数引数に宣言された変数は使用されていません。

変数を確認してください。

C1619 (W) function inlining made dummy return value

値を返すべき inline 関数が、値を返さない return 文を持っています。

値を返すように return 文を変更してください。

C1620 (W) function must be far

関数を near 型で宣言しています。

関数は far 型で宣言してください。

C1621 (W) handler function called

#pragma HANDLER で指定した関数を呼び出しています。

ハンドラ関数は呼び出さないようにしてください。

C1622 (W)handler function can't return value

#pragma HANDLER で指定した関数が、戻り値を使用しています。

#pragma HANDLER で指定した関数は、戻り値を使用できません。戻り値を削除してください。

C1623 (W) handler function has argument

#pragma HANDLER で指定した関数が、引数を使用しています。

#pragma HANDLER で指定した関数は、引数を使用できません。引数を削除してください。

C1625 (W) interrupt function called

#pragma INTERRUPT で指定した関数を呼び出しています。

割り込み処理関数は呼び出さないようにしてください。

C1626 (W) interrupt function can't return value

#pragma INTERRUPT で指定した割り込み処理関数が、戻り値を使用しています。

割り込み関数では戻り値を使用できません。戻り値を削除してください。

C1627 (W) interrupt function has argument

#pragma INTERRUPT で指定した割り込み処理関数が、引数を使用しています。

割り込み関数では引数を使用できません。引数を削除してください。

C1628 (W) invalid function argument

関数引数が正しく記述されていません。

関数引数を正しく記述してください。

C1629 (W) invalid storage class for function, change to extern

関数宣言において、不当な記憶域クラスを使用しています。extern として処理します。

記憶域クラスを extern にしてください。

C1630 (W) non-prototyped function declared

定義されている関数の関数原型宣言が存在しません(コンパイルオプション-Wnon_prototype 指定時のみ表示)。

関数原型宣言を行ってください。

C1631 (W) non-prototyped function used

関数原型宣言していない関数を呼び出しています。コンパイルオプション-Wnon_prototype を指定した時のみ出力されます。

関数の関数原型宣言を記述するか、コンパイルオプション-Wnon_prototype を指定しないでください。

C1632 (W) old style function declaration

ANSI(ISO)C 以前の形式で関数定義を記述しています。

ANSI(ISO)形式で 関数定義を記述してください。

C1633 (W) prototype function is defined as nonprototyped function before

関数原型宣言していない関数を再度関数原型宣言しています。

関数の宣言方法を統一してください。

C1635 (W) register parameter function used before as stack parameter function

レジスタ引数の関数が以前にスタック引数の関数として使用しています。

関数を使用する前に関数原型宣言を行ってください。

C1636 (W) static variable in inline function

記憶クラス inline で宣言した関数内で static データを宣言しています。

インライン関数内で、static データを宣言しないでください。

C1637 (W) task function called
#pragma TASK で指定した関数を呼び出しています。
タスク関数は呼び出さないようにしてください。

C1638 (W) task function can't return value
#pragma TASK で指定した関数が、戻り値を使用しています。
#pragma TASK で指定した関数は、戻り値を使用できません。戻り値を削除してください。

C1639 (W) task function has invalid argument
#pragma TASK で指定した関数が、引数を使用しています。
#pragma TASK で指定した関数は、引数を使用できません。引数を削除してください。

C1640 (W) this function used before with non-default argument
関数呼び出しの後に、デフォルト引数を持つ関数として宣言しています。
関数を使用する前にデフォルト引数を宣言してください。

C1641 (W) this interrupt function is called as normal function before
関数呼び出しの後に、 #pragma INTERRUPT で宣言しています。
割り込み関数は呼び出せません。#pragma の内容を確認してください。

C1642 (W) inline function is called as normal function before, change to static function
関数呼び出しの後に、その関数を inline 関数として宣言しています。
inline 関数は、最初の呼び出しより先に定義してください。

C1643 (W) xxx was declared but never referenced
参照されない宣言があります。
宣言を削除してください。

C1644 (W) inline function have invalid argument or return code
インライン関数呼び出しの引数の数が、関数原型宣言と一致しません。
インライン関数呼び出しの引数の数は、関数原型宣言と一致させてください。

C1645 (W) function '関数名' size is out of range
インライン関数の定義が大きすぎるので、インライン関数として展開しません。
インライン関数の定義を小さくしてください。

C1671 (W) argument is define by 'typedef', 'typedef' ignored
引数の宣言において typedef を使用しています。typedef を無視します。
typedef を削除してください。

C1672 (W) illegal storage class for argument, 'extern' ignore
関数定義の引数リストにおいて、不当な記憶域クラスを使用しています。
正しい記憶域クラスを指定してください。

C1673 (W) enum declared inside parameter list
仮引数リスト中で宣言した列挙型は、関数外から型を参照できません。
仮引数リストではなく、関数の外で列挙型を宣言してください。

C1674 (W) mismatch prototyped parameter type
関数原型宣言で宣言した時と引数の型が異なります。
引数の型を確認してください。

C1675 (W) struct declared inside parameter list
仮引数リスト中で宣言した構造体型は、関数外から型を参照できません。
仮引数リストではなく、関数の外で構造体型を宣言してください。

C1676 (W) struct/union/enum declared inside parameter list
仮引数リスト中で宣言した構造体型、共用体、列挙型は、関数外から型を参照できません。
仮引数リストではなく、関数の外で型を宣言してください。

C1677 (W) too few parameters
関数原型宣言で宣言した時より引数が足りません。
関数原型引数の数を確認してください。

C1678 (W) too many parameters
関数原型宣言で宣言した時より引数が多すぎます。
引数の数を確認してください。

C1679 (W) union declared inside parameter list
仮引数リスト中で宣言した共用体型は、関数外から型を参照できません。
仮引数リストではなく、関数の外で共用体型を宣言してください。

C1680 (W) uncomplete struct member
構造体または共用体のメンバに不完全型が含まれています。
構造体または共用体のメンバは完全型を使用してください。

C1691 (W) 'auto' is illegal storage class
不当な記憶域クラスを使用しています。
正しい記憶域クラスを指定してください。

C1692 (W) inline & static conflicted, inline ignored
inline と static はどちらも記憶クラス指定子なので、同時指定はできません。
どちらか一方だけを指定してください。

C1693 (W) block level extern variable initialize forbid, ignored
関数内の extern 変数宣言で初期化式を記述しています。
初期化式を削除するか、記憶域クラスを変更してください。

C1694 (W) external variable initialized, change to public

extern で宣言した変数に対して、初期化式を記述しています。extern を無視します。

extern を削除してください。

C1695 (W) no volatile in previous declaration

同一の宣言が既に存在しており、以前の宣言では volatile がありません。

同じ変数、関数の宣言は、型を合わせてください。

C1696 (W) no const in previous declaretion

const 修飾子がない関数、変数の宣言に対して、実体定義された関数、変数で、const 修飾しています。

関数、変数の宣言とその実体定義で、const 修飾を統一してください。

C1697 (W) static declaration of 識別子 follows non-static

同一の宣言が既に存在しており、以前の宣言では static がありません。

同じ変数、関数の宣言は、記憶クラスを合わせてください。

C1698 (W) extern/static 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。

可視の範囲でリンケージだけが異なる、同名かつ同じ型の宣言を複数個書かないでください。

C1711 (W) char array initialized by wchar_t string

char 型の初期化式を wchar_t 型の文字列で初期化しています。

初期化式の型を合わせてください。

C1712 (W) size of array shall be a value greater than zero

配列の要素数として 0 以下の値で宣言しています。

配列を宣言する場合の要素数は、1 以上としてください。

C1713 (W) string size bigger than array size

初期化する変数のサイズより、初期化式のサイズが大きい。

初期化式のサイズは、変数と同じか、変数より小さくしてください。

C1714 (W) wchar_t array initialized by char string

wchar_t 型の初期化式を char 型の文字列で初期化しています。

初期化式の型を合わせてください。

C1716 (W) enumerator value overflow size of unsigned char

コンパイルオプション -fCE 使用時において、列挙子の値が 255 を越えました。

255 以下で表現できるように記述してください。

C1717 (W) enumerator value overflow size of unsigned int

列挙子の値が 65535 を越えました。

65535 以下で表現できるように記述してください。

C1718 (W) enum's bitfield

ビットフィールドのメンバに列挙型を用いて定義しています。

違う型のメンバを用いてください。

C1719 (W) string terminator not added

配列の要素数と初期化式のサイズが同じであるため、文字列の最後に付加する'¥0'は、付加しません。

配列の要素数を増やしてください。

C1731 (W) identifier (変数名) is duplicated

変数名が重複して定義されています。この宣言を無視します。

変数名の宣言を1つにしてください。

C1732 (W) identifier (変数名) is shadowed

引数で宣言した変数名と同じ変数名のauto変数を使用しています。auto変数を無視します。

引数で使用した変数名以外を使用してください。

C1733 (W) identifier (メンバ名) is duplicated, this declare ignored

メンバ名が重複して定義されています。この宣言を無視します。

メンバ名の宣言を1つにしてください。

C1734 (W) can't get address from register storage class variable

register記憶域クラスの変数に&（アドレス）演算子を記述しています。

register記憶域クラスの変数に&（アドレス）演算子を記述しないでください。

C1735 (W) No storage class & data type in declare, global storage class & int type assumed

記憶域クラス指定子、型指定子なしに、変数を宣言しています。intとして処理します。

記憶域クラス指定子、型指定子を記述してください。

C1736 (W) 'register' is illegal storage class

不当な記憶域クラスを使用しています。

正しい記憶域クラスを指定してください。

C1737 (W) near/far is conflict beyond over typedef

near/far を指定して typedef した型を参照時に再度、near/far を指定して宣言しています。

型指定子を正しく記述してください。

C1754 (W) invalid return type

return文の式が関数の型と異なっています。

リターン値を関数の型に合わせるか、関数の型をリターン値の型に合わせてください。

C1755 (W) redefined type

typedefで、定義済みの型名を再定義しています。

別の型名を使用するか、記述ミスがないか確認してください。

C1756 (W) redefined type name of (識別子)
typedef で同じ識別子名を 2 回以上定義しています。
識別子名を正しく記述してください。

C1800 (W) section name 'interrupt' no more used
#pragma SECTION で指定されたセクション名に'interrupt'を使用しています。
'interrupt'は使用できません。別の名称に変更してください。

C1803 (W) the same identifier is stored in a different section, previous section is used
同じ変数、関数を複数回宣言する時に、#pragma SECTION で異なるセクション配置を指定しています。
同一の変数、関数の宣言では、同じセクションを指定してください。

C1814 (W) non initialized variable '変数名' is used
未初期化の自動変数を参照しています。
参照する前に、値を設定してください。

C1831 (W) case value is out of range
case の値が switch 文の分岐条件の式の表現できる範囲を越えています。
case 値は switch 引数の範囲を超えない値にしてください。

C1832 (W) compile option -fauto_over_255 is specified, #pragma SBDATA was ignored
オプション-fauto_over_255 指定時は#pragma SBDATA を指定できません。
どちらか一方だけを指定してください。

C1833 (W) init elements overflow, ignored
初期化式が初期化しようとする変数のサイズを超えました。
初期化式の数が、初期化する変数のサイズを超えないようにしてください。

C1834 (W) keyword (キーワード) are reserved for future
将来のために予約されているキーワードを使用しています。
別の名前に変更してください。

C1835 (W) large type was implicitly cast to small type
大きい型から小さい型への代入により、値の上位バイト(ワード)が失われる可能性があります。
型を確認してください。記述が正しい場合は、このウォーニングは無視してください。

C1836 (W) No initialized of 変数名
レジスタ変数を初期化しないまま使用している可能性があります。
レジスタ変数に対する代入を行なってください。

C1837 (W) no restrict in previous declaration
同一の宣言が既に存在しており、以前の宣言では restrict がありません。
同じ変数、関数の宣言は、型を合わせてください

C1838 (W) overflow in floating value converting to integer
整数型には格納できない巨大な浮動小数点値を、整数型に代入しています。
代入式を再確認してください。

C1839 (W) standard library "関数名()" need "インクルードファイル名"
標準ライブラリ関数をヘッダファイルをインクルードしないで使用しています。
ヘッダファイルをインクルードしてください。

C1840 (W) this feature not supported now, ignored
文法エラーです。この記述は、将来拡張用の文法ですので使用しないでください。
正しく記述してください。

C1841 (W) underflow in floating value converting to integer
整数型では表現不可能な大きさの浮動小数点定数を、整数型に変換しようとしています。
変換後の整数型で表現可能な範囲の値を使ってください。

C1842 (W) zero divide in constant folding
除算演算子、剰余算演算子において除数が0です。
除数は、0以外を使用してください。

C1843 (W) zero divide,ignored
除算演算子、剰余算演算子において除数が0です。
除数は、0以外を使用してください。

C1844 (W) zero width for bitfield
ビットフィールドの幅が0です。
ビット幅を1以上で記述してください。

C1847 (W) no _ext4mptr is previous declaration
同一の宣言が既に存在しており、以前の宣言では_ext4mptrがありません。
同じ変数、関数の宣言は、型を合わせてください。

C1848 (W) meaningless statements deleted in optimize phase
無意味な記述が最適化で削除されました。
無意味な記述を削除してください。

C1849 (W) this comparison is always true
常に真になる比較を行っています。
条件式を確認してください。

C1850 (W) this comparison is always false
常に偽になる比較を行っています。
条件式を確認してください。

C1851 (W) compile option -fSB_auto(-fSBA) is specified, #pragma SBDATA was ignored
オプション-fSB_auto と#pragma SBDATA を同時に使うことはできません。
どちらか一方だけを指定してください。

C1860 (W) -OR, -OS duplicated option
-OR と -OS を同時に使うことはできません。
どちらか一方だけを指定してください。

C1861 (W) オプション名A, オプション名B duplicated option,オプション名C is ignore
オプション名A と オプション名B は同時に指定できません。オプション名C を無視します。
どちらか一方だけを指定してください。

C1862 (W) Can't use オプション名A with オプション名B, オプション名A is ignored.
オプション名A と オプション名B は同時に指定できません。オプション名A を無視します。
どちらか一方だけを指定してください。

C1863 (W) Invalid オプション名 value (値)
オプション名に設定する値が不正です。
正しい値を設定してください。

C1864 (W) Unknown オプション種別 option (オプション名)
オプション（オプション名）は存在しません。
正しいオプション名を使用してください。

C1865 (W) Unknown option (オプション名)
オプション（オプション名）は存在しません。
正しいオプション名を使用してください。

C2004 (E) can't open command file
@で指定されたコマンドファイルがオープンできません。
正しいファイル名を指定してください。

C2005 (E) command-file line characters exceed 2048.
コマンドファイルの1行の文字数が 2048 文字を越えています。
コマンドファイルの1行の文字数を 2048 文字以下にしてください。

C2008 (E) Invalid suffix '.xxx'
NC30 が認識できないファイル拡張子(.c、.cpp、.cc、.cp、.i、.a30、.obj 以外の拡張子)を使用しています。
正しい拡張子でファイルを指定してください。

C2009 (E) Invalid option '-?'
不正なコンパイルオプション-?が指定されています。または、コンパイルオプション-?に必要なパラメータがありません。
コンパイルオプション-?が正しいか確認してください。または、コンパイルオプション-?の次に必要なパラメータを指定してください。

C2010 (E) Too many command files
@コマンドファイルが複数指定されています。
@コマンドファイルの指定は1つにしてください。

C2011 (E) too many options
指定されたコンパイルオプションの数が100以上です。
コンパイルオプションは99個までにしてください。

C2012 (E) -r8ce, -r8c duplicated option
-R8Cオプションと -R8CEオプションが同時に指定されています。
-R8Cオプションと -R8CEオプションは同時に指定しないでください。

C2013 (E) Can't specify twice option 'オプション'
同じオプションが2回以上指定されています、または相反するオプションが同時に指定されています。
指定されたオプションは1回のみの指定としてください。または、相反するオプションはどちらか一方のみの指定としてください。

C2014 (E) Can't specified 'オプション' with -S option
-Sオプションと同時指定できないオプションが指定されています。
指定されたオプションと -Sオプションは同時に指定しないようにしてください。

C2015 (E) Invalid NCKIN value 'xxxx'
環境変数：NCKIN に無効な値が設定されています。
環境変数：NCKIN への設定は、SJIS か EUC の何れかにしてください。

C2017 (E) Illegal option 'オプション' can't specify together with -lang=ecpp option
-lang=ecpp と同時に指定できないオプションが指定されています。
指定されたオプションは -lang=ecpp と同時には指定しないでください。

C2018 (E) Illegal option 'オプション' can't specify together with -rtti,-exception,-template option
-lang=ecpp オプションが -rtti=on または -exception と同時に指定されています。
-lang=ecpp オプションは -rtti=on または -exception と同時には指定しないでください。

C2024 (E) Can't be specified to a file name
-o オプションのパラメータに -(ハイフン) で始まる文字列が指定されています。
-o オプションのパラメータ（ファイル名）には、-(ハイフン) で始まるもの以外を指定してください。

C2026 (E) Can't specify 'オプションA' with 'オプションB' option
指定されたオプションA は オプションB と一緒に指定できません。
オプションA とオプションB は同時に指定しないでください。

C2029 (E) No directory 'ディレクトリ', environment variable '環境変数名'
環境変数に設定されたディレクトリが存在しません。
環境変数に設定されたディレクトリが正しいか確認してください。

C2500 (E) Sorry, compilation terminated because of too many errors

ソースファイル中のエラーがエラーの上限(50個)を超えました。
このメッセージが出力される以前のエラーを修正してください。

C2501 (E) Sorry, compilation terminated because of these errors in 関数名

関数名で示される関数でエラーが発生しました。コンパイルを中止します。
このメッセージが出力される以前のエラーを修正してください。

C2502 (E) can't read C source from filename line 行数 for error message

エラーが発生したソースラインを表示できません。filenameで示されるファイルがないか、行番号が、ファイルに存在しません。
ファイルの存在を確認してください。

C2504 (E) can't open C source filename for error message

エラーが発生したソースファイルがオープンできません。
ファイルの存在を確認してください。

C2505 (E) Sorry stack frame memory exhaust, max '最大引数合計値' bytes(argument) but now '現在の引数合計値' bytes

スタック渡しとなる引数の合計サイズが大きすぎます。
表示されている最大値以内にしてください

C2506 (E) Sorry stack frame memory exhaust, max '関数でのスタック使用可能値' bytes(auto) but now '現在の合計値' bytes

スタック渡しとなる引数とauto変数の合計サイズが大きすぎます。
表示されている最大値以内にしてください。

C2508 (E) can't refer to the range outside of the stack frame

スタックフレーム領域外を参照しています。
正しく指定してください。

C2509 (E) too many operators

1行中の演算子の数が多すぎます。
1行中の演算子の数は、1000個未満にしてください。

C2512 (E) #pragma プラグマ名 & function prototype mismatched

#pragma プラグマ名 で指定した関数と関数原型宣言の引数の内容が異なっています。
関数原型宣言の引数と合わせてください。

C2514 (E) Invalid #pragma OS拡張機能 interrupt number

#pragma OS拡張機能 で記述したINT番号は、指定することができません。
正しく指定してください。

C2514 (E) Invalid #pragma INTCALL interrupt number

#pragma INTCALL で記述した INT 番号は、指定することができません。

正しく指定してください。

C2515 (E) Invalid #pragma SPECIAL special page number

#pragma SPECIAL で記述した番号またはフォーマット指定が間違っています。

正しく指定してください。

C2516 (E) Invalid #pragma INTERRUPT vector number

#pragma INTERRUPT で記述した番号またはフォーマット指定が間違っています。

正しく指定してください。

C2518 (E) multiple #pragma EXT4MPTR's pointer, ignored

#pragma EXT4MPTR が 2 個以上宣言されています。

#pragma EXT4MPTR は 2 個以上宣言しないでください。

C2519 (E) asm()'s string must have 1 $$

この asm 関数は$$を 1 個持たなくてはなりません。

$$を 1 個使用してください。

C2520 (E) asm()'s string must have 1 $$ or $@

この asm 関数は$$または$@を 1 個持たなくてはなりません。

$$または$@を 1 個使用してください。

C2521 (E) asm()'s string must have 1 $@

この asm 関数は$@を 1 個持たなくてはなりません。

$@を 1 個使用してください。

C2522 (E) asm()'s string must have only 1 $b

asm ステートメントで $b は 一度しか記述することはできません。

$b の記述を 1 回にしてください。

C2523 (E) asm()'s string must not have more than 3 $$ or $@

asm ステートメントで $$または $@を 3 回以上記述しています。

$$($@)の記述を 2 回以下にしてください。

C2525 (E) floating type's bitfield

不当な型のビットフィールドを宣言しています。

ビットフィールドは、整数型を使用してください。

C2525 (E) invalid asm()'s argument

asm ステートメントに使用できる変数は、auto 変数と引数です。

auto 変数か引数で記述してください。

C2526 (E) #pragma PARAMETER functions register not allocated

#pragmaPARAMETERで指定した関数で指定したレジスタは、記述できません。
レジスタを正しく記述してください。

C2527 (E) #pragma プラグマ名's function must be declared before use, #pragma is ignored

関数呼び出しの後に、#pragmaを指定しています。
#pragmaは対象となる関数を呼び出すより前に指定してください。

C2528 (E) #pragma BITADDRESS variable is not _Bool type

#pragma BITADDRESSで指定された変数が、_Bool型ではありません。
#pragma BITADDRESSに指定する変数は、_Bool型にしてください。

C2529 (E) #pragma プラグマ名 format error,ignored

#pragma プラグマ名に続く内容が間違っています。
正しいフォーマットで書いてください。

C2531 (E) #pragma INTCALL function's argument on stack

#pragma INTCALLで宣言した関数の実体をC言語で記述した場合に引数がスタック渡しになっています。
#pragma INTCALLで宣言した関数の実体をC言語で記述する場合は引数にはレジスタ渡しとなる型を指定してください。

C2532 (E) #pragmaプラグマ名 function argument is long-long or double

#pragma プラグマ名 で指定した関数の引数に、long long型、double型が使用されています。
"#pragma プラグマ名 関数名"で指定した関数には、long long型、およびdouble型を指定できません。別の型を使用してください。

C2533 (E) #pragma プラグマ名 function argument is struct or union

#pragma プラグマ名 で指定した関数の関数原型宣言で struct / union型 を指定しています。
関数原型宣言でint、short型又は、サイズが2バイトのポインタ型、列挙型を指定してください。

C2534 (E) #pragma プラグマ名 must be declared before use

#pragma プラグマ名 で指定した関数の定義が、その関数の呼び出しの後に記述されています。
関数の呼び出しを行う前に宣言してください。

C2535 (E) #pragmaプラグマ名 関数名 redefined

#pragma プラグマ名 において同じ関数を重複して定義しています。
#pragma プラグマ名の宣言を1回にしてください。

C2537 (E) #pragma プラグマ名 function must be prototyped

#pragma プラグマ名 で指定した関数を関数原型宣言無しで呼び出しています。
#pragma プラグマ名 で指定した関数は、呼び出す前に関数原型宣言してください。

C2551 (E) mismatch prototyped parameter type

関数原型宣言で宣言した時と引数の型が異なります。
引数の型を確認してください。

C2552 (E) '関数名' function has struct argument
インライン関数は引数として構造体を持つことができません。
構造体型の引数を持つ関数は、インライン関数にしないでください。

C2554 (E) '関数名' is recursion, a function of recursive call can not be described inline qualifier
インライン関数は再帰呼び出しできません。
再帰呼び出しする関数から inline 指定を外してください。

C2555 (E) can't get inline function's address by '&' operator
インライン関数に&演算子を記述しています。
インライン関数に&演算子を記述しないでください。

C2556 (E) conflict function argument type of 変数名
引数リストに同じ変数名があります。
変数名を変更してください。

C2557 (E) declared register parameter function's body declared
#pragma PARAMETER で宣言した関数を C 言語で実体の定義を行っています。
#pragma PARAMETER で宣言した関数は C 言語での実体記述を行わないでください。

C2558 (E) function initialized
関数の宣言に対して初期化式を記述しています。
初期化式を削除してください。

C2559 (E) function member declared
構造体、共用体のメンバで関数型を指定しています。
メンバを正しく記述してください。

C2560 (E) function returning a function declared
関数の宣言において戻り値の型が関数型になっています。
戻り値の型を関数へのポインタ型等に変更してください。

C2561 (E) function returning an array
関数の宣言において戻り値の型が配列型になっています。
戻り値の型を関数へのポインタ型等に変更してください。

C2562 (E) handler function called
#pragma HANDLER で指定した関数を呼び出しています。
ハンドラ関数は呼び出さないようにしてください。

C2563 (E) default function argument conflict
関数原型宣言において、引数のデフォルト値を 2 回以上宣言しています。
引数のデフォルト値は、1回だけ宣言してください。

C2564 (E) inline function have invalid argument or return code
インライン関数に、不正な引数または、不正な戻り値があります。
正しい、引数、戻り値を指定してください。

C2565 (E) inline function is called as normal function before
インライン関数が通常の関数として、宣言前に呼び出されています。
関数を確認してください。

C2566 (E) inline function's address used
インライン関数のアドレスを参照しています。
インライン関数のアドレスは使用しないでください。

C2567 (E) inline function's body is not declared previously
インライン関数の実体定義がありません。
インライン関数を使用する際には、関数を呼び出すよりも前に関数の実体を定義してください。

C2568 (E) interrupt function called
#pragma INTERRUPT で指定した関数を呼び出しています。
割り込み処理関数は呼び出さないようにしてください。

C2569 (E) invalid function argument
関数定義中の引数の宣言において、引数リストに含まれない引数を宣言しています。
引数リストに存在する変数を宣言してください。

C2570 (E) invalid function declare
関数定義に誤りがあります。
エラーが発生した行か、その直前の関数定義を確認してください。

C2571 (E) invalid function default argument
関数のデフォルト引数が正しくありません。
デフォルト引数を持つ関数の関数原型宣言と、関数定義部で引数の整合が取れていない場合等に発生します。整合が取れるように記述してください。

C2572 (E) invalid function[] operand
関数型の配列は使えません。
関数ポインタの配列を使ってください。

C2573 (E) invalid function's argument declaration
関数の引数の宣言に誤りがあります。
正しく記述してください。

C2574 (E) redefine function 関数名
関数名で示される関数が重複定義されています。
関数は１度しか定義できません。関数定義を１度にしてください。

C2575 (E) return expression is in void type function
void を返す関数定義の中に、値を返す return 文があります。
void を返す関数定義中の return 文は、値を返さないでください。

C2576 (E) task function called
#pragma TASK 指定した関数は、通常関数同様に呼び出すことはできません。
#pragma TASK 指定した関数の呼び出し方は RTOS のマニュアルを参照してください。

C2577 (E) unknown function argument 変数名
引数リストにない引数を指定しています。
引数を確認してください。

C2591 (E) array of functions declared
配列宣言において関数のポインタ配列ではなく関数自身の配列を宣言しています。
関数のポインタ配列等に変更してください。

C2592 (E) array size is not constant integer
配列の宣言において要素数が定数ではありません。
要素数を定数で記述してください。

C2593 (E) incomplete array access
不完全型の多次元配列を参照しています。
多次元配列のサイズを明記してください。

C2594 (E) invalid initializer on array
初期化式に誤りがあります。
括弧内の初期化式の数が、配列要素の数、構造体メンバの数と一致しているかを確認してください。

C2595 (E) invalid initializer on char array
初期化式に誤りがあります。
括弧内の初期化式の数が、配列要素の数、構造体メンバの数と一致しているかを確認してください。

C2596 (E) size of incomplete array type
大きさが不明な配列の sizeof を求めています。無効なサイズです。
配列の大きさを指定してください。

C2597 (E) size of uncomplete type's array
不完全な配列のサイズは取得てきません。
サイズ取得が必要な場合は、完全型にしてください。

C2598 (E) too large array size : バイト数
配列のサイズが大きすぎます。
配列のサイズを小さくしてください。

C2599 (E) uncomplete array pointer operation

不完全型の配列に対してポインタ参照しようとしています。

完全な配列を先に定義してください。

C2600 (E) void array is invalid type, int array assumed

void 型の配列は宣言できません。int 型の配列として処理を継続します。

型指定子を正しく記述してください。

C2601 (E) zero size array member

サイズがゼロの配列です。

サイズを明確にしてください。

構造体のメンバにサイズがゼロの配列があります。

サイズがゼロの配列を構造体のメンバにすることはできません。

C2603 (E) incomplete struct get by [ ]

メンバが確定していない(不完全な)構造体、共用体の配列を参照又は初期化しています。

完全な構造体、共用体を先に定義してください。

C2604 (E) incomplete struct initialized

メンバが確定していない(不完全な)構造体、共用体を初期化しています。

完全な構造体、共用体を先に定義してください。

C2605 (E) incomplete struct return function call

メンバが確定していない(不完全な)構造体、共用体の型をリターン値にもつ、関数を呼び出しています。

完全な構造体、共用体を先に定義してください。

C2606 (E) incomplete struct / union(タグ名)'s member access

メンバが確定していない(不完全な)構造体、共用体のメンバを参照しています。

完全な構造体、共用体を先に定義してください。

C2607 (E) incomplete struct / union's member access

メンバが確定していない(不完全な)構造体、共用体のメンバを参照しています。

完全な構造体、共用体を先に定義してください。

C2608 (E) invalid initializer on struct

初期化式に誤りがあります。

括弧内の初期化式の数が、配列要素の数、構造体メンバの数と一致しているかを確認してください。

C2609 (E) invalid struct or union type

列挙型のデータに対して構造体, 共用体のメンバを参照しています。

正しく記述してください。

C2610 (E) not struct or union type
->の左辺式が、構造体,共用体型ではありません。
->の左辺式を、構造体,共用体型で記述してください。

C2611 (E) redefinition tag of struct タグ名
構造体を二重に定義しています。
構造体の定義は１回にしてください。

C2612 (E) struct or enum's tag used for union
構造体, 列挙型のタグ名を共用体のタグ名として使用しています。
タグ名を変更してください。

C2613 (E) struct or union's tag used for enum
構造体, 共用体のタグ名を列挙型のタグ名として使用しています。
タグ名を変更してください。

C2614 (E) union or enum's tag used for struct
共用体、列挙型のタグ名を構造体のタグ名として使用しています。
タグ名を変更してください。

C2615 (E) unknown pointer to structure idetifier "変数名"
-> の左辺式が、構造体,共用体型ではありません。
-> の左辺式を、構造体,共用体型で記述してください。

C2616 (E) unknown size of struct or union
大きさの確定していない、不完全な構造体、共用体を使用しています。
構造体、共用体の変数を宣言する前に、構造体、共用体を宣言してください。

C2617 (E) unknown structure idetifier "変数名"
. の左辺式が、構造体,共用体型ではありません。
. の左辺式を、構造体,共用体型で記述してください。

C2618 (E) redefinition tag of union タグ名
共用体を二重に定義しています。
共用体の定義は１回にしてください。

C2619 (E) invalid enumerator initialized
列挙子の初期値に変数名を記述するなど誤った指定を行っています。
列挙子の初期値を正しく記述してください。

C2620 (E) redefinition tag of enum タグ名
列挙を二重に定義しています。
列挙の定義は１回にしてください。

C2621 (E) bitfield width exceeded

ビットフィールドの幅が、データ型のビット幅を超えています。

ビットフィールドで宣言したデータ型のビット幅以内で記述してください。

C2622 (E) bitfield width is not constant integer

ビットフィールドのビット幅が定数ではありません。

ビット幅を定数で記述してください。

C2623 (E) can't get bitfield address by '&' operator

ビットフィールドタイプに&演算子を記述しています。

ビットフィールドタイプに&演算子を記述しないでください。

C2624 (E) can't get size of bitfield

ビットフィールドのサイズを取得しようとしています。

ビットフィールドのサイズを取得することはできません。

C2626 (E) invalid bitfield declare

ビットフィールドの宣言で誤りがあります。

正しく記述してください。

C2627 (E) invalid size of bitfield

ビットフィールドのサイズを取得しようとしています。

この宣言ではビットフィールドを記述しないでください。

C2628 (E) invalid type's bitfield

不当な型のビットフィールドを宣言しています。

ビットフィールドは整数型を使用してください。

C2629 (E) long long type's bitfield

long long 型のビットフィールドを記述しています。

long long 型はビットフィールドに宣言できません。別の型で宣言してください。

C2630 (E) invalid array type

不完全型の配列を宣言することはできません。

多次元配列の宣言時、配列の要素数を明示してください。

C2651 (E) not static initializer for 変数名

static 変数の初期化式に誤りがあります。初期化式が関数呼び出しになっているなど。

初期化式を正しく記述してください。

C2652 (E) 'static' is illegal storage class for argument

引数の宣言において不適当な記憶域クラスを使用しています。

正しい記憶域クラスを使用してください。

C2661 (E) do while(void) statement
do while 文の式に void 型を使用しています。
do while 文の式は、スカラ型を記述してください。

C2662 (E) do while(struct/union) statement
do while 文の式に struct/union 型を使用しています。
do while 文の式は、スカラ型を記述してください。

C2663 (E) for(; struct/union;) statement
for 文の 2 番目の式に struct/union 型を使用しています。
for 文の 2 番目の式は、スカラ型を記述してください。

C2664 (E) if(struct/union) statement
if 文の式に struct/union 型を使用しています。
if 文の式は、スカラ型を記述してください。

C2665 (E) if(void) statement
if 文の式に void 型を使用しています。
if 文の式は、スカラ型を記述してください。

C2666 (E) invalid break statements
break 文を記述できないところで使用しています。
switch、while、do-while、for のなかで記述してください。

C2667 (E) invalid case statements
case ラベル文を switch 文の中以外で記述しています。
switch 文の中以外で記述しないでください。

C2668 (E) invalid continue statements
continue 文を記述できないところで使用しています。
while 文、do-while 文および for 文の中以外で記述しないでください。

C2669 (E) invalid default statements
switch 文に誤りがあります。
switch 文を正しく記述してください。

C2670 (E) invalid switch statement
switch 文の記述に誤りがあります。
正しく記述してください。

C2671 (E) while(struct/union) statement
while 文の式に struct/union 型を使用しています。
while 文の式は、スカラ型を記述してください。

C2672 (E)while(void) statement

while 文の式に void 型を使用しています。

while 文の式は、スカラ型を記述してください。

C2673 (E) for(; void ;) statement

for 文の 2 番目の式に void 型を使用しています。

for 文の 2 番目の式は、スカラ型を記述してください。

C2691 (E) auto variable's size is zero

auto 領域に要素数が 0 の配列、あるいは要素数のない配列を宣言しています。

正しく宣言してください。

C2692 (E) invalid environment variable : 環境変数名

環境変数 NCKIN/NCKOUT で指定された変数名が SJIS、EUC 以外が指定されています。

環境変数を確認してください。

C2693 (E) unknown variable 変数名 used

未定義の変数名を使用しています。

変数を定義してください。

C2694 (E) unknown variable 変数名

未定義の変数名を使用しています。

変数を定義してください。

C2695 (E) unknown variable "変数名" used in asm()

asm ステートメントにおいて、未定義の変数名を使用しています。

変数を定義してください。

C2696 (E) can't get void value

式の中で void 型の値を参照しようとしています。

データの型を確認してください。

C2697 (E) case value is duplicated

case の値を重複して使用しています。

1 つの switch 文で同じ case の値を使用しないでください。

C2698 (E) floating point value overflow

浮動小数点型の定数の値が表現できる範囲を超えています。

範囲以内の値にしてください。

C2699 (E) invalid case value

case の値に誤りがあります。

整数型、列挙型の定数を記述してください。

C2701 (E) void value can't return

void 型にキャストした値を関数の戻り値に記述しています。
正しく記述してください。

C2702 (E) argument type given both places

関数定義中の引数の宣言において、引数リストと重複して引数の宣言を行っています。
引数リストか、引数の宣言のどちらかで引数の宣言を行ってください。

C2705 (E) illegal storage class for argument, 'interrupt' ignored

関数内での宣言文において割り込み関数を宣言しています。
関数外で宣言してください。

C2706 (E) invalid lvalue

代入式の左辺が代入可能ではありません。
左辺に代入可能なオブジェクトを示す式を記述してください。

C2707 (E) can't set argument

関数原型宣言と実引数との型の不一致により、レジスタ(引数)に実引数をセットできません。
型の不一致を修正してください。

C2708 (E) illegal storage class for argument, 'inline' ignored

関数内での宣言文においてインライン関数を宣言しています。
関数外で宣言してください。

C2721 (E) switch's condition is floating

switch 文の式に浮動小数点型を使用しています。
整数型、列挙型を使用してください。

C2722 (E) switch's condition is void

switch 文の式に void 型を使用しています。
整数型、列挙型を使用してください。

C2723 (E) switch's condition must integer

switch 文の式に整数型、列挙型以外の型を使用しています。
整数型、列挙型を使用してください。

C2743 (E) 'const' is duplicate

const を 2 回以上記述しています。
型修飾子を正しく記述してください。

C2744 (E) default: is duplicated

default の値を重複して使用しています。
1 つの switch 文の中に default ラベルが 2 つ以上使用されています。
1 つの switch 文で default ラベルは 1 つだけにしてください。(ネストしている switch 文の中の default ラベルは含みません。)

C2745 (E) identifier (変数名) is duplicated
変数が重複して定義されています。
変数の定義を正しく指定してください。

C2746 (E) 'restrict' is duplicate
宣言中の restrict 修飾子が重複しています。
1つの修飾対象について1回だけにしてください。

C2747 (E) 'volatile' is duplicate
volatile を2回以上記述しています。
型修飾子を正しく記述してください。

C2748 (E) '_ext4mptr' is duplicated
_ext4mptr を重ねて書いています。
重複する_ext4mptr は1個だけにしてください。

C2761 (E) conflict declare of 変数名
1度目と2度目で記憶域クラスの異なる重複定義を行っています。
変数を2度宣言する場合は、同じ記憶域クラスで行ってください。

C2763 (E) duplicate frame position defind 変数名
同じ識別子を持つ auto 変数が2回以上記述されています。
正しく記述してください。

C2764 (E) Empty declare
記憶域クラス指定子、型指定子しかありません。
宣言子を記述してください。

C2765 (E) 'far' & 'near' conflict
同じ変数(関数)に対して near / far の宣言が一致していません。
near / far を正しく記述してください。

C2766 (E) parse error at near '文字列'
解釈不可能な文字列が存在します。
C/C++の文法に従った書き方に変更してください。

C2767 (E) parse error at near
解釈不可能な文字列が存在します。
C/C++の文法に従った書き方に変更してください。

C2780 (E) redeclare of 変数名もしくは列挙子
変数名もしくは列挙子が重複して定義されています。
どちらかの変数名を変更してください。

C2781 (E) invalid lvalue at '=' operator

代入式の左辺が代入可能ではありません。

左辺に代入可能なオブジェクトを示す式を記述してください。

C2782 (E) invalid ' ? : ' operand

?: 演算子の記述に誤りがあります。

演算子の各式を確認してください。また、：の左右の式の型は、互換型である必要があります。

C2782 (E) invalid '!=' operands

!=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '&&' operands

&&演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '&' operands

&演算子の記述に誤りがあります。

演算子の右辺式を確認してください。

C2782 (E) invalid '&=' operands

&=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '()' operand

( )の左辺式が関数ではありません。

( )の左辺式は関数又は関数へのポインタを記述してください。

C2782 (E) invalid '＊' operands

乗算の場合＊演算子の記述に誤りがあります。ポインタ演算子の場合、右辺式がポインタ型でありません。

乗算の場合、演算子の左辺式、右辺式を確認してください。ポインタの場合、右辺の型を確認してください。

C2782 (E) invalid '*=' operands

*=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '+' operands

+演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '+=' operands

+=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '-' operands
-演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '-=' operands
-=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '/=' operands
/=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '<<' operands
<<演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '<<=' operands
<<=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '<=' operands
<=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '=' operand
=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '= =' operands
= =演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '>=' operands
>=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '>>' operands
>>演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '>>=' operands
>>=演算子の記述に誤りがあります。
演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '[ ]' operands

[ ] の左辺式が配列、ポインタ型ではありません。

[ ] の左辺式は配列、又はポインタ型を記述してください。

C2782 (E) invalid '^=' operands

^=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '¦=' operands

¦=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '¦¦' operands

¦¦ 演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid '%=' operands

%=演算子の記述に誤りがあります。

演算子の左辺式、右辺式を確認してください。

C2782 (E) invalid ++ operands

++単項演算子又は後置演算子の記述に誤りがあります。

単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。

C2782 (E) invalid -- operands

--単項演算子又は後置演算子の記述に誤りがあります。

単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。

C2782 (E) invalid (? ;)'s condition

三項演算子の記述に誤りがあります。

三項演算式を確認してください。

C2782 (E) invalid CAST operand

cast 演算子に誤りがあります。void 型を他の型にキャスト及び、構造体, 共用体からもしくは他の構造体、共用体へのキャストはできません。

正しく記述してください。

C2784 (E) invalid unary '!' operands

! 単項演算子の記述に誤りがあります。

演算子の右辺式を確認してください。

C2784 (E) invalid unary '+' operands

+単項演算子の記述に誤りがあります。

演算子の右辺式を確認してください。

C2784 (E) invalid unary '-' operands

-単項演算子の記述に誤りがあります。
演算子の右辺式を確認してください。

C2784 (E) invalid unary '~' operands

~単項演算子の記述に誤りがあります。
演算子の右辺式を確認してください。

C2785 (E) invalid cast operator

cast 演算子の記述に誤りがあります。
正しく記述してください。

C2786 (E) invalid (? :)'s condition

条件演算子(? :)の条件式が不正です。
条件式を正しく書いてください。

C2787 (E) invalid -> used

->の左辺式が、構造体または共用体へのポインタ型ではありません。
->の左辺式を、構造体または共用体へのポインタ型の式で記述してください。

C2788 (E) invalid operation for pointer to incomplete type

不完全型へのポインタに対して無効な演算をしています。
構造体のメンバを定義する、または配列の要素数を明示して完全型にしてください。

C2789 (E) can't get address from register storage class variable

レジスタ変数のアドレスは取得できません。
アドレス取得が必要な場合はレジスタ修飾を外してください。

C2801 (E) invalid redefined type name of (識別子)

typedef で同じ識別子名を 2 回以上定義しています。
識別子名を正しく記述してください。

C2802 (E) invalid return type

関数の戻り値の型が正しくありません。
正しく記述してください。

C2803 (E) invalid type specifier

int int i; 等のように同じ型指定子を 2 回以上記述しているか、float int i; 等のように矛盾した型指定子を記述しています。
型指定子を正しく記述してください。

C2804 (E) invalid type specifier,long long long

long を 3 個以上記述して型宣言しています。
型宣言を確認してください。

C2805 (E) invalid void type, int assumed

void 型の変数は宣言できません。int 型として処理を継続します。

型指定子を正しく記述してください。

C2807 (E) type redeclaration of 変数名

1 度目と 2 度目で型の異なる重複定義を行っています。

変数を 2 度宣言する場合は同じ型で行ってください。

C2808 (E) too many storage class of typedef

宣言中に extern/typedef/static/auto/register などの記憶域クラス指定子を 2 以上記述しています。

記憶域クラス指定子を 2 回以上記述しないでください。

C2809 (E) typedef initialized

typedef で宣言した変数に初期化式を記述しています。

初期化式を削除してください。

C2821 (E) invalid initializer

初期化式に誤りがあります。括弧が多過ぎる、初期化式の数が多い、関数内の static 変数を auto 変数で初期化している、変数を変数で初期化しているなど。

初期化式を正しく記述してください。

C2822 (E) invalid initializer of 変数名

初期化式に誤りがあります。ビットフィールドの初期化式に対して変数を記述しているなど。

初期化式を正しく記述してください。

C2823 (E) invalid initializer on scalar

初期化式に誤りがあります。

括弧内の初期化式の数が、配列要素の数、構造体メンバの数と一致しているかを確認してください。

C2824 (E) invalid initializer, too many brace

auto 記憶域クラスのスカラ型の初期化式において括弧{}が多過ぎます。

括弧{}の数を減らしてください。

C2825 (E) invalid member

メンバ参照の記述に誤りがあります。

正しく記述してください。

C2826 (E) invalid member used

メンバ参照の記述に誤りがあります。

正しく記述してください。

C2827 (E) invalid push

関数引数等で void 型を push しています。

void を push することはできません。

C2828 (E) invalid strage class for data
記憶クラスの指定に誤りがあります。
正しく記述してください。

C2829 (E) invalid truth expression
条件式(?:)の１つめの式で void, struct, union 型を使用してします。
スカラ型で記述してください。

C2830 (E) label ラベル redefine
１つの関数内で同じラベルを２度定義しています。
どちらかのラベルの名前を変更してください。

C2834 (E) size of incomplete type
sizeof 演算子のオペランドに、定義されていない構造体、共用体を記述しています。
構造体、共用体を先に定義してください。
sizeof 演算子のオペランドに定義されている配列の要素数が決定していません。
配列の宣言時に要素数を指定してください。

C2835 (E) No declarator
宣言文が不完全です。
完全な宣言文を記述してください。

C2836 (E) reinitialized of 変数名
同じ変数に対して初期化式を２度指定しています。
初期化式を１つにしてください。

C2851 (E) size of void
void のサイズを求めています。無効なサイズです。
void のサイズは求められません。

C2852 (E) too big address
32 ビット以上のアドレスを設定しようとしています。
ご使用になられるマイコンに応じたアドレスに収まる値を設定してください。

C2853 (E) too big data-length
32 ビット以上のアドレスを取得しようとしています。
ご使用になられるマイコンに応じたアドレスに収まる値を設定してください。

C2854 (E) undefined label "ラベル" used
goto の分岐先のラベルが関数内に定義されていません。
関数内に分岐先のラベルを定義してください。

C2855 (E)unknown member メンバ名 used
構造体,共用体のメンバに登録されていないメンバを参照しています。
メンバ名を確認してください。

C2856 (E) syntax error

文法エラーです。
正しく記述してください。

C3001 (F) Arg list too long

各処理系を起動するときのコマンドラインがシステムで定義された文字数を超えています。
システムで定義された文字数を超えないようにコンパイルオプションを指定してください。各処理系のコマンドラインは、コンパイルオプション-vで確認することができます。

C3002 (F) Permission denied

各処理系が実行できません。
各処理系のアクセス権を確認してください。又、パーミッションが正しい場合は、各処理系が格納されているディレクトリを環境変数で正しく設定しているかを確認してください。

C3003 (F) Invalid argument

内部エラーです（通常は発生しません)。
弊社までご連絡ください。

C3004 (F) Too many open files

内部エラーです（通常は発生しません)。
弊社までご連絡ください。

C3005 (F) No such file or directory

各処理系が実行できません。
各処理系が格納されているディレクトリを環境変数で正しく設定しているかを確認してください。

C3006 (F) Exec format error

各処理系の実行ファイルが壊れています。
再度、インストールしなおしてください。

C3007 (F) Not enough core

スワップ領域が不足しています。
スワップ領域を増やしてください。

C3008 (F) Result too large

内部エラーです（通常は発生しません)。
弊社までご連絡ください。

C3010 (F) Cannot analyze error

通常は発生しません(内部エラーです)。
弊社までご連絡ください。

C3012 (F) Can't get environment variable(環境変数名)

環境変数の値が指定されていません。または値が不正です。
環境変数の値を設定してください。

C3013 (F) Core dump(command_name)

処理系がCore Dumpを起こしました。カッコ内はCore dumpを起こした処理系です。
各処理系が正しく実行されていません。環境変数又は各処理系が存在するディレクトリを確認してください。その上で正しく起動しない場合は、弊社にご連絡ください。

C3014 (F) Can't create temporary file

一時ファイルのオープンに失敗しました。
ディスク容量やシステムの状態を確認してください。

C3507 (F) can't open ファイル名

ファイルがオープンできません。
ファイルのパーミッションを確認してください。

C3508 (F) can't output to ファイル名

ファイルに書き込みができません。
ディスクの残り容量又はファイルのアクセス権を確認してください。

C3514 (F) No #pragma ENDASM

#pragma ASM と対になる#pragma ENDASM がありません。
#pragma ENDASM を記述してください。

C3517 (F) Not enough memory

メモリ空間が不足しています。
メモリを増やすか、Windowsの仮想メモリを増やしてください。

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。

C5001 (E) Last line of file ends without a newline

ファイルの最終行の末尾に改行文字がありません。

C5002 (E) Last line of file ends with a backslash

ファイルの最終行の末尾がバックスラッシュになっています。

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)(W) 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)(W) 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。

C5023 (E) Integer constant is too large
整数定数の値が大きすぎます。

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) (W) 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 (E)(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 (E)(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)(W) 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) “マクロ名” is predefined; attempted redefinition ignored
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二重定義しています。

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。オーバー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。

C5076 (W) Argument to macro is empty
関数マクロに対して引数が指定されていません。

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。

C5078 (E) A parameter declaration may not have an initializer
パラメーター宣言には初期化子を指定できません。

C5079 (E) Expected a type specifier
型指定子が必要です。

C5080 (E)(W) 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 (W) Storage class is not first
記憶クラスがデータ型の前に指定されていません。

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 (W) 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。

C5092 (E) Identifier-list parameters may only be used in a function definition
識別子リストパラメータは関数定義以外の場所で使用できません。

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。

C5096 (W) A translation unit must contain at least one declaration
翻訳単位内には最低1つの宣言が必要です。

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。

C5099 (E)(W) A declaration here must declare a parameter
この関数宣言はパラメータを宣言する必要があります。

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。

C5102 (E) Forward declaration of enum type is nonstandard
enum 型の前方宣言は標準形式ではありません。

C5103 (E) Class is too large
クラスのサイズが大きすぎます。

C5104 (E) Struct or union is too large
構造体または共用体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)(W) 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 (W) 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)(W) 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 (W) 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)(W) 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。

C5138 (E)(W) Taking the address of a register field is not allowed
レジスタフィールドのアドレスを参照することはできません。

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)(W) Too many arguments in function call
関数呼び出しの実引数の数が多すぎます。

C5141 (E) Unnamed prototyped parameters not allowed when body is present
定義された関数の関数原型宣言のパラメータに名前がありません。

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)(W) 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。

C5152 (W) Conversion of nonzero integer to pointer
ゼロ以外の整数をポインタに変換しようとしました。

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。

C5155 (W) Old-fashioned assignment operator
古いスタイルの代入オペレータが使用されました。

C5156 (W) Old-fashioned initializer
古いスタイルの初期化子が使用されました。

C5157 (E)(W) 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"と名前が重複しています。

C5161 (W) 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)(W) 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 (W)(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)(W) 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 (E)(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) Entity-kind "名前" was declared but never referenced
参照されない宣言があります。

C5178 (W) "&" applied to an array has no effect
配列名の前に"&"があります。無視します。

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。

C5180 (W)(I) Argument is incompatible with formal parameter
引数が古い形式のパラメータと合致しません。

C5181 (W) Argument is incompatible with corresponding format string conversion
引数が対応する文字列変換形式と合致しません。

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と符号な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。

C5188 (W) Enumerated type mixed with another type
列挙型が他の列挙型またはデータ型に変換されています。

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。

C5190 (F) Invalid intermediate language file
不正な中間言語ファイルです。

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。

C5194 (E) Expected an asm string
asm文字列が必要です。

C5195 (E) An asm function must be prototyped
asm 関数は関数原型宣言されている必要があります。

C5196 (E) An asm function may not have an ellipsis
asm 関数のパラメータに省略記号(…)は使用できません。

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。

C5220 (E) Integral value does not fit in required floating-point type
整数値を要求された浮動小数点型に変換できません。

C5221 (E) 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。

C5222 (E) Floating-point operation result is out of range
浮動所数点演算の結果が値の範囲を超えました。オーバーフローした上位ビットを無視した値を仮定します。

C5223 (W) Function 関数名 declared implicitly
関数が暗黙的に宣言されました。

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。

C5227 (E) Macro recursion
再帰的なマクロの展開レベルが 300 を超えています。

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。

C5230 (W) Nonstandard type for a bit field
ビットフィールドとして標準形式でないデータ型を使用しています。

C5231 (W) Declaration is not visible outside of function
関数原型宣言内のタイプ宣言は関数の外からは見えません。

C5232 (W) Old-fashioned typedef of "void" ignored
古い形式である void の typedef は無効になります。

C5233 (W) Left operand is not a struct or union containing this field
左オペランドの構造体または共用体には無いフィールドを指定しました。

C5234 (W) Pointer does not point to struct or union containing this field
ポインタの指す構造体または共用体には無いフィールドを指定しました。

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。

C5260 (W) Explicit type is missing ("int" assumed)
型を指定していません。int 型を仮定します。

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)(W) 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"が曖昧です。

C5268 (E) Declaration may not appear after executable statement in block
宣言がブロックの実行文の後にありません。

C5269 (E) Conversion to inaccessible base class "型" is not allowed
参照不可能な基底クラス "型" に変換できません。

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 (E)(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 "型" is ambiguous
基底クラスの型が曖昧です。

C5287 (E) Derived class "型" 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曖昧です。

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。

C5296 (E)(W) Invalid use of non-lvalue array
左辺値でない配列の使用が不正です。

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)(W) 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。

C5301 (E) Typedef name has already been declared (with same type)
typdef の名前がすでに同じタイプで定義されています。

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 "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"が曖昧です。

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 cv-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"の置き換えが曖昧です。

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)(W) "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 "型" 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) (W) 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"のオペランドが曖昧です。

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 "型" 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 (W) "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) In-class initializer for nonstatic member is nonstandard
非スタティックなメンバを初期化するのは標準形式ではありません。

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。

C5390 (E) Function "main" may not be called or have its address taken
main関数の呼び出し、またはアドレスの取得を行ってはいけません。

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。

C5398 (W) Cast to array type is nonstandard (treated as cast to “型”)
配列型へのキャストは標準形式ではありません。(“型”へのキャストと仮定します)

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) Invalid redeclaration of member “関数名”
メンバ関数の不正な再宣言です。

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曖昧です。

C5417 (E) More than one conversion function from "型1" to "型2" applies:
"型1"から"型2"への変換関数が曖昧です。

C5418 (E) More than one conversion function from "型" to a built-in type applies:
"型"から組み込み型への変換関数が曖昧です。

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2"の引数"名前1"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"が曖昧です。

C5450 (E) The type "long long" is nonstandard
long long 型は標準形式ではありません。

C5451 (E) Omission of “class” is nonstandard
“class”の無いfriend宣言は標準形式ではありません。

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。

C5460 (W) Declaration of “変数名” hides function parameter
関数内の変数宣言が関数の引数を隠しました。

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 "名前" 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)(I) 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。

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。

C5494 (E)(W) Declaring a void parameter list with a typedef is nonstandard
typedef された void パラメータリストを宣言するのは標準形式ではありません。

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。

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"が曖昧です。

C5504 (E) Nonstandard form for taking the address of a member function
メンバ関数のアドレスを取得するのは標準形式ではありません。

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)(W) 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。
(W) 型1と型2がそれぞれ、互いに互換性のない型へのポインタ。
(E) 型1と型2が、互いに互換性のない型。

C5514 (W) Pointless comparison of unsigned integer with a negative constant
負の定数と符号な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。

C5518 (E) Nonstandard preprocessing directive
標準形式ではないプリプロセッサのキーワードがあります。

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)(W) 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。

C5523 (W) "." used in place of "::" to form a qualified name
“.” が スコープ解決子 ”::” の代わりに使用されています。

C5525 (W) A dependent statement may not be a declaration
条件式はスコープを持ちません。

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 (W) 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。

C5560 (E) “キーワード” is reserved for future use as a keyword
キーワードは将来実装される予約語です。

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。

C5599 (E) Excessive recursive instantiation of entity-kind “名前” due to instantiate-all mode
instantiate-allモードの指定によってテンプレート”名前”のインスタンスが再帰的に生成されます。

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。

C5605 (E) Floating-point template parameter is nonstandard
浮動小数点のテンプレートパラメータは標準形式ではありません。

C5606 (E) This pragma must immediately precede a declaration
このpragmaは宣言の前に記述しなければいけません。

C5607 (E) This pragma must immediately precede a statement
このpragmaは式の直前に記述しなければいけません。

C5608 (E) This pragma must immediately precede a declaration or statement
このpragmaは宣言または式の直前に記述しなければいけません。

C5609 (E) This kind of pragma may not be used here
この種類のpragmaはここで使用してはいけません。

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。

C5615 (E) Parameter type involves pointer to array of unknown bound
引数の型に要素数の指定がない配列へのポインタがふくまれています。

C5616 (E) Parameter type involves reference to array of unknown bound
引数の型に要素数の指定がない配列への参照が含まれています。

C5617 (W) Pointer-to-member-function cast to pointer to function
メンバ関数ポインタを関数ポインタにキャストしています。

C5618 (I) Struct or union declares no named members
構造体または共用体に名前付きのメンバが含まれていません。

C5619 (E) Nonstandard unnamed field
標準形式ではない名前の無いフィールドです。

C5620 (E) Nonstandard unnamed member
標準形式ではない名前の名前の無いメンバです。

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。

C5643 (E) “restrict” is not allowed
“restrict”を指定することはできません。

C5644 (E) A pointer or reference to function type may not be qualified by "restrict"
関数へのポインタまたは参照型は”restrict”によって修飾してはいけません。

C5647 (E) Conflicting calling convention modifiers
呼び出し規約修飾子が競合しています。

C5650 (W) Calling convention specified here is ignored
ここで指定された呼び出し規約は無視されます。

C5651 (E) A calling convention may not be followed by a nested declarator
呼び出し規約の後にネストされた宣言子が続いてはいけません。

C5652 (I) Calling convention is ignored for this type
この型に対する呼び出し規約は無視されます。

C5654 (E) Declaration modifiers are incompatible with previous declaration
宣言子が前に宣言されたものと互換性がありません。

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。

C5661 (E) Expected an integer constant
整数定数がありません。

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。

C5665 (E) "asm" is not allowed
asm 指定子は使用できません。

C5666 (E) "asm" must be used with a function definition
asm 指定子は関数定義と共に指定してください。

C5667 (E) "asm" function is nonstandard
asm 関数は標準形式ではありません。

C5668 (E) Ellipsis with no explicit parameters is nonstandard
省略指定(...)のみのパラメータは標準形式ではありません。

C5669 (E) "&..." is nonstandard
“&...” のパラメータは標準形式ではありません。

C5670 (E) Invalid use of "&..."
“&...” が不正に使われています。

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。

C5676 (W) Using out-of-scope declaration of “シンボル名”
Using 宣言がシンボルのスコープ外です。

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。

C5691 (E)(W) “シンボル”, required for copy that was eliminated, is inaccessible
コピーコンストラクタにアクセスできません。

C5692 (E)(W) “シンボル”, required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue
コピーコンストラクタを呼び出すことができません。

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。

C5731 (E) Array with incomplete element type is nonstandard
不完全な要素型を持つ配列は標準形式ではありません。

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。

C5748 (W) Calling convention specified more than once
呼び出し規約が1回以上指定されています。

C5749 (E) A type qualifier is not allowed
型修飾子を指定できません。

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。

C5758 (E) Explicit instantiation is not allowed in the current scope
現在のスコープ範囲でインスタンスを明示的に生成することはできません。

C5759 (E) “シンボル名” cannot be explicitly instantiated in the current scope
シンボルは現在のスコープで明示的にインスタンス化できません。

C5760 (W) “シンボル” explicitly instantiated more than once
シンボルを具現化できませんでした。

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。

C5765 (E) Nonstandard character at start of object-like macro definition
非標準の文字列がオブジェクト的マクロ定義の始まりに含まれています。

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。

C5769 (E) “シンボル1”, implicitly called from “シンボル2”, is ambiguous
operator delete の呼び出しが曖昧です。

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。

C5780 (W) Reference is to “シンボル1” -- under old for-init scoping rules it would have been “シンボル2”
“シンボル1”を参照しています。

C5782 (E) Definition of virtual entity-kind "名前" is required here
仮想関数の定義"名前"が必要です。

C5783 (W) Empty comment interpreted as token-pasting operator "##"
空のコメントは字句連結オペレータ"##"と仮定します。

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 "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。

C5799 (E) Specializing “シンボル名” without "template<>" syntax is nonstandard
“template<>”なしでシンボルを特殊化するのは標準形式ではありません。

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) Unexpected end of default argument expression
デフォルト引数式が正しくありません。

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 (E)(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。

C5816 (E) In a function definition a type qualifier on a "void" return type is not allowed

関数定義において“void”型の戻り値に型修飾子を指定することはできません。

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。

C5819 (E) "..." is not allowed

“...” は使用できません。

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。

C5828 (E) Support for "new[]" and "delete[]" is disabled

new[] と delete[] はサポートされていません。

C5829 (W) "double" used for "long double" in generated C code

C コード生成時に ”long double” は “double” に変換されます。

C5830 (W) "シンボル” has no corresponding operator deletes (to be called if an exception is thrown during initialization of an allocated object)

対応する operator delete がありません。

C5831 (W)(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。

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)(W) 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。

C5863 (I) Effect of this "#pragma pack" directive is local to “シンボル”
#pragma pack ディレクティブの影響はシンボル内にとどまります。

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。

C5866 (I) Exception specification ignored
例外指定は無視されます。

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。

C5869 (E) Could not set locale to allow processing of multibyte characters
多バイト文字にロケール設定ができませんでした。

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。

C5872 (E) Ambiguous guiding declaration -- more than one function template no matches type “型”
テンプレート関数が曖昧です。

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++仕様は新形式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。

C5886 (E) Invalid suffix on integral constant
整数定数への接尾辞が不正です。

C5890 (E) Variable length array with unspecified bound is not allowed
可変長配列に大きさが指定されていません。

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。

C5892 (E) An entity with linkage cannot have a type involving a variable length array
リンケージ指定子がある宣言は可変長配列を含む型を持つことはできません。

C5893 (E) A variable length array cannot have static storage duration
可変長配列は静的記憶期間を持つことができません。

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。

C5907 (E) Option "nonstd_qualifier_deduction" can be used only when compiling C++
“nonstd_qualifier_deduction”オプションはC++コンパイル時のみ使用できます。

C5912(W) Ambiguous class member reference – “シンボル1” used in preference to “シンボル2”
曖昧なクラスメンバの参照です。シンボル1をシンボル2に優先して参照します。

C5915 (E) A segment name has already been specified
すでに指定されたセグメント名です。

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。

C5925 (W) Type qualifiers on function types are ignored
関数型への型修飾子を無視します。

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。

C5934 (E) A member with reference type is not allowed in a union
参照型は共用体のメンバにできません。

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。

C5938 (E) Return type "int" omitted in declaration of function "main"
int 型の戻り値は main 関数の宣言において除外されます。

C5939 (E) pointer-to-member representation “シンボル1” is too restrictive for “シンボル2”
メンバへのポインタの宣言が正しくありません。

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。

C5942 (W) enum bit-fields are always unsigned, but enum “名前” includes negative enumerator
列挙型のビットフィールドは常に unsigned ですが、列挙型 “名前” には値が負の列挙定数が含まれています。

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。

C5948 (E)(W) Nonstandard local-class friend declaration -- no prior declaration in the enclosing scope
非標準形式のローカルクラスのフレンド宣言です。クラスの定義内に前方宣言がありません。

C5949 (I) Specifying a default argument on this declaration is nonstandard
この宣言にデフォルト引数を指定するのは標準形式ではありません。

C5951 (E)(W) Return type of function "main" must be "int"
main 関数の戻り値は int でなければいけません。

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。

C5955 (E) Ordinary and extended designators cannot be combined in an initializer designation

指示子が正しくありません。

C5956 (E) The second subscript must not be smaller than the first

2 番目の添え字は 1 番目の添え字より大きくなければいけません。

C5959 (W) Declared size for bit field is larger than the size of the bit field type; truncated to "ビット数" bits

指定されたビット数がビットフィールドの型の"ビット数"を超えています。ビット数をビットフィールドの型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。

C5969 (E) The identifier __VA_ARGS__ can only appear in the replacement lists of variadic macros
__VA_ARGS__識別子 は可変個引数を持つマクロの置換リスト内以外に記述できません。

C5970 (W) The qualifier on this friend declaration is ignored
このフレンド宣言への修飾子は無視されます。

C5971 (E) Array range designators cannot be applied to dynamic initializers
配列範囲名は動的初期化子に適用できません。

C5972 (E) Property name cannot appear here
プロパティ名はここに存在できません。

C5973 (W) "inline" used as a function qualifier is ignored
関数修飾子として使用された"inline"を無視します。

C5975 (E) A variable-length array type is not allowed
可変長配列型は使用できません。

C5976 (E) A compound literal is not allowed in an integral constant expression
複合リテラルは整数定数式で使用することはできません。

C5977 (E) A compound literal of type "型" is not allowed
指定の複合リテラル型は使用できません。

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。

C5983 (E) typedef name has already been declared (with similar type)
typedef名はすでに同等の型で宣言されています。

C5984 (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。

C5985 (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。

C5987 (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。

C5988 (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。

C5989 (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。

C5990 (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。

C5991 (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。

C5992 (E) Invalid macro definition:
不正なマクロ定義です。

C5993 (W) Subtraction of pointer types “シンボル名1” and “シンボル名2” is nonstandard
ポインタ型のシンボル1とシンボル2の減算は標準形式ではありません。

C5994 (E) An empty template parameter list is not allowed in a 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。

C5995 (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。

C5996 (E) The "class" keyword must be used when declaring a 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。

C5997 (W) "関数名1" is hidden by "関数名2" -- virtual function override intended?
"関数名1"が"関数名2"を隠しています。仮想関数をオーバーライドしようとしていないか確認してください。

C5998 (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。

C5999 (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。

C6000 (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。

C6001 (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。

C6006 (E) A 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。

C6007 (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。

C6009 (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。

C6010 (E) Destructor name must be qualified
不正なデストラクタ名です。

C6011 (E) Friend class name may not be introduced with "typename"
フレンドクラスの名前を ”typename” に続けて記述してはいけません。

C6012 (E) A using-declaration may not name a constructor or destructor
using 宣言でコンストラクタまたはデストラクタを指定してはいけません。

C6013 (E) A qualified friend template declaration must refer to a specific previously 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。

C6014 (E) Invalid specifier in class template declaration
不正な指定子がクラステンプレート宣言に含まれています。

C6015 (E) Argument is incompatible with formal parameter
引数が定義された引数と互換性がありません。

C6017 (E) Loop in sequence of "operator->" functions starting at class “シンボル”
operator->が正しくありません。

C6018 (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。

C6019 (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。

C6020 (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。

C6021 (E) Access declarations and using-declarations cannot appear in unions
union で using 指定は使えません。

C6022 (E) "名前" is not a class member
クラスのメンバではありません。

C6023 (E) Nonstandard member constant declaration is not allowed
非標準形式の const メンバは宣言することができません。

C6028 (W) Invalid redeclaration of nested class
クラス内でクラスを二重定義しています。

C6029 (E) Type containing an unknown-size array is not allowed
サイズが未定の配列を持つ構造体または共用体はメンバにできません。

C6030 (W) A variable with static storage duration cannot be defined within an inline function
静的なスコープを持つ変数はインライン関数内に宣言できません。

C6031 (W) An entity with internal linkage cannot be referenced within an inline function with external linkage
内部リンケージを持つ識別子は外部リンケージを持つインライン関数内で参照することはできません。

C6032 (E) Argument type “型” does not match this type-generic function macro
引数の型がジェネリック関数生成マクロの型に合いません。

C6034 (E) Friend declaration cannot add default arguments to previous declaration
フレンド関数が宣言された場合、フレンド関数の定義にデフォルト引数をいれることはできません。

C6035 (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。

C6036 (E) The reserved identifier “シンボル” may only be used inside a function
関数外で__FUNC__を使用しています。

C6037 (E) This universal character cannot begin an identifier
この汎用文字で識別子名を始めることはできません。

C6038 (E) Expected a string literal
文字列リテラルがありません。

C6039 (E) Unrecognized STDC pragma
認識できない STDC プラグマです。

C6040 (E) Expected "ON", "OFF", or "DEFAULT"
“ON”、”OFF”、”DEFAULT”がありません。

C6041 (E) A STDC pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope

STDC プラグマが現れるのはグローバルスコープ内の宣言の間、いかなる式の間またはブロックスコープ内の宣言の間だけです。

C6042 (E) Incorrect use of va_copy

va_copy マクロの使用方法が不正です。

C6043 (E) “型” can only be used with floating-point types

“型"が浮動小数点型以外の型と使用しています。

C6044 (E) Complex type is not allowed

複素数型を使えません。

C6045 (E) Invalid designator kind

不正なフィールド識別子です。

C6046 (W) Floating-point value cannot be represented exactly

浮動小数点数値に誤差が生じています。

C6047 (E) Complex floating-point operation result is out of range

複素数型浮動小数点演算の結果が表現可能な値の範囲を超えました。

C6048 (E) Conversion between real and imaginary yields zero

実数と虚数の相互変換後の値が 0 になりました。

C6049 (E) An initializer cannot be specified for a flexible array member

可変長配列メンバに初期化子を指定することはできません。

C6050 (W) imaginary *= imaginary sets the left-hand operand to zero

虚数 *= 虚数は左辺値を 0 にします。

C6051 (E)(W) Standard requires that “シンボル” be given a type by a subsequent declaration ("int" assumed)

暗黙の型は使用できません。

C6052 (E) A definition is required for inline “シンボル”

インライン関数の定義がありません。

C6053 (W) Conversion from integer to smaller pointer

整数がより小さいサイズのポインタへ変換されました。

C6054 (E) A floating-point type must be included in the type specifier for a _Complex or _Imaginary type

浮動小数点型は複素数または虚数型の指定子に含まれてなければいけません。

C6055 (E) Types cannot be declared in anonymous unions
型を無名共用体内で宣言することはできません。

C6056 (W) Returning pointer to local variable
ローカル変数へのポインタを返しています。

C6057 (W) Returning pointer to local temporary
ローカルな領域へのポインタを返しています。

C6061 (E) Declaration of “シンボル名” is incompatible with a declaration in another translation unit
“シンボル名”の宣言はもう一つの翻訳単位内の宣言と互換性がありません。

C6062 (E) The other declaration is "行”
別の宣言があります。

C6065 (E) A field declaration cannot have a type involving a variable length array
フィールド宣言は可変長配列が存在する型を含むことができません。

C6066 (E) declaration of “インスタンス” had a different meaning during compilation of “シンボル”
コンパイル時に宣言が異なっています。

C6067 (E) Expected "template"
“template”がありません。

C6072 (E)(W) A declaration cannot have a label
宣言はラベルを持つことはできません。

C6075 (E) “インスタンス名” already defined during compilation of “シンボル”
コンパイル時にすでに定義されています。

C6076 (E) “シンボル” already defined in another translation unit
すでに別の翻訳単位で定義されています。

C6081 (E) A field with the same name as its class cannot be declared in a class with a user-declared constructor
クラス名と同じ名前のメンバを宣言することはできません。

C6083 (F) Exported template file ファイル名 is corrupted
エクスポートされたテンプレートファイルは破損しています。

C6086 (E) the object has cv-qualifiers that are not compatible with the member “シンボル”
オブジェクトの持つ CV 修飾子はメンバ“シンボル”と互換性がありません。

C6087 (E) No instance of “クラス名” matches the argument list and object (the object has cv-qualifiers that prevent a match)

“クラス名”のインスタンスは引数リストとオブジェクトと合致しません。(オブジェクトの持つ CV 修飾子が合致を抑止しています)

C6089 (E) There is no type with the width specified

幅が指定された型がありません。

C6105 (W) #warning directive: “文字列”

“文字列”を出力しました。

C6139 (E) The "template" keyword used for syntactic disambiguation may only be used within a template

キーワード”template”を構文上の曖昧さを解消するのに使用できるのは template 内のみです。

C6144 (E) Storage class must be auto or register

記憶クラスは auto または register でなければいけません。

C6145 (W) “型1” would have been promoted to “型2” when passed through the ellipsis parameter; use the latter type instead

型1は型2へと拡張されます。型2を使用します。

C6146 (E) “シンボル” is not a base class member

基底クラスのメンバではありません。

C6151 (F) Mangled name is too long

マングルされた名前が長すぎます。

C6158 (E) void return type cannot be qualified

void 型の戻り値は修飾できません。

C6161 (E) A member template corresponding to “シンボル” is declared as a template of a different kind in another translation unit

テンプレート宣言が他コンパイル単位と異なっています。

C6163 (E) va_start should only appear in a function with an ellipsis parameter

va_start が使用されるのは省略記号を引数とする関数のみです。

C6192 (W) Null (zero) character in input line ignored

入力ライン中の null 文字が無視されました。

C6193 (W) Null (zero) character in string or character constant

文字列または文字定数内に null 文字が含まれています。

C6194 (W) Null (zero) character in header name

ヘッダ名に null 文字が含まれています。

C6197 (W) The prototype declaration of “シンボル” is ignored after this unprototyped redeclaration
関数原型を無視します。

C6201 (E) Typedef “シンボル” may not be used in an elaborated type specifier
詳述型指定子に使用できません。

C6203 (E) Parameter “引数名” may not be redeclared in a catch clause of function try block
“引数名”を try ブロックの catch 句の中で再宣言してはいけません。

C6204 (E) The initial explicit specialization of “シンボル名” must be declared in the namespace containing the template
シンボルに対する最初の明示的な特殊化はテンプレートを含む名前空間の中に宣言されなければいけません。

C6206 (E) "template" must be followed by an identifier
“template”の後には識別子が必要です。

C6211 (W) Nonstandard cast to array type ignored
非標準形式の配列型へのキャストが無視されました。

C6212 (E) This pragma cannot be used in a _Pragma operator (a #pragma directive must be used)
このプラグマは_Pragma operator 内では使用できません。(#pragma ディレクティブを使用してください)

C6213 (W) Field uses tail padding of a base class
フィールドは基底クラスの終端パディングを使用しています。

C6218 (W) Base class “クラス名1” uses tail padding of base class “クラス名2”
基底クラス１は基底クラス２の終端パディングを使用しています。

C6222 (W) Invalid error number
不正なエラー番号です。

C6223 (W) Invalid error tag
不正なエラータグです。

C6224 (W)Expected an error number or error tag
エラー番号またはエラータグがありません。

C6227 (E) Transfer of control into a statement expression is not allowed
式文への制御の転移はできません。

C6229 (E) This statement is not allowed inside of a statement expression
この式は式文内にあってはいけません。

C6230 (E) A non-POD class definition is not allowed inside of a statement expression
非 POD クラスは式文内に定義できません。

C6235 (W) Nonstandard conversion between pointer to function and pointer to data
非標準形式の変換がポインタ関数と不完全なオブジェクト間で行われました。

C6254 (E) Integer overflow in internal computation due to size or complexity of “型”
データ型のサイズまたは複雑さに伴い、内部の計算結果にて整数のオーバーフローが発生しました。

C6255 (E) Integer overflow in internal computation
内部の計算結果にて整数のオーバーフローが発生しました。

C6273 (W) Alignment-of operator applied to incomplete type
オペレータのアライメントが不完全な型に対して適用されました。

C6280 (E) Conversion from inaccessible base class “クラス名” is not allowed
派生クラスにプライベートで継承された基底クラス型のポインタを継承クラス型のポインタへ変換することはできません。

C6282 (E) String literals with different character kinds cannot be concatenated
違う種類の文字列リテラルを連結することはできません。

C6285 (W) Nonstandard qualified name in namespace member declaration
非標準形式の修飾子名が名前空間のメンバの宣言に使用されています。

C6290 (W) Non-POD class type passed through ellipsis
非 POD クラス型が省略記号に渡されています。

C6291 (E) A non-POD class type cannot be fetched by va_arg
非 POD 型のクラスは va_arg によって取得することができません。

C6292 (E) The 'u' or 'U' suffix must appear before the 'l' or 'L' suffix in a fixed-point literal
固定小数点リテラルにおいて、‘u’ または ’U’ 型の接尾辞は ‘l’ または ‘L’ の接尾辞の前に現れなければいけません。

C6294 (W) Integer operand may cause fixed-point overflow
整数オペランドは固定小数点オーバーフローを起こす可能性があります。

C6295 (E) Fixed-point constant is out of range
固定小数点定数が表現可能な範囲を超えています。

C6296 (W) Fixed-point value cannot be represented exactly
固定小数点では 16 進数表記を完全に表現することができません。

C6297 (W) Constant is too large for long long; given unsigned long long type (nonstandard)
定数は long long 型としては大きすぎます。 Unsigned の long long 型に変更します。(非標準形式)

C6301 (W) “シンボル” declares a non-template function -- add <> to refer to a template instance
非テンプレート関数を宣言しています。

C6302 (W) Operation may cause fixed-point overflow
演算によって固定小数点オーバーフローが起こる可能性があります。

C6303 (E) Expression must have integral, enum, or fixed-point type
式には整数型、列挙型または固定小数点型を含んでください。

C6304 (E) Expression must have integral or fixed-point type
式には整数型または固定小数点型を含んでください。

C6307 (W) Class member typedef may not be redeclared
クラスメンバの typedef を再宣言してはいけません。

C6308 (W) Taking the address of a temporary
ローカルな領域のアドレスを取得しています。

C6310 (W) Fixed-point value implicitly converted to floating-point type
固定小数点値が浮動小数点型に暗黙的に変換されました。

C6311 (E) Fixed-point types have no classification
浮動小数点型の区分がありません。

C6312 (E) A template parameter may not have fixed-point type
テンプレート引数には固定小数点型を指定できません。

C6313 (E) Hexadecimal floating-point constants are not allowed
16 進数の浮動小数点定数は使用できません。

C6315 (E) Floating-point value does not fit in required fixed-point type
浮動小数点数値は要求された固定小数点型に収まりません。

C6316 (W) Value cannot be converted to fixed-point value exactly
値を固定小数点値にすると誤差が生じます。

C6317 (E) Fixed-point conversion resulted in a change of sign
負の整数値を固定小数点型へ変換した結果、正の値になりました。

C6318 (E) Integer value does not fit in required fixed-point type
整数値は要求された固定小数点型に収まりません。

C6319 (E)(W) Fixed-point operation result is out of range
固定小数点演算の結果が表現可能な値の範囲をこえました。

C6320 (E) Multiple named address spaces
同一の名前アドレス空間が複数存在します。

C6321 (E) Variable with automatic storage duration cannot be stored in a named address space
局所的なスコープを持つ変数は名前付きアドレス空間に保持することはできません。

C6322 (E) Type cannot be qualified with named address space
名前付きアドレス空間によって型を識別することはできません。

C6323 (E) Function type cannot be qualified with named address space
名前付きアドレス空間によって関数型を識別することはできません。

C6324 (E) Field type cannot be qualified with named address space
フィールド型は名前付き空間によって識別することはできません。

C6325 (E) Fixed-point value does not fit in required floating-point type
固定小数点値は要求された浮動小数点型に収まりません。

C6326 (E) Fixed-point value does not fit in required integer type
固定小数点値は要求された整数型に収まりません。

C6327 (E) Value does not fit in required fixed-point type
値は要求された固定小数点値に収まりません。

C6335 (F) Cannot open predefined macro file: “ファイル名”
定義済みマクロファイルを開けません。

C6336 (F) Invalid predefined macro entry at line “行数”: “マクロ名”
不正な定義済みマクロの entry 宣言が ”行数” にあります。

C6337 (F) Invalid macro mode name “マクロモード名”
不正なマクロモード名です。

C6338 (F) Incompatible redefinition of predefined macro “マクロ名”
互換性の無い定義済みマクロの再定義です。

C6342 (W) const_cast to enum type is nonstandard
const_cast で列挙型をキャストするのは標準形式ではありません。

C6344 (E) A named address space qualifier is not allowed here
名前付きアドレス空間識別子はここで使用できません。

C6345 (E) An empty initializer is invalid for an array with unspecified bound
空の初期化子で境界が指定されていない配列を初期化するのは不正です。

C6346 (W) Function returns incomplete class type “クラス名”
関数が不正なクラス型を返しています。

C6348 (I) Declaration hides “変数名”
局所変数が他の局所変数の宣言によって隠蔽されました。

C6349 (E) A parameter cannot be allocated in a named address space
引数は名前付きアドレス空間に配置できません。

C6350 (E) Invalid suffix on fixed-point or floating-point constant
不正な接尾辞が固定または浮動小数点定数についています。

C6351 (E) A register variable cannot be allocated in a named address space
レジスタ変数は名前付きアドレス空間に配置できません。

C6352 (E) Expected "SAT" or "DEFAULT"
“SAT”または”DEFAULT”がありません。

C6353 (I) “シンボル名” has no corresponding member operator delete “シンボル名” (to be called if an exception is thrown during initialization of an allocated object)
シンボルは new オペレータの対となる delete オペレータを持ちません。(取得したオブジェクトの初期化時に例外が発生した場合に呼ばれます。)

C6355 (E) A function return type cannot be qualified with a named address space
関数の戻り値を名前付きアドレス空間で修飾することはできません。

C6361 (W) Negation of an unsigned fixed-point value
符号なしの固定小数点を無効にします。

C6365 (E) Named-register variables cannot have void type
名前付きレジスタ変数は void 型にできません。

C6372 (E) Nonstandard qualified name in global scope declaration
非標準形式の修飾された名前がグローバルなスコープに宣言されています。

C6373 (W) Implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
64 ビット整数型がより小さい整数型へと暗黙的に変換されています。移植性の問題になる可能性があります。

C6374 (W) Explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
64 ビット整数型がより小さい整数型へと明示的に変換されています。移植性の問題になる可能性があります。

C6375 (W) Conversion from pointer to same-sized integral type (potential portability problem)
ポインタから同サイズの整数型へと変換しています。移植性の問題になる可能性があります。

C6380 (E)(I) Virtual “関数名” was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace)
仮想関数の定義がありません。また、無名空間のメンバである為それ以外の場所で定義することができません。

C6381 (E)(I) Carriage return character in source line outside of comment or character/string literal
改行文字がコメントまたは文字列リテラル以外のところにあります。

C6382 (E) Expression must have fixed-point type
式に固定小数点を含めなくてはいけません。

C6386 (W) Storage specifier ignored
記憶クラス指定子を無視します。

C6396 (W) White space between backslash and newline in line splice ignored
行接合部のバックスラッシュと改行の間の空白を無視します。

C6398 (E) Invalid member for anonymous member class -- class “シンボル” has a disallowed member function
無名のメンバクラスに対して不正なメンバ関数を宣言しています。

C6400 (W) Positional format specifier cannot be zero
位置フォーマット指定子に0を指定することはできません。

C6403 (E) A variable-length array is not allowed in a function return type
可変長配列を関数の戻り値型とすることはできません。

C6404 (E) Variable-length array type is not allowed in pointer to member of type “型”
クラスメンバへのポインタとして可変長配列型メンバへのポインタは禁止されています。

C6405 (E) The result of a statement expression cannot have a type involving a variable-length array
式文の演算結果に可変長配列型が含まれてはいけません。

C6420 (E)(W) Some enumerator values cannot be represented by the integral type underlying the enum type
整数型で表せない列挙値です。

C6421 (E) Default argument is not allowed on a friend class template declaration
デフォルト引数をフレンドクラスのテンプレート宣言に指定することはできません。

C6422 (W) Multicharacter character literal (potential portability problem)
複数文字リテラルです。移植性の問題を引き起こす可能性があります。

C6424 (E) Second operand of offsetof must be a field
マクロ offsetof の二番目のオペランドはフィールドでなくてはいけません。

C6425 (E) Second operand of offsetof may not be a bit field
マクロ offsetof の二番目のオペランドはフィールドであってはいけません。

C6426 (E) Cannot apply offsetof to a member of a virtual base
マクロ offsetof を仮想基底クラスのメンバに適用することはできません。

C6427 (W) offsetof applied to non-POD types is nonstandard
マクロ offsetof を非 POD 型に適用するのは標準形式ではありません。

C6428 (E) Default arguments are not allowed on a friend declaration of a member function
デフォルト引数をフレンド宣言のメンバ関数に指定することはできません。

C6429 (E) Default arguments are not allowed on friend declarations that are not definitions
デフォルト引数を定義ではないフレンド宣言に指定することはできません。

C6430 (E) Redeclaration of “関数名” previously declared as a friend with default arguments is not allowed
デフォルト引数を持つフレンドとしてすでに宣言した関数を再宣言することはできません。

C6431 (E) Invalid qualifier for “シンボル” (a derived class is not allowed here)
限定子が正しくありません。

C6432 (E) Invalid qualifier for definition of class “クラス名”
不正な修飾子をクラスの定義に指定しました。

C6439 (E) Template argument list of “シンボル” must match the parameter list
テンプレート引数リストに合わなければなりません。

C6440 (E) An incomplete class type is not allowed
不完全なクラス型です。

C6445 (E) Invalid redefinition of “シンボル名”
列挙型が再定義されています。

C6449 (E) Explicit specialization of “シンボル” must precede its first use “シンボル2”
テンプレートをすでに具現化しています。

C6623 (W) The destructor for “クラス1” has been suppressed because the destructor for “クラス2” is inaccessible
クラス2のデストラクタにアクセスできないため、クラス1のデストラクタは抑制されました。

C6648 (W) '=' assumed following macro name “マクロ名” in command-line definition
コマンドライン定義内のマクロ名の後ろには'='がついているとみなします。

C6649 (E)(W) White space is required between the macro name “マクロ名” and its replacement text
”マクロ名” とその置換テキストの間には空白が必要です。

C6655 (E) “シンボル” cannot be declared inline after its definition “定義名”
inline が抑止されているため、シンボルは inline 関数として宣言することができません。

C6671 (W) __assume expression with side effects discarded
副作用のある __assume 式が破棄されました。

C6674 (E) __evenaccess qualifier is applied to only integer type
__evenaccess 修飾子は整数タイプのみに指定できます。

C6675 (E) Expected a section name string
__sectop/__secend/__secsize にセクション名がありません。

C6676 (E) Expected a section name
セクション名がありません。

C6677 (E) Invalid pragma declaration
#pragma の構文が不正です。

C6678 (E) "シンボル名" 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。

C6679 (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。

C6680 (E) Invalid kind of pragma is specified to this symbol
不正な#pragma を指定しました。

C6681 (I) This pragma has no effect
この#pragma は無効です。

C6682 (E) “シンボル名” must be qualified for function type
シンボルは関数型でなければいけません。

C6683 (E) Illegal “プラグマ名” specifier
不正な#pragma です。

C6684 (E) Multiple pointer qualifiers
ポインタ型修飾子が重複しています。

C6685 (E) __ptr16 must be qualified for data pointer type
__ptr16 はデータポインタ型以外を修飾できません。

C6686 (E) Invalid binary digit
不正な 2 進数です。

C6688 (E) "this" pointer of "クラス名" is cast implicitly to near pointer
“this”を暗黙的に near ポインタでキャストしました。

C6689 (E) Can not specify near or far for member
メンバ関数に対して near または far を指定することはできません。

C6690 (E) A member “関数名” qualified with near or far is declared
メンバ関数に対して near または far が指定されています。

C6691 (E) Near or far specifier on a reference type is not allowed
near または far 指定を参照タイプに指定することはできません。

C6692 (E) Can not specify near or far for member function
メンバ関数に対して near または far を指定することはできません。

C6693 (E) Can not specify near or far for function types
関数タイプに対して near または far を指定することはできません。

C6694 (E) "this" pointer offset is too large
this ポインタへのオフセットがオーバーフローしました。
該当クラスの基底クラスのサイズを小さくしてください。

C6695 (E) Number of virtual function is too many
仮想関数の数が上限値を超えています。
仮想関数の数を減らしてください。

C6696 (W) Assertion warning
#assert の定数式が 0 です。

C6697 (I) Enumeration type is signed
C++の列挙型は符号付きなので C と結果が異なる可能性があります。

付録G SBDATA 宣言&SPECIAL ページ関数宣言ユーティリティ(utl30)

SBDATA 宣言&SPECIAL ページ関数宣言ユーティリティ utl30 の起動方法と起動オプションの機能を説明します。

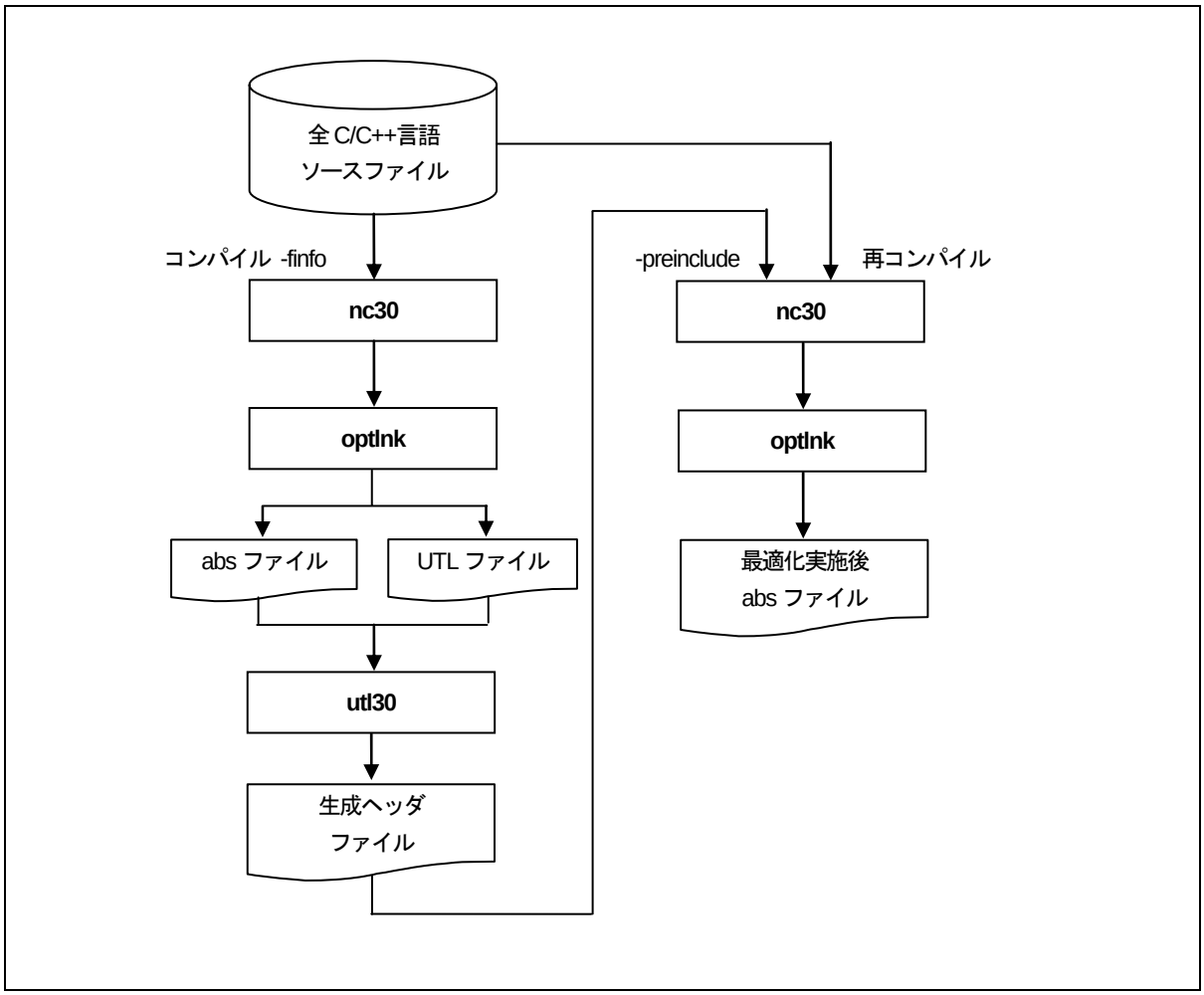
G.1 utl30 の概要

G.1.1 utl30 の処理概要

utl30 は、使用頻度の高い変数や関数を、自動的に SBDATA 宣言や SPECIAL ページ関数宣言にすることで、SB 領域やスペシャルページ領域に割り当てるための最適化機能を実現します。

【図G.1】はutl30 の処理フローを表しています。まず初めに、全てのC/C++言語ソースファイルをコンパイルオプション-finfoを使用しコンパイル・リンクを行います。リンク時にoptlnkへ-utlオプションを付けることでUTLファイル(拡張子.utl)を生成させることができます。

utl30 は、UTL ファイルとアブソリュートファイル(拡張子.abs)の両方を読み込み、最適化に必要なヘッダファイルを生成します。次に、全ての C/C++言語ソースファイルを再コンパイル・リンクします。再コンパイル時には-preinclude オプションを用いて utl30 が生成したヘッダファイルを指定してください。以上の手順により最適化されたアブソリュートファイル(.abs)を生成することができます。



図G.1　UTL30 の処理フロー

G.2 utl30 の起動方法

G.2.1 入力書式

utl30 を起動するためには、以下の図に示す書式に従って、必要な情報、パラメータを指定する必要があります。

```
% utl30△{ -sp30 | -sb30 } [△起動オプション ]△アブソリュートファイル

% : プロンプトを示します。
< > : 必須項目を示します。
[ ] : 必要に応じて記述する項目を示します。
△ : スペースを示します。
複数の起動オプションを記述する場合はスペースで区切ってください。
```

図G.2　utl30 コマンドの入力書式

utl30 を使用するためには、本コンパイラの起動オプションに、

- インスペクタ情報の出力..... -finfo オプション

を指定して下さい。以下に入力例を示します。

```
% nc30 -finfo ncrt0.a30 sample.c
                    -finfo オプションを付けてコンパイルします。

% optlnk -utl -output=samle.abs   ncrt0.obj sample.obj
                    abs と utl ファイルを生成させます。

% utl30 -sb30 -o samle sample.abs
                    utl30 を実行させ、ヘッダを生成させます。出力ファイル名に拡張子を付けないでください。

% nc30 -finfo -preinclude=sample.h ncrt0.a30 sample.c
                    preinclude オプションでヘッダを指定し、再コンパイルします。

% optlnk -output=sample.abs ncrt0.obj sample.obj
                    最適化実施後の abs ファイルを生成します。
```

図G.3　utl30 コマンドの入力例

G.2.2 出力情報の切り替え

utl30 で "SBDATA 宣言" と"SPECIAL ページ関数宣言"の出力を切り替えるには、以下のオプションを指定してください。どちらのオプションも指定しない場合はエラーになります。

(1)　SBDATA 宣言を出力
　　オプション"-sb30"

(2)　SPECIAL ページ関数宣言を出力
　　オプション "-sp30"

G.2.3　オプションリファレンス

-all

全変数の SBDATA 宣言または全関数への SPECIAL ページ関数宣言の出力

機　　能:
- オプション-sb30 と同時に使用する場合
  使用頻度が低いため、SB 領域に入らない変数に対しても、コメントの形で SBDATA 宣言を出力します。
- オプション-sp30 と同時に使用する場合
  使用頻度が低いため、SPECIAL ページ領域に入らない関数に対しても、コメントの形で SPECIAL 宣言を出力します。

補足説明:　本オプションを使用することにより、一度も呼出される事がない関数を見つけ出す事ができます。
ただし、間接でのみ呼出される関数は、呼出し回数が 0 回と表示されますので、お客様側で確認してください。

-fover_write　-fOW

SBDATA 宣言または SPECIAL 関数宣言のファイルへの出力

機　　能:　オプション-o で指定された出力ファイルが既に存在するか否かをチェックしません。ファイルが存在する場合は上書きします。
オプション-o と同時に指定してください。

-fsection

#pragma SECTION 内の SBDATA 宣言、SPECIAL ページ関数宣言の出力

機　　能:　処理の対象として、#pragma SECTION でセクション変更された領域に配置された変数および関数も含めます。

注　　意:　#pragma SECTION を使用して特定の変数および関数を、任意のアドレスへ意図的に配置する事を目的としている場合は、SBDATA 宣言あるいは SPECIAL ページ宣言により、意図したアドレスとは異なるアドレスへ配置されますので、本オプションは指定しないでください。

-o 出力ファイル名

SBDATA 宣言または SPECIAL 関数宣言のファイルへの出力

機　能:　SBDATA 宣言、または SPECIAL 関数宣言の結果をファイルに出力します。指定が無い場合は表示をホストマシンの標準出力に出力します。
また、指定されたファイルが既に存在する場合は、標準出力に出力します。
出力ファイル名には拡張子を付けないでください。
utl30 が自動的にヘッダ拡張子（.h）を付けて出力します。

-sb30

SBDATA 宣言の出力

機　能:　SBDATA 宣言を出力します。
オプション-sb30 またはオプション-sp30 のいずれかを指定してください。
指定がない場合はエラーになります。

-sp30

SPECIAL ページ関数宣言の出力処理

機　能:　SPECIAL ページ関数宣言を出力します。
オプション-sb30 またはオプション-sp30 のいずれかを指定してください。
指定がない場合はエラーになります。

-sp=*番号*

SPECIAL ページ関数として使用しない番号の指定

機　能:　SPECIAL ページ関数として使用しない番号を指定します。
オプション-sp30 と同時に使用してください。
複数指定する場合はカンマ、範囲を指定する場合はハイフンを使用できます。
＜書式例＞
-sp=18,19　　// 18 と 19 のスペシャルページベクタ番号を使用しません。
-sp=200-255　// 200 から 255 に含まれるスペシャルページベクタ番号を使用しません。

-Wstdout

エラーメッセージの標準出力への表示

機　能:　エラー、およびウォーニングメッセージをホストマシンの標準出力に出力します。

G.3 制限事項

(1) SBDATA 宣言を出力する場合、アセンブラで記述されたファイル中で宣言されているアセンブラ指示命令.sbsym をカウントすることができません。したがって指示命令.sbsym によって定義された外部変数がある場合は、utl30 実行後生成された結果に対して外部変数が SB 領域内に収容できるように調整する必要があります。

(2) SPECIAL ページ関数宣言を出力する場合、アセンブラで記述されたファイル中で宣言されている SPECIAL ページ関数をカウントすることができません。したがってアセンブラで宣言された SPECIAL ページ関数がある場合は、ut30 実行後生成された結果に対して SPECIAL ページ領域内に入るように調整する必要があります。

G.4 utl30 が処理対象とする変数および関数

G.4.1 SBDATA 宣言が処理対象とする変数

下記の条件のいずれかを満たす変数は、処理の対象外になります。

- #pragma SECTION で変更されたセクションに配置された変数
- #pragma ADDRESS で定義された変数
- コンパイルオプション-fconst_not_ROM(-fCNR)を使用しない場合の const 修飾変数

既に#pragma SBDATA を用いて宣言された変数がプログラム中に存在する場合は、utl30 はその宣言を優先し、SB 領域の残りから割り当てられる変数を選択します。

G.4.2 SPECIAL ページ関数宣言が処理対象とする関数

utl30 の処理対象となる関数は、以下の外部関数に対してのみです。

- static で宣言されていない関数
- 4 回以上呼ばれている関数

上記の関数でも下記の条件を満たす場合は、処理の対象外になります。

- #pragma SECTION で変更されたセクションに配置された関数
- 各#pragma で定義された関数

既に#pragma SPECIAL を用いて宣言された関数がプログラム中に存在する場合は、utl30 はその宣言を優先し、SPECIAL ページ領域の残りから割り当てられる関数を選択します。

G.5 utl30 の使用例

G.5.1 SBDATA 宣言の場合

a. SBDATA 宣言ファイルの出力

utl30 に-sb30 オプションを付けることで、SBDATA 宣言ファイルを出力することができます。

```
/*
 *      #pragma SBDATA Utility
 */
/* SBDATA Size [256] */
#pragma   SBDATA   gi                          /* size=(     2) / ref=[        2] / gi */
/*                                               (A)              (B)         (C)
 *      End of File
 */
```

(A) データのサイズを示します。
(B) データの使用頻度を示します。
(C) ソース上の名前を示します。

図G.4　-sb30 オプション指定時の生成されたヘッダ例

b. アセンブラで SB 宣言がある場合の調整

アセンブラ指示命令.sbsym で外部変数を定義している場合は、utl30 により生成されたヘッダファイルを調整する必要が有ります。

アセンブリ言語プログラム:

```
        .sbsym      _sym
            :
        (省略)
            :
        .glb        _sym
_sym:
        .blkb       2
```

utl30 生成ファイル:

```
/*
 * #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma SBDATA      data3           /*  size = (4) / ref = [2] / data3  */
#pragma SBDATA      data2           /*  size = (1) / ref = [1] / data2  */
            :
        (省略)
            :
#pragma SBDATA      data1           /*  size = (2) / ref = [1] / data1  */
/*
 * End of File
 */
```

アセンブラルーチン内で 2 バイトデータを SB 宣言しているため utl30 生成ファイルから 2 バイト分の SBDATA 宣言を削除します。

例:

```
            :
        (省略)
            :
//#pragma SBDATA    data1           /*  size = (2) / ref = [1] / data1  */
/*コメントアウト*/
```

図G.5　utl30 生成結果調整例

G.5.2　SPECIAL ページ関数宣言の場合

a.　SPECIAL ページ関数宣言ファイルの出力

utl30 に-sp30 オプションを付けることで、SPECIAL ページ関数宣言ファイルを出力することができます。

```
/*
 *      #pragma SPECIAL PAGE Utility
 */
/*      special page definition */
#pragma   SPECIAL   255   func()                    /* size=(     6) / ref=[        4] / func() */
/*                                                     (A)             (B)             (C)
 *      End of File
 */
(A) 関数のサイズを示します。
(B) 関数の参照頻度を示します。
(C) ソース上の名前を示します。
```

図G.6　-sp30 オプション指定時の生成されたヘッダ例

G.6 utl30 のエラーメッセージ

G.6.1 エラーメッセージ

【表G.1】【表G.2】にutl30 が出力するエラーメッセージとその内容及び対処方法を示します。

表G.1 utl30 エラーメッセージ一覧表 (1/2)

エラー番号	種別	メッセージ	エラー内容と対策
U2100	エラー	Illegal file extension '拡張子'	• 入力ファイルの拡張子が “abs” ではありません。 ⇒ 入力ファイルを確認してください。
U2101	エラー	Illegal file extension ''	• 入力ファイルの拡張子がありません。 ⇒ 正しいファイル名を指定してください。
U2200	エラー	ignore option '入力オプション名'	• 無効なオプションが入力されています。 ⇒ オプションを確認してください。
U2201	エラー	ignore option '-sp'	• -sb30 オプション選択時に-sp オプションが選択されています。 ⇒ -sp オプションは-sp30 オプションと同時に指定できます。
U2301	エラー	Option '-sp' is not appropriate	• -sp オプションの指定に数値以外の文字が含まれています。
U2402	エラー	No input 'abs' file specified	• 入力ファイルに abs ファイルが選択されていません。または、abs ファイルが読み込めません。
U2403	エラー	No input '入力 abs ファイル名' file specified	• abs ファイルが読み込めません。
U2600	エラー	'-SB30/-SP30' is missing	• -sb30 もしくは-sp30 オプションのどちらも選択されていません。 ⇒ どちらか一方を指定してください。
U2700	エラー	cannot open '入力 utl ファイル名' file	• utl ファイルを開くことができません。
U2701	エラー	cannot read header file '入力 abs ファイル名'	• abs ファイルに誤りがあります。壊れている可能性があります。
U2702	エラー	cannot read symbol table	• abs ファイルに誤りがあります。壊れている可能性があります。
U2703	エラー	cannot read section header	• abs ファイルに誤りがあります。壊れている可能性があります。
U2704	エラー	cannot read section data	• abs ファイルに誤りがあります。壊れている可能性があります。
U2705	エラー	cannot read ELF header	• abs ファイルに誤りがあります。壊れている可能性があります。
U2706	エラー	cannot open output file '出力ファイル名.h'	• 出力ファイルを開くことが出来ません。
U2707	エラー	cannot close file '出力ファイル名.h'	• 出力ファイルを閉じることが出来ません。
U2800	エラー	Illegal File Format '入力 utl ファイル名' file	• utl ファイルに誤りがあります。壊れている可能性があります。
U2801	エラー	Illegal File Format '入力 abs ファイル名'	• abs ファイルに誤りがあります。壊れている可能性があります。
U2802	エラー	Illegal file format	• abs ファイルに誤りがあります。壊れている可能性があります。
U2900	エラー	not enough memory	• メモリが不足しています。不要なアプリケーションを終了してください。
U0001	インフォメーション	Since '出力ファイル名.h' file exists, it makes a standard output.	• -o オプションで選択されたファイルが既に存在しています。

表G.2　utl30 エラーメッセージ一覧表 (2/2)

エラー番号	種別	メッセージ	エラー内容と対策
U1000	ウォーニング	warning：conflict declare of 変数名	• 該当する変数が異なるファイル間で、異なる記憶域クラス、型、などで宣言されています。
U1001	ウォーニング	warning：conflict declare of 関数名	• 該当する関数が異なるファイル間で、異なる記憶域クラス、型、などで宣言されています。

付録H ライブラリジェネレータ

ライブラリジェネレータ(lbg30)は、ユーザが指定したオプションに応じた標準ライブラリファイル(.lib)を作成するツールです。

作成した標準ライブラリを使ってリンクする場合は、以下のようにライブラリファイルを指定してください。

$ nc30 -fno_lib -l 生成された標準ライブラリ…

リンカから指定する場合

$ optlnk -library=生成された標準ライブラリ…

H.1 コマンド記述形式

% lbg30 [△オプション１] [△オプション２]

- オプション１には、「H.3 ライブラリジェネレータオプション」を指定します。
- オプション２には、「H.4 ライブラリジェネレータに指定可能なコンパイラオプション」を指定します。

例) lbg30 -output=mylib.lib -head=stdio -exception -rtti=on

H.2 ご使用にあたっての注意点

ライブラリジェネレータは、nc30 コンパイラを呼び出すので、コンパイラ実行に必要な環境変数をあらかじめ設定してください。ライブラリジェネレータは以下のフォルダを使用します。

- 環境変数 TMP30 で指定されたフォルダ
- カレントディレクトリ

ライブラリジェネレータは、これらのフォルダに書き込みを行いますので、フォルダに書き込み許可をあらかじめ設定してください。

H.3 ライブラリジェネレータオプション

ライブラリジェネレータのコマンドラインオプション「オプション1」に指定できるオプションを示します。

表H.1 ライブラリジェネレータオプション一覧

オプション	内容
-head=<sub>[,…]	**構築対象のライブラリを指定**
<sub>:{ all	**すべてのライブラリ関数とランタイムライブラリ**
| runtime	**ランタイムライブラリ**
| ctype	ctype.h(C89)**とランタイムライブラリ**
| math	math.h(C89)**とランタイムライブラリ**
| mathf	mathf.h(C89)**とランタイムライブラリ**
| stdarg	stdarg.h(C89)**とランタイムライブラリ**
| stdio	stdio.h(C89)**とランタイムライブラリ**
| stdlib	stdlib.h(C89)**とランタイムライブラリ**
| string	string.h(C89)**とランタイムライブラリ**
| ios	ios(EC++)**とランタイムライブラリ**
| new	new(EC++)**とランタイムライブラリ**
| complex	complex(EC++)**とランタイムライブラリ**
| cppstring	string(EC++)**とランタイムライブラリ**
}	
-output=<**ファイル名**>	**出力ライブラリファイル名を指定**
-nofloat	**簡易入出力関数の生成**

-head

書　式　-head=<sub>[,…]
<sub>:{ all
| runtime | ctype | math | mathf | stdarg | stdio | stdlib | string | ios | new | complex | cppstring
}

説　明　構築対象をヘッダファイル名で指定します。
-head=all を指定した場合、全てのヘッダファイル名が構築対象として指定されます。
ランタイムライブラリは常に構築対象になります。
本オプションの省略時解釈は、-head=all です。

-output

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。

-nofloat

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、コード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　本オプションを指定して作成したライブラリでは、対象関数で浮動小数点数の入出力をした場合の動作は保証しません。

H.4 ライブラリジェネレータに指定可能なコンパイラオプション

ライブラリジェネレータのコマンドラインオプション「オプション２」に指定できるオプションを示します。

表H.2 ライブラリジェネレータに指定可能なコンパイラオプション一覧

No.	オプション
1	-exception
2	-noexception
3	-rtti=on
4	-rtti=off
5	-O
6	-O1
7	-O2
8	-O3
9	-O4
10	-OR
11	-OS
12	-fdouble_32 (-fD32)
13	-fptrdifft_16 (-fP16)
14	-fsizet_16 (-fS16)
15	-R8C
16	-R8CE
17	-goptimize
18	-fno_align (-fNA)
19	-Ostack_frame_align (-OSFA)

付録I NC30 の C 言語の振舞い

この章では、ANSI 規格が示すところの「未定義の動作（undefined behavior ）」、「処理系定義の動作（implementation - defined behavior ）」、および「文化圏固有動作（locale - specific behavior ）」に対する C コンパイラ NC30 の C 言語での振舞いについて説明します。各説明は ANSI 規格「ANSI / ISO 9899－1990 」の節に対応付けて行います。

ANSI 規格では、「未定義の動作」、「処理系定義の動作」および「文化圏固有動作」を次のように定義しています。

a. 未定義の動作

移植性を考えていないプログラムやエラーのあるプログラム構造、エラーのあるデータ、または不定値を含んだオブジェクトなどを使用した場合、ANSI 規格の必要条件として課されていない振舞い。

b. 処理系定義の動作

正しいプログラム構成と正しいデータに対する動作で、そのインプリメンテーションの特性に依存する動作(各種コンパイラごとに規定される動作)。

c. 文化圏固有動作

国籍、文化、言語といった地域の習慣に依存した動作。

I.1 ANSI 規格未定義の動作

ANSI 規格で「未定義の動作」扱いとなっている動作は、C コンパイラでの動作が保証されません。ほとんどの場合、無視する、エラーなどの警告を出す、実行時にエラーが発生する、のいずれかとなる可能性があります。したがって、「未定義の動作」に該当しないようなコーディングを行うことが推奨されます。

以下より、「未定義の動作」扱いの動作に対する、C コンパイラ NC30 で予測される振舞い（保証されるものではありません）を示します。「■ ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI / ISO 9899 － 1990」の節番号と節タイトルです。

■ ANSI 規格 5.1.1.2 翻訳段階（ソースファイルの終わり）

ソースファイルの最後に改行文字が無い場合、改行文字は自動的に付加されます（ファイルの最後の行は改行文字で終わる必要はありません）。

ソースファイルが前にバックスラッシュのついた改行文字で終わっている場合、バックスラッシュと改行文字が削除され改行文字が 2 個追加されます。ソースファイルが前処理トークン[注1)]やコメント文の途中で終わっている場合、エラーとなります。

[注1)]前処理トークン（詳細はANSI 規格 6.1 参照）はCソースファイル内テキストの基本処理単位で次のものがあります：ヘッダファイル名、識別子、前処理数、文字定数、文字列リテラル、演算子、句切り文字、および、左記以外の単一文字（空白を除く）。

■ ANSI 規格 5.2.1 文字セット(文字セット以外の文字)

ソースファイル中に使用できる文字セット以外の文字が出現した場合（ただし、トークンに変換されない前処理トークン、文字定数、文字列リテラル、ヘッダ名、およびコメント文は除く）、エラーとなります。

■ ANSI 規格 5.2.1.2 マルチバイト文字

コメント、文字定数、文字列リテラル以外でマルチバイト文字を使用した場合、本コンパイラにより生成されたコードの動作は、保証されません。また、コメントの終わり「*/」の直前が、シフト状態（初期シフト状態以外）の場合、コメントの終わりを認識できない場合があります。

■ ANSI 規格 6.1 字句要素（引用符の対）

ソース中に対になっていない‘または“が出現した場合、エラーとなります。

■ ANSI 規格 6.1.2.1 識別子のスコープ

同一関数内で同じ識別子を複数回ラベルとして使用するとエラーとなります。
カレントスコープ(有効範囲）にない識別子を使用する場合、生成されたコードの動作は保証されません。

■ ANSI 規格 6.1.2 識別子

同じものを指す識別子において、有意文字以降の文字が異なる場合、保証されません。

■ ANSI 規格 6.1.2.2 識別子のリンケージ

関数を表す同じ識別子を内部識別子と外部識別子の両方で宣言した場合、生成されたコードの動作は保証されません。

■ ANSI 規格 6.1.2.4 オブジェクトの記憶持続期間

自動記憶持続期間をもつオブジェクト用に予約されていた記憶域がもはや保証されなくなった場合、そのオブジェクトを参照するポインタ値を使用すると、コンパイルエラーにはなりませんがプログラムの動作は保証されません。

■ ANSI 規格 6.1.2.6 型の適合、合成型

同じオブジェクトまたは関数に対して２つの宣言があり、それらが適合した型（compatible type ）でない場合、生成されたコードの動作は保証されません。

■ ANSI 規格 6.1.3.4 文字定数

文字定数または文字列リテラル中にサポートされていないエスケープシーケンスが出現した場合、エラーとなります（例：' ＼ C ’はエラーとなります)。

■ ANSI 規格 6.1.4 単純文字列リテラル

単純文字列リテラルとワイド文字列リテラルが隣り合っている場合、それぞれの型に合わせることなく、そのまま単純に結合します。

■ ANSI 規格 6.1.7 ヘッダ名

ヘッダ名中に＼、”“、または/*が出現しても、それはファイル名を構成する文字として認識します（特殊文字として処理しません)。

■ ANSI 規格 6.2.1 算術演算

算術型変換の結果、与えられたスペースで表せない結果（精度不足）となった場合、近似値をとります。ただし、整数へ変換の場合、小数点以下の桁と、納まりきらなかった上位桁のビットパターンは捨てられます。

■ ANSI 規格 6.2.2.1 左辺値（lvalue）

初期化式で配列を初期化する以外、不完全な型を左辺値に使用するとエラーとなります。

■ ANSI 規格 6.2.2.2 void

void 型の値を使用してアクセスしたり、void 式に暗黙的型変換（void への変換を除く）を適用したりすると、エラーとなります。

■ ANSI 規格 6.3 式

副作用

式のシーケンスポイント間における副作用は不定です。副作用によって演算結果が異なる可能性のあるコードは記述しないようにしてください。

例えば、「*p++=*p+5 」というコードは、*p+5 が p++よりも前に評価される場合と後に評価される場合があり、*p+5 の代入先が不定となります。この場合、目的の処理に応じて、以下のどちらかでコーディングしてください。

```
*p=*p+5;
++p;
```

または、

```
*p = *(p+1)+5;
p++;
```

無効演算、定義域エラー

無効な演算（0 除算など）の場合、演算結果が定義域エラー（オーバーフロー、アンダフローなど）の場合、生成されたコードの動作は保証されません。

■ ANSI 規格 6.3.2.2 関数呼び出し

関数への引数がvoid 式の場合

実引数として、空引数以外の void 式を指定するとエラーになります。また、空引数（void 式）指定時、呼び出した関数に仮引数が 1 つ以上定義されている場合、関数に渡される値は不定です。

実引数と仮引数の型の不一致

関数原型宣言がない場合の関数呼び出しにおいて、その関数が関数宣言の見えないところで定義されており、かつ、拡張後（暗黙の型変換実行後）の実引数と仮引数の型が一致しない場合、実引数の値は保証されません。

関数プロトタイプと関数定義の型の不一致

関数原型宣言が見えている場合の関数呼び出しで、かつ、その関数が宣言と適合する型で定義されていない場合、生成されたコードの動作は保証されません。

可変個引数のプロトタイプ宣言

可変個引数列を受け付ける関数が、「…」で終わる関数原型が関数原型有効範囲外の場合、生成されたコードの動作は保証されません。

■　ANSI 規格 6.3.3.2 単項演算子（&、＊）

アドレス演算子&および間接参照演算＊によって、以下のような参照を行った場合、動作は保証されません。

・無効な配列を参照した場合

・NULL ポインタを参照した場合

・有効範囲が終了した自動記憶持続期間をもつオブジェクトを参照した場合

■　ANSI 規格 6.3.4 キャスト演算子

関数へのポインタを異なる型の関数へのポインタにキャストし、元の型に適合しない型の関数を呼び出した場合、動作は保証されません。

ポインタを整数型（文字型を含む）およびポインタ型以外へキャストすると多くの場合エラーとなります。また、エラーとならない場合でもプログラムの動作は保証されません。

■　ANSI 規格 6.3.6 加法演算子

配列へのポインタを加算／減算し、ポインタが配列要素の領域以外を指す結果となっても、コンパイルエラーにはなりません。この場合、ポインタの指す内容を＊演算子で参照することはできますが、そのデータは配列の要素ではないため、プログラムの動作は保証されません。

■　ANSI 規格 6.3.7 シフト演算子

シフト演算におけるシフト量の指定が、負数あるいはシフトされる式のビット幅以上の場合、動作は保証されません（動作例：シフト量が負数の場合、シフト方向が逆転することがある。シフト量がシフトされる式のビット幅以上の場合、型のサイズで表現できる限り、正常にシフトされることがある）。

■　ANSI 規格 6.3.8 関係演算子

関係演算子（<,<=,>,>=）で比較されるポインタが、同じ集成体（構造体または配列）に包含されるオブジェクトを指していない場合でも、コンパイルエラーにはなりません。またプログラムの動作も保障されません。

■　ANSI 規格 6.3.16.1 代入演算子（単純代入=）

あるオブジェクトを重複するオブジェクトに代入する場合、生成されたコードの動作は保証されません。

■　ANSI 規格 6.5 宣言

リンケージなしで宣言されたオブジェクトが、その宣言が終わっても、あるいは、（そのオブジェクトが初期値をもっているならば）その初期宣言が終わっても、不完全な型である場合、エラーとなります。

■　ANSI 規格 6.5.1 記憶クラス指示子

関数が、ブロックスコープにおいて extern 以外の記憶クラス指定子で宣言される場合、動作は保証されません。

■　ANSI 規格 6.5.2.1 構造体／共用体の指定子

名前のないメンバ

名前のないメンバだけからなる構造体あるいは共用体を定義した場合、動作は保証されません。

構造体のビットフィールドの型

構造体のビットフィールドの宣言に有効な型は、signed または unsigned の、char 、short 、int 、long、および_Bool です。それ以外の型を宣言した場合、動作は保証されません。

■　ANSI 規格 6.5.3 型修飾子

const 以外の左辺値によって const 宣言されたオブジェクトを変更しようとした場合、すなわち、const 宣言された領域をキャストなどで const でないかのように処理しようとした場合、動作は保証されません（エラーとならない場合もあります）。

volatile 以外の左辺値によって volatile 宣言されたオブジェクトを変更しようとした場合、すなわち、volatile 宣言された領域をキャストなどで volatile でないかのように処理しようとした場合、動作は保証されません（エラーとならない場合もあります）。

■　ANSI 規格 6.5.7 初期化

初期化されていない自動記憶域期間をもつオブジェクトを、値を代入する前に使用した場合、その値は不定です。

■　ANSI 規格 6.6.6.4　return 文

関数値を参照したが、関数側で値を return しなかった場合、参照した関数値は不定値です。

■　ANSI 規格 6.7 外部定義

外部リンケージをもつ、２つ以上の同一識別子を定義する場合、それらが同一ソース中にあるならばコンパイル時にエラーとなり、複数のソースに散在するならばリンク時にエラーとなります。

ただし、関数・外部変数の定義と変数が、下記の状態で複数のソースに散在する場合はエラーにならない場合があります。

- 関数
  - ・関数原型と K&R が混在している。
  - ・引数が異なっている。
- 変数
  - ・型が異なっている。

■　ANSI 規格 6.7.1 関数定義

可変個引数を受け付ける関数が、その関数定義の仮引数リストが「...」で終わっていない場合、仮引数リストで宣言した個数以上の引数を渡した場合、動作は保証されません。

■　ANSI 規格 6.7.2 外部オブジェクト定義

内部リンケージをもつ不完全型オブジェクトの識別子が、あいまいな定義で宣言された場合、動作は保証されません（ウォーニングとなる場合もあります）。

■　ANSI 規格 6.8.1 条件付き取り込み

if あるいは#elif 前処理指令の展開中に生成されるトークン defined は演算子として扱われます。

■　ANSI 規格 6.8.2 ソースファイル取り込み

include 前処理指令が、以下の形式のどちらにも合わない場合、エラーとなります。

- <ファイル名>
- "ファイル名"

■　ANSI 規格 6.8.3 マクロ置換

引数のない関数形式マクロ呼び出しはエラーとなります。

マクロコールの実引数の並びの中に、#で始まる行、すなわち前処理指令があれば、それは前処理指令とみなします。

■ ANSI 規格 6.8.3.2 #演算子（文字列化）

前処理用演算子#による文字列化の結果、有効な文字列定数にならない場合、動作は保証されません。展開時にエラーとなる場合もあります。

■ ANSI 規格 6.8.3.3 ##演算子（トークン結合）

前処理用演算子##でトークンを結合した結果、有効な前処理トークンにならない場合、動作は保証されません。例えば、func##1 が展開されると func1 となりますが、func1 が意味のないトークンである場合、コンパイル時にウォーニングとなり、リンク時にエラーとなる可能性があります。

■ ANSI 規格 6.8.4 #line

展開後の#line 前処理指令が文法に合わなければエラーとなります。このとき、行情報は更新されません。

■ ANSI 規格 6.8.8 予約マクロ名

__LINE__、__FILE__、__DATE__、__TIME__は、すでに定義済みのマクロで、#define や#undef による定義や定義の取消しを行うとウォーニングとなります。

■ ANSI 規格 7 ライブラリ

memmove 以外のライブラリ関数の使用によって、重複オブジェクトにオブジェクトをコピーしようとした場合、重複部分のデータは保証されません。

■ ANSI 規格 7.1.2 標準ヘッダ

外部定義内でのインクルード

C 標準ライブラリで提供される、関数宣言、オブジェクト宣言、型定義、マクロ定義、およびキーワードと同じ名前のマクロ定義は、初回参照前に、対応する標準ヘッダファイルをインクルードする必要があります。参照後にインクルードした場合、正しく動作しません。

予約外部名の再定義

プログラム予約外部名（ヘッダ内の外部名など）を定義した場合の処理はリンク順序に依存します。

■ ANSI 規格 7.1.4 エラー＜errno.h>

errno は、外部変数で定義しています。

■ ANSI 規格 7.1.6 共通定義＜stddef.h >

offsetof のマクロの第 2 パラメータに構造体のビットフィールドメンバを指定するとエラーとなります。

■ ANSI 規格 7.1.7 ライブラリ関数の使用方法

ライブラリ関数の実引数が無効な値の場合、プログラムの動作は保証されません。

可変個引数を受け付けるライブラリ関数が、ヘッダのインクルードなどによって宣言されていない場合、該当関数は正しく動作しないことがあります。

■ ANSI 規格 7.2 診断関数ヘッダ＜assert.h >

assert はマクロにより実現されています。関数をアクセスするためにマクロ呼び出しを無効にして assert を呼び出すと、コンパイル時にはウォーニングとなり、リンク時には外部シンボルが存在しないためエラーとなります。

■　ANSI 規格 7.3 文字操作関数ヘッダ＜ctype.h＞

文字操作関数への引数が unsigned char で表現可能であるか またはマクロ EOF の値に等しく無い場合に、動作は保証されません。

■　ANSI 規格 7.6 大域ジャンプ関数ヘッダ＜setjmp.h＞

setjmp はマクロ定義を無効にした場合でもエラーとはなりません。

■　ANSI 規格 7.6.1.1 setjmp マクロ

setjmp マクロは以下に示す用途で利用が推奨されます。これら以外の用途で利用してもエラーとはなりませんが、複雑な式の中で使用した場合、現在の実行環境の一部（式の評価の途中結果など）が失われる可能性があります。

- 選択文、繰り返し文、および整数定数式の比較における、オペランドの制御（単項演算子！による暗黙処理など）
- 選択文や繰り返し文のオペランドの制御
- 式文（void へのキャストなど）

■　ANSI 規格 7.6.2.1　longjmp 関数

setjmp 実行から longjmp 呼び出しまでの間に、volatile 指定されていない自動記憶クラスのオブジェクトが変更された場合、そのオブジェクトの値は保証されません。

■　ANSI 規格 7.7.1.1　signal 関数

本コンパイラの標準ライブラリは signal 関数を実装していません。

■　ANSI 規格 7.8.1　可変個の実引数アクセス関数ヘッダ＜stdarg.h>

ある関数（関数 A とする）において、va_arg マクロの引数 ap（可変個引数列）を実引数として呼び出された関数（関数 B とする）の中で、その ap を使って va_arg マクロを呼び出した、引数の参照は次のようになります。

- 関数 B （ある関数 A から呼び出された側）では、呼び出された時点での ap が指す可変個引数から参照できます。
- 関数 A(関数 B を呼び出した側）では、関数 B が可変個引数を参照する／しないにかかわらず、関数 B を呼び出した時点の ap が指す可変個引数から参照できます。

ただし、ap のアドレスを引数にして渡したり、集成体（ap が集成体であるとき）を引数として渡した場合、関数 B から戻ってきた後の関数 A の ap は関数 B の終了時点からの続きになります。

va_start、va_arg 、va_end はマクロにより実現されています。関数をアクセスするためにマクロ呼び出しを無効にしてこれらを呼び出すと、リンク時には外部シンボルが存在しないためエラーとなります。

■　ANSI 規格 7.8.1. 1　va_start マクロ

va_start マクロの第 2 引数の型宣言が、レジスタクラス変数、関数型、配列型であるとき、または、規定実引数拡張後の型（引数に対する暗黙の型変換後の型）と合わないとき、動作は保証されません。

■　ANSI 規格 7.8.1.2　va_arg マクロ

va _ arg を呼び出したとき、処理指定の引数が実際には存在しない場合、動作は保証されません。

va _ arg を呼び出したとき、処理指定の引数が指定された型でない場合、動作は保証されません。

■　ANSI 規格 7.8.1.3　va_end マクロ

va_start マクロを呼び出す前に va_end を呼び出してもエラーとはならず正常に動作します。

va_end マクロを呼び出す前に、va_start マクロによって初期化された可変引数リストをもつ関数が return してもエラーにはなりませんが、プログラムの動作は保証されません。

■ ANSI 規格 7.9.5.2 fflush 関数

何もせず、0 を返します。

■ ANSI 規格 7.9.5.3 fopen 関数

M16C C コンパイラの C 標準ライブラリは fopen 関数を実装していません。

■ ANSI 規格 7.9.6 書式指定入出力関数

printf 系／scanf 系

関数仕様の型と引数リストの対応する数が一致しない場合、および、変換指定子の数より引数の数が少ない場合、エラーとはなりませんが動作は保証されません。変換指定子の数より引数が多い場合、余分な引数は無視されます。

printf 系、scanf 系関数において無効な変換指定に対する入出力結果は不定です。ほとんどの場合、エラーメッセージは出力されません。期待どおりに入出力が得られない場合、変換指定のコーディングが正しい書式かどうか確認してください。

printf 系／scanf 系%%変換

printf 系または fscanf 系関数の変換指定%%で、%の次の文字を変換指定子として処理します。

■ ANSI 規格 7.9.6.1 printf 系

修飾子

printf 系の変換指定において、修飾子（サイズ指定文字 h 、l ）が該当変換指定子（o, x , X , e , E , f , g , G ）以外の変換指定子の前の h または l の前に指定された場合、その修飾子は無視されます。

フラグ

printf 系の変換指定において、フラグ#が該当変換指定子（o , x , X , e , E , f , g , G ）以外の前に指定された場合、そのフラグは無視されます。

printf 系の変換指定において、フラグ 0 が該当変換指定子（d , i , o , u , x , X , e , E , f , g, G ）以外の前に指定された場合、そのフラグは無視されます。

変換結果

集成体、共用体、または集成体か共用体へのポインタが、printf 系の%p および%s 以外に指定された場合、動作は保証されません。printf 関数による単一の%変換の変換結果が 30 文字を超える文字出力となる場合、動作は保証されません。

■ ANSI 規格 7.9.6.2 scanf 系

修飾子

scanf 系の変換指定において、修飾子（サイズ指定文字 h 、l、L ）が以下のように該当変換指定子以外の前に指定された場合、その修飾子は無視されます。

- d , i , n , o , u , x 以外の変換指定子の前の h または l
- e , f , g 以外の変換指定子の前の L

printf系の%pとの互換

printf 系の%p 変換の出力形式と scanf 系の%p に代入されるアドレス形式には互換性があります。

変換結果の格納領域

scanf 系関数による変換結果値を代入する領域が、容量不足あるいは適合しない型である場合、動作は保証されません。

■ ANSI 規格 7.10.1 文字列変換関数（文字列から数値への変換）

文字列を数値に変換する関数（atof 、atoi 、atol）の変換結果が定義域エラーで表現できない値の場合、最小値または最大値を返します。errno には ERANGE を設定します。

■ ANSI 規格 7.10.3 メモリ管理関数（free 関数、real loc 関数）

free 関数または realloc 関数によって解放された領域を参照した場合、動作は保証されません。

free 関数または realloc 関数の第 1 引数（解放対象領域へのポインタ）として次のような値を渡した場合、動作は保証されません。

- calloc 、malloc 、または realloc 関数の戻り値（割り当て完了領域へのポインタ）でない値。
- 以前に free 関数または realloc 関数によって解放された領域へのポインタ。

■ ANSI 規格 7.10.4.3 exit 関数

プログラムが 2 回以上 exit 関数の呼び出しを実行する場合の動作は保証されません。1 度目の exit 関数で無限ループとなります。

■ ANSI 規格 7.10.6 整数型の算術演算関数

整数型の算術演算関数（abs 、div 、labs 、ldiv）の結果が表現できない場合、その値は保証されません。

■ ANSI 規格 7.10.7 マルチバイト文字関数（シフト状態）

C ロケールのみをサポートしているため、ロケールを変えることはできません。

■ ANSI 規格 7.11.2 , 7.11.3 複製／連結関数

以下の場合、動作は保証されません。

memcpy 、memmove 、strcpy 、strncpy 関数において、複写先のサイズが複写元のサイズより小さい場合。strcat 、strncat 関数において、連結される文字列の格納領域が、連結結果を格納するために十分でない場合。

■ ANSI 規格 7.12.3.5 strftime 関数

M16C C コンパイラの C 標準ライブラリは strftime 関数を実装していません。

I.2 処理系定義の動作

ANSI 規格で「処理系定義の動作」扱いとなっている動作について、C コンパイラ NC30 における C 言語での振舞いを以下に示します。おもに、エラーメッセージの通知のしかた、識別子として有効な文字の数、整数や浮動小数点のフォーマットなどを規定しています。

「■ ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI / ISO 9899 － 1990 」の節番号と節タイトルです。また、「処理系定義の動作」扱いとなっている事項を＜＞内に、それに対する NC30 の動作を＜＞の後に示します。

I.2.1 変換（Translation）

■ ANSI 規格 5.1.1.3 診断

＜C コンパイラのメッセージ出力形式＞

C コンパイラの診断メッセージには、ウォーニングメッセージ、エラーメッセージ、重大エラーメッセージがあります。メッセージの出力形式および詳細については、付録 F「エラーメッセージ一覧表」を参照してください。

I.2.2 環境（Envi ronment）

■ ANSI 規格 5.1.2.2.1 プログラムの開始

＜main 関数への引数の意味＞

main 関数に渡される引数は、ユーザーが作成するスタートアッププログラムの仕様に依存します。

■ ANSI 規格 5.1.2.3 プログラムの実行

＜対話型デバイスを構成するもの＞

入出力デバイスの動作は、ユーザーが作成する低水準関数の仕様に依存します。

I.2.3 識別子（Identifiers）

■ ANSI 規格 6.1.2 識別子

＜外部リンケージを持たない識別子では、先頭から（31 を越えて）何文字まで認識されるか。＞

外部リンケージを持たない識別子は先頭から 255 文字まで有効です。 256 文字以降の文字は無視されます。

＜外部リンケージをもつ識別子では、先頭から（6 を越えて）何文字まで認識されるか。＞

外部識別子は先頭から 255 文字まで有効です。256 文字以降の文字は無視されます。また、外部識別子は大文字／小文字が区別されます。

I.2.4 文字（Characters）

■ ANSI 規格 5.2.1 文字セット

＜ソース用、および実行用の文字セット（本国際標準において明白に指定されたものを除く）の種類＞

ソース文字セット、実行文字セット共、JIS X0201 ,0208 で定義される文字が使用できます。ただし、JIS X 0201 のラテン文字部分は ASCII と見なして処理します。実際の文字コード(エンコード)には、EUC (Expanded Unix Code)、シフト JIS を使用できます。

■ ANSI 規格 5.2.1.2 マルチバイト文字

＜マルチバイト文字のシフト状態＞

マルチバイト文字のためのシフト状態（マルチバイト文字の始まりと終わりを示す文字列）はありません。

■ ANSI 規格 5.2.4.2.1 整数型のサイズ

＜実行用の文字セットの 1 文字のビット数＞

実行文字セットの 1 文字は 8 ビットです。

■ ANSI 規格 6.1.3.4 文字定数

＜ソース用文字セットの実行用文字セットへのマッピング＞

ソース文字セット中の文字と、実行文字セットとの配置対応は 1 対 1 です。

＜基本実行文字セットやワイド文字定数の拡張文字セットにない文字を含む整数文字定数の値＞

最左端の 2 文字を Big-Endian で連結したものになります。

＜2 文字以上からなる整数文字定数、または、2 文字以上のマルチバイトからなるワイド文字定数の値＞

ワイド文字ではない文字定数は、最左端の文字の値となります。ワイド文字定数は、環境変数 NCKOUT により変化します。

＜マルチバイト文字を、対応するワイド文字（コード）に変換するためのロケール＞

“C“以外のロケールはサポートしていません。

■ ANSI 規格 6.2.1.1 文字と整数

＜char は signed char と unsigned char のどちらに相当するか。＞

char は生成されるコード上では unsigned char と同様に振る舞います。

I.2.5 整数（Integers）

■ ANSI 規格 6.1.2.5 型

＜整数型の表現＞

各種整数型データの内部表現および極限値については付録 D.1「データの内部表現」を参照してください。なお、C コンパイラは、int と signed int 、short と signed short 、long と signed long はそれぞれ同じものと解釈します。

■ ANSI 規格 6.2.1.2 符号付き／符号なし整数

＜値が表現できない場合、整数データをより短い符号付き整数型に変換した結果、または、符号なし整数データを同サイズの符号付き整数型に変換した結果＞

整数を「よりサイズの小さい符号付き整数」に変換するときは、もとの整数の下位ビット値がそのまま符号付き整数に変換されます。変換後の符号付き整数の最上位ビットは符号ビットです。

符号なし整数を「同一サイズの符号付き整数」に変換するときは、もとの整数の下位ビット値がそのまま符号付き整数に変換されます。

■ ANSI 規格 6.3 式

＜符号付き整数のビット演算結果＞

符号付き整数のビット演算は、符号なし整数のように扱われます。

■ ANSI 規格 6.3.5 乗法演算子

＜整数の除算結果の剰余の符号＞

余りの符号は被除数の符号と同じです。

■ ANSI 規格 6.3.7 ビット単位シフト演算子

＜負の値をもつ符号付き整数型の右シフト＞

符号付きの負の整数型の右シフトは算術シフトです。

I.2.6 浮動小数点（Floating Point）

■ ANSI 規格 6.1.2.5 型

＜浮動小数点数型の表現＞

各種浮動小数点型データの内部表現および極限値については、付録 D.1.2「浮動小数点型」を参照してください。

■ ANSI 規格 6.2.1.3 浮動小数点数と整数の演算

＜整数が浮動小数点数型に変換され、もとの数値を正確に表現できないときの丸め＞

変換後の浮動小数点数型で表現できる範囲で、もとの値に最も近い値に丸められます。

■ ANSI 規格 6.2.1.4 浮動小数点数の演算

＜浮動小数点数型がより小さいサイズの浮動小数点型に変換されたときの丸め＞

変換後の浮動小数点数型で表現できる範囲で、もとの値に最も近い値に丸められます。

I.2.7 配列とポインタ（Arrays and Pointers）

■ ANSI 規格 6.3.3.4 sizeof 演算子、7.1.1 ライブラリ用語の定義

＜sizeof 演算子の型 size_t＞

sizeof 演算子の型 size_t は、デフォルトは unsigned long、コンパイルオプション-fsizet_16(-fS16)指定時は unsigned int として定義しています。

■ ANSI 規格 6.3.4 キャスト演算子

＜ポインタ型の整数型へのキャスト、またはその逆＞

ポインタを整数に変換、あるいは、整数をポインタに変換するときは、ポインタを符号なし整数とみなした変換が行われます。

変換前の型と変換後の型のビット数が同じならば、ビットパターンがそのまま使用されます。

変換後の型のビット数が少なければ、最下位ビットから変換後のビット数分が使用されます。

変換後の型のビット数が多ければ、ポインタから整数への変換はゼロ拡張し、符号付き整数からポインタへの変換は符号拡張し、符号なし整数からポインタへの変換は符号拡張します。変換前のビット数に相当する下位のビットパターンは変化しません。

■ ANSI 規格 6.3.6 加法演算子、7.1.1 ライブラリ用語の定義

＜prtdiff_t の型＞

二つのポインタの差を保持する整数の型 ptrdiff_t は、デフォルトでは signed long 、コンパイルオプション-fptrdifft_16(-fP16)指定時は signed int として定義されています。

I.2.8 レジスタ（Registers）

■ ANSI 規格 6.5.1 記憶クラス指定子

＜register 宣言できるオブジェクトの数＞

register 宣言できるオブジェクトの数に制限はありません。

デフォルトでは、記憶クラス指定子 register は無視されます。

コンパイルオプション-fenable_register(-fER)指定時は、register 宣言された 32 ビット以下の整数型またはポインタ型の変数は、アクセス時にレジスタに配置されます。

同時にレジスタに配置できない場合は、レジスタに配置しているオブジェクトの一部をスタック上に一時的に退避します。

I.2.9 構造体、共用体、ビットフィールド（structures , unions , enumerators , and Bit - fields）

■ ANSI 規格 6.3.2.3 構造体／共用体のメンバ

＜union のメンバが異なる型のメンバによってアクセスされる場合＞

union のメンバに格納されているビットパターンがアクセスされ、その値は、アクセスされたメンバの型に従って解釈されます。

■ ANSI 規格 6.5.2.1 構造体／共用体指定子

＜構造体のメンバのパディング（データ詰め）とアライメント（整合条件）＞

ビットフィールドのパディングおよびアライメントの詳細は、付録 D.1「データの内部表現」を参照してください。

＜「int 」型ビットフィールドは「signed int 」か「unsigned int 」か。＞

signed または unsigned が明示されていないビットフィールドは、符号なしとして扱います。

＜ビットフィールドの記憶装置上への割り付け順＞

ビットフィールドは下位ビット側から上位ビット側へ割り付けられます。

＜ビットフィールドは記憶領域の境界をまたぐかどうか。＞

１つのビットフィールドが、記憶領域の境界をまたいで配置されることはありません。

記憶領域は、そのビットフィールドを宣言した型の、ビットフィールドでないときのビット数(例えば char なら 8 ビット, int なら 16 ビット)の単位で作成されます。

■ ANSI 規格 6.5.2.2 列挙型指定子

＜列挙型の値の型＞

デフォルトでは、列挙型は unsigned int 型と適合する型として取り扱います。

コンパイルオプション-fchar_enumerator(-fCE)指定時は、列挙型を unsigned char 型と適合する型として取り扱います。

I.2.10 修飾子（qualifiers）

■ ANSI 規格 6.5.3 型修飾子

＜volatile 修飾子をもつオブジェクトへのアクセス方法＞

volatile のオブジェクト名を参照するたびに、そのオブジェクトをアクセスします。volatile オブジェクトの最適化は行いません。

I.2.11 宣言子（Declarators）

■ ANSI 規格 6.5.4 宣言子

＜算術演算、構造体、共用体の型が修正可能な宣言子の最大数＞

宣言子の最大数に制限はありません。

I.2.12 文（Statements）

■ ANSI 規格 6.6.4.2 switch 文

＜switch 文中の case 値の最大数＞

switch 文中の case 値の最大数は、ホストマシンの使用可能なメモリ容量に依存します。

I.2.13 前処理指令（Preprocessing Directives）

■ ANSI 規格 6.8.1 条件付きインクルード

＜条件つきインクルードを制御する定数式中の単一文字からなる文字定数の値が、実行文字セット中の同じ文字定数の値と一致するか。また、そのような文字定数が負の値を持つか。＞

条件付きインクルードを制御する定数式中の 1 文字定数の値は、実行文字セット中の同じ文字定数の値と一致します。このような文字定数は全て符号なしの正の値になります。

■ ANSI 規格 6.8.2 ソースファイルインクルード

＜インクルードファイル（ヘッダファイル）の検索方法＞

#include で指定されたヘッダファイルの検索順序は、以下の通りです。

<>で囲ったヘッダファイルの場合

(1) NC30 の起動オプション-I で指定されたディレクトリ
(2) 環境変数 INC30 により設定された標準ディレクトリ

""で囲ったヘッダファイルの場合

(1) ソースファイルの存在するディレクトリ
(2) 起動オプション-I で指定されたディレクトリ
(3) 環境変数 INC30 により設定された標準ディレクトリ

＜引用符で囲まれたインクルードファイル名＞

include 前処理指令では、インクルードファイル名を引用符で指定することもできます。

＜ソースファイル文字シーケンスの配置＞

ソースファイルの文字にはそれぞれ ASCII 文字の値が割り当てられます。

■ ANSI 規格 6.8.6 ＃pragma

＜コンパイラにおける#pragma 前処理指令の振舞い＞

解釈できない #pragma 前処理指令は無視されます。

コンパイルオプション -Wunknown_pragma (-WUP) 指定時は、解釈できない #pramga を警告します。

■ ANSI 規格 6.8.8 予約マクロ名

＜__DATE__と__TIME__の定義＞

__DATE__ と__TIME__ は常時使用可能です。

I.2.14 ライブラリ関数（Library Functions）

■ ANSI 規格 7.1.6 共通定義ヘッダ＜stddef.h>

＜NULL で展開される null ポインタ＞

マクロ定数 NULL（null ポインタ）の定義は 0 です。

■ ANSI 規格 7.2 診断関数用ヘッダ＜assert.h＞

＜assert 関数による診断メッセージ＞

assert 関数の終了によって出力される診断メッセージは

「Assertion failed: 式, ファイル名(line 行番号)」です。

マクロ NDEBUG が定義されているとき、assert 関数に偽の式を渡すと、assert 関数から戻らず、abort ライブラリ関数の中で無限ループします。

■ ANSI 規格 7.3.1 文字判定関数

＜文字判定関数で検査される文字セット＞

isalnum 、isalpha 、iscntrl 、islower 、isupper 、isprint が真を返す文字（ASCII 文字）は次のとおりです。

関数名	文字セット
isalnum	0～9、A～Z、a～z
isalpha	A～Z、a～z
iscntrl	0x00～0x1F、0x7F
islower	a～z
isupper	A～Z
isprint	0x20～0x7E

■ ANSI 規格 7.5.1 エラー処理

＜定義域エラー（domain error ）発生時、数学関数が返す値＞

数学関数において定義域エラーが起きた場合、errno にマクロ EDOM が設定されます。

＜アンダーフローエラー発生時、数学関数が errno を ERANGE マクロの値にセットするかどうか。＞

数学関数においてアンダーフローエラーが起きた場合、errno にマクロ ERANGE が設定されます。

■ ANSI 規格 7.5.6.4 fmod 関数

＜fmod 関数の第 2 引数が 0 のとき＞

fmod 関数の第 2 引数が 0 の場合、ドメインエラーとなり、errno にマクロ EDOM が設定されます。このとき、戻り値は 0 が返されます。

■ ANSI 規格 7.7.1.1 signal 関数
M16C C コンパイラの C 標準ライブラリは signal 関数を実装していません。

■ ANSI 規格 7.9.2 ストリーム
＜テキストストリーム（テキストファイル）の最終行には改行文字が必要か。＞
標準ライブラリの関数は、最終行に改行コードがなくても動作するように設計されています。

＜改行文字の直前にテキストストリームへ書き出された空白文字は読み込み時に出力されるか。＞
改行文字の前の空白文字も出力されます。

＜バイナリストリームに追加される null 文字の数＞
バイナリストリームの末尾に null 文字は付加されません。

■ ANSI 規格 7.9.3 ファイル
＜追加モードストリームのファイル位置指示子の位置＞
M16C C コンパイラのストリームは、ファイル位置指示子をサポートしていません。

＜テキストストリームへの書き込みがそのポイントを超えて関連ファイルを切り詰めるかどうか。＞
テキストストリームの切り詰めは起こりません。

＜ファイルバッファリングの特性＞
M16C C コンパイラでは、ファイルへの入出力ライブラリのバッファリングを行いません。

＜長さ 0 のファイルが実際に存在するかどうか。＞
M16C C コンパイラでは、ファイルへの入出力ライブラリでは、実際の出力先のファイルの管理を行いません。ファイルシステムは M16C C コンパイラ のサポート範囲外になります。

＜有効なファイル名を作るための規則＞
M16C C コンパイラでは、ファイルへの入出力ライブラリでは、実際の出力先のファイルの管理を行いません。ファイルシステムは M16C C コンパイラ のサポート範囲外になります。

＜同一のファイルを何回もオープンできるか。＞
M16C C コンパイラでは、ファイルのオープン、クローズに関するライブラリ関数をサポートしていません。

■ ANSI 規格 7.9.4.1 remove 関数
＜オープンファイルにおける remove 関数の効果＞
M16C C コンパイラの C 標準ライブラリでは、ファイル管理をサポートしていません。
remove 関数は実装していません。

■ ANSI 規格 7.9.4.2 rename 関数
＜rename 関数を呼び出す前に新しい名前を持つファイルがあった場合、そのファイルはどうなるか。＞
M16C C コンパイラの C 標準ライブラリでは、ファイル管理をサポートしていません。
rename 関数は実装していません。

■ ANSI 規格 7.9.6.1 fprintf 関数
<fprintf 関数における%p 変換の出力＞
printf 系の変換指定%p の出力は、24 ビットアドレスとして 6 桁の 16 進数を出力します。上位 2 桁と下位 4 桁の間に：が出力されます。

■ ANSI 規格 7.9.6.2 fscanf 関数

＜fscanf 関数における%p 変換の入力＞

scanf 系の変換指定%p の入力は、24 ビットアドレスとして 16 進数を入力します。上位 2 桁と下位 4 桁の間に：が必要です。

＜scanf 系におけるハイフンの解釈＞

%[変換でのハイフンは、通常の文字として解釈します。

■ ANSI 規格 7.9.9.1 fgetpos 関数、7.9.9.4 ftell 関数

M16C C コンパイラの C 標準ライブラリは fgetpos 関数、ftell 関数を実装していません。

■ ANSI 規格 7.9.10.4 perror 関数

＜perror 関数によって生成されるメッセージ＞

以下のいずれかの文字列を生成します。

- 引数として受け取った文字列
- domain error
- range error

■ ANSI 規格 7.10.3 メモリ操作関数

＜0 バイトサイズのメモリが要求された場合の calloc 、malloc 、realloc 関数の動作＞

calloc 、malloc 、realloc 関数に 0 バイトのサイズのメモリを要求したときは、NULL ポインタを返します。

■ ANSI 規格 7.10.4.1 abort 関数

＜オープンファイルと一時ファイルに関する abort 関数の動作＞

無限ループになります。

■ ANSI 規格 7.10.4.3 exit 関数

＜ exit 関数が返す終了ステータス（引数の値が 0 、EXIT_SUCCESS、または EXIT_FAILER のいずれでもない場合）＞

何も返しません。

■ ANSI 規格 7.10.4.4 getenv 関数

＜getenv 関数における環境名のセットおよび環境リストの変更方法＞

M16C C コンパイラの C 標準ライブラリは getenv 関数を実装していません。

■ ANSI 規格 7.10.4.5 system 関数

＜system 関数による文字列の内容および実行モード＞

M16C C コンパイラの C 標準ライブラリでは system 関数を実装していません。

■ ANSI 規格 7.11.6.2 strerror 関数

＜strerror 関数に返されるエラーメッセージ＞

M16C C コンパイラの C 標準ライブラリでは strerror 関数内では、関数を呼び出しません。

■ ANSI 規格 7.12.1 時刻データの内容

＜現地時間と夏時間＞

M16C C コンパイラではサポートしていません。

■ ANSI 規格 7.12.2.1 clock 関数

<clock 関数における経過時間＞

M16C C コンパイラの C 標準ライブラリは clock 関数を実装していません。

I.3 文化圏固有動作

ANSI 規格で「文化圏固有動作」扱いとなっている動作について、M16C C コンパイラ NC30 における C 言語での振舞いを以下に示します。

「■ ANSI 規格」に続く番号と見出しは、対応する ANSI 規格「ANSI / ISO 9899 － 1990 」の節番号と節タイトルです。また、「文化圏固有動作」扱いとなっている事項を＜＞内に、それに対する NC30 の動作を＜＞の後に示します。

■ ANSI 規格 5.2.1 文字セット

＜拡張実行用文字セットの内容＞

JISX0201 （ラテン文字を除く）、JISX0208 で定義される文字が使用できます。

■ ANSI 規格 5.2.2 文字表示の意味

＜印字方向＞

印字方向は左から右です。

■ ANSI 規格 7.1.1 ライブラリ用語定義

＜小数点を表す文字＞

小数点をあらわす文字は、全てのロケールにおいて 0x2E（'.'） です。

■ ANSI 規格 7.3 文字操作関数用ヘッダ＜ctype.h＞

＜文字判定関数における判定文字セット＞

文字操作関数の文字判定関数における、実装に依存する判定文字セットについては、4.2 「処理系定義の動作」の 4.2.14 「ライブラリ関数」にある「ANSI 規格 7.3.1 文字判定関数」の部分を参照してください。全てのロケールにおいて、ctype.h に定義および宣言されたマクロや関数は、“C”環境（” C” locale ）と同じ動作をします。

■ ANSI 規格 7.11.4.4 strncmp 関数

＜文字セットの照合順序＞

全てのロケールにおいて、strncmp 関数における文字セットの照合順序、ASCII の照合順序と同じです。

■ ANSI 規格 7.12.3.5 strftime 関数

M16C C コンパイラの C 標準ライブラリは strftime 関数を実装していません。

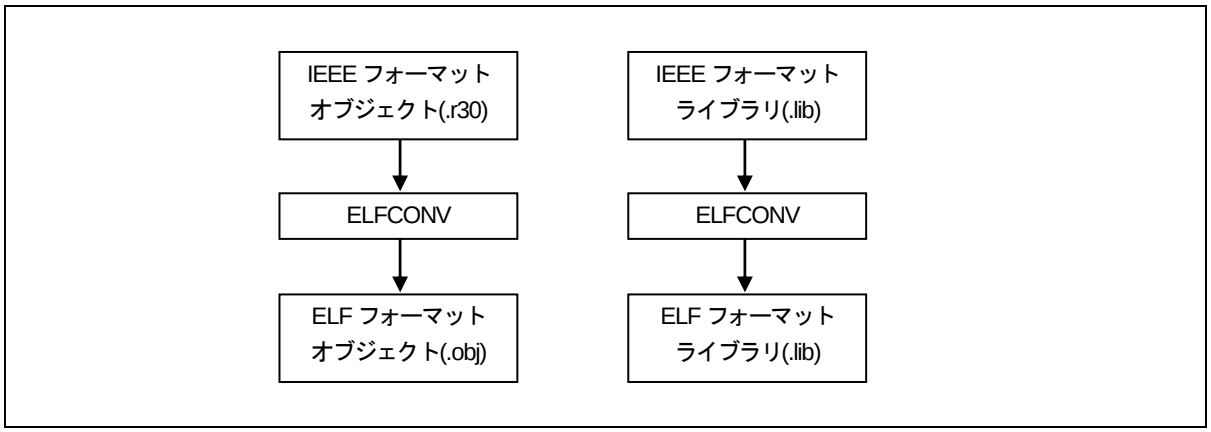
付録J ELF フォーマットコンバータ ELFCONV

ELF フォーマットコンバータ ELFCONV の起動方法と起動オプションの機能と使用上の注意事項について説明します。

J.1 概要

ELF コンバータ ELFCONV は、IEEE フォーマット形式のファイルを ELF フォーマット形式のファイルに変換するユーティリテイツールです。

NC30 V.6 以前のバージョンで生成された IEEE フォーマットのオブジェクトファイルや IEEE フォーマットのライブラリを、本ユーティリテイツールで ELF フォーマット形式へ変換することで NC30 V.6.00 以降のプロジェクトに含めることができます。



図J.1 ELFCONV の処理フロー

J.2 起動方法

J.2.1 書式

ELFCONV の起動方法を示します。入力には IEEE フォーマットのオブジェクトファイル名(拡張子.r30) もしくは IEEE フォーマットのライブラリファイル名(拡張子.lib) を指定します。

ELFCONV△[オプション]△IEEE フォーマットのオブジェクトファイル名(.r30)△-o△出力ファイル名△-cpu△CPU 種別<RET>

ELFCONV△[オプション]△IEEE フォーマットのライブラリファイル名(.lib)△-o△出力ファイル名△-cpu△CPU 種別<RET>

J.2.2 オプション

ELFCONV に指定できるオプションを示します。

表J.1　ELFCONV オプション一覧

オプション	機能
-o△出力ファイル名	出力ファイル名を指定します。 入力ファイルと出力ファイルは異なるファイルを指定してください。入力ファイルと出力ファイルが同名の場合、変換処理は行わず、エラーメッセージを出力します。
-V	ELFCONV のバージョンを表示して終了します。同時に内部ツールのバージョンも表示します。
-cpu△CPU 種別	生成する ELF ファイルの CPU 種別を指定します。 CPU 種別としては、以下のものが指定可能です。 M16C(m16C)、R8C(r8C)、R8CE(R8ce) 本オプションで指定された CPU 種別と入力オブジェクトファイルの CPU 種別が異なる場合、出力オブジェクトファイルの CPU 種別はオプションで指定されたものとなります。このとき、ウォーニングを出力します。
-Wno_check_cpu	本オプション指定時には、CPU 種別の相違によるウォーニングを出力しません。

J.3 使用上の注意事項

ELFCONV を使用する場合には、以下の点に注意してください。

- 入力ファイルのオブジェクトやライブラリ中に含まれるデバッグ情報は、変換の対象になりません。
- ELFCONV で変換したファイルをリンクする場合、optlnk の-cpu=stride オプションを使用しないでください。
- ELFCONV ではバイナリで同一となる変換を行っていますが、以下の場合には同一となりません。
  (1)　複数のオブジェクトで同名のセクションが個別に存在する場合
  (2)　各セクションでアライメント指定が異なっている場合
  (3)　セクションの開始位置が(リンクオプションなどで)奇数アドレスに指定される場合
- #pragma address で指定された変数シンボルは絶対アドレスとして変換されます。このため、リンカでの変数情報について、以下のような制限事項があります。
  (1)　list 及び、show=reference,xreference オプションを指定した際に出力される.map ファイルにおいて、#pragma address で指定した変数シンボルの参照回数は 0 回となります。
  (2)　msg_unused オプションを指定した際に出力されるシンボル未参照メッセージが#pragma address で指定した変数シンボルに対して出力されます。
- ELFCONV で変換されたオブジェクトは、UTL30 による SBDATA/SPECIAL 機能の対象外になります。
- 同一名称のセクションが複数定義されている場合は 1 つのセクションにまとめられます。また各セクションの間に空き領域が存在する場合は NOP コード(0x04)が埋め込まれます。また、複数の同名セクションで属性が異なる場合は、ELFCONV は以下の表に従ってエラーまたは警告表示を出しています。

表J.2 同名セクションで属性が異なる場合の表示

セクションアドレス		セクション種別		
比較元	比較先	CODE	ROMDATA	DATA
相対	相対	正常	正常	正常
絶対	相対	正常	正常	正常
絶対	絶対(上位)	Error	Error	Warning
絶対	絶対(重複)	Error	Error	Warning
絶対	絶対(下位)	Error	Error	Warning
相対	絶対	Error	Error	Error

J.4 ELFCONV のメッセージ

ELFCONV が表示するメッセージを示します。

表J.3 ELFCONV メッセージ一覧

番号	エラーもしくはウォーニング	メッセージ	対応方法
H1001	ウォーニング	Address is overlapped in 'DATA' section 'セクション名'	DATA セクションに重複が発生しました。
H1002	ウォーニング	Absolute-section 'セクション名' is written after the same name of Absolute-section	絶対指定されたセクションに対して、同名のセクションが再度絶対指定されています。
H1005	ウォーニング	Specified CPU type 'CPU 種別' is different from the object CPU type 'CPU 種別' in '入力モジュール名'. 'CPU 種別' is adopted	入力オブジェクトの CPU 種別が、オプション指定された CPU 種別と異なります。
H2001	エラー	can not open file(ファイル名)	入力ファイルが読み取りモードで開けません。
H2003	エラー	file format error	入力ファイルが IEEE695 形式ではありません。
H2005	エラー	input file name is not specified	入力ファイルが指定されていません。
H2010	エラー	unknown option(入力オプション名)	無効なオプションが入力されています。
H2013	エラー	Address is overlapped in 'CODE' section 'セクション名'	CODE セクションに重複があります。
H2014	エラー	Address is overlapped in 'ROMDATA' section 'セクション名'	RAMDATA セクションに重複があります。
H2015	エラー	Absolute-section 'セクション名' is written after the same name of Relocatable-section	相対指定されたセクションに対して、同名のセクションが再度絶対指定されています。

付録K バージョンアップ内容と移行方法

K.1 バージョンアップ内容

旧バージョンからのバージョンアップ内容と、ユーザーアプリケーションを移行する際の方法と注意事項について説明します。

K.1.1 C++言語対応

C++言語で記述されたファイル（拡張子は.cpp, .cc, .cp のいずれか）をコンパイルすることができます。

K.1.2 統合開発環境(High-performance Embedded Workshop)プロジェクト変換

旧バージョンのコンパイラの統合開発環境(High-performance Embedded Workshop)のプロジェクトを、本バージョンのプロジェクトへプロジェクト変換することができます。
アセンブラスタートアップを使用している場合、-order が存在しないためプロジェクト移行後 optlnk のオプションに下記-start を設定する必要があります。

```
-start=data_SE,bss_SE,data_SO,bss_SO,data_NE,bss_NE,data_NO,bss_NO,stack,
istack,heap,rom_NE,rom_NO/0400,data_FE,bss_FE,data_FO,bss_FO/010000,
rom_FE,rom_FO,data_SEI,data_SOI,data_NEI,data_NOI,data_FEI,data_FOI,
switch_table,program,interrupt/0E0000,program_S/0F0000,vector/0FFD00
```

また、各セクションの開始アドレスはデフォルト値になっていますので、ユーザー設定に変更する必要があります。

K.1.3 オブジェクトフォーマット変更

オブジェクトフォーマットを従来の IEEE-695 から、ELF/DWARF2 フォーマットへ変更しました。デバッガで読み込むオブジェクトフォーマットの指定時は ELF/DWARF2 を指定してください。
IEEE-695 のオブジェクトファイルを ELF フォーマットへ変換するオブジェクトコンバータ(elfconv.exe)を用意しましたので、ご使用ください。

K.1.4 拡張子の変更

オブジェクトファイルの拡張子を、(.r30)から(.obj)へ変更しました。また、アブソリュートファイルの拡張子を、(.x30)から(.abs)へ変更しました。
統合開発環境(High-performance Embedded Workshop)を使っている場合には、特に意識する必要はありません。メイクファイルを使用している場合には変更が必要です。

K.1.5 ライブラリアンの変更

ライブラリアンを従来の lb30 から optlnk へ変更しました。コマンドラインで使用する場合にはオプション等を含めて修正してください。

機能	lb30 オプション	optlnk でのオプション
メッセージ出力の抑止	-.	無し
モジュール追加	-A file.lib file1.r30 file2.r30	-form=library -library=file.lib file1.obj file2.obj
ライブラリファイル生成	-C file.lib file1.r30 file2.r30	-form=library -output=file.lib file1.obj file2.obj
モジュール削除	-D file.lib file1.r30	-form=library -library=file.lib -delete=file1
ライブラリリストファイル作成	-L file.lib -L file.lib file1.r30	-form=library -library=file.lib -list -show=symbol 無し
モジュール置換	-R file.lib file1.r30	-form=library -library=file.lib -replace=file1.obj
モジュール更新	-U file.lib file1.r30	無し
バージョン表示	-V	無し
モジュール抽出	-X file.lib file1.r30	-library=file.lib -extract=file1 -form=object
コマンドファイル	@file	-subcommand=file.cmd

K.1.6 リンケージエディタの変更

リンケージエディタを従来の ln30 から optlnk へ変更しました。コマンドラインで使用する場合にはオプション等を含めて修正してください。

機能	ln30 オプション	optlnk でのオプション
メッセージ出力の抑止	-.	無し
エントリポイント指定	-E point -E 0f0000	-entry=point -entry=0f0000
デバッグ情報出力	-G -G 無し	-debug -nodebug
分岐命令の最適化	-JOPT	-optimize=branch
ライブラリ指定	-l file.lib	-library=directory¥file.lib
	-LD directory	
	環境変数：LIB30	環境変数：HLNK_DIR
ROM 化支援	-LOC PP=0fe000 -ORDER PP=00400	-rom=PP=PP_ram -start=PP_ram/00400,PP/fe000
リンケージリスト出力	-M	-list
	-MS	-list -show=symbol
	-MSL	
メッセージ数の上限指定	-NOSTOP	無し
出力ファイル名指定	-O file 拡張子は.x30	-output=file 拡張子省略時は.abs
配置アドレス指定	-ORDER AA=f0000,BB,CC	-start=AA,BB,CC/f0000
エラータグファイル出力	-T	無し
不要シンボル通知機能	-U	-msg_unused -message
バージョン表示	-V	無し
ベクタ指定	-VECT label -VECT 0ff000	-VECT=label -VECT=0ff000
	-VECTN label,20 -VECTN 0ff000,21	-VECTN=20=label -VECTN=21=0ff000
Warning 時のロードモジュール生成抑止	-W	-change_message=error
コマンドファイル	@file	-subcommand=file.cmd
MCU 指定	-M60 -M61 -R8C -R8CE	無し

必要に応じて optlnk のためのライブラリファイルのサーチパス用環境変数 HLNK_DIR を設定してください。
アドレス値の先頭が a-f の場合、ln30 オプションはゼロが必要です。なお optlnk オプションはゼロが不要です。

K.1.7 ロードモジュールコンバータの変更

ロードモジュールコンバータを従来のlmc30からoptlnkへ変更しました。コマンドラインで使用する場合にはオプション等を含めて修正してください。

機能	lmc30 オプション	optlnk でのオプション
メッセージ出力の抑止	-.	無し
出力アドレス範囲指定	-A 1000:11ff -A 1000	-output=file.mot=1000-11ff -output=file.mot=1000-fffff
実行開始アドレス指定	-E 0f0000	-entry=0f0000
空き領域データの設定	-F 00 -F 00:1000:10ff -F 00:1000	-space=00 無し 無し
インテルHEX形式生成	-H	-form=hexadecimal
IDコード設定	-ID	アセンブラ指示命令.IDで設定し、optlnkでIDファイルを出力します。
データ長の選択	-L	-byte_count=20
出力ファイル名指定	-O file	-output=file
OFSREG設定値指定	-ofsregx -protect1 -protectx	アセンブラ指示命令.OFSREGで設定します アセンブラ指示命令.PROTECTで設定します。
バージョン表示	-V	無し
モトローラS形式生成	デフォルト	-form=stype
生成コード制御	-R8C -R8CE	無し

K.1.8 スタック使用量算出ユーティリテイの変更

スタック使用量算出ユーティリティが従来のstk.exe, stkviewer.exe, CallWalker.exeからCallWalker.exeのみになりました。optlnkに-stackオプションを付けてリンクするとスタックファイル(.sni)が生成されます。

K.1.9 マップ情報を表示するツールの変更

マップ情報を表示するツールが従来のMapViwer.exeから統合開発環境(High-performance Embedded Workshop)のマップウインドウのみになりました。

K.1.10 ライブラリジェネレータの使用

ユーザが指定したオプションに応じた標準ライブラリを生成するライブラリジェネレータ(lbg30.exe)を導入しました。統合開発環境(High-performance Embedded Workshop)を使用している場合には特に意識する必要はありません。ただし、コマンドラインでご使用になる場合には、ライブラリジェネレータを直接起動するか、もしくはLIB30フォルダ以下の標準ライブラリファイル(nc30lib.lib, nc30s16.lib, r8celib.lib, r8ces16.lib, r8clib.lib, r8cs16.lib)をご使用ください。

K.1.11 表示メッセージの変更

コンパイラおよびアセンブラが出力するエラー／ウォーニングメッセージの出力形式を変更しました。これにより統合開発環境(High-performance Embedded Workshop)のエラーメッセージジャンプ機能が使用可能になります。

K.1.12 コンパイルオプションの追加

下記のオプションを追加しました。

(1) インクルードファイルを指定する-preinclude を追加しました。
(2) コンパイルする言語を指定する-lang を追加しました。
(3) C++言語の例外処理機能を ON／OFF する-exception, -noexception を追加しました。
(4) C++言語の実行時型情報機能の-rtti を追加しました。
(5) リンカに渡すオプションを指定する-lnkcmd を追加しました。
(6) モジュール間最適化用付加情報を出力する-goptimize を追加しました。

K.1.13 アセンブラ指示命令の追加

下記のアセンブラ指示命令を追加しました。

(1) 予約領域を設定する.reserve_area を追加しました。
(2) 符号付き２バイト長データを格納する.words を追加しました。
(3) インスペクタ情報の関数呼び出し情報を定義する.callind を追加しました。

K.1.14 アセンブラオプションの追加

下記のアセンブラオプションを追加しました。

(1) モジュール間最適化用付加情報を出力する-goptmize を追加しました。
(2) アセンブラへのオプションをファイルで渡す-subcommand を追加しました。

K.1.15 外部分岐最適化の設定方法変更

旧バージョンでは外部分岐最適化を実施するためにコンパイルオプション-OGJ、アセンブラオプション-JOPT およびリンクオプション-JOPT を使用していましたが、本バージョンではコンパイルオプション-goptimize、アセンブラオプション-goptimize を使用してください。

K.1.16 廃止された機能

アセンブラオプション	-M60 -M61 -M62E -JOPT	廃止しました。
コンパイラオプション	-gbool_to_char -gold -Oglb_jmp -Werror_file -Wmake_tagfile -Wstdout -ln30 -fJSRW	廃止しました。
アセンブラ指示命令	.ver .optj .sjmp	廃止しました。
プラグマ	#pragma JSRA	-Wunknown_pragma(-WUP)もしくは-Wall を使用すると、#pragma JSRA の記述に対してウォーイングメッセージが発生しますので、該当記述を削除してください。
	#pragma JSRW	-Wunknown_pragma(-WUP)もしくは-Wall を使用すると、#pragma JSRW の記述に対してウォーイングメッセージが発生しますので、該当記述を削除してください。
	#pragma STRUCT xxx arrange #pragma PAGE	C++言語では使用できません。C 言語では従来と同じく使用できます。
	#pragma ROM	廃止しました。
ツール	アブソリュートリスタ(abs30.exe)	デバッガの逆アセンブルウインドウで、「ファイルの保存」を使ってください。
	クロスリファレンサ(xrf30.exe)	optlnk が生成するマップファイルをご使用ください。
ファイル	SPECIAL ページベクタ定義ファイル special.inc	廃止しました。

K.2 移行時の注意事項

K.2.1 -R8C オプションで生成したオブジェクトのリンクについて

-R8C オプションで生成したオブジェクト同士以外のリンクができなくなりましたので、ご注意ください。

K.2.2 -R8CE オプションで生成したオブジェクトのリンクについて

-R8CE オプションで生成したオブジェクト同士以外のリンクができなくなりましたので、ご注意ください。
また、-R8CE オプションを使用する場合には、以下のいずれかのライブラリとリンクしてください。

(1) ライブラリジェネレータに-R8CE オプションを使用して生成した標準ライブラリ
(2) r8celib.lib(ただし、以前に nc30lib.lib を使用されていた場合に限ります)
(3) r8ces16.lib(ただし、以前に nc30s16.lib を使用されていた場合に限ります)

K.2.3 複数のライブラリファイル中に同名のシンボル名が存在する場合の仕様変更について

従来の ln30 では複数ライブラリ中に同名のシンボルが存在する場合、ライブラリファイル作成時に最初に入力されたオブジェクトファイルのシンボルを有効にしていました。
optlnk では同名のシンボルがある場合、ウォーニングを出す機能があります。また、optlnk にはライブラリのモジュール単位にリンク順序を指定する機能と、グローバルシンボルをローカル属性に変更する機能があります。これらを使って意図したシンボルを強制的にリンクさせることが可能になりました。

K.2.4 #pragma ASM/ENDASM に.section の記述の取り扱いについて

関数内の#pragma ASM/ENDASM の領域内にアセンブラ指示命令.section を記述しないでください。コンパイラはそれを検出しませんので、ご注意ください。関数外での記述は可能です。

K.2.5 モジュール間最適化時のウォーニング表示について

アセンブリ言語スタートアップ(ncrt0.a30)を使用し、移行後のプロジェクトでモジュール間最適化を使用する場合に以下のウォーニングを出力する場合があります。

** L1001 (W) Option "optimize=symbol_delete" is ineffective without option "entry"

この場合、optlnk に-entry=start を指定してください。ここで、"start"は、リセットベクタに設定するシンボルです。

K.2.6　標準ライブラリの sprintf、vsprintf, sscanf 関数だけを使用する場合

標準Cライブラリの sprintf 関数、vsprintf 関数、sscanf 関数のみを使ったプログラムをリンクする場合、__sget、__iob、$_fp、$_sput シンボルが未定義エラーになることがあります。その際には、「2.2.2 アセンブラスタートアッププログラムのカスタマイズ」に記載したダミーのスタブ関数を作成してリンクしてください。

K.2.7　アセンブラスタートアップの変更

アセンブラスタートアップは以下のように変更してください。

(1)　C++言語ライブラリ初期化／終了処理の呼び出し

C++言語を使う場合には、下記のように ncrt0.a30 内の_main の呼び出しの前後に C++言語ライブラリの初期化／終了処理の呼び出しを行ってください。

```
                .glb    __CALL_INIT
                .call   __CALL_INIT,G
                jsr.a   __CALL_INIT

                .glb    _main
                .call   _main,G
                jsr.a   _main

                .glb    _exit
                .glb    $exit
                .glb    __exit_loop
_exit:
$exit:
                .glb    __CALL_END
                .call   __CALL_END,G
                jsr.a   __CALL_END
```

(2)　セクションの初期化の記述の変更

旧バージョンのアセンブラスタートアップを引き継ぐ場合、ncrt0.a30 ファイルのセクション初期化処理を、下記のようなセクション先頭を示す(top of xxxx)を使った記述へ必ず変更してください。

```
;------------------------------------------------------------------------
; bss zero clear
;------------------------------------------------------------------------
                        N_BZERO         (topof bss_SE),bss_SE
                        N_BZERO         (topof bss_SO),bss_SO
                        N_BZERO         (topof bss_NE),bss_NE
                        N_BZERO         (topof bss_NO),bss_NO

;------------------------------------------------------------------------
; initialize data section
;------------------------------------------------------------------------
                        N_BCOPY         (topof data_SEI),(topof data_SE),data_SE
                        N_BCOPY         (topof data_SOI),(topof data_SO),data_SO
                        N_BCOPY         (topof data_NEI),(topof data_NE),data_NE
                        N_BCOPY         (topof data_NOI),(topof data_NO),data_NO

;========================================================================
; FAR area initialize.
;------------------------------------------------------------------------
; bss zero clear
;------------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                        BZERO           (topof bss_FE),bss_FE
                        BZERO           (topof bss_FO),bss_FO
.endif

;------------------------------------------------------------------------
; initialize data section
;------------------------------------------------------------------------
.if __FAR_RAM_FLG__ != 0
                        BCOPY           (topof data_FEI),(topof data_FE),data_FE
                        BCOPY           (topof data_FOI),(topof data_FO),data_FO
.if __STACKSIZE__ != 0
```

(3)　SECT30.INC の記述の変更

旧バージョンのアセンブラスタートアップファイル引き継ぐ場合、optlnk 起動時ウォーニングメッセージ“L1322 (W) Section alignment mismatch“を出力することがあります。上記ウォーニングが出力された場合、sect30.inc ファイルから data_NEI と data_FEI のセクション定義(.section)から ALIGN の記述を削除してください。

また、-fno_align(-fNA)または-OR_MAX(-ORM)を指定する場合、sect30.inc ファイルから以下の記述を削除してください。

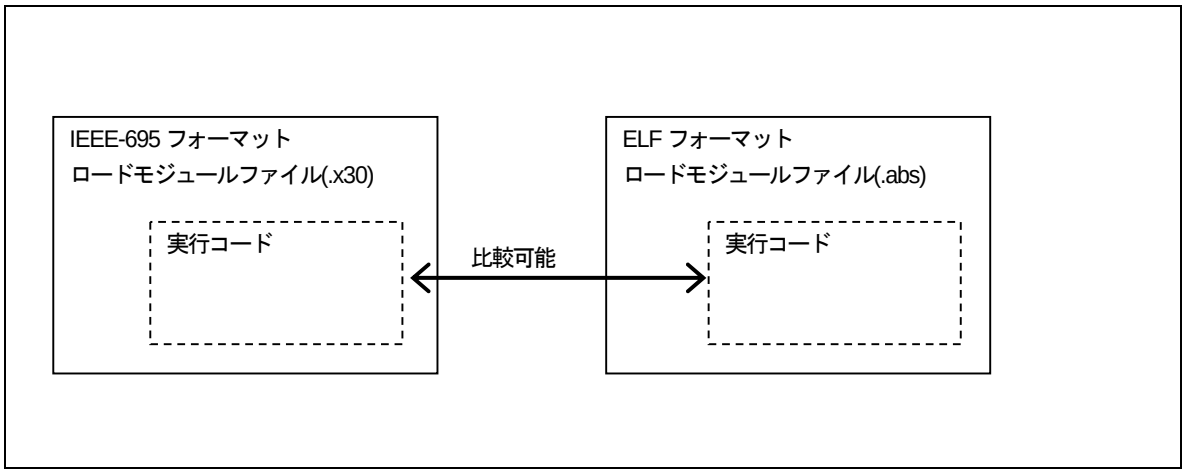
```
        .section    program,CODE,ALIGN
        .section    program_S,CODE,ALIGN
```

K.3 オブジェクトフォーマット変換後の実行コードの比較検証方法について

オブジェクトフォーマット変換ツール(elfconv.exe)は、旧バージョンで生成された IEEE-695 フォーマットのオブジェクト(.r30)[以降、IEEE-r30 と称す]やライブラリ(.lib)[以降、IEEE-lib と称す]を、NC V.6.00 で使用する ELF フォーマットのオブジェクト(.obj)[以降、ELF-obj と称す]やライブラリ(.lib)[以降、ELF-lib と称す]に変換するツールです。elfconv.exe はフォーマット変換のみを行い、実行コードには影響を与えません。そのため、既に実績のある実行コードはフォーマット変換することで、NC V.6.00 で使用することができます。ここではオブジェクトフォーマット変換ツールが実行コードに影響を与えてないことを検証するための方法について以下に説明します。

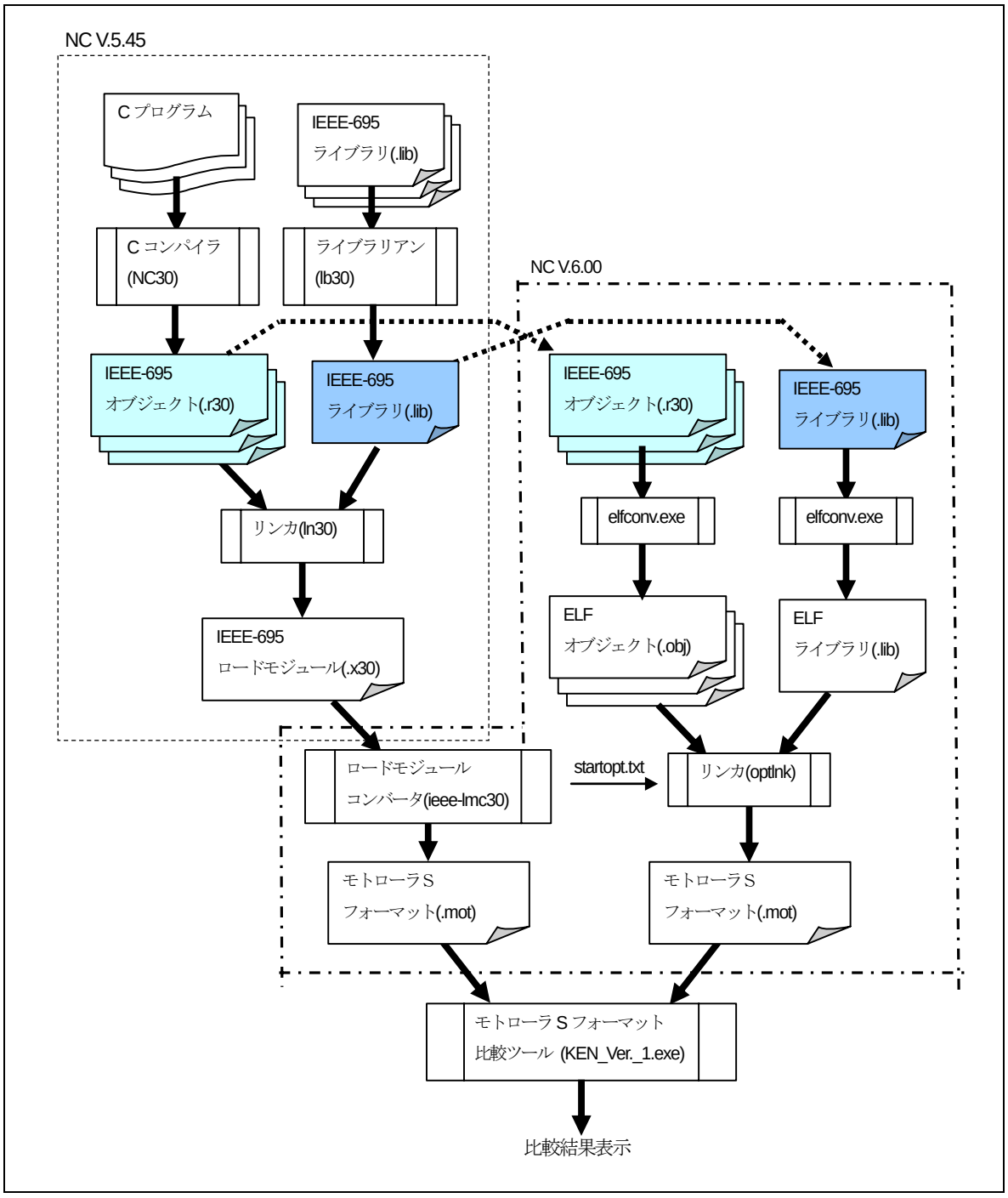
K.3.1 検証方法の考え方

IEEE-r30 と、ELF-obj は、フォーマット形式が異なるので、そのままで比較検証はできません。そのため、上記ファイルから機械語ファイルを生成し、比較を実施します。



K.3.2 検証方法の手順

検証方法の具体的な手順について説明します。



手順1．準備

旧バージョンで使用したIEEE-r30や、IEEE-libと、リンク後のIEEE-695フォーマットのロードモジュールファイル(.x30)[以降、IEEE-x30と称す]を準備します。複数のライブラリを使用している場合にはライブラリアン(lb30)で全オブジェクトファイルを取り出し、１つのライブラリにまとめてください。

また、NC V.6.00で必要な環境変数(BIN30,LIB30,INC30,TMP30,PATH)が正しく設定されている必要があります。

＜例＞

sample1.r30　(IEEE-695フォーマットのオブジェクトファイル)
sample2.r30　(IEEE-695フォーマットのオブジェクトファイル)
sample3.lib　(IEEE-695フォーマットのライブラリファイル)
sample.x30　(IEEE-695フォーマットのロードモジュールファイル)

手順2．NC V.5.45のMOTファイル生成

IEEE-x30から、NC V.6.00パッケージに含まれるieee-lmc30.exeを用いて、モトローラSフォーマットのMOTファイルおよびstartopt.txt(セクション配置情報)を生成します。

なお、ieee-lmc30.exeを起動する際は必ず -optlnkオプションを設定してください。本オプションが設定されていない場合、startopt.txtは生成されません。

＜例＞

```
$" %BIN30%¥ieee-lmc30.exe" -optlnk -o sample_ieee sample.x30
```

（sample_ieee.motとstartopt.txtができます）

手順3．オブジェクト変換の実施

IEEE-r30や、IEEE-libを、elfconv.exeを使用し、ELF-objや、ELF-libにオブジェクトフォーマット変換します。ライブラリファイルの拡張子は、IEEE-695もELFも同じ.libですので注意してください。

elfconv.exeへの-cpuオプションは付録Jを確認してください。

＜例＞

```
$" %BIN30%¥elfconv.exe" -o sample1.obj sample1.r30 -cpu m16c
$" %BIN30%¥elfconv.exe" -o sample2.obj sample2.r30 -cpu m16c
$"%BIN30%¥elfconv.exe" -o sample3_elf.lib sample3.lib -cpu m16c
```

(sample1.obj,sample2.obj,sample3_elf.libができます)

手順4．ELFフォーマットのロードモジュールファイルの生成

手順3で作ったELF-objやELF-libをリンケージエディタoptlnk.exeを使ってリンクし、モトローラSフォーマットのMOTファイルを生成します。optlnk.exeの-startオプションには手順3で生成されたstartopt.txtを指定します。

＜例＞

```
$" %BIN30%¥optlnk.exe" -form=stype -output=sample_elf.mot
        -list=sample_elf.map -sub=startopt.txt
        sample1.obj sample2.obj -library=sample3_elf.lib
```

（sample_elf.motができます）

手順5．MOTファイルの比較

<日本語対応PC上のみ>

手順2で生成したIEEE-MOTファイルと、手順4で生成したELF-MOTファイルを弊社Sフォーマット比較ツール玄1号で比較します。玄1号の入手方法および使い方については、「K.3.4. Sフォーマットファイル比較ツール玄1号について」を参照ください。

英語PC上での比較方法はリリースノートをご覧ください。

K.3.3 注意事項

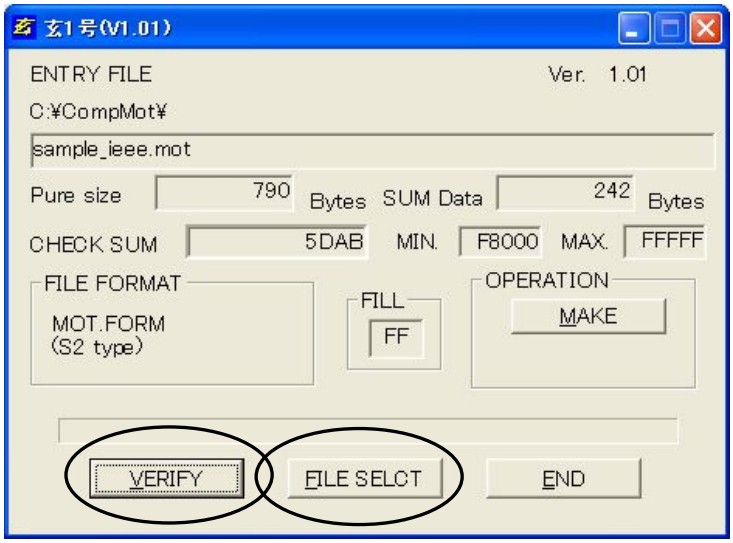
- 複数のライブラリ間でモジュール名が重複している場合は必要なモジュールをユーザが選別し、必要な側をリンクしてください。
- リンク時に未定義シンボルエラーが出るファイル群を比較検証する場合には、スタブ(stub)などを作成し、未定義シンボルを解決してから比較検証を行ってください。
- リンク後の実行コード順が異なると比較できませんので、実行コード順が必ず同じになるようにリンク時に-start オプションで調整してください。マップファイルを参照すると便利です。
- ID 情報、プロテクト情報およびオプション機能選択レジスタが未設定の場合、MOT ファイルに差分が出る場合があります。その際にはアセンブラ指示命令 .id、.protect または.ofsreg を使用して各情報を設定してください。

K.3.4 Sフォーマットファイル比較ツール玄１号について

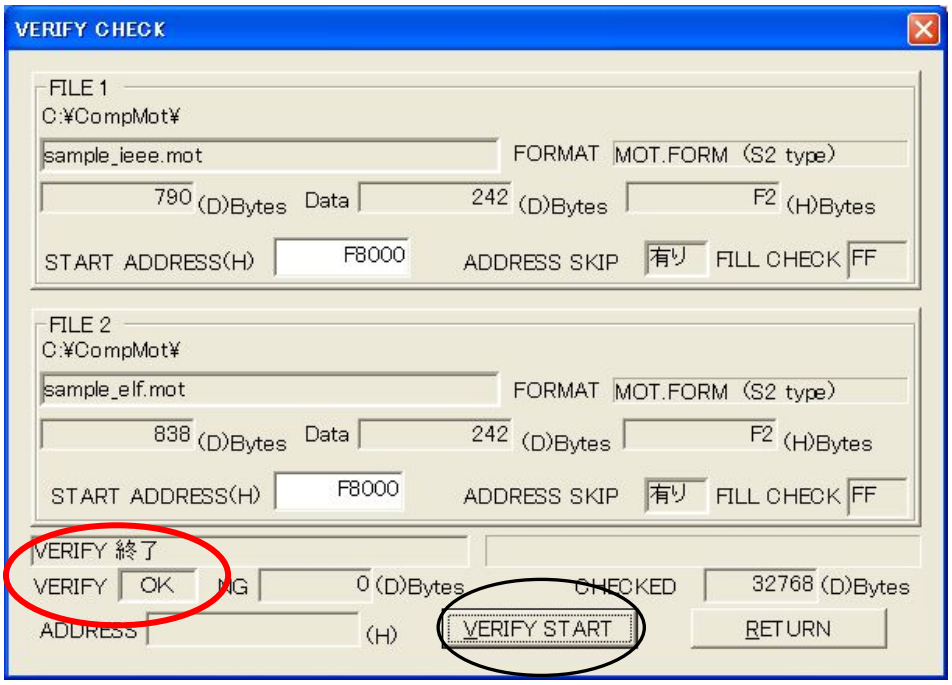
Sフォーマットファイル比較ツールは、弊社 ROM 発注の際に用いるチェックサム値算出プログラムのことです。

1. 入手先について
   下記 URL より最新バージョンをダウンロードし、適当なフォルダーに解凍してください。
   http://japan.renesas.com/products/mpumcu/rom_ordering/superh_rom_ordering/child_folder/checksum_child.jsp

2. 解凍された KEN_Ver._1.exe をダブルクリックして起動すると次のようなダイアログが表示されます。ダイアログ中の「FILE SELCT」ボタンにより、比較対象の MOT ファイルを１つ指定します。次に「VERIFY」ボタンで、他方の MOT ファイルを指定します。



3. 次のようなダイアログに表示が切り替わります。「VERIFY START」を押して比較を開始します。



4. 4．ダイアログの左下に「VERIFY 終了」と「VERIFY OK」と表示されれば MOT ファイルに差分が無く終了したことです。

付録L 注意事項

本製品をご使用いただく際に以下の注意事項があります。

L.1 機種依存部に関する注意事項

L.1.1 SFR 領域のアクセスに関する注意事項

SFR 領域のレジスタをアクセスする場合には、特定の命令を使用しなければならないことがあります。
この特定の命令は機種毎に異なりますので詳しくは各機種のユーザーズマニュアルなどを参照してください。本注意事項にかかわる命令は、asm 関数等のインラインアセンブル機能を使用してプログラム中に命令を直接記述してください。

L.1.2 M16C/62 4M 拡張モードに関して

プログラムは、内部 ROM に配置するようにしてください。

L.1.3 オンチップデバッガ選択時の FirmRam_NE セクションと SB レジスタの値に関して

新規プロジェクトワークスペース作成時に OnChipDebugger 選択画面でデバッガを選択した場合、FirmRam_NE セクションが 400H から配置されることがあります。SB レジスタの初期値は 400H で設定されているため、SB 相対アドレッシングモードで正しい領域をアクセスできなくなります。
リンクした結果、FirmRam_NE セクションが 400H から配置されている場合は、SB レジスタの初期値を bss_SE セクションの先頭アドレスの値に変更してください。bss_SE セクションの先頭アドレスは、マップファイルで確認してください。
次の 2 箇所の値を bss_SE セクションの先頭アドレスにしてください。

<resetprg.c>

```
void start(void)
{
        :
        _sb_    = 0x400;    // 400H fixation (Do not change)

}
```

<resetprg.h>

```
#define   DEF_SBREGISTER          _asm("          .glb        __SB__¥n"¥
                                      "__SB__    .equ        0400H");
```

該当 MCU は以下の通りです。(2009 年 5 月 16 日現在)

M16C/26, M16C/26A, M16C/28, M16C/29,
M16C/30P,
M16C/62P,
M16C/6N4, M16C/6N5, M16C/6NK, M16C/6NL, M16C/6NM, M16C/6NN,
M16C/6S,
M16C/64,
M16C/64A,
M16C/65

L.2 コンパイラ、アセンブラ、リンケージエディタ及びユーティリティに関する注意事項

L.2.1 -ffar_pointer(-fFP)について

-ffar_pointer を使用した場合、near 属性の変数のアドレスを取得する&演算子を使用すると、16 ビットで扱われます。&演算子の前に far ポインタでキャストするようにしてください。

また、sizeof でポインタサイズを取得した場合、戻り値は 2 となります。関数原型宣言のない関数を呼び出すとアドレスを 2 バイトしか積みません。必ず関数原型宣言をしてください。

L.2.2 標準入出力関数について

printf関数等の標準入出力関数は、多くのRAMを消費します。そのため**R8Cファミリ MCU用ライブラリ(-R8C オプション指定時)**において、標準入出力関数をご使用になる場合は、%e,%E,%f,%g,%Gの変換指定記号は使用できません。本件の対象は、r8clib.lib, r8cs16.libまたはライブラリジェネレータlbg30.exeの-nofloatオプションを使用して作成されたライブラリです。

L.2.3 インラインアセンブル機能(#pragma ASM～#pragma ENDASM、asm 関数)に関する注意事項

- #pragma ASM～#pragma ENDASM、asm 関数内の記述について

(1) コンパイラは、レジスタの生存区間、変数の生存区間についてプログラムフローを解析して処理を行っているため asm 関数などでフローに影響を与えるようなブランチ(条件ブランチ含む)を記述しないようにしてください。

(2) コンパイラはレジスタを介して渡される引数およびレジスタ変数に対して、これらの有効範囲を解析しコードを生成します。しかし、インラインアセンブル機能(#pragma ASM～#pragma ENDASM または asm 関数)を使用してレジスタ値を操作する記述を行った場合、C コンパイラはインラインアセンブル機能で記述されたプログラム部分で有効となるこれらの引数およびレジスタ変数の範囲の情報を保持することができません。したがって、インラインアセンブル機能を使用してレジスタを操作する記述を行う場合は、必ずレジスタの退避・復帰を行ってください。

L.2.4 メモリ管理関数 malloc()、calloc()および realloc()の注意事項

NC30WA のメモリ管理関数"malloc()"、"calloc()"および"realloc()"は、一度に 64KB 以上の領域を確保することはできません。

L.3 MISRA C ルール適合に関して

L.3.1 標準関数ライブラリ

M3T-NC30WAの標準関数ライブラリのCソースコードは、MISRA Cルールに対していくつかのルール違反[1]が認められますが、これらの違反は動作に支障はありません。

L.3.2 ルール違反の要因

M3T-NC30WA の標準関数ライブラリの C ソースコードにおいて、ルール違反となった主な要因は次の通りです。

- C コンパイラの仕様 (near/far 修飾、asm()関数、#pragma)
- ANSI 規格に基づく関数の宣言
- 条件文における評価順序をカッコ()により明示的に記述していない
- 暗黙の型変換

L.3.3 ルール違反となった検査番号

ルール違反になった検査番号は次のとおりです。

1	12	13	14	18	21	22	28	34	35
36	37	38	39	43	44	45	46	48	49
50	54	55	56	57	58	59	60	61	62
65	69	70	71	72	76	77	82	83	85
99	101	103	104	105	110	111	115	118	119
121	124	-	-	-	-	-	-	-	-

L.3.4 評価環境

コンパイラ	M3T-NC30WA V.5.30 Release 1
コンパイルオプション	-O -c -as30 "-DOPTI=0" -gnone -finfo -fNII -misra_all -r $*.csv
MISRA C チェッカ	SQMlint V.1.00 Release 1A

L.3.5 統合開発環境(High-performance Embedded Workshop)が自動生成するソースコード

統合開発環境(High-performance Embedded Workshop)がが自動生成するソースコードは、MISRA C ルールに対し、いくつかのルール違反が認められますが、これらの違反は動作に支障はありません。

L.3.6 ルール違反の要因

統合開発環境(High-performance Embedded Workshop)生成ソースコードにおいて、ルール違反となった主な要因は次の通りです。

- C コンパイラの仕様 (#pragma 等)
- ヘッダで定義された変数のスコープ
- ビットフィールドで使用する型の定義

[1] MISRA C ルールチェッカSQMLintによる検査結果値です。

L.3.7 ルール違反となった検査番号

ルール違反になった検査番号は次のとおりです。

13	14	22	34	36	37	43	45	46
49	54	59	69	76	82	85	99	104
110	111	115	124	126	-	-	-	-

L.3.8 評価環境

コンパイラ	M3T-NC30WA V.5.45 Release00
コンパイルオプション	-c -misra_all
MISRA C チェッカ	SQMlint V.1.03 Release 00

L.3.9 C スタートアップ中で使用する#pragma 拡張機能(Misra C ルール 99)

拡張機能	宣言ファイル	内容	機能
#pragma STACKSIZE	resetprg.h	ユーザスタックサイズを定義します。	スタックセクション(stack)の出力及び、スタックのトップラベル名を生成します。
#pragma ISTACKSIZE	resetprg.h	割り込みスタックサイズを定義します。	割り込みスタックセクション(istack)の出力及び、割り込みスタックのトップラベル名を生成します。
#pragma CREG	resetprg.h	MCU の内部レジスタを宣言します。	本pragma で宣言された内部レジスタにアクセスする場合、専用命令を使用してアクセスするコードを生成します。
#pragma sectaddress	resetprg.h fvector.c	セクションの定義を行います。同時に配置アドレスの宣言をすることができます。	本pragma で宣言されたセクション名でセクション定義を行います。 同時にアドレス指定された場合は、疑似命令.org を使用したアドレス定義を出力します。
#pragma entry	resetprg.h	reset 時に実行する関数を宣言します。	本pragma で宣言された関数に対してスタックフレームを構築する enter 命令を出力しません。 これは、スタックポインタ初期化前に enter 命令を生成しないようにするためです。
#pragma interrupt/V	fvetor.c	ベクタテーブルを生成します。	本pragma で宣言された関数に対して割り込みベクタのみを定義します。
#pragma inline	resetprg.h	inline 関数を宣言します。	本pragma で宣言された関数に対してインライン展開します。
#pragma interrupt	intprg.c fvector.c	割り込み関数を宣言します。	本pragma を使用して宣言された関数に対して、割り込み関数のコードを生成します。
#pragma section	heap.c resetprg.c initsct.h resetprg.c firm.c	セクション名を変更します。	本pragma で定義されたセクション名に変更します。
#pragma ADDRESS	各 sfr ヘッダファイル	I/O のアドレス定義及び変数宣言を行います。	本 pragma で定義された sfr に対して.equ でアドレス定義を行います。

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
M16Cシリーズ,R8Cファミリ用C/C++コンパイラパッケージ V.6.00
C/C++コンパイラ ユーザーズマニュアル

発行年月日　2011年1月16日　Rev.1.00

発行　ルネサス エレクトロニクス株式会社
〒211-8668 神奈川県川崎市中原区下沼部1753

編集　株式会社ルネサス ソリューションズ

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）

(03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

Colophon 1.0

M16Cシリーズ,R8Cファミリ用
C/C++コンパイラパッケージ V.6.00
C/C++コンパイラ ユーザーズマニュル

ルネサス エレクトロニクス株式会社

RJJ10J2777-0100