お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

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。

User's Manual

M32C/90, 80, M16C/80, 70シリーズ用 Cコンパイラパッケージ V.5.20

Cコンパイラユーザーズマニュアル

ルネサス エレクトロニクス
www.renesas.com

Rev.2.00　2005.03

- Microsoft、MS-DOS、Windows および Windows NT は、米国 Microsoft Corporation の米国およびその他の国における商標または登録商標です。
- HP-UX は、米国 Hewlett-Packard Company のオペレーティングシステムの名称です。
- Sun、Solaris、Java およびすべての Java 関連の商標およびロゴは、米国およびその他の国における米国 Sun Microsystems, Inc.の商標または登録商標です。
- UNIX は、The Open Group の米国ならびにその他の国における登録商標です。
- Linux は、Linus Torvalds 氏の米国およびその他の国における登録商標あるいは商標です。
- Turbolinux の名称およびロゴは、Turbolinux, Inc.の登録商標です。
- IBM および AT は、米国 International Business Machines Corporation の登録商標です。
- HP 9000 は、米国 Hewlett-Packard Company の商品名称です。
- SPARC および SPARCstation は、米国 SPARC International, Inc.の登録商標です。
- Intel, Pentium は、米国 Intel Corporation の登録商標です。
- Adobe および Acrobat は、Adobe Systems Incorporated（アドビシステムズ社）の登録商標です。
- Netscape および Netscape Navigator は、米国およびその他の諸国の Netscape Communications Corporation 社の登録商標です。

その他すべてのブランド名および製品名は個々の所有者の登録商標もしくは商標です。

安全設計に関するお願い

- 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

- 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報について株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは責任を負いません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズは、予告なしに、本資料に記載した製品又は仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前に株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ（http://www.renesas.com）などを通じて公開される情報に常にご注意ください。
- 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズはその責任を負いません。
- 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及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。株式会社ルネサス テクノロジ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。
- 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店へご照会ください。
- 本資料の転載、複製については、文書による株式会社ルネサス テクノロジおよび株式会社ルネサス ソリューションズの事前の承諾が必要です。
- 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたら株式会社ルネサス テクノロジ、株式会社ルネサス ソリューションズ、株式会社ルネサス販売又は特約店までご照会ください。

製品の内容及び本書についてのお問い合わせ先

インストーラが生成する以下のテキストファイルに必要事項を記入の上、ツール技術サポート窓口support_tool@renesas.comまで送信ください。

¥SUPPORT¥製品名¥SUPPORT.TXT

株式会社ルネサス ソリューションズ

ツール技術サポート窓口	support_tool@renesas.com
ユーザ登録窓口	regist_tool@renesas.com
ホームページ	http://www.renesas.com/jp/tools

NC308 ユーザーズマニュアル

目　次

目次

はじめに

NC308は、ルネサス 32/16ビットマイクロコンピュータ M32C/80シリーズ,16ビットマイクロコンピュータ M16C/80, M16C/70シリーズ用のCコンパイラです。NC308は、C言語で記述したプログラムをM32C/80シリーズ用,M16C/80,M16C/70シリーズ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進数形式ファイルまで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。

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	M3T-NC308WA に含まれるコンパイラ・パッケージを意味します。
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	M3T-NC308WA に含まれるアセンブラ・パッケージを意味します。
as30	ﾘﾛｹｰﾀﾌﾞﾙﾏｸﾛｱｾﾝﾌﾞﾗ、又はその実行ファイルを意味します。
TM	付属の統合環境を意味します
プロフェッショナル版	本格的プログラミングに対応したプロユースのコンパイラです。
エントリー版	スターターキット等に付属の簡易版コンパイラを意味します。

使用する記号の説明

NC308 のマニュアルでは、以下に示す記号を使用します。

記号	表示内容
#	ルートユーザのプロンプトを示します。
%	UNIXのプロンプトを示します。
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。

NC308

ユーザーズマニュアル

第１章
NC308の処理概要

この章では、NC308が行うコンパイル処理の概要と、NC308を使用したプログラム開発の事例を説明します。

1.1　NC308の構成

NC308は、以下に示す8つの実行ファイルで構成されています。

1. nc308 ..... コンパイルドライバ
2. cpp308 ..... プリプロセッサ
3. ccom308 ..... コンパイラ
4. aopt308 ..... アセンブラオプティマイザ
5. StkViewer & stk ..... STK ビューワ & スタックサイズ計算ユーティリティ
6. utl308 ..... SBDATA宣言 & SPECIALページ関数宣言ユーティリティ
7. MapViewer ..... マップビューワ(Windows(TM)版のみに付属)

（4. ～ 7. は、エントリー版には含まれません）

1.2　NC308の処理フロー

NC308の処理フローを【図1.1】に示します。

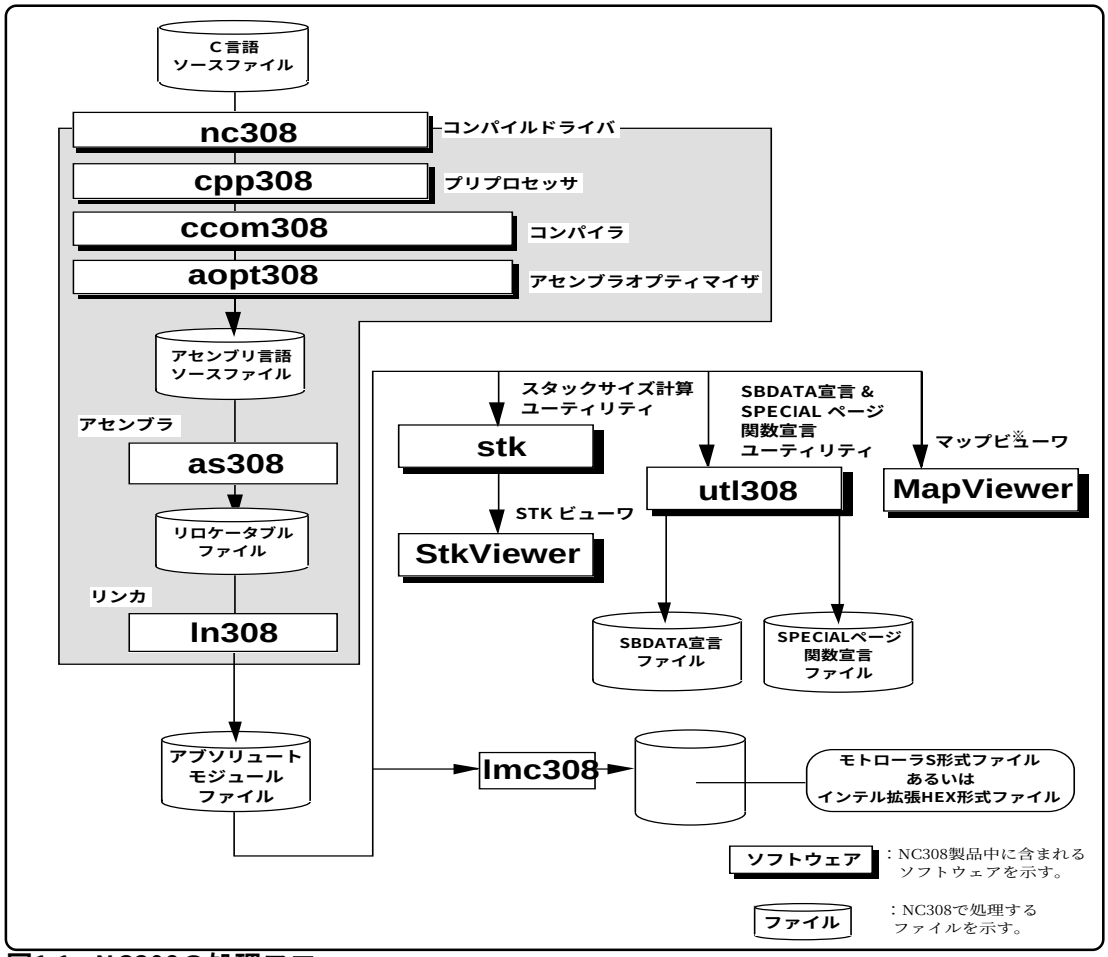


図1.1　NC308の処理フロー

※1.MapViewer は、Windows(TM)版のみに付属します。UNIX版をご使用でマップ情報を確認するには、リンカでマップファイルを生成してそのファイルで確認してください。

1.2.1 nc308

nc308は、コンパイルドライバの実行ファイルです。nc308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、nc308の起動オプション　-as308、-ln308 に続けてリロケータブルマクロアセンブラ as308、リンケージエディタ ln308 のオプションを指定することができます。

1.2.2 cpp308

cpp308 は、プリプロセッサの実行ファイルです。cpp308 は、# で始まるマクロ(#define、#include 等)と条件コンパイル(#if ～ #else ～ #endif 等)の処理を行います。

1.2.3 ccom308

ccom308は、コンパイラ本体の実行ファイルです。cpp308によって処理されたＣ言語ソースプログラムを as308 で処理可能なアセンブリ言語ソースプログラムに変換します。

1.2.4 aopt308

aopt308は、アセンブラオプティマイザです。ccom308が出力したアセンブラコードに対して、最適化を行います。（エントリー版には含まれません。）

1.2.5 StkViewer & stk

StkViewer は、プログラムの動作に必要な、スタックサイズと関数の呼び出し関係を、グラフィカルに表示するユーティリティの実行ファイルです。また、stk は、StkViewer で必要な情報の解析を行うユーティリティの実行ファイルです。

StkViewer は、stk を呼び出して、アブソリュートモジュールファイル（ .x30 ）に付加されているインスペクタ情報※1を処理し、プログラムの動作に必要なスタックサイズと関数の呼び出し関係を求め、表示します。

また、インスペクタ情報だけでは解析しきれなかった情報を、Stk Viewer で指定するとスタックサイズと関数の呼び出し関係を再度計算し表示します。

StkViewer & stk を使用するには、コンパイル時にコンパイルドライバの起動オプション -finfo を指定して、アブソリュートモジュールファイル（ .x30 ）にインスペクタ情報が付加されるようにしてください。（エントリー版には含まれません。）

1.2.6 utl308

utl308 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。utl308 は、アブソリュートモジュールファイル（ .x30 ）を処理し、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。

utl308 を使用するには、コンパイル時にコンパイルドライバの起動オプション　-finfo　を指定して、アブソリュートモジュールファイル（ .x30 ）を生成してください。（エントリー版には含まれません。）

1.2.7 MapViewer

MapViewer は、マップビューワの実行ファイルです。MapViewer は、アブソリュートモジュールファイル（ .x30 ）を処理し、リンク後のメモリ配置をグラフィカルに表示します。

MapViewer を使用するには、コンパイル時にコンパイルドライバの起動オプション -finfo を指定して、アブソリュートモジュールファイル（ .x30 ）を生成してください。

なお、MapViewer は、Windows(TM)版のみに付属します。UNIX 版をご使用でマップ情報を確認するには、リンカでマップファイルを生成してそのファイルで確認してください。（エントリー版には含まれません。）

※1.インスペクタ情報とは、コンパイルオプション ”-finfo“ により生成される解析情報です。

1.3　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。株式会社ルネサス テクノロジ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。

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時に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。

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。

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。

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には、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。

以下の例のような C 言語記述を SFR 領域に行った場合、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う可能性があります。

[例：SFR領域に対するCソースコード記述]

```
#pragma ADDRESS TA0IC   006Ch   /* M16C/80 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 */
  }
```

1.4　プログラム開発例

NC308を使用したプログラム開発例の流れを【図1.2】に示します。このプログラムの概要を以下に示します（項目の(1)～(4)は【図1.2】の[1]～[4]に対応します）。

(1)C言語ソースプログラム（AA.c）をnc308でコンパイル、アセンブラas308でアセンブルし、リロケータブルオブジェクトファイル（AA.r30）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sect308.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、リロケータブルオブジェクトファイル（ncrt0.r30）を作成します。

(4)2つのリロケータブルオブジェクトファイル、AA.r30とncrt0.r30をnc308から実行されるリンケージエディタln308でリンクし、アブソリュートモジュールファイル（AA.x30）を作成します。

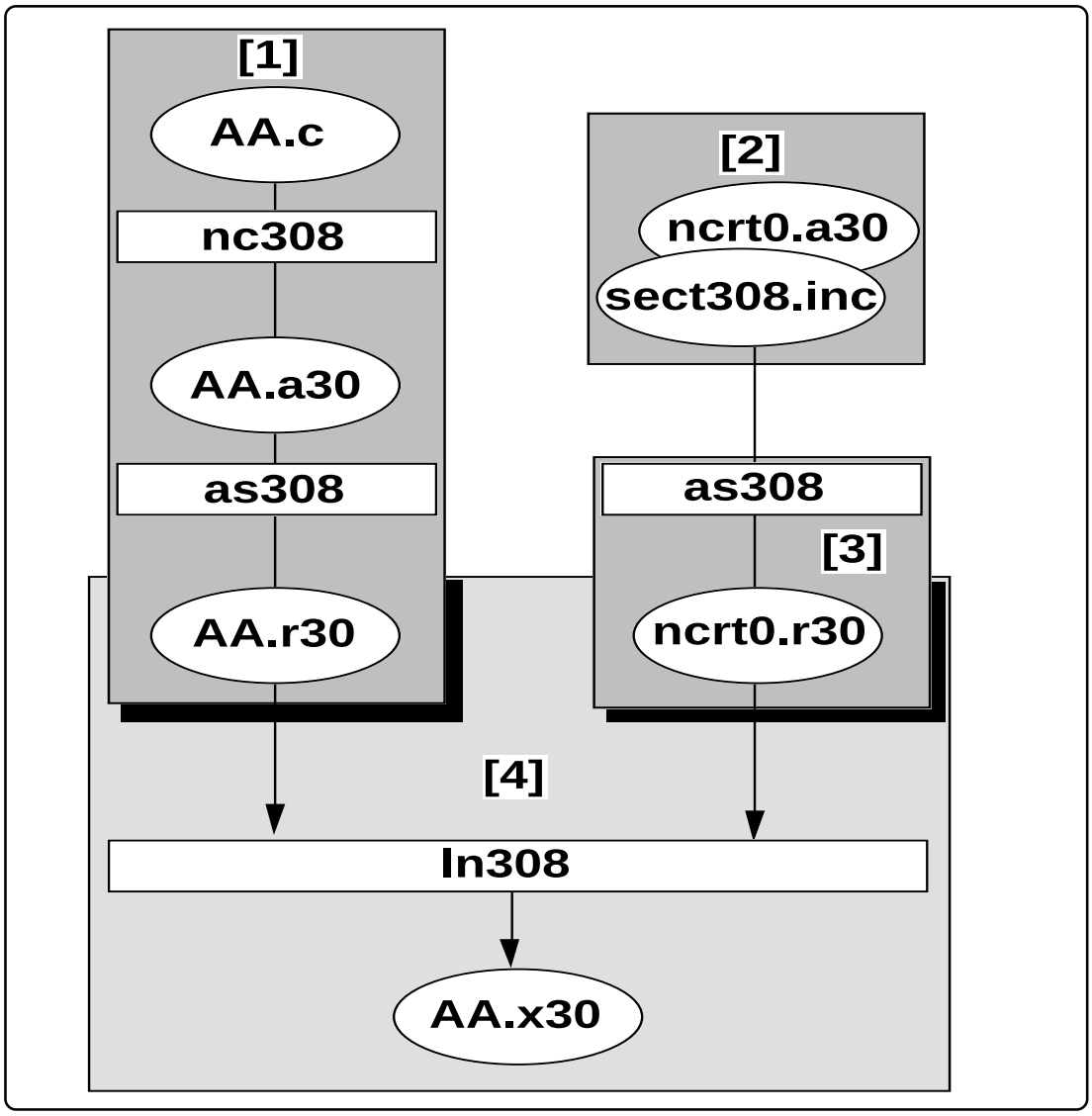


図1.2　プログラム開発フロー

【図1.2】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1.3】に示します。

```
AA.x30 : ncrt0.a30 AA.r30
        nc308 -oAA ncrt0.r30 AA.r30

ncrt0.r30 : ncrt0.a30
        as308 ncrt0.a30

AA.r30 : AA.c
        nc308 -c AA.c
```

図1.3　実行手順ファイル(makefile)の記述例

また、コンパイルドライバ nc308 では、【図 1.3】と同様の処理をコマンドラインから【図1.4】に示すように入力できます。

```
% nc308 -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。

※リンク処理を行うときは必ずスタートアッププログラムを最初に指定してください。

図1.4　nc308コマンドの入力例

1.5　NC308の出力ファイル

サンプルプログラム smp.c を NC308 でコンパイルした結果、出力されるプリプロセス結果 C 言語ソースプログラム、アセンブリ言語ソースプログラムの概要を説明します。

1.5.1　出力ファイルの概要

コンパイルドライバ nc308 は、起動オプションによって【図 1.5】に示すファイルを出力します。次項から【図 1.6】に示す C 言語ソースファイル smp.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。

なお、as308 及び ln308 が出力するリロケータブルオブジェクトファイル（拡張子 .r30 ）、プリントファイル（拡張子.lst)、マップファイル（拡張子.map)、等に関しては「AS308 ユーザーズマニュアル」を参照してください。

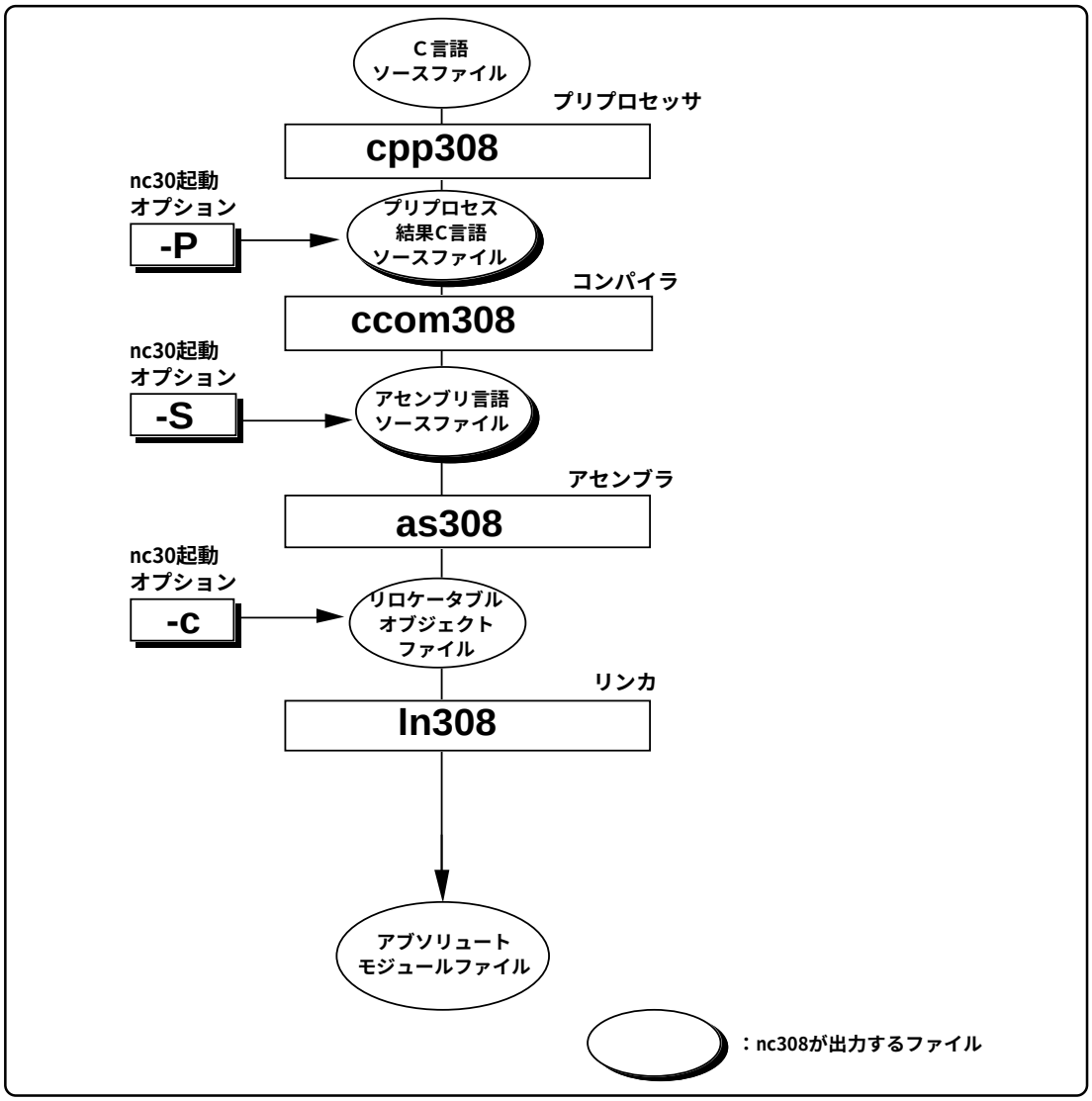


図1.5　nc3088起動オプションと出力ファイルの相互関係図

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

void main()
{
          int flag;
          flag = CLR;

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

図1.6　C言語ソースファイル例(smp.c)

1.5.2　プリプロセス結果C言語ソースファイル

プリプロセッサ cpp308 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。

プリプロセス結果Ｃ言語ソースファイルは、cpp308 が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 1.7】、【図 1.8】に示します。

```
                                                        ①
typedef struct _iobuf {
    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1.7　プリプロセス結果C言語ソースファイル例(1)(smp.i)

```
                                                                    ①
extern int      getc(FILE _far *);
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      ungetc(int, FILE _far *);
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     init_prn(void);
extern  int     _sget(void);
extern  int     _sput(int);
extern  int     _pput(int);
extern  char    _far *_print(int(*)(), char _far *, int _far * _far *, int _far
*);
```

```
void main()                                          ②
 {
        int flag;
        flag = 0 ;      ←③


         printf("flag = %d\n",flag);       ←④

 }
```

図1.8　プリプロセス結果C言語ソースファイル例(2)(smp.i)

プリプロセス結果 C 言語ソースファイルの内容を以下に説明します。項目番号の①～④は【図 1.7】、【図 1.8】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 1 であるため、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　アセンブリ言語ソースファイル

このファイルは、コンパイラccom308がプリプロセス結果C言語ソースファイルをAS308で処理可能なアセンブリ言語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リ言語ソースファイルです。

ファイルの出力例を【図1.9】、【図1.10】に示します。なお、アセンブリ言語ソースファイルの出力には、nc308の起動オプション-dsource(-dS)を指定して行単位でC言語ソースファイルの内容をコメントとして表示させています。

```
    ._LANG     'C','X.XX.XX','REV.X'

 ;## NC308 C Compiler   OUTPUT
 ;## ccom308 Version X.XX.XX
 ;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
 ;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RE-
SERVED
 ;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,2000,2001,2002,2003

 ;## COMMAND_LINE: ccom308  D:\MTOOL\nc308wa5\TMP\smp.i -o .\smp.a30 -dS

┌──────────────────────────────────────────────────────────────────────────┐
│ ;## Normal Optimize       OFF                                          ① │
│ ;## ROM size Optimize     OFF                                            │
│ ;## Speed Optimize        OFF                                            │
│ ;## Default ROM is        far                                            │
│ ;## Default RAM is        near                                           │
└──────────────────────────────────────────────────────────────────────────┘
    .GLB  __SB__
    .SB   __SB__
    .FB   0

 ;## #    FUNCTION main
 ;## #    FRAME AUTO  (    flag) size  2,   offset -2
 ;## #    ARG Size(0)Auto Size(2)     Context Size(8)


    .SECTION   program,CODE,ALIGN
    ._file     'smp.c'
    .align
    ._line     6
 ;## # C_SRC :  {
    .glb  _main
 _main:
    enter #02H
    ._line     8
 ;## # C_SRC :       flag= CLR;
```

図1.9　アセンブリ言語ソースファイル例(1)(smp.a30)

```
    mov.w #0000H,-2[FB]    ;  flag
    ._line     11
;## # C_SRC :        printf("flag = %d\n",flag);                ←②
    push.w     -2[FB]      ;  flag
    push.l     #___TO
    jsr   _printf
    add.l #06H,SP
    ._line     13
;## # C_SRC :  }
    exitd

          :
        (省略)
          :

    .glb  _sscanf
    .glb  _fflush
    .glb  _clearerr
    .glb  _perror
    .glb  _init_dev
    .glb  $speed

          :
        (省略)
          :

    .SECTION    rom_FO,ROMDATA
___TO:
    .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
    .END

;## Compile End Time Mon Jun 17 14:40:21 20xx
```

図1.10　アセンブリ言語ソースファイル例(2)(smp.a30)

アセンブリ言語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1.9】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のnear/far属性の初期設定の情報を示しています。

②nc308の起動オプション-dsource(-dS)を指定したときにC言語ソースファイルの内容がコメントで表示されます。

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、コンパイルドライバ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。

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、コンパイラの各コマンドとアセンブルコマンド及びリンクコマンドを起動し、機械語データファイルを生成します。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言語ソースファイル
2.アセンブリ言語ソースファイル
3.リロケータブル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、のいずれか一つは最低限、入力してください。

【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言語ソースプログラムsample.cをコンパイル / アセンブル
3.リロケータブルオブジェクトファイルncrt0.a30とsample.r30をリンク

を行い、アブソリュートモジュールファイルsample.x30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、

●アブソリュートモジュールファイル名sample.x30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..... -as308 "-l"オプション
●リンク時のマップファイル(拡張子.map)の出力指定..... -ln308 "-ms"オプション

を行っています。

```
% nc308△[起動オプション]△<[アセンブリ言語ソースファイル名]△
        [リロケータブルオブジェクトファイル名]△[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

```
% nc308 -osample -as308 "-l" -ln308 "-ms" ncrt0.a30 sample.c<RET>

<RET>：リターンキーの入力を示します。
※リンク時には必ずスタートアッププログラムを先に指定してください。
```

図2.2　コンパイルドライバコマンドの入力例

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、Windows(TM)等のコマンド行の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

```
% nc308△[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

```
% nc308 -c @test.cmd -g<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。

```
test.cmdの記述  ──────▶  ncrt0.a30<CR>
                         sample1.c sample2.r30<CR>
                         -g -as308 -l<CR>
                         -o<CR>
                         sample<CR>

<CR>：改行を示します。
```

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文字までです。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は、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には、エラーとなります。
リンク時にファイルを指定するために拡張子 ".cm$" のln308用コマンドファイルを自動生成します。このため、拡張子が ".cm$" のファイルがある場合には、上書きされる可能性があります。拡張子が ".cm$" のファイルは、使用しないでください。
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には " 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、そのオプションによる機能は取り消されます。

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 --->
```

従って、例えば、
“ -c ”: リロケータブルファイル(拡張子.r30)を作成して処理を終える
“ -S ”: アセンブリ言語ソースファイル(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。つまり、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。

この場合は、アセンブリ言語ソースファイルのみが生成されます。
リロケータブルファイルを作成し、かつ、アセンブリ言語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。

2.1.4　nc308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】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。

表2.1　コンパイルドライバ制御オプション

オプション	機　能
-c	リロケータブルファイル(拡張子.r30)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。
-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。ディレクトリは最大16個まで指定可能です。
-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。※1
-P	プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成します。※1
-S	アセンブリ言語ソースファイル(拡張子.a30)を作成し、処理を終了します。※1
-U*ﾌﾟﾘﾃﾞﾌｧｲﾝﾄﾞﾏｸﾛ名*	指定したプリデファインドマクロを未定義にします。
-silent	起動時のコピーライトメッセージを出力しません。
-dsource (短縮形 -dS)	C言語ソースリストをコメントとして出力した、アセンブリ言語ソースファイル（拡張子 ".a30 "）を生成します。（アセンブル後も削除しません。）
-dsource_in_list (短縮形 -dSL)	" -dsource " の機能に加えて、アセンブリ言語リストファイル(.lst)を生成します。

b.　出力ファイル指定オプション

【表2.2】に出力するアブソリュートモジュール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2.2　出力ファイル指定オプション

オプション	機　能
-o*ファイル名*	ln308が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)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。
-dir ディレクトリ名	ln308が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)の出力先ディレクトリを指定できます。

※1.起動オプション-c、-E、-P、及び-Sを指定しない場合、nc308はln308まで制御を行い、アブソリュートモジュールファイル(拡張子.x30)まで作成します。

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2.3】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2.3　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 2.4】に 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2.4　デバッグ用オプション

オプション	機　能
-g	デバッグ情報をアセンブリ言語ソースファイル（拡張子.a30）に出力します。これにより、C言語レベルデバッグが可能になります。
-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には必ずこのオプションを指定してください。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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。 エントリー版では、本オプションは指定できません。

e. 最適化オプション

【表2.5】にプログラムの実行速度及び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。エントリー版では、すべての最適化オプションが使用できません。

表2.5　最適化オプション（１）

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容量ともに最小にする最大限の最適化を行います。
-OR	なし	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
-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。
-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。
-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_logical_or_combine	-ONLOC	論理ORをまとめる最適化を抑止します。
-Ono_asmopt	-ONA	アセンブラオプティマイザ ”aopt308 ” による最適化を抑止します。
-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。
-Ostatic_to_inline	-OSTI	static宣言された関数を、inline宣言扱いにします。
-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。
-Oglb_jmp	-OGJ	外部分岐の最適化を行います。
-Ofloat_to_inline	-OFTI	浮動小数点のランタイムライブラリをインライン展開します。

f. 生成コード変更オプション

【表2.6】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2.6　生成コード変更オプション（１）

オプション	短縮形	機　能
-fansi	なし	-fnot_reserve_far_and_near, -fnot_reserve_asm, -fnot_reserve_inline、及び-fextend_to_intを有効にします。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fnot_reserve_asm	-fNRA	asmを予約語にしません（”_asm ”のみ有効になります）。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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）。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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。) エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fextend_to_int	-fETI	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）。※1 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fchar_enumerator	-fCE	enumerator(列挙子)の型をint型ではなくunsigned char型で扱います。
-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。
-ffar_RAM	-fFRAM	RAMデータのデフォルト属性をfarにします。
-fnear_ROM	-fNROM	ROMデータのデフォルト属性をnearにします。 エントリー版では、本オプションは指定できません。
-fnear_pointer	-fNP	ポインタ及びアドレスのデフォルトをnearにします。 エントリー版では、本オプションは指定できません。
-fconst_not_ROM	-fCNR	constで指定した型をROMデータとして扱いません。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#pragma EQU)で指定した変数をvolatileで指定した変数とみなしません。

※１.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。

表2.7　生成コード変更オプション（２）

オプション	短縮形	機　能
-fsmall_array	-fSA	far 型の配列を参照する場合、その総サイズが64Kバイト以内であれば、添字の計算を16ビットで行ないます。 エントリー版では、本オプションは指定できません。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-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。 エントリー版では、本オプションは指定できません。
-fJSRW	なし	関数呼び出しの命令のデフォルトをJSR.W命令に変更します。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。 エントリー版では、本オプションは指定できません。
-finfo	なし	インスペクタ、”Stk Viewer”、”Map Viewer ”、”utl308 ”、に必要な情報を出力します。 エントリー版では、本オプションは指定できません。
-fswitch_table	-fST	switch文に対してジャンプテーブルを生成します。
-M82	なし	M32C/80シリーズに対応したコードを生成します。
-fswitch_other_section	-fSOS	switch文に対するテーブルジャンプ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。
-ferase_static_fucntion=関数名	-fESF=関数名	本オプションで指定された関数がstatic 関数の場合、コード生成を行いません。 エントリー版では、本オプションは指定できません。
-fdouble_32	-fD32	本オプション指定時に、double 型をfloat 型として処理します。 エントリー版では、本オプションは指定できません。
-fno_switch_table	-fNST	switch文に対して、比較を行ってから分岐するコードを、生成します。 エントリー版では、本オプションは指定できません。
-fmake_vector_table	-fMVT	可変ベクタテーブルを自動生成します。
-fmake_special_table	-fMST	スペシャルページテーブルを自動生成します。

g.　ライブラリ指定オプション

【表 2.8】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2.8　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。

h. 警告オプション

【表2.9】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(ワ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2.9　警告オプション

オプション	短縮形	機　能
-Wnon_prototype	-WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	-WUP	サポートしていない＃pragmaを使用した場合、警告を出します。
-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。
-Wstdout	なし	エラーメッセージをホストマシンの標準出力（stdout）に出力します。
-Werror_file<file name>	-WEF	タグファイルを出力します。
-Wstop_at_warning	-WSAW	ワーニング発生時にコンパイル処理を停止します。
-Wnesting_comment	-WNC	コメント中に/*を記述した場合に警告を出します。
-Wccom_max_warnings=ワーニング回数	-WCMW	ccom308の出力するワーニングの回数の上限を指定できます。
-Wall	なし	検出可能な警告（ただし、”-Wlarge_to_small”、”Wno_used_argument”で出力される警告を除く）をすべて表示します。
-Wmake_tagfile	-WMT	error および warning が発生した場合にファイル毎にタグファイルを出力します。
-Wuninitialize_variable	-WUV	初期化されていない auto変数に対してワーニングを出力します。
-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、ワーニングを出力します。
-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、「プロトタイプ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。
-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、ワーニングを出力します。
-Wno_used_static_function	-WNUSF	コード生成が不要な static 関数名を、出力します。 エントリー版では、本オプションは指定できません。
-Wno_used_function	-WNUF	未使用のグローバル関数を、リンク時に表示します。
-Wundefined_macro	-WUM	未定義のマクロを#ifの中で使用した場合に警告します
-Wstop_at_link	-WSAL	リンク時に、ワーニングが発生した場合、アブソリュートモジュールファイルの生成を抑止します。

i. アセンブル / リンクオプション

【表 2.10】に as308 及び ln308 のオプションを指定する起動オプションを示します。

表2.10　アセンブル / リンクオプション

オプション	機　能
-as308△<*オプション*>	アセンブルコマンド as308 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。 エントリー版では、本オプションは指定できません。
-ln308△<*オプション*>	リンクコマンドln308 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。 エントリー版では、本オプションは指定できません。

2.2　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、本コンパイラでは、マイコンの初期設定、セクションの配置、割り込みベクタアドレステーブル、等を設定するアセンブリ言語で記述したサンプルのスタートアッププログラムを製品に付属しています。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必要があります。

ここでは、スタートアッププログラムについてと、そのカスタマイズの仕方について説明します。

2.2.1　スタートアッププログラムのサンプル

スタートアッププログラムは、以下の2つのファイルで構成しています。

1. 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。

2. sect308.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【図2.6】から、【図2.10】に、sect308.incのソースプログラムリストを【図2.11】から、【図2.18】、にそれぞれ示します。

```
;***********************************************************************
;
;     C COMPILER for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;     ncrt0.a30 : NC308 startup program
;
;     This program is applicable when using the basic I/O library
;
;     $Id: ncrt0.a30,v 1.23 XXXX/XX/XX XX:XX:Xx xxxxxx Exp $
;
;***********************************************************************

;------------------------------------------------------------------
; HEEP SIZE definition                                  ←[1]
;------------------------------------------------------------------
.if __HEAP__ == 1       ; for HEW

HEAPSIZE    .equ    0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE    .equ    300h

.else                           ; for HEW

HEAPSIZE    .equ    __HEAPSIZE__

.endif
.endif
```

[1]　使用するheapサイズを定義します。

図2.6　スタートアッププログラムリスト(1)（ncrt0.a30）

```
;-----------------------------------------------------------------------
; STACK SIZE definition                                        ←[2]
;-----------------------------------------------------------------------
.if __USTACKSIZE__ == 0

STACKSIZE  .equ   300h

.else                           ; for HEW

STACKSIZE  .equ   __USTACKSIZE__

.endif

;-----------------------------------------------------------------------
; INTERRUPT STACK SIZE definition                              ←[3]
;-----------------------------------------------------------------------
.if __ISTACKSIZE__ == 0

ISTACKSIZE .equ   300h

.else                           ; for HEW

ISTACKSIZE .equ   __ISTACKSIZE__

.endif

;-----------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  definition                         ←[4]
;-----------------------------------------------------------------------
VECTOR_ADR .equ   0fffd00h
SVECTOR_ADR.equ   0fffe00h

;-----------------------------------------------------------------
; special page definition
;-----------------------------------------------------------------
;       macro define for special page
;
;Format:
;   SPECIAL       number
;

SPECIAL    .macro NUM
    .org   0FFFFFEH-(NUM*2)
    .glb   __SPECIAL_@NUM
    .word  __SPECIAL_@NUM  & 0FFFFH
.endm
;-----------------------------------------------------------------------
; Section allocation
;-----------------------------------------------------------------------
    .list OFF
    .include sect308.inc                                       ←[5]
    .list ON

;-----------------------------------------------------------------------
; SBDATA area definition
;-----------------------------------------------------------------------
    .glb   __SB__
__SB__     .equ   data_SE_top
```

[2]　ユーザー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]　割り込みベクタテーブルの開始アドレスを定義します。
[5]　sect30.incをインクルードします。

図2.7　スタートアッププログラムリスト(2)(ncrt0.a30)

```
;=====================================================================
; Initialize Macro declaration
;---------------------------------------------------------------------
;
; when copy less 64K byte
BZERO      .macro TOP_ ,SECT_
    mov.b  #00H, R0L
    mov.l  #TOP_, A1
    mov.w  #sizeof SECT_ , R3
    sstr.b
    .endm

BCOPY .macro      FROM_,TO_,SECT_
    mov.l  #FROM_ ,A0
    mov.l  #TO_ ,A1
    mov.w  #sizeof SECT_ , R3
    smovf.b
    .endm

; when copy over 64K byte
;BZEROL    .macro TOP_,SECT_
;   push.w #sizeof SECT_ >> 16
;   push.w #sizeof SECT_  & 0ffffh
;   pusha  TOP_
;   .stk   8
;
;   .glb   _bzero
;   .call  _bzero,G
;   jsr.a  _bzero
;   .endm
;
;
;BCOPYL    .macro FROM_ ,TO_ ,SECT_
;   push.w #sizeof SECT_  >> 16
;   push.w #sizeof SECT_  & 0ffffh
;   pusha  TO_
;   pusha  FROM_
;   .stk   12
;
;   .glb   _bcopy
;   .call  _bcopy,G
;   jsr.a  _bcopy
;   .endm
;


;=====================================================================
; Interrupt section start
;---------------------------------------------------------------------
    .insf  start,S,0
    .glb   start
    .section     interrupt
start:                                                  ←[6]
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
    ldc    #istack_top,  isp    ;set istack pointer
    mov.b  #02h,0ah
    mov.b  #00h,04h             ;set processer mode      ←[7]
    mov.b  #00h,0ah
    ldc    #0080h,       flg                             ←[8]
    ldc    #stack_top,   sp     ;set stack pointer
    ldc    #data_SE_top, sb     ;set sb register

    fset   b                    ;switch to bank 1
    ldc    #data_SE_top, sb     ;set sb register
    fclr   b                    ;switch to bank 0

    ldc    #VECTOR_ADR,intb
```

[6]　リセット直後はこのラベルstartからスタートします。

[7]　プロセッサ動作モードを設定します。

[8]　割り込み許可レベル及び各種フラグの設定を行います。

図2.8　スタートアッププログラムリスト(3)(ncrt0.a30)

```
;=====================================================================
; NEAR area initialize.
;---------------------------------------------------------------------
; bss zero clear                                          ←[9]
;---------------------------------------------------------------------
    BZERO  bss_SE_top,bss_SE
    BZERO  bss_SO_top,bss_SO
    BZERO  bss_NE_top,bss_NE
    BZERO  bss_NO_top,bss_NO

;---------------------------------------------------------------------
; initialize data section                                 ←[10 ]
;---------------------------------------------------------------------
    BCOPY  data_SEI_top,data_SE_top,data_SE
    BCOPY  data_SOI_top,data_SO_top,data_SO
    BCOPY  data_NEI_top,data_NE_top,data_NE
    BCOPY  data_NOI_top,data_NO_top,data_NO

;=====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss zero clear                                          ←[11 ]
;---------------------------------------------------------------------
;   BZERO  bss_SE_top,bss_SE
;   BZERO  bss_SO_top,bss_SO
;   BZERO  bss_6E_top,bss_6E
;   BZERO  bss_6O_top,bss_6O
    BZERO  bss_FE_top,bss_FE
    BZERO  bss_FO_top,bss_FO

;---------------------------------------------------------------------
; Copy edata_E(O) section from edata_EI(OI) section       ←[12 ]
;---------------------------------------------------------------------
;   BCOPY  data_SEI_top,data_SE_top,data_SE
;   BCOPY  data_SOI_top,data_SO_top,data_SO
;   BCOPY  data_6EI_top,data_6E_top,data_6E
;   BCOPY  data_6OI_top,data_6O_top,data_6O
    BCOPY  data_FEI_top,data_FE_top,data_FE
    BCOPY  data_FOI_top,data_FO_top,data_FO

    ldc    #stack_top,sp

;   .stk   -??    ; Validate this when use BZEROL,BCOPYL

;=====================================================================
; heap area initialize                                    ←[13 ]
;---------------------------------------------------------------------
.if __HEAP__ != 1
    .glb   __mbase
    .glb   __mnext
    .glb   __msize
    mov.l  #(heap_top&0FFFFFFFH), __mbase
    mov.l  #(heap_top&0FFFFFFFH), __mnext
    mov.l  #(HEAPSIZE&0FFFFFFFH), __msize
.endif
;=====================================================================
; Initialize standard I/O                                 ←[14 ]
;---------------------------------------------------------------------
.if __STANDARD_IO__ == 1
    .glb   _init
    .call  _init,G
    jsr.a  _init
.endif
```

[10] near及びSBDATA領域のdataセクションの初期値をRAM領域に転送します。

[11] far領域のbssセクションのゼロクリア処理を行います。※1

[12] far領域のdataセクションの初期値をRAM領域に転送します。※1

[13] 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。

[14] 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。

図2.9　スタートアッププログラムリスト(4)（ncrt0.a30）

※1.far領域を使用しない場合はコメントにします。

```
;=====================================================================
; Call main() function                                    ←[15 ]
;---------------------------------------------------------------------
    ldc    #0h,fb ; for debuger

    .glb   _main
    jsr.a  _main

;=====================================================================
; exit() function                                         ←[16 ]
;---------------------------------------------------------------------
    .glb   _exit
    .glb   $exit
_exit:                      ; End program
$exit:
    jmp    _exit
    .einsf

;=====================================================================
; dummy interrupt function                                ←[17 ]
;---------------------------------------------------------------------
    .glb   dummy_int
dummy_int:
    reit
    .end
;*****************************************************************************
;
;     C COMPILER for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;*****************************************************************************
```

[15] main関数の呼び出しを行います。※1

[16] exit関数部です。

[17] ダミーの割り込み処理関数です。

図2.10　スタートアッププログラムリスト(5)（ncrt0.a30）

※1.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。

```
;*****************************************************************************
;
;     C Compiler for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;     Written by X.Xxxxx
;
;     sect30.inc     : section definition
;     This program is applicable when using the basic I/O library
;
;     $Id: sect308.inc,v 1.16 XXXX/XX/XX XX:XX:XX xxxxxxxx Exp $
;
;*****************************************************************************
;-------------------------------------------------------------
;   Arrangement of section
;
;-------------------------------------------------------------
; Near RAM data area
;-------------------------------------------------------------
; SBDATA area
    .section    data_SE,DATA
    .org   400H
data_SE_top:

    .section    bss_SE,DATA,ALIGN
bss_SE_top:

    .section    data_SO,DATA
data_SO_top:

    .section    bss_SO,DATA
bss_SO_top:

; near RAM area
    .section    data_NE,DATA,ALIGN
data_NE_top:

    .section    bss_NE,DATA,ALIGN
bss_NE_top:

    .section    data_NO,DATA
data_NO_top:

    .section    bss_NO,DATA
bss_NO_top:

;-------------------------------------------------------------
; Stack area
;-------------------------------------------------------------
    .section    stack,DATA,ALIGN
    .blkb  STACKSIZE
    .align
stack_top:

    .blkb  ISTACKSIZE
    .align
istack_top:


;-------------------------------------------------------------
;   heap section
;-------------------------------------------------------------
.if __HEAP__ != 1
    .section    heap,DATA
heap_top:
    .blkb  HEAPSIZE
.endif
```

図2.11　スタートアッププログラムリスト(6)（sect308.inc）

```
;------------------------------------------------------------
; Near ROM data area
;------------------------------------------------------------
    .section    rom_NE,ROMDATA,ALIGN
rom_NE_top:

    .section    rom_NO,ROMDATA
rom_NO_top:

;------------------------------------------------------------
; Far RAM data area
;------------------------------------------------------------
; SBDATA area for #pragma SB16DATA
;   .section    data_SE,DATA
;   .org        10000H
;data_SE_top:
;
;   .section    bss_SE,DATA,ALIGN
;bss_SE_top:
;
;   .section    data_SO,DATA
;data_SO_top:
;
;   .section    bss_SO,DATA
;bss_SO_top:
;
;   .section    data_6E,DATA,ALIGN
;data_6E_top:
;
;   .section    bss_6E,DATA,ALIGN
;bss_6E_top:
;
;   .section    data_6O,DATA
;data_6O_top:
;
;   .section    bss_6O,DATA
;bss_6O_top:
;
    .section    data_FE,DATA
    .org        20000H
data_FE_top:

    .section    bss_FE,DATA,ALIGN
bss_FE_top:

    .section    data_FO,DATA
data_FO_top:

    .section    bss_FO,DATA
bss_FO_top:


;------------------------------------------------------------
; Far ROM data area
;------------------------------------------------------------
    .section    rom_FE,ROMDATA
    .org        0FE0000H
rom_FE_top:

    .section    rom_FO,ROMDATA
rom_FO_top:
```

図2.12　スタートアッププログラムリスト(7)（sect308.inc）

```
;------------------------------------------------------------
; Initial data of 'data' section
;------------------------------------------------------------
    .section    data_SEI,ROMDATA
data_SEI_top:

    .section    data_SOI,ROMDATA
data_SOI_top:

;   .section    data_6EI,ROMDATA
;data_6EI_top:
;
;   .section    data_6OI,ROMDATA
;data_6OI_top:

    .section    data_NEI,ROMDATA
data_NEI_top:

    .section    data_NOI,ROMDATA
data_NOI_top:

    .section    data_FEI,ROMDATA
data_FEI_top:

    .section    data_FOI,ROMDATA
data_FOI_top:

;------------------------------------------------------------
; code area
;------------------------------------------------------------
    .section    interrupt,ALIGN

    .section    program,ALIGN

    .section    program_S
    .org        0FF0000H

.if __MVT__ == 0
;------------------------------------------------------------
; variable vector section
;------------------------------------------------------------
    .section    vector,ROMDATA          ; variable vector table
    .org   VECTOR_ADR

    .lword dummy_int            ; BRK (software int 0)
    .lword dummy_int            ;
    .lword dummy_int            ;
    .lword dummy_int            ;
    .lword dummy_int            ;
    .lword dummy_int            ;
    .lword dummy_int            ;
    .lword dummy_int            ; DMA0 (software int 8)
    .lword dummy_int            ; DMA1 (software int 9)
    .lword dummy_int            ; DMA2 (software int 10)
    .lword dummy_int            ; DMA3 (software int 11)
    .lword dummy_int            ; TIMER A0 (software int 12)
    .lword dummy_int            ; TIMER A1 (software int 13)
    .lword dummy_int            ; TIMER A2 (software int 14)
    .lword dummy_int            ; TIMER A3 (software int 15)
    .lword dummy_int            ; TIMER A4 (software int 16)
    .lword dummy_int            ; uart0 trance (software int 17)
    .lword dummy_int            ; uart0 receive (software int 18)
    .lword dummy_int            ; uart1 trance (software int 19)
    .lword dummy_int            ; uart1 receive (software int 20)
    .lword dummy_int            ; TIMER B0 (software int 21)
    .lword dummy_int            ; TIMER B1 (software int 22)
    .lword dummy_int            ; TIMER B2 (software int 23)
    .lword dummy_int            ; TIMER B3 (software int 24)
    .lword dummy_int            ; TIMER B4 (software int 25)
    .lword dummy_int            ; INT5 (software int 26)
    .lword dummy_int            ; INT4 (software int 27)
    .lword dummy_int            ; INT3 (software int 28)
    .lword dummy_int            ; INT2 (software int 29)
    .lword dummy_int            ; INT1 (software int 30)
    .lword dummy_int            ; INT0 (software int 31)
```

図2.13　スタートアッププログラムリスト(8)(sect308.inc)

```
    .lword dummy_int            ; TIMER B5 (software int 32)
    .lword dummy_int            ; uart2 trance/NACK (software int 33)
    .lword dummy_int            ; uart2 receive/ACK (software int 34)
    .lword dummy_int            ; uart3 trance/NACK (software int 35)
    .lword dummy_int            ; uart3 receive/ACK (software int 36)
    .lword dummy_int            ; uart4 trance/NACK (software int 37)
    .lword dummy_int            ; uart4 receive/ACK (software int 38)
    .lword dummy_int            ; uart2 bus collision (software int 39)
    .lword dummy_int            ; uart3 bus collision (software int 40)
    .lword dummy_int            ; uart4 bus collision (software int 41)
    .lword dummy_int            ; A-D Convert (software int 42)
    .lword dummy_int            ; input key (software int 43)
    .lword dummy_int            ; software int 44
    .lword dummy_int            ; software int 45
    .lword dummy_int            ; software int 46
    .lword dummy_int            ; software int 47
    .lword dummy_int            ; software int 48
    .lword dummy_int            ; software int 49
    .lword dummy_int            ; software int 50
    .lword dummy_int            ; software int 51
    .lword dummy_int            ; software int 52
    .lword dummy_int            ; software int 53
    .lword dummy_int            ; software int 54
    .lword dummy_int            ; software int 55
    .lword dummy_int            ; software int 56
    .lword dummy_int            ; software int 57
    .lword dummy_int            ; software int 58
    .lword dummy_int            ; software int 59
    .lword dummy_int            ; software int 60
    .lword dummy_int            ; software int 61
    .lword dummy_int            ; software int 62
    .lword dummy_int            ; software int 63
.else       ; __MVT__
    .section      __NC_rvector,ROMDATA
    .org          VECTOR_ADR
.endif      ; __MVT__

.if __MST__ == 0
;=============================================================
; fixed vector section
;-------------------------------------------------------------
    .section      svector,ROMDATA           ; specialpage vector table
    .org          SVECTOR_ADR
;=============================================================
; special page defination
;-------------------------------------------------------------
;   macro is defined in ncrt0.a30
;   Format: SPECIAL number
;
;-------------------------------------------------------------
;   SPECIAL 255
;   SPECIAL 254
;   SPECIAL 253
;   SPECIAL 252
;   SPECIAL 251
;   SPECIAL 250
;   SPECIAL 249
;   SPECIAL 248
;   SPECIAL 247
;   SPECIAL 246
;   SPECIAL 245
;   SPECIAL 244
;   SPECIAL 243
;   SPECIAL 242
;   SPECIAL 241
;   SPECIAL 240
;   SPECIAL 239
;   SPECIAL 238
;   SPECIAL 237
;   SPECIAL 236
;   SPECIAL 235
;   SPECIAL 234
;   SPECIAL 233
;   SPECIAL 232
;   SPECIAL 231
;   SPECIAL 230
```

図2.14　スタートアッププログラムリスト(9)(sect308.inc)

```
;    SPECIAL 229
;    SPECIAL 228
;    SPECIAL 227
;    SPECIAL 226
;    SPECIAL 225
;    SPECIAL 224
;    SPECIAL 223
;    SPECIAL 222
;    SPECIAL 221
;    SPECIAL 220
;    SPECIAL 219
;    SPECIAL 218
;    SPECIAL 217
;    SPECIAL 216
;    SPECIAL 215
;    SPECIAL 214
;    SPECIAL 213
;    SPECIAL 212
;    SPECIAL 211
;    SPECIAL 210
;    SPECIAL 209
;    SPECIAL 208
;    SPECIAL 207
;    SPECIAL 206
;    SPECIAL 205
;    SPECIAL 204
;    SPECIAL 203
;    SPECIAL 202
;    SPECIAL 201
;    SPECIAL 200
;    SPECIAL 199
;    SPECIAL 198
;    SPECIAL 197
;    SPECIAL 196
;    SPECIAL 195
;    SPECIAL 194
;    SPECIAL 193
;    SPECIAL 192
;    SPECIAL 191
;    SPECIAL 190
;    SPECIAL 189
;    SPECIAL 188
;    SPECIAL 187
;    SPECIAL 186
;    SPECIAL 185
;    SPECIAL 184
;    SPECIAL 183
;    SPECIAL 182
;    SPECIAL 181
;    SPECIAL 180
;    SPECIAL 179
;    SPECIAL 178
;    SPECIAL 177
;    SPECIAL 176
;    SPECIAL 175
;    SPECIAL 174
;    SPECIAL 173
;    SPECIAL 172
;    SPECIAL 171
;    SPECIAL 170
;    SPECIAL 169
;    SPECIAL 168
;    SPECIAL 167
;    SPECIAL 166
;    SPECIAL 165
;    SPECIAL 164
;    SPECIAL 163
;    SPECIAL 162
;    SPECIAL 161
;    SPECIAL 160
;    SPECIAL 159
;    SPECIAL 158
;    SPECIAL 157
;    SPECIAL 156
;    SPECIAL 155
;    SPECIAL 154
```

図2.15　スタートアッププログラムリスト(10)（sect308.inc）

```
;    SPECIAL 153
;    SPECIAL 152
;    SPECIAL 151
;    SPECIAL 150
;    SPECIAL 149
;    SPECIAL 148
;    SPECIAL 147
;    SPECIAL 146
;    SPECIAL 145
;    SPECIAL 144
;    SPECIAL 143
;    SPECIAL 142
;    SPECIAL 141
;    SPECIAL 140
;    SPECIAL 139
;    SPECIAL 138
;    SPECIAL 137
;    SPECIAL 136
;    SPECIAL 135
;    SPECIAL 134
;    SPECIAL 133
;    SPECIAL 132
;    SPECIAL 131
;    SPECIAL 130
;    SPECIAL 129
;    SPECIAL 128
;    SPECIAL 127
;    SPECIAL 126
;    SPECIAL 125
;    SPECIAL 124
;    SPECIAL 123
;    SPECIAL 122
;    SPECIAL 121
;    SPECIAL 120
;    SPECIAL 119
;    SPECIAL 118
;    SPECIAL 117
;    SPECIAL 116
;    SPECIAL 115
;    SPECIAL 114
;    SPECIAL 113
;    SPECIAL 112
;    SPECIAL 111
;    SPECIAL 110
;    SPECIAL 109
;    SPECIAL 108
;    SPECIAL 107
;    SPECIAL 106
;    SPECIAL 105
;    SPECIAL 104
;    SPECIAL 103
;    SPECIAL 102
;    SPECIAL 101
;    SPECIAL 100
;    SPECIAL 99
;    SPECIAL 98
;    SPECIAL 97
;    SPECIAL 96
;    SPECIAL 95
;    SPECIAL 94
;    SPECIAL 93
;    SPECIAL 92
;    SPECIAL 91
;    SPECIAL 90
;    SPECIAL 89
;    SPECIAL 88
;    SPECIAL 87
;    SPECIAL 86
;    SPECIAL 85
;    SPECIAL 84
;    SPECIAL 83
;    SPECIAL 82
;    SPECIAL 81
;    SPECIAL 80
;    SPECIAL 79
;    SPECIAL 78
```

図2.16　スタートアッププログラムリスト(11)(sect308.inc)

```
;    SPECIAL 77
;    SPECIAL 76
;    SPECIAL 75
;    SPECIAL 74
;    SPECIAL 73
;    SPECIAL 72
;    SPECIAL 71
;    SPECIAL 70
;    SPECIAL 69
;    SPECIAL 68
;    SPECIAL 67
;    SPECIAL 66
;    SPECIAL 65
;    SPECIAL 64
;    SPECIAL 63
;    SPECIAL 62
;    SPECIAL 61
;    SPECIAL 60
;    SPECIAL 59
;    SPECIAL 58
;    SPECIAL 57
;    SPECIAL 56
;    SPECIAL 55
;    SPECIAL 54
;    SPECIAL 53
;    SPECIAL 52
;    SPECIAL 51
;    SPECIAL 50
;    SPECIAL 49
;    SPECIAL 48
;    SPECIAL 47
;    SPECIAL 46
;    SPECIAL 45
;    SPECIAL 44
;    SPECIAL 43
;    SPECIAL 42
;    SPECIAL 41
;    SPECIAL 40
;    SPECIAL 39
;    SPECIAL 38
;    SPECIAL 37
;    SPECIAL 36
;    SPECIAL 35
;    SPECIAL 34
;    SPECIAL 33
;    SPECIAL 32
;    SPECIAL 31
;    SPECIAL 30
;    SPECIAL 29
;    SPECIAL 28
;    SPECIAL 27
;    SPECIAL 26
;    SPECIAL 25
;    SPECIAL 24
;    SPECIAL 23
;    SPECIAL 22
;    SPECIAL 21
;    SPECIAL 20
;    SPECIAL 19
;    SPECIAL 18
;
.else       ; __MST__
    .section      __NC_svector,ROMDATA
    .org          SVECTOR_ADR
.endif      ; __MST__
```

図2.17　スタートアッププログラムリスト(12)(sect308.inc)

```
;=============================================================
; fixed vector section
;-------------------------------------------------------------
    .section     fvector,ROMDATA
    .org   0FFFFDCh
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
;
;*****************************************************************************
;
;     C Compiler for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;*****************************************************************************
;
```

図2.18　スタートアッププログラムリスト(13)(sect308.inc)

2.2.2　スタートアッププログラムのカスタマイズ

a.　スタートアッププログラムの処理概要

「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。

●SBDATA領域（SB相対アドレッシングモードでアクセスする領域）の先頭アドレス値(__SB__)を設定します。
●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。
1. デフォルトでは
bss_NE、bss_NO、bss_SE、bss_SO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_NEI、data_NOI、data_SEI、data_SOI)の初期値をRAM領域(data_NE、data_NO、data_SE、data_SO)に転送する処理を行います。
2. #pragma SB16DATA 拡張機能 を使用する場合では
上記 1. での処理の代わりに、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_NEI、data_NOI)の初期値をRAM領域(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。
1. デフォルトでは
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。
2. #pragma SB16DATA 拡張機能 を使用する場合では
上記 1. での処理の代わりに、bss_SE、bss_SO、bss_6E、bss_6O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_6EI、data_6OI)の初期値をRAM領域(data_SE、data_SO、data_6E、data_6O)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。
●FBレジスタの初期化を行います。
●main関数の呼び出しを行います。

b. スタートアッププログラムの変更手順

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概略を【図2.19】に示します。

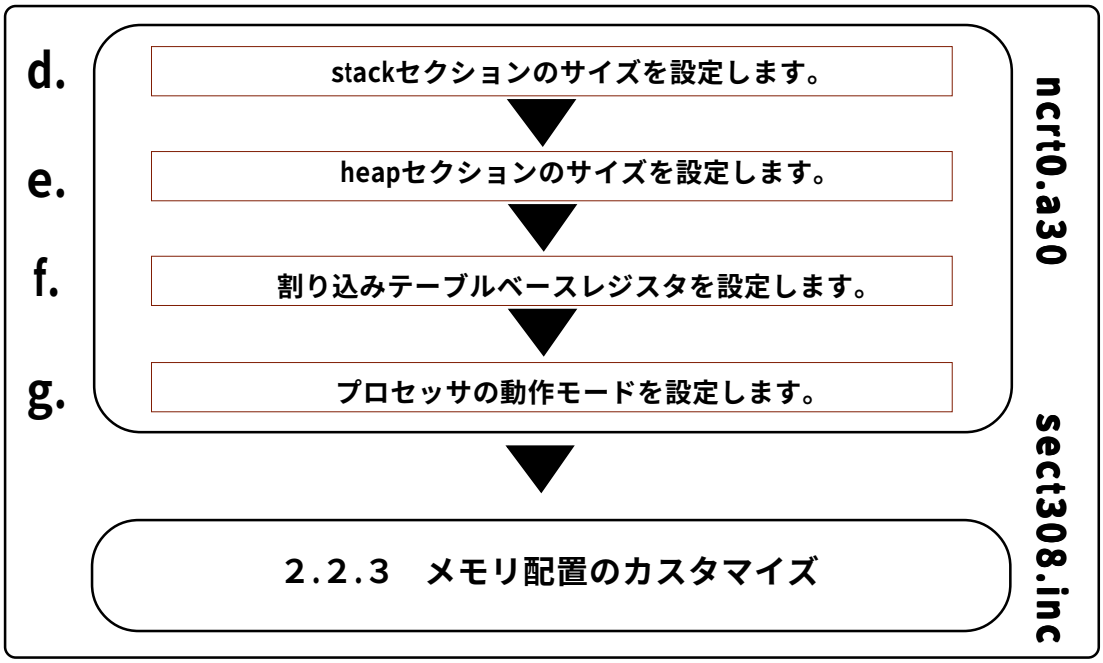


図2.19　スタートアッププログラムの変更手順例

c. 注意を要する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*1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2.20】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、ncrt0.a30内のinit関数の呼び出し部をコメントにしてください。

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

図2.20　init関数呼び出し部（ncrt0.a30）

なお、sprintf、sscanfのみを使用する場合、init関数を呼び出す必要はありません。

※1.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。
デフォルトでは、M16C/80を指定した初期化を行っています。
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。

(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セクションの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 ＊__mbaseの初期化
(2)外部変数char ＊__mnextの初期化
　　heapセクションの先頭アドレスを表すラベルheap_topで初期化します。
(3)外部変数unsigned long __msizeの初期化
　　「2.2.2 e. heapセクションのサイズの設定」で設定した"HEAPSIZE"で初期化します。

【図2.21】にncrt0.a30内の初期化部を示します。

```
;=====================================================================
; heap area initialize
;---------------------------------------------------------------------
.if __HEAP__ != 1
    .glb  __mbase
    .glb  __mnext
    .glb  __msize
    mov.l #(heap_top&0FFFFFFFH), __mbase
    mov.l #(heap_top&0FFFFFFFH), __mnext
    mov.l #(HEAPSIZE&0FFFFFFFH), __msize
.endif
```

図2.21　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ときは、この初期化部をすべてコメントにしてください。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サイズを節約できます。

(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項目に注意してください。
　　①B、Dフラグはクリアした状態で呼び出してください。
　　②Uフラグはセットした状態で呼び出してください。

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。

スタックは必ず使用しますので、必ず領域を確保してください。スタックサイズは、使用する最大サイズを設定してください*1。

スタックサイズは、スタックサイズ計算ユーティリティ Stk Viewer を使用して求めることができます。

e.　heapセクションのサイズの設定

heapセクションのサイズは、プログラム中でメモリ管理関数calloc、mallocを使用して確保する最大のメモリ使用量を設定します。メモリ管理関数を使用しないときはサイズを0に設定してください。heapセクションは物理的なRAM領域を越えないように設定してください。

```
;------------------------------------------------------------------------
; HEEP SIZE definition
;------------------------------------------------------------------------
.if __HEAP__ == 1           ; for HEW

HEAPSIZE  .equ  0h

.else
.if __HEAPSIZE__ == 0

HEAPSIZE  .equ  300h

.else                       ; for HEW

HEAPSIZE  .equ  __HEAPSIZE__

.endif
.endif
```

図2.22　heapサイズの設定例（ncrt0.a30）

f.　割り込みベクタテーブルの設定

ncrt0.a30中の【図2.23】の部分において割り込みベクタテーブルの先頭アドレスを設定します。設定した値で、INTBレジスタが初期化されます。

```
;------------------------------------------------------------------------
; INTERRUPT VECTOR ADDRESS  definition
;------------------------------------------------------------------------
VECTOR_ADR      .equ  0fffd00h
SVECTOR_ADR     .equ  0fffe00h
```

図2.23　割り込みベクタテーブルの先頭アドレスの設定例（ncrt0.a30）

サンプルのスタートアッププログラムでは、

0FFFD00H ～ 0FFFDFFH　　：割り込みベクタテーブル
0FFFE00H ～ 0FFFFFFH　　：スペシャルページベクタテーブル
　　および、固定ベクタテーブル

で使用するための値が、設定されています。

通常は、設定値を変更する必要はありません。

※1. スタートアッププログラム内でもスタックを使用しています。main()関数を呼び出す前に初期値を再ロードしていますが、main()関数等で使用するスタックサイズが少ない場合は、考慮が必要です。

g.　プロセッサモードレジスタの設定

ncrt0.a30中の【図2.24】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
;------------------------------------------------------------------------
       :
    (omitted)
       :
    mov.b #00h,04h          ;set processer mode
       :
    (omitted)
       :
;========================================================================
```

図2.24　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、NC308のようなクロス環境のコンパイラでは、ユーザがメモリ配置を決定する必要があります。

NC308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、割り込みベクトルアドレス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構成されます。

表2.12　セクション名

セクションベース名	_	属性

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	初期値をもつ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、#pragma ROM、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/6	N...near属性※1	data, bss, rom
	F...far属性※1	
	S...SBDATA属性	data, bss
	6...SB16DATA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	

※1. near、farとは、NC308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000000H番地から00FFFFH番地まで

far.....アクセス範囲は000000H番地から0FFFFFFH番地まで

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。

表2.15　セクションの名称

セクション名	内　容
stack	スタックとして使用する領域です。アドレス0400Hから0FFFFHの間に配置してください。
heap	メモリ管理関数(malloc等)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。
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。
fvector	マイコンの固定ベクタの内容を格納します。
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。
switch_table	switch文に対するテーブルコードを格納します。 このセクションは、オプション ”-fswitch_other_section (-fSOS) ”を使用した場合のみ生成されます。
_NC_vector	#pragma INTERRUPT ベクタ番号指定機能を用いた場合に生成します。
_NC_svector	#pragma SPECIAL で指定されたベクタを格納します。

これらのセクションの配置は、スタートアッププログラムのインクルードファイルsect308.incで設定します。このインクルードファイルの内容を変更することで、セクションの配置を変更することができます。

サンプルのスタートアッププログラムのインクルードファイルsect308.incのセクション配置例を【図2.25 (1/2)】に示します。

また、「#pragma SB16DATA 拡張機能」を使用した場合のセクション配置例を【図2.26 (2/2)】に示します。

「#pragma SB16DATA 拡張機能」については、”付録B 拡張機能リファレンス　B.7 #pragma　拡張機能 ” を参照してください。

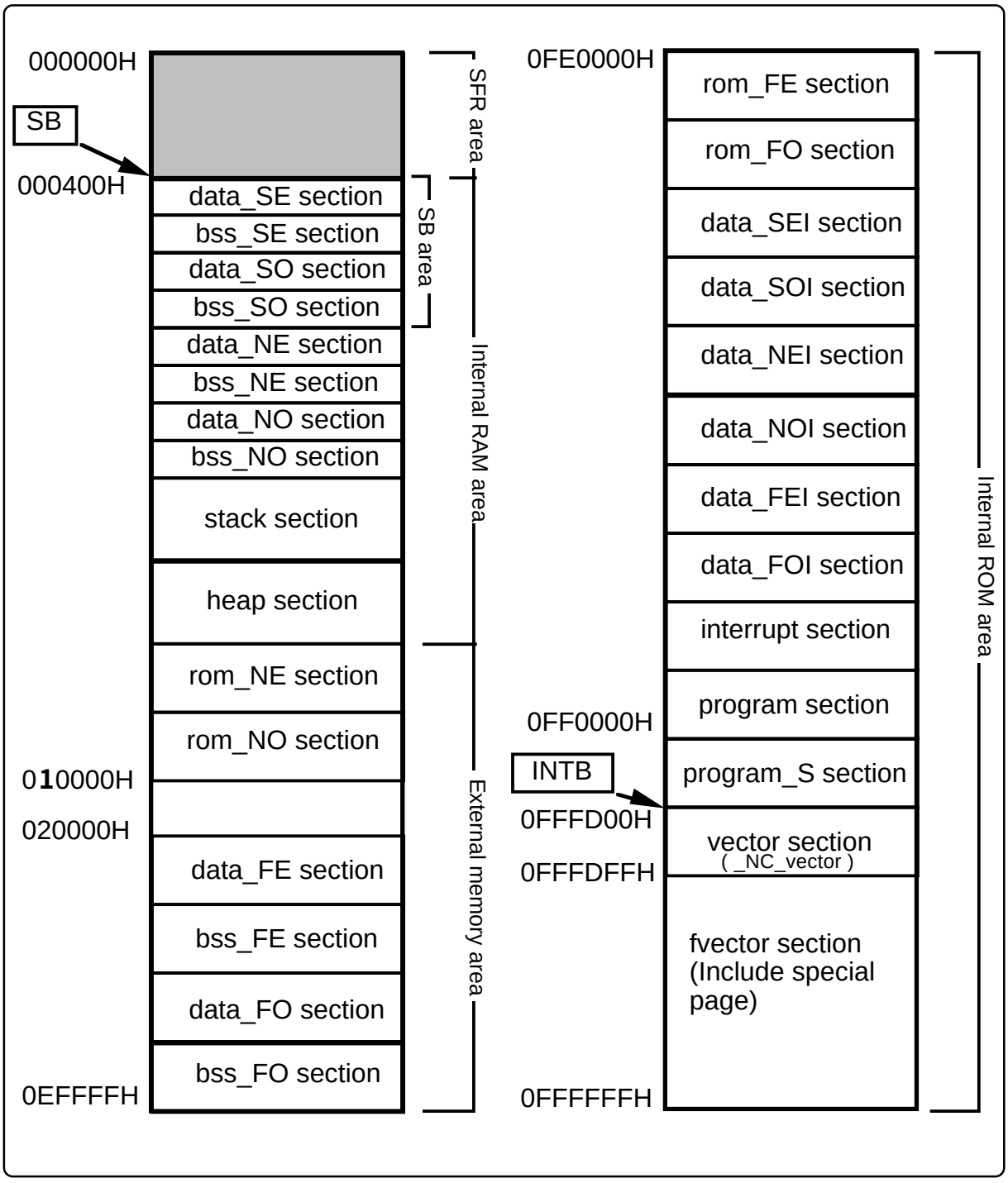


図2.25　セクション配置例 (1/2)

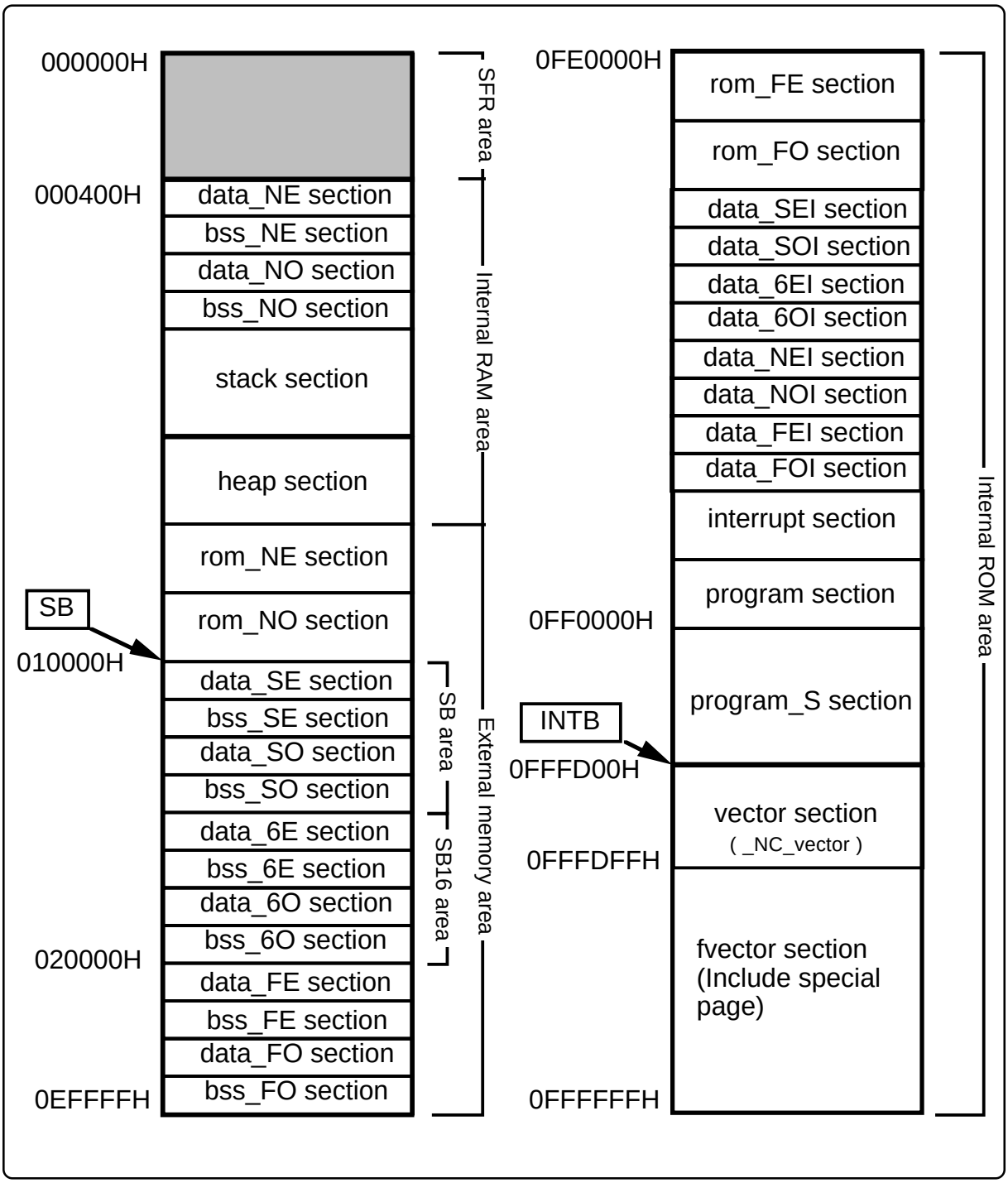


図2.26　セクション配置例 (2/2)

b.　メモリ配置設定用ファイルの概要

「sect308.inc について」

このプログラムは、ncrt0.a30からインクルードされます。このプログラムは主に以下の処理を行います。

●各セクション配置(順序)の設定を行います。
●セクション開始アドレスの設定を行います。
●stackセクション及びheapセクション領域のサイズを定義します。
●割り込みベクタテーブルを設定します。
●固定ベクタテーブルを設定します。

c.　sect308.incの変更手順

スタートアッププログラムのメモリ配置設定用ファイルを、組み込むシステムに合わせて変更する方法の概略を【図2.27】に示します。

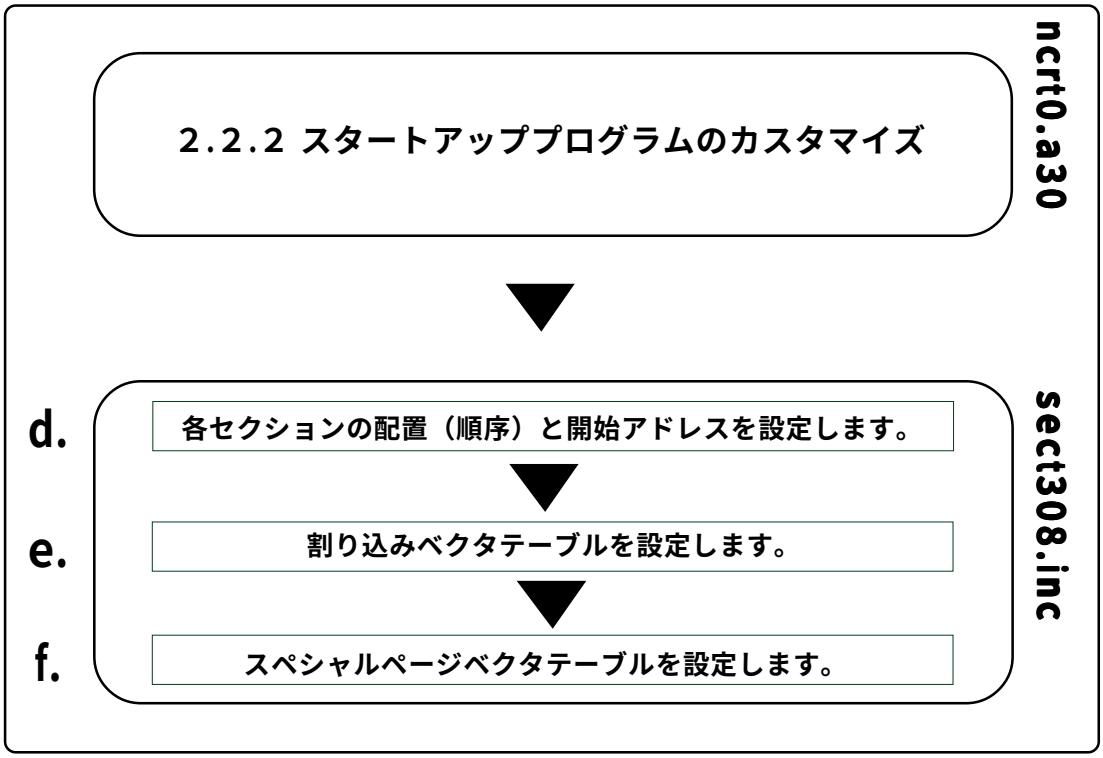


図2.27　メモリ配置の変更手順例

d. セクション配置(順序)と開始アドレスの設定

セクションの配置(順序)と開始アドレスの設定(プログラム及びデータのROM/RAMへの配置設定)は、スタートアッププログラムのインクルードファイルsect308.incで行います。

セクションの配置は、sect308.incに定義された順に配置されます。また、そのセクションの開始アドレスはas308の疑似命令.ORGを用いて指定します。【図2.28】に設定例を示します。

```
        .section        program_S
        .org            0FF0000H        ←programセクションの開始アドレスの設定
;
```

図2.28　セクション開始アドレスの設定例

セクションに対する開始アドレスの指定がない場合、前に定義したセクションに連続してメモリに配置されます。

(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。セクションの配置には、以下の規則に従ってください。

(a)RAM領域に配置するセクション

- ●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
- ●data_6Eセクション
- ●data_6Oセクション
- ●stackセクション
- ●heapセクション
- ●bss_SEセクション
- ●bss_SOセクション
- ●bss_NEセクション
- ●bss_NOセクション
- ●bss_FEセクション
- ●bss_FOセクション
- ●bss_6Eセクション
- ●bss_6Oセクション

(b)ROMに配置するセクション

- ●rom_NEセクション
- ●rom_NOセクション
- ●rom_FEセクション
- ●rom_FOセクション
- ●programセクション
- ●interruptセクション
- ●fvector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
- ●data_6E1セクション
- ●data_6O1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。

(a)0H～0FFFFH(near領域)にのみ配置できるセクション

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

(b)0FF0000H～0FFFFFFHにのみ配置できるセクション

●program_S
●fvector

(c)M32C/80シリーズの全メモリ空間に配置できるセクション

●data_FE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
●bss_FEセクション
●data_6Eセクション
●data_6Oセクション
●program
●data_FO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
●bss_FOセクション
●data_6EIセクション
●data_6OIセクション
●bss_6Eセクション
●bss_6Oセクション
●vector
●_NC_vector

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, data_SEIセクション
●data_SO, data_SOIセクション
●data_NE, data_INEセクション
●data_NO, data_INOセクション
●data_FE, data_IFEセクション
●data_FO, data_IFOセクション
●bss_SEセクション
●bss_SOセクション
●data_6E,data_6EIセクション
●bss_6E,bss_6Oセクション
●bss_NEセクション
●bss_NOセクション
●bss_FEセクション
●bss_FOセクション
●rom_NEセクション
●rom_NOセクション
●rom_FEセクション
●rom_FOセクション
●data_6O,data_6OIセクション

(2) シングルチップモードにおけるセクション配置例

シングルチップモードにおけるセクション配置を行うインクルードファイルsect308.incの記述例を【図2.29】、【図2.30】、【図2.31】に示します。

```
;***********************************************************************
;
;     C Compiler for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;     Written by X.Xxxxx
;
;     sect30.inc     : section definition
;     This program is applicable when using the basic I/O library
;
;     $Id: sect308.inc,v 1.16 XXXX/XX/XX XX:XX:XX xxxxxxxx Exp $
;
;***********************************************************************
;-----------------------------------------------------------------
;
;   Arrangement of section
;
;-----------------------------------------------------------------
; Near RAM data area
;-----------------------------------------------------------------
; SBDATA area
    .section    data_SE,DATA
    .org   400H
data_SE_top:

    .section    bss_SE,DATA,ALIGN
bss_SE_top:

    .section    data_SO,DATA
data_SO_top:

    .section    bss_SO,DATA
bss_SO_top:

; near RAM area
    .section    data_NE,DATA,ALIGN
data_NE_top:

    .section    bss_NE,DATA,ALIGN
bss_NE_top:

    .section    data_NO,DATA
data_NO_top:

    .section    bss_NO,DATA
bss_NO_top:

;-----------------------------------------------------------------
; Stack area
;-----------------------------------------------------------------
    .section    stack,DATA,ALIGN
    .blkb  STACKSIZE
    .align
stack_top:

    .blkb  ISTACKSIZE
    .align
istack_top:


;-----------------------------------------------------------------
;   heap section
;-----------------------------------------------------------------
.if __HEAP__ != 1
    .section    heap,DATA
heap_top:
    .blkb  HEAPSIZE
.endif
```

図2.29　シングルチップモードにおけるsect308.incリスト(1)

```
;------------------------------------------------------------
; Near ROM data area
;------------------------------------------------------------
    .section    rom_NE,ROMDATA,ALIGN
rom_NE_top:

    .section    rom_NO,ROMDATA
rom_NO_top:

;------------------------------------------------------------
; Far RAM data area
;------------------------------------------------------------
; SBDATA area for #pragma SB16DATA
;   .section    data_SE,DATA
;   .org        10000H
;data_SE_top:
;
;   .section    bss_SE,DATA,ALIGN
;bss_SE_top:
;
;   .section    data_SO,DATA
;data_SO_top:
;
;   .section    bss_SO,DATA
;bss_SO_top:
;
;   .section    data_6E,DATA,ALIGN
;data_6E_top:
;
;   .section    bss_6E,DATA,ALIGN
;bss_6E_top:
;
;   .section    data_6O,DATA
;data_6O_top:
;
;   .section    bss_6O,DATA
;bss_6O_top:
;
    .section    data_FE,DATA
    .org        20000H
data_FE_top:

    .section    bss_FE,DATA,ALIGN
bss_FE_top:

    .section    data_FO,DATA
data_FO_top:

    .section    bss_FO,DATA
bss_FO_top:
```

←セクションサイズがゼロですので、取り除く事ができます。

この場合、ncrt0.a30のfar領域の初期化プログラムも取り除く必要があります。

```
;------------------------------------------------------------
; Far ROM data area
;------------------------------------------------------------
    .section    rom_FE,ROMDATA
    .org        0FE0000H
rom_FE_top:

    .section    rom_FO,ROMDATA
rom_FO_top:

;------------------------------------------------------------
; Initial data of 'data' section
;------------------------------------------------------------
    .section    data_SEI,ROMDATA
data_SEI_top:

    .section    data_SOI,ROMDATA
data_SOI_top:

;   .section    data_6EI,ROMDATA
;data_6EI_top:
;
;   .section    data_6OI,ROMDATA
;data_6OI_top:
```

図2.30　シングルチップモードにおけるsect308.incリスト(2)

```
    .section    data_NEI,ROMDATA
data_NEI_top:

    .section    data_NOI,ROMDATA
data_NOI_top:

    .section    data_FEI,ROMDATA
data_FEI_top:

    .section    data_FOI,ROMDATA
data_FOI_top:

;------------------------------------------------------------
; code area
;------------------------------------------------------------
    .section    interrupt,ALIGN

    .section    program,ALIGN

    .section    program_S
    .org        0FF0000H

.if __MVT__ == 0
;------------------------------------------------------------
; variable vector section
;------------------------------------------------------------
    .section    vector,ROMDATA          ; variable vector table
    .org   VECTOR_ADR

    .lword dummy_int            ; BRK (software int 0)
        :
        :
    (omitted)
        :
        :

    .lword dummy_int            ; software int 63
.else     ; __MVT__
    .section    __NC_rvector,ROMDATA
    .org        VECTOR_ADR
.endif    ; __MVT__

.if __MST__ == 0
;============================================================
; fixed vector section
;------------------------------------------------------------
    .section    svector,ROMDATA         ; specialpage vector table
    .org        SVECTOR_ADR
;============================================================
; special page defination
;------------------------------------------------------------
;   macro is defined in ncrt0.a30
;   Format: SPECIAL number
;
;------------------------------------------------------------
;   SPECIAL 255
        :
        :
    (omitted)
        :
        :
;   SPECIAL 18
;
.else     ; __MST__
    .section    __NC_svector,ROMDATA
    .org        SVECTOR_ADR
.endif    ; __MST__
;============================================================
; fixed vector section
;------------------------------------------------------------
    .section    fvector,ROMDATA
    .org   0FFFFDCh
UDI:
    .lword dummy_int
OVER_FLOW:
    .lword dummy_int
```

図2.31　シングルチップモードにおけるsect308.incリスト(3)

```
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
;
;******************************************************************************
;
;     C Compiler for M16C/80
; COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
; AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;
;
;******************************************************************************
;
```

図2.32　シングルチップモードにおけるsect308.incリスト（4）

e. 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、

1.コンパイラオプション ”-fmake_vector_table(-fMVT)”を使用して、割り込みベクタテーブル中の可変ベクタテーブルを自動設定する。

2. sect308.inc中のvectorセクションの割り込みベクタテーブルの設定を行なう。

のいずれかを行って、割り込みベクタテーブルの設定してください。

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.コンパイラオプション ”-fmake_vector_table(-fMVT)”を使用して設定

’付録A コマンドオプションリファレンス”の ”-fmake_vector_table(-fMVT)”、’付録 B 拡張機能リファレンス”の ’#pragma INTERRUPT”の項を参照してください。

2.sect308.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、sect308.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2.33】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

    .lworddummy_int         ; BRK (software int 0)
      :
    (omitted)
      :
    .lworddummy_int         ; DMA0 (software int 8)
    .lworddummy_int         ; DMA1 (software int 9)
    .lworddummy_int         ; DMA2 (software int 10)
      :
    (omitted)
      :
    .lworddummy_int         ; uart0 trance (software int 17)
    .lworddummy_int         ; uart0 receive (software int 18)
    .lworddummy_int         ; uart1 trance (software int 19)
    .lworddummy_int         ; uart1 receive (software int 20)
    .lworddummy_int         ; TIMER B0 (software int 21)
      :
    (omitted)
      :
    .lworddummy_int         ; INT5 (software int 26)
    .lworddummy_int         ; INT4 (software int 27)
      :
    (omitted)
      :
    .lworddummy_int         ; uart2 trance/NACK (software int 33)
    .lworddummy_int         ; uart2 receive/ACK (software int 34)
      :
    (omitted)
      :
    .lworddummy_int         ; software int 63
```

※dummy_intはダミーの割り込み処理関数です。

図2.33　割り込みベクタテーブルの設定例

次の手順でsect308.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疑似命令.GLBで外部参照宣言します。
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。

② 使用する割り込みも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2.34】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 trance (for user)
    .lword   dummy_int      ; uart0 receive (for user)
    .glb    _uarttrn                                    ←上記①の処理
    .lword   _uarttrn       ; uart1 trance (for user)   ←上記②の処理

    (以下省略)
```

図2.34　割り込みベクタテーブルの設定例

f.　スペシャルページベクタテーブルの設定

#pragma SPECIALをご使用になる場合には、スペシャルページベクタテーブルの設定が必要です。

スペシャルページベクタテーブルの設定は、

1.コンパイラオプション ”-fmake_special_table(-fMST)”を使用して、スペシャルページベクタテーブルを自動設定する。

2. sect30.incでスペシャルページベクタテーブルの設定を行なう。

のいずれかで行ってください。

1.コンパイラオプション ”-fmake_special_table(-fMST)”を使用して設定

’付録A コマンドオプションリファレンス ”の ”-fmake_special_table(-fMST) ”、’付録 B 拡張機能リファレンス ”の ’#pragma SPECIAL ”の項を参照してください。

2.sect308.incでスペシャルページベクタテーブルの設定

【図2.35】にスペシャルページベクタテーブルの設定例を示します。

```
;=============================================================
; special page defination
;-------------------------------------------------------------
;       macro is defined in ncrt0.a30
;       Format: SPECIAL number
;
;-------------------------------------------------------------
;
        SPECIAL 42
        SPECIAL 41
        SPECIAL 40
;
        SPECIAL 31
        SPECIAL 30
;
        SPECIAL 22
        SPECIAL 21
        SPECIAL 20
        SPECIAL 19
        SPECIAL 18
;
```

【図2.35】スペシャルページベクタテーブルの設定例

デフォルトでは、スペシャルページベクタテーブルは、コメントになっています。
“ SPECIAL ’は、マクロであり、 ’#pragma SPECIAL ”で定義した関数名と関連づけを行う働きをしています。
使用するスペシャルページ番号の定義は、コメントを外すことにより、設定します。
スペシャルページ番号は必ずしも連続になっている必要はありませんが、必ず降順に設定してください。
スペシャルページベクタテーブルは、 ”SBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 ” を使用して、SPECIALページベクタ定義ファイルとして出力することができます。詳しくは、 ”付録G　SBDATA宣言&SPECIALページ関 数宣言ユーティリティ ”を参照してください。

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。株式会社ルネサス テクノロジ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時に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。

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。

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には、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。

以下の例のようなC言語記述をSFR領域に行った場合、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う可能性があります。

[例：SFR領域に対するCソースコード記述]

```
#pragma ADDRESS TA0IC   006Ch   /* M16C/80 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 */
  }
```

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、以下の図に示される変数portは、読み出し結果を使用しないので、読み出し動作が削除されます。

```
extern int port;
funC()
{
    port;
}
```

図3.1　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3.2】に示すようにvolatile修飾子を付加してください。

```
extern int volatile port;
funC()
{
    port;
}
```

図3.2　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3.3　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文を記述された場合、本コンパイラでは、アセンブラ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、アセンブラコードは、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3.4　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。

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

図3.5　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？
a = ++b;       // a = b なのか、a = (b+1) なのか？
```

図3.6　volatile修飾子の解釈が曖昧になる例

連続するビット操作については、volatile修飾子の指定があっても、最適化を行うと、まとめてビット操作を行うコードが生成されます。（参照の順序が無くなり、同時に操作される）
まとめて操作されないようにするには、 ”-Ono_bit (短縮形 -ONB) ”オプションを使用してください。

3.1.4　register変数の使用に関する注意事項

a.　register修飾と ”-fenable_register“ オプションについて

　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。むやみに使用すると、逆効果となりますので、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。

3.1.5　スタートアップの扱いについて

　ご使用のマイコン機種、お客様のシステムによ、りスタートアップを変更していただく必要があります。

　機種により変更を必要とする内容は対応機種のデータブック等を参照いただき、添付のスタートアップファイルを修正して、ご使用ください。

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16ビット幅で計算します。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3.7】に示しますように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 -fsmall_array (-fSA)※2 オプションを付加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3.7　far型配列のextern宣言例

※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.-fsmall_array(-fSA)オプションは、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。エントリー版では、本オプションは指定できません。

c.　配列の添え字に関して

配列の添え字は、その配列の1つの要素のサイズにより演算時において型拡張されます。

① サイズが2バイト以上（char型もしくはsigned char型以外）の場合
　添え字は必ずint型に拡張されて演算されます。
② サイズが64K以上のfar型配列の場合
　添え字は必ずlong型に拡張されて演算されます。

したがって、配列の添え字になる変数をchar型で宣言するとint型への拡張が参照する度に行われコード効率が良くありません。このような場合、配列の添え字になる変数をint型の変数にしてください。

d. 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のプロトタイプ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みます。

表3.1　引き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、関数のプロトタイプ宣言を行なわない場合、冗長な型拡張を行なう場合があります。
関数のプロトタイプ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。

e.　SBレジスタの活用

SBレジスタ※1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サイズ)を削減することができます。本コンパイラでは【図3.8】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。

※1.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。

```
#pragma  SBDATA       val

int      val;
```

図3.8　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

f.　-fJSRWオプションによるROMサイズ圧縮

本コンパイラでは、ファイル外で定義された関数を呼び出す場合には、"JSR.A"命令で呼び出しを行ないます。

しかし、プログラムサイズがあまり大きくない場合、大半の関数が"JSR.W"命令で呼び出せる場合があります。

このような場合にオプション"-fJSRW"を指定してコンパイルをおこない、リンク時にエラーの発生した関数のみを"#pragma JSRA 関数名"を用いて宣言することによりROMサイズの圧縮が期待できます。

※ ”-OGJ ”オプションを使用すると、リンク時に佐井駅なjmp命令を選択します。

g.　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。

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3.9】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※1のクリア処理）が必要になります。

```
static int i;
```

図3.9　初期値を持たない変数の宣言例

応用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。この場合に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。

```
;=====================================================================
; NEAR area initialize.
;---------------------------------------------------------------------
; bss zero clear
;---------------------------------------------------------------------
;       BZERO bss_SE_top,bss_SE
;       BZERO bss_SO_top,bss_SO
;       BZERO bss_NE_top,bss_NE
;       BZERO bss_NO_top,bss_NO

                :
             (省略)
                :
;=====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss zero clear
;---------------------------------------------------------------------
;       BZERO   bss_FE_top,bss_FE
;       BZERO   bss_FO_top,bss_FO
```

図3.10　bss領域のクリア処理のコメントアウト例

※1.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。

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。呼び出すC言語プログラム中には、必ずアセンブラ関数のプロトタイプ宣言を記述してください。

【図3.11】にアセンブラ関数asm_funcを呼び出すときの記述例を示します。

```
extern void         asm_func( void );  ←アセンブラ関数の
                                           プロトタイプ宣言
void main()
{
                :
             (省略)
                :
        asm_func();          ←アセンブラ関数の呼び出し
}
```

図3.11　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3.12　smp1.cのコンパイル結果（抜粋）(smp1.a30)

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宣言を記述する前にアセンブラ関数のプロトタイプ宣言を行います。このときには、必ず引数の型宣言を行なってください。

② #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。

【図 3.13】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。

```
extern unsigned int     asm_func(unsigned int, unsigned int);
#pragma PARAMETER       asm_func(R0, R1)        ←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()
{
        int         i = 0x02;
        int         j = 0x05;

        asm_func(i, j);                 ←アセンブラ関数の呼び出し
}
```

図3.13　引数がある場合のアセンブラ関数の呼び出し例(smp2.c)

```
    .glb  _main
_main:
    enter #04H
    pushm R1
    ._line6
;## # C_SRC :           int         i = 0x02;
    mov.w #0002H,-4[FB]     ;  i
    ._line7
;## # C_SRC :           int         j = 0x05;
    mov.w #0005H,-2[FB]     ;  j
    ._line9
;## # C_SRC :           asm_func(i, j);

    mov.w -2[FB],R1   ;  j  ←引数をR0、R1レジスタを介して
    mov.w -4[FB],R0   ;  i    アセンブラ関数に渡しています

    jsr   _asm_func ←アセンブラ関数の呼び出し('_'を付加しています。)
    ._line10
;## # C_SRC :   }
    popm  R1
    exitd
```

図3.14　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

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の記述手順を以下に示します。

① アセンブラの疑似命令 .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疑似命令 .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。※1
⑤ 関数内で破壊されるレジスタを退避してください。※2

アセンブラ関数の出口処理の記述手順を以下に示します。

⑥ 関数の入口処理で退避したレジスタを復帰してください。
⑦ 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。※1
⑧ RTS 命令を記述します。

また、アセンブラ関数内で SB、FB レジスタ内容を書き換える操作は行わないでください。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。

【図 3.15】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         ←①
        .GLB            _asm_func       ←②
_asm_func:                              ←③
        PUSHC           FLG             ←④
        PUSHM           R3,R1           ←⑤
        MOV.L           SYM1, R3R1

        POPM            R3,R1           ←⑥
        POPC            FLG             ←⑦
        RTS                             ←⑧
        .END
```

※①～⑧は、上記の手順に対応しています。

図3.15　アセンブラ関数の記述例

※1.通常、アセンブラ関数内では、B及びUフラグの変更は行なわないでください。
※2.R0レジスタおよび、戻り値に使用するレジスタは、関数の呼び出し側で退避します。このため、R0レジスタおよび、戻り値に使用するレジスタを退避する必要はありません。

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 3.16】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	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.I   #01A000H, R2R0
        RTS
        .END
```

図3.16　long型の戻り値を返すアセンブラ関数の記述例

c.　C言語の変数の参照方法

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

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疑似命令 .GLB で外部参照宣言する必要があります。

【図 3.17】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

main()
{
                :
            (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
_asm_func:                                外部参照宣言
                :
            (省略)
                :
        MOV.W           _counter, R0    ←参照
```

図3.17　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 3.18】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3.18　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 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 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 言語の関数は、関数の入口処理で、R0 レジスタおよび、戻り値に使用するレジスタの退避を行いません。このため、アセンブラから C 言語の関数を呼び出す場合、その前に R0 レジスタおよび、戻り値に使用しているレジスタの退避を行ってください。

(3)アセンブラ関数中で使用しているレジスタは C 言語関数を呼び出す前に退避し、C 言語関数から戻った後に復帰してください。

3.3.3　アセンブラ関数の記述に関する注意事項

C言語プログラム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言語プログラムへ正常に復帰できなくなります。したがって、アセンブラ関数中でFBの値を変更しないでください。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を除く、R1、R2、R3)及びアドレスレジスタ(A0、A1)の内容を変更する場合、アセンブラ関数の入口処理でそれらを退避し、出口処理で復帰する必要があります。

ただし、#pragma PARAMETER /C で宣言されたアセンブラ関数は、呼び出した側で待避・復帰を行うコードが生成されますので、アセンブラ関数内で、待避・復帰を行う必要はありません。（多少コードサイズは、大きくなります）

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3.19】に示します(図中のasm_funcはアセンブラ関数名です)。

```
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
#pragma PARAMETER asm_func(R0, R1)
```

図3.19　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、#pragma PARAMETER宣言の前には必ずアセンブラ関数のプロトタイプ宣言を行なってください。

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

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「NCxxx」とは、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、移植する時に、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
NC79	near	near	far
NC77	near	near	far

3.4.2　NC308とNC30間の移植に関する注意事項

a.　コーリングコンベンションの違い

NC30では、関数呼び出し時のレジスタの退避を関数の呼び出し側で行いますが、NC308では、関数の呼び出され側（関数の実体側）で行います。

このためNC308で、Ｃ言語で記述した関数から、アセンブラで記述した関数を呼ぶ場合には、以下の手順で呼び出すようにしてください。

条件　-　アセンブラで記述した関数によって、破壊されるレジスタが存在する場合。

1. 破壊されるレジスタを、関数の入口で退避、
2. 関数の出口で、そのレジスタを復帰

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

```
% nc308△[起動オプション]△<[アセンブリ言語ソースファイル名]△
        [リロケータブルオブジェクトファイル名]△[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

```
% nc308 -osample -as308 "-l" -ln308 "-ms" ncrt0.a30 sample.c<RET>

<RET>：リターンキーの入力を示します。
※リンク時には必ずスタートアッププログラムを先に指定してください。
```

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

【表A.1】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。

表A.1　コンパイルドライバの制御オプション

オプション	機　能
-c	リロケータブルファイル(拡張子.r30)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。
-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。ディレクトリは最大8個まで指定可能です。
-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。※1
-P	プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成します。※1
-S	アセンブリ言語ソースファイル(拡張子.a30)を作成し、処理を終了します。※1
-U*ﾌﾟﾘﾃﾞﾌｧｲﾝﾄﾞﾏｸﾛ名*	指定したプリデファインドマクロを未定義にします。
-silent	起動時のコピーライトメッセージを出力しません。
-dsource (短縮形 -dS)	C言語ソースリストをコメントとして出力した、アセンブリ言語ソースファイル（拡張子 ”.a30 ”）を生成します。（アセンブル後も削除しません。）
-dsource_in_list (短縮形 -dSL)	“ -dsource ”の機能に加えて、アセンブリ言語リストファイル（拡張子 ”.lst ”）を生成します。

※1.起動オプション-c、-E、-P、及び-Sを指定しない場合、nc308はln308まで制御を行い、アブソリュートモジュールファイル(拡張子.x30)まで作成します。

-C

コンパイラドライバの制御

機　能：　リロケータブルオブジェクトファイル(拡張子.r30)を作成し、処理を終了します。

実行例：

```
%nc308 -c sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

% ls sample.*
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
-rw-r-----  1 toolusr       450 Aug 17 11:28 sample.r30
%
```

注意事項：　このオプションを指定したときは、アブソリュートモジュールファイル(拡張子.x30)等、ln308で処理した結果出力されるファイルは生成されません。

-D*識別子名*

コンパイラドライバの制御

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。

書　式：　nc308△-D*識別子名=定数*△<C言語ソースファイル名>
※[*=定数*]は省略できます。

実行例：

```
% nc30 -c -DMYDEBUG=1 -DMSDOS=1 -DUNIX sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
```

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。

-Iディレクトリ名

コンパイラドライバの制御

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。
最大16個のディレクトリを指定できます。

書　式：　nc308△-I*ディレクトリ名*△<C言語ソースファイル名>

実行例：

```
% nc308 -c -I./test/include -I./test/inc sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

%


※この例では、2つのディレクトリ./test/includeと./test/incを指定しています。
```

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のOSのコマンドラインの最大文字数により制限されることがあります。

-E

コンパイラドライバの制御

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。

実行例：

```
% nc308 -E sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

#line 1 "sample.c"
                    :
                (省略)
                    :
#line 1 "/usr3/tool/toolusr/work30/inc308/stdio.h"
                    :
                (省略)
                    :
```

注意事項：　このオプションを指定したときは、アセンブリ言語ソースファイル(拡張子.a30)、リロケータブルオブジェクトファイル(拡張子.r30)、アブソリュートモジュールファイル(拡張子.x30)等、ccom308、as308、及びln308で処理した結果出力されるファイルは生成されません。

-P

コンパイラドライバの制御

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。

実行例：

```
% nc308 -P sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
%ls sample.*
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
-rw-r-----  1 toolusr      2322 Aug 17 11:30 sample.i
%
```

注意事項：
1. このオプションを指定したときは、アセンブリ言語ソースファイル(拡張子.a30)、リロケータブルオブジェクトファイル(拡張子.r30)、アブリュートモジュールファイル(拡張子.x30)等、ccom308、as308、及びln308で処理した結果出力されるファイルは生成されません。
2. この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には、-Eオプションを指定し、リダイレクトしてください。

-S

コンパイラドライバの制御

機　能：　アセンブリ言語ソースファイル(拡張子.a30)を作成し、処理を終了します。

実行例：

```
% nc308 -S sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
% ls sample.*
-rw-r-----  1 toolusr      2059 Aug 17 11:30 sample.a30
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
%
```

注意事項：　このオプションを指定したときは、リロケータブルオブジェクトファイル(拡張子.r30)、アブソリュートモジュールファイル(拡張子.x30)等、as308及びln308で処理した結果出力されるファイルは生成されません。

-U*プリデファインドマクロ名*

コンパイラドライバの制御

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：　nc308△-U*プリデファインドマクロ名*△<C言語ソースファイル名>

実行例：

```
% nc308 -c -UNC308 -UM16C sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c

%



※この例では、2つのマクロ定義NC308とM16Cを未定義にしています。
```

注意事項：未定義にできるマクロの数は、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より制限されることがあります。
__STDC__、__LINE__、__FILE__、__DATE__、__TIME__は未定義にすることはできません。

-silent

コンパイラドライバの制御

機　能：　起動時のコピーライトメッセージを出力しません。

実行例：

```
% nc308 -c -silent sample.c
sample.c

%
```

-dsource　　-dS

コメントオプション

機　能：　C言語ソースリストをコメントとして出力した、アセンブリ言語ソースファイル（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：　-S オプションを使用した場合、自動的に、-dsource オプションが有効になります。また、生成された “ .a30 ”、 ”.r30 ”、を削除しません。
本オプションは、アセンブルリストファイルに、C言語ソースリストを出力したいときに使用します。

-dsource_in_list　　-dSL

リストファイルオプション

機　能：　“ -dsource ”の機能に加えて、アセンブリ言語リストファイル(拡張子 ”.lst ”)を生成します。

A.2.2　出力ファイル指定オプション

【表A.2】に出力するアブソリュートモジュール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A.2　出力ファイル指定オプション

オプション	機　能
-oファイル名	ln308が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)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。
-dir ディレクトリ名	ln308が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)の出力先ディレクトリを指定できます。

-oファイル名

出力ファイル指定

機　能：　ln308が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)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。

書　式：　nc308△-oファイル名△<C言語ソースファイル名>

実行例：

```
% nc308 -o./test/sample ncrt0.a30 sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
sample.c
% cd test
% ls
total 65
drwxr-x---  2 toolusr       512 Aug 17 16:13 ./
drwxrwxrwx 11 toolusr      3584 Aug 17 16:14 ../
-rw-r-----  1 toolusr     44040 Aug 17 16:14 sample.x30

%
```

※この例では、ディレクトリ./testにアブソリュートモジュールファイル sample.x30のファイルを出力する設定を行っています。

-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*

実行例：

```
% nc30 -dir./test/sample ncrt0.a30 sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
sample.c
% cd test/sample
% ls
total 65
drwxr-x---  2 toolusr       512 Aug 17 16:13 ./
drwxrwxrwx 11 toolusr      3584 Aug 17 16:14 ../
-rw-r-----  1 toolusr     44040 Aug 17 16:14 ncrt0.x30

%
```

※この例では、ディレクトリ./test/sampleにアブソリュートモジュールファイルncrt0.x30のファイルを出力する設定を行っています。

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。

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

【表A.3】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A.3　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）。

-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。

実行例：

```
% nc308 -c -v sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp308 sample.c -o sample.i -DM16C -DNC308
ccom308 sample.i -o ./sample.a30
as308 -. -N sample.a30

%
```

注意事項：　このオプションは、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。

実行例：

```
D:\MTOOL\nc308wa>nc308 -V
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

   C Compiler Driver            Version X.XX.XX
   C Preprocessor               Version X.XX.XX
   C Compiler                   Version X.XX.XX (NC_CORE Version X.XX.XX)
   Assembler Optimizer (aopt308) for M32C/80,M16C/80 Series Version X.XX.XX
M32C/80,M16C/80 Series Assembler system Version X.XX ReleaseX
   Assembler Driver (as308) for M32C/80,M16C/80 Series Version X.XX.XX
   Macro Processor (mac308) for M32C/80,M16C/80 Series Version X.XX.Xx  (core
X.XX.XX)
   Structured Processor (pre30) for M16C Family Version X.XX.XX
   Assembler Processor (asp308) for M32C/80,M16C/80 Series Version X.XX.XX
   Linkage Editor (ln308) for M32C/80,M16C/80 Series Version X.XX.XX
   Librarian (lb308) for M32C/80,M16C/80 Series Version X.XX.XX
   Load Module Converter (lmc308) for M32C/80,M16C/80 Series Version X.XX.XX
(core X.XX.XX)
   Cross Referencer (xrf308) for M32C/80,M16C/80 Series Version X.XX.XX
   Absolute Lister (abs308) for M32C/80,M16C/80 Series Version X.XX.XX

D:\MTOOL\nc308wa>
```

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。
インストール方法の詳細は「M3T-NC308WA ガイドブック」を参照してください。

注意事項：　1.このオプションは、大文字のVを記述します。
2.このオプションを指定したときは、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

【表A.4】に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A.4　デバッグ用オプション

オプション	機　能
-g	デバッグ情報をアセンブリ言語ソースファイル(拡張子.a30)に出力します。これにより、C言語レベルデバッグが可能になります。
-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には必ずこのオプションを指定してください。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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。 エントリー版では、本オプションは指定できません。

-g

デバッグ情報の出力

機　能：　デバッグ情報をアセンブリ言語ソースファイル(拡張子.a30)に出力します。

実行例：

```
% nc308 -g -v sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp308 sample.c -o sample.i -DM16C80 -DNC308
ccom308 sample.i -o ./sample.a30 -g

as308 -. -N --N sample.a30


ln308 sample.r30 -. -G -MS -o sample
                :
             (省略)
                :
% ls sample.*
-rw-r--r--  1 toolusr      2894 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr      7048 Aug 17 15:53 sample.map
-rw-r-----  1 toolusr     53570 Aug 17 15:53 sample.x30
%
```

注意事項：C言語レベルデバッグを行なう場合には、必ず指定してください。本オプションを指定しても、コンパイラの生成コードには影響を与えません。

-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。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

注意事項：　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このオプションを指定してください。指定しない場合は、正しい結果が得られません。
このオプションを指定した場合、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。
エントリー版では、本オプションは指定できません。

補足説明：　レジスタ変数に対するデバッグ情報が必要でない場合には本オプションを指定して、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

【表A.5】にプログラムの実行速度及び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。

エントリー版では、すべての最適化オプションが使用できません。

表A.5　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容量ともに最小にする最大限の最適化を行います。
-OR	なし	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、変数の参照を定数で置き換える最適化を行います。
-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。
-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	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。これにより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。
-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_logical_or_combine	-ONLOC	論理ORをまとめる最適化を抑止します。
-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。
-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。
-Ostatic_to_inline	-OSTI	static宣言された関数を、inline宣言扱いにします。
-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。
-Oglb_jmp	-OGJ	分岐命令に関する外部参照の最適化を行います。
-Ofloat_to_inline	-OFTI	浮動小数点のランタイムライブラリをインライン展開します。

[主な最適化オプションの効果を以下に示します]

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サイズ	良	良	悪	良	同
消費スタック	良	同	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。

-O[1-5]

最適化

機　能：　速度及びROM容量ともに効果のある最大限の最適化を行います。このオプションは、-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。
エントリー版では、本オプションは指定できません。

-O1:　-O3,-Ono_bit,-Ono_break_source_debug,-Ono_float_const_fold,
-Ono_stdlibを有効にしたものと同等です。

-O2:　-O1と同じです。

-O3:　速度及びROM容量ともに効果のある最大限の最適化を行います。

-O4:　-O3に加え、-Oconstを有効にします。

-O5:　共通部分式の最適化(-OR同時指定時)、文字列転送,比較(-OS同時指定時)などを強化した最大限の最適化を行ないます。
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
・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;

test1()
{
        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)
```

次ページへつづく

-O[1-5]

最適化

注意事項：M16C/80の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、アセンブラコードに対して、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。

[例：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   006Ch   /* M16C/80 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 */
  }
```

もし、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。いずれの場合にも、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。

-OR

最適化

機　能：　速度は低下する場合がありますが、ROM容量を最小にする最大限の最適化を行います。このオプションは、-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。
エントリー版では、本オプションは指定できません。

注意事項：　この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、速度重視の最大限の最適化を行います。このオプションは、-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。
エントリー版では、本オプションは指定できません。

-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。
エントリー版では、本オプションは指定できません。

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

記述例：　最適化の行われる記述を以下に示します。

```
int const i = 10;
const double ad[3] = {0.0,0.1,0.2};

func()
{
    int k = i;  /* iを10に置き換える。*/
    double d = ad[1]; /*  ad[1] を 0.1 に置き換える*/
          :
          :
}
```

-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。
エントリー版では、本オプションは指定できません。

補足説明：　-O[3～5](もしくは-OR、-OS)オプションを指定した場合は、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。

注意事項：　●この最適化は、volatile修飾子の有無に関係なく、行われます。
●-O[3～5](もしくは-OR、-OS)オプションを指定したときのみ効果があります。

-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。
エントリー版では、本オプションは指定できません。

補足説明：　-OR、-O[3～5] 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。

注意事項：　-OR、-O[3～5] オプションを指定したときのみ効果があります。

-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。
エントリー版では、本オプションは指定できません。

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。

-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。
エントリー版では、本オプションは指定できません。

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。

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、本オプションを指定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。
エントリー版では、本オプションは指定できません。

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、１回の補正となる。

```
int  func1(int, int);
int  func2(int);

void  main( void ) {
        int  i = 1;
        int  j  = 2;
        int  k;

        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。

-Oloop_unroll[=ループ回数]　-OLU

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。
エントリー版では、本オプションは指定できません。

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サイズは増加します。

-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。
エントリー版では、本オプションは指定できません。

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などの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るので、本オプションを指定して、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_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。
エントリー版では、本オプションは指定できません。

-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。
エントリー版では、本オプションは指定できません。

-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展開したアセンブルコードを生成します。
エントリー版では、本オプションは指定できません。

補足説明：　以下の条件を満たした場合に、static関数を、inline関数として扱い、inline展開したアセンブルコードを生成します。

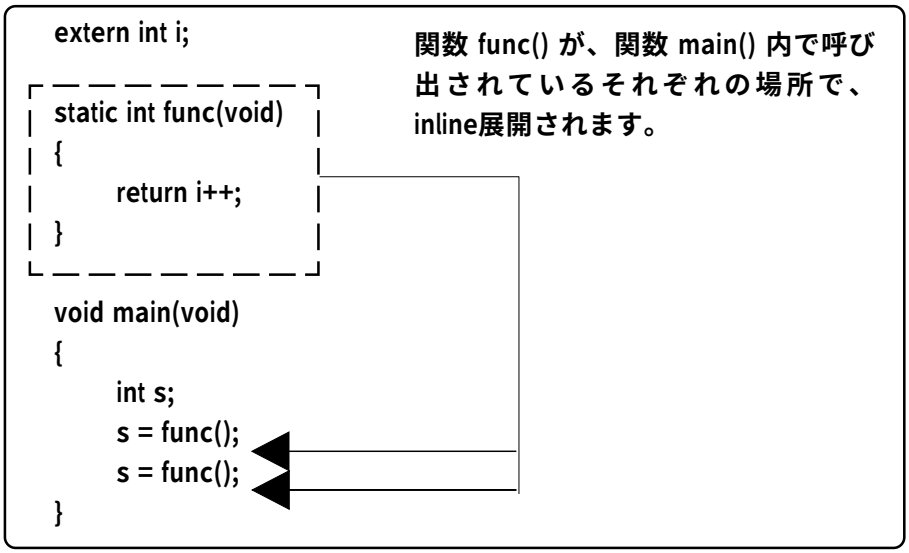
(1) 関数呼び出しの前に、実体が記述されている static関数を対象とします。
（関数の呼び出しと、その関数の実体が、同じソースファイル内に記述されていなければなりません。）
（-Oforward_function_to_inlineオプションを指定した場合は、本条件を無視してください。）

(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ル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オプションを指定した場合は、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：

- 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ラコードは、常に生成されます。
”-ferase_static_fucntion ”オプションを指定した場合、コード生成を行いません。
- 関数を、強制的に inline関数扱いにする場合には、関数に inline宣言をしてください。

-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。
エントリー版では、本オプションは指定できません。

補足説明：　インライン関数の宣言とその実体定義は、インライン関数の宣言をした後に、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、インライン関数の宣言を行う前に、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：
- ●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。
- ●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。
- ●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。
- ●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。

-Oglb_jmp　-OGJ

最適化

機　能：　分岐命令に関する外部参照の最適化を行います。
エントリー版では、本オプションは指定できません。

-Ofloat_to_inline　-OFTI

最適化

機　能：　浮動小数点のランタイムライブラリをインライン展開し、浮動小数点演算処理を高速化します。（比較、乗算のみ）
エントリー版では、本オプションは指定できません。

注意事項：　本機能を使用する場合は、必ず、コンパイルオプション“ -M82 ”を指定してください。

A.2.6　生成コード変更オプション

【表A.6】にnc30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A.6　生成コード変更オプション(1)

オプション	短縮形	機　能
-fansi	なし	-fnot_reserve_far_and_near, -fnot_reserve_asm, -fnot_reserve_inline、及び-fextend_to_intを有効にします。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fnot_reserve_asm	-fNRA	asmを予約語にしません("_asm "のみ有効になります)。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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)。 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-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。) エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fextend_to_int	-fETI	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)。※1 エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。
-fchar_enumerator	-fCE	enumerator(列挙子)の型をint型ではなくunsigned char型で扱います。
-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。
-ffar_RAM	-fFRAM	RAMデータのデフォルト属性をfarにします。
-fnear_ROM	-fNROM	ROMデータのデフォルト属性をnearにします。 エントリー版では、本オプションは指定できません。
-fnear_pointer	-fNP	ポインタおよびアドレスのデフォルトを nearにします。 エントリー版では、本オプションは指定できません。
-fconst_not_ROM	-fCNR	constで指定した型をROMデータとして扱いません。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#pragma EQU)で指定した変数をvolatileで指定した変数とみなしません。
-fsmall_array	-fSA	コンパイル時に総サイズが不明の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 エントリー版では、本オプションは指定できません。

※１.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。

表A.7　生成コード変更オプション(2)

オプション	短縮形	機　能
-fenable_register	-fER	レジスタ記憶クラスを有効にします
-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。 エントリー版では、本オプションは指定できません。
-fJSRW	なし	関数呼び出しの命令のデフォルトをJSR.W命令に変更します。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。 エントリー版では、本オプションは指定できません。
-finfo	なし	インスペクタ、”Stk Viewer”、”Map Viewer ”、”utl30 ”、に必要な情報を出力します。 エントリー版では、本オプションは指定できません。
-M82	なし	M32C/80シリーズに対応したコードを生成します。
-fswitch_other_section	-fSOS	switch文に対するテーブルジャンプ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。
-ferase_static_fucntion=関数名	-fESF=関数名	本オプションで指定された関数がstatic 関数の場合、コード生成を行いません。 エントリー版では、本オプションは指定できません。
-fdouble_32	-fD32	本オプション指定時に、double 型をfloat 型として処理します。 エントリー版では、本オプションは指定できません。
-fno_switch_table	-fNST	switch文に対して、比較を行ってから分岐するコードを、生成します。 エントリー版では、本オプションは指定できません。
-fmake_vector_table	-fMVT	可変ベクタテーブルを自動生成します。
-fmake_special_table	-fMST	スペシャルページテーブルを自動生成します。

-fansi

生成コードの変更

機　能：　以下に示す起動オプションをすべて有効にします。
-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。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

補足説明：このオプションを指定することにより、ANSI規格に基づいたコード生成を行います。

-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。ただし、同じ機能の_asmは予約語として扱われます。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

-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。ただし、同じ機能の_far、_nearは予約語として扱われます。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

-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。ただし、同じ機能の_inlineは予約語として扱われます。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

-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）。
エントリー版では、本オプションは常に指定された状態で使用されます。従って本オプションの有無の制御はできません。

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。
以下に例を示します。

```
main()
{
        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。
本オプションを指定することにより、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。

-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：　型デバッグ情報には型のサイズ情報は含まれていません。
このため、このオプションを指定した場合、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。

-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_INO, data_IFO, 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。例えば、

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

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。

-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。

-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。
エントリー版では、本オプションは指定できません。

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。本オプションを指定することにより、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。

-fnear_pointer　　-fNP

生成コードの変更

機　能：　ポインタ型のデフォルトをnearとして扱います。本オプションを指定した場合にポインタ型のデータは16bitサイズで扱います。
エントリー版では、本オプションは指定できません。

補足説明：　C言語のポインタ型変数のポインタサイズはデフォルトで32bit（実質は24bit）サイズです。
これを16bitサイズに変更するときに本オプションを使用します。
本オプションを指定することにより、生成コードサイズ及び使用RAMサイズを圧縮できる場合があります。反面、near / far の制御を厳密に行う必要が出てきます。near / far の制御はできるだけ、near / far 修飾子を用いずに、const修飾子を用いることを推奨します。

注意事項：　本オプションは、NC308にのみ有効です。

-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。
例えば、

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

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。

-fnot_address_volatile　　-fNAV

生成コードの変更

機　能：　#pragma ADDRESS又は#pragma EQUで指定した大域変数又は関数外に宣言したstatic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又は#pragma EQU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。

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。

-fsmall_array　-fSA

生成コードの変更

機　能：　コンパイル時に、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。
エントリー版では、本オプションは指定できません。

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32bitで計算します。これは、配列のサイズが64kbyte以上の場合に、対応するためです。
例えば、

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

の場合、配列「array」の総サイズがコンパイル時には分からないので、添字「j」を32bitで計算します。
本オプションを指定することにより、配列「array」の総サイズを64Kバイト以下と仮定して、添字「j」を16bit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。

-fenable_register　-fER

レジスタ記憶クラス

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において、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、上記状況下で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ました。
本オプションを指定することにより、register指定された、

1. 整数型変数
2. ポインタ変数

を強制的にレジスタに割り当てます。

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るので必ず生成されたアセンブリ言語を確認の上、使用してください。

-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。
エントリー版では、本オプションは指定できません。

-fJSRW

生成コードの変更

機　能：　関数呼び出し命令のデフォルトをJSR.W命令に変更します。

補足説明：デフォルトでは、ソースファイルの外で定義された関数を呼び出す時、JSR.A命令を用いて呼び出します。
本オプションにより、それをＪＳＲ.Ｗ命令に変更することができます。JSR.W命令に変更することにより、生成コードサイズを圧縮することができます。反面、JSR.W命令は、呼び出し位置から前後32Kバイトを超える位置にある関数を呼び出すと、リンク時にエラーが発生します。#pragma JSRAを組み合わせることにより、このエラーを回避することができます。
本オプションは、プログラムサイズが32Kバイト以内、もしくは比較的小さな場合に、ROM圧縮をしたいときに有効です。

-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。
エントリー版では、本オプションは指定できません。

補足説明：　除算時に、被除数が4byte値、除数が2byte値 で、かつ、演算結果が2byte値の場合や、被除数が2byte値、除数が1byte値 で、かつ、演算結果が1byte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：　本オプションを指定した場合に、除算結果が over flow すると ANSIの規定とは異なる動作になります。

M16Cのdiv命令は演算結果が over flow した場合には、結果は不定になります。このためNC308でデフォルトでコンパイルを行った場合には、結果を保証する為に、被除数が4byte、除数が2byte で、かつ、結果が2byte のような場合にもランタイムライブラリを呼び出します。

-finfo

生成コードの変更

機　能：　“ TM ”、 ”Stk Viewer ”、 ”Map Viewer ”、 ”utl308 ”、に必要な情報を出力します。
エントリー版では、本オプションは指定できません。

補足説明：“ Stk Viewer ”、 ’Map Viewer ”、 ”utl308“ を使用する時は、このオプションで出力されたアブソリュートファイル、 ”.x30 ”ファイルが必要です。

注意事項：　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、utl308 でも、asm関数の使用は無視されます。

-M82

生成コードの変更

機　能：　M32C/80シリーズに対応したコードを生成します。

-fswitch_other_section　　-fSOS

生成コードの変更

機　能：　switch文に対するテーブルコード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。
本オプションは通常、使用する必要はありません。

-ferase_static_fucntion=*関数名*　　-fESF=*関数名*

生成コードの変更

機　能：　本オプションで指定された関数がstatic 関数の場合、コード生成を行いません。
エントリー版では、本オプションは指定できません。

例)

nc308 -fESF=func1 -fESF=func2 test.c<ret>

関数func1 と関数func2 がstatic 関数の場合、
これらの関数のコード生成は行われません。

補足説明：　HEW から本オプションを指定した場合、下記に示す警告が出力されますが無視してください。
==> The Library field is empty. Input a library name in the field.

-fdouble_32　　-fD32

生成コードの変更

機　能：　本オプション指定時に、double 型をfloat 型として処理します。
エントリー版では、本オプションは指定できません。

注意事項：　1.本オプション指定時には、必ず、関数のプロトタイプ宣言を行なってください。もし、プロトタイプ宣言がない場合には、不正なコードを生成する場合があります。

2.本オプションを指定した場合の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PD308,シミュレータPD308SIM 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。

-fno_switch_table　-fNST

生成コードの変更

機　能：　switch文に対して、比較を行ってから分岐するコードを、生成します。
エントリー版では、本オプションは指定できません。

補足説明：　本オプションを指定しない場合は、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。

-fmake_vector_table　-fMVT

生成コードの変更

機　能：　可変ベクタテーブルを自動生成します。

補足説明：　”#pragma INTERRUPT（付録 B.7 を参照）”で指定された、割り込みベクタ番号、割り込み処理関数名を基に、可変ベクタテーブルを自動生成します。

[#pragma INTERRUPT の書式]

#pragma INTERRUPT△割り込みベクタ番号△割り込み処理関数名

#pragma INTERRUPT△割り込み処理関数名(vect=割り込みベクタ番号)

生成された可変ベクタテーブルは、スタートアッププログラムファイルに対応するリロケータブルオブジェクトファイルに格納されます。

注意事項：　1.本オプションで生成される割り込みベクタテーブルの内容は、リンケージエディタが生成するマップファイルに出力されます。
2.本オプションを使用した場合、スタートアッププログラムファイルに記述された可変ベクタテーブルは、全て無視されます。
3.本オプションを指定した場合は、アセンブラ AS308、リンケージエディタ ln308 にも、 ”-fMVT ”オプションを指定してください。

-fmake_special_table　　-fMST

生成コードの変更

機　能：　スペシャルページベクタテーブルを自動生成します。

補足説明：　”#pragma SPECIAL（付録 B.7 を参照）”で指定された、呼び出し番号、関数名を基に、スペシャルページベクタテーブルを自動生成します。

[#pragma SPECIAL の書式]

#pragma SPECIAL△呼び出し番号△関数名

#pragma SPECIAL△関数名(vect=呼び出し番号)

生成されたスペシャルページベクタテーブルは、スタートアッププログラムファイルに対応するリロケータブルオブジェクトファイルに格納されます。

注意事項：
1. 本オプションで生成されるスペシャルページベクタテーブルの内容は、リンケージエディタが生成するマップファイルに出力されます。
2. 本オプションを使用した場合、スタートアッププログラムファイルに記述されたスペシャルページベクタテーブルは、全て無視されます。
   スタートアップファイルに記述されたスペシャルページベクタテーブルと、本オプションで生成されるスペシャルページベクタテーブルは、同時に使用することはできません。
3. 本オプションを指定した場合は、アセンブラ AS308、リンケージエディタ ln308 にも、”-fMST ”オプションを指定してください。

A.2.7　ライブラリ指定オプション

【表A.7】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A.8　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。

-lライブラリファイル名

ライブラリファイル指定

機　能：　ln308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：　nc308△-lファイル名△<C言語ソースファイル名>

実行例：

```
% nc308 -v -lusrlib ncrt0.a30 sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
as308 -. -N ncrt0.a30

sample.c
cpp308 sample.c -o sample.i -DM16C -DNC308
ccom308 sample.i -o ./sample.a30
main
as308 -. -N sample.a30


ln308 ncrt0.r30 sample.r30 -. -l usrlib -o ncrt0

%
```

※この例では、ライブラリusrlib.libを指定しています。

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. NC308は環境変数LIB308で指定されたディレクトリ内にあるライブラリnc308lib.libをデフォルトでリンクします。(複数のライブラリを指定した場合nc308lib.libを参照する優先順位は最も低くなります。)

A.2.8　警告オプション

【表A.8】にnc308の言語仕様に関する記述の間違いに対して警告(ワ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A.9　警告オプション

オプション	短縮形	機　能
-Wnon_prototype	-WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	-WUP	サポートしていない＃pragmaを使用した場合、警告を出します。
-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。
-Wstdout	なし	エラーメッセージをホストマシンの標準出力(stdout)に出力します。
-Werror_file<file name>	-WEF	タグファイルを出力します。
-Wstop_at_warning	-WSAW	コンパイル時にワーニングが発生した場合、コンパイルを停止します。
-Wnesting_comment	-WNC	コメント中に/*を記述した場合に警告を出します。
-Wccom_max_warnings=ワーニング回数	-WCMW	ccom308の出力するワーニングの回数の上限を指定できます。
-Wall	なし	検出可能な警告(ただし、 ”-Wlarge_to_small ”、 ”-Wno_used_argument ”で出力される警告を除く)をすべて表示します。
-Wmake_tagfile	-WMT	error および warning が発生した場合にファイル毎にタグファイルを出力します。
-Wuninitialize_variable	-WUV	初期化されていない auto変数に対してワーニングを出力します。
-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、ワーニングを出力します。
-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、「プロトタイプ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。
-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、ワーニングを出力します。
-Wno_used_static_function	-WNUSF	コード生成が不要な static 関数名を、表示します。 エントリー版では、本オプションは指定できません。
-Wno_used_function	-WNUF	未使用のグローバル関数を、リンク時に表示します。
-Wundefined_macro	-WUM	未定義のマクロを#ifの中で使用した場合に警告します
-Wstop_at_link	-WSAL	リンク時に、ワーニングが発生した場合、アブソリュートモジュールファイルの生成を抑止します。

-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、または関数のプロトタイプ宣言を行っていない場合に警告を出します。

補足説明：　プロトタイプ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、プロトタイプ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。

-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、警告を出します。

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警告を出しません。
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。
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。

-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。その場合には、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。

-Wstdout

警告オプション

機　能：　エラーメッセージをホストマシンの標準出力(stdout)に出力します。

実行例：

```
A> nc308 -c -Wstdout sample.c > err.doc

A> type err.doc
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
[Error(ccom):sample.c,line 39] unknown valuable port00
   ===>          port00 = 0x00;
Sorry, compilation terminated because of these errors in main().

A>
```

補足説明：本オプションは、Windows版(パソコン版)においてエラー出力等をリダイレクトを用いて、ファイルに保存する場合に使用します。

注意事項：コンパイルドライバから呼び出されるas308、ln308のエラー出力は、本オプションに関係なくWindows版(パソコン版)であれば標準出力に出力されます。

-Werror_file <file name>　　-WEF

警告オプション

機　能：　エラーメッセージを指定したファイルに出力します。

書式：　nc308△-Werror_file△<エラー出力ファイル名>

補足説明：ファイルに出力されるエラーメッセージの出力フォーマットは、ディスプレイに表示されるエラーメッセージとは異なり、一部のエディタの持つ「タグジャンプ機能」に適したフォーマットで出力されます。

出力例)

```
test.c  12  Error(ccom):unknown variable i
```

-Wstop_at_warning　-WSAW

警告オプション

機　能：　コンパイル時にワーニングが発生した場合、コンパイルを停止し、コンパイラの終了コード ”10 ”を戻します。

補足説明：　デフォルトでは、コンパイル時にワーニングが発生した場合、コンパイルの終了コードは、 ” 0 （正常終了）”で終了します。
本オプションは、makeユーティリティ等を用いている場合に、ワーニングが発生した場合にコンパイル処理を停止したいときに使用します。

-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警告を発生します。

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。

-Wccom_max_warnings=ワーニング回数　-WCMW

警告オプション

機　能：　コンパイラ本体の出力するワ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、ワーニングの出力には上限がありません。
本オプションは、多量のワ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：ワ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回を指定するとワーニング出力を完全に抑止します。

.

-Wall

警告オプション

機　能：　検出可能な警告をすべて表示します。
ただし、-Wlarge_to_small（ -WLTS ）、及び“ -Wno_used_argument (-WNUA) ”、 ”-Wno_used_static_function(-WNUSF) ”を使用した場合の警告は除きます。
オプション ”-Wnon_prototype(-WNP) ”、 ”-Wunknown_pragma(-WUP) ”、 ”-Wnesting_comment(-WNC) ”、 ”-Wuninitialize_variable(-WUV) ”と同等の警告を表示します。またこれらに加えて、以下の場合にも警告を表示します。

(1)　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。

例）
```
if ( i = 0 )
        func();
```

(2)　代入演算子 ”= ”を間違って ”== ”と記述した場合。

例）
```
i == 0;
```

(3)　古い形式の関数定義を行った場合。

例）
```
func(i)
int i ;
{
   ...
}
```

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。

-Wmake_tagfile　-WMT

警告オプション

機　能：　error および warning が発生した場合に、 ファイル単位で、タグファイル に生成メッセージの内容を出力します。

補足説明：本オプションを指定した場合に　-Werror_file <file name> (-WEF) を同時に指定した場合には、エラーとなります。

-Wuninitialize_variable　-WUV

警告オプション

機　能：　初期化されていない auto変数に対してワ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、警告が出力されます。

例）

```
main()
{
        int i;
        int val;
        for ( i = 0; i<2; I++ ) {
                f();
                val =1 ; //論理上、ここで必ず初期化される
        }
        ff( val );
}
```

-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、ワ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、警告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、sigbed 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、警告が出力されます。
- 本オプションは、多量のワーニングを出力するため、以下の型変換のみワ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

-Wno_warning_stdlib　-WNWS

警告オプション

機　能：　”-Wnon_prototype ”指定や、”-Wall ”指定と同時に本オプションを指定すると、「プロトタイプ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。

-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に、使用していない引数に対してワーニングを出力します。

-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。

エントリー版では、本オプションは指定できません。

補足説明：　本オプション指定により、出力された static関数は、-ferase_static_function オプションにより、コード生成を抑止することができます。

注意事項：
- -Ostatic_to_inline(-OSTI)オプション指定時に出力されたstatic 関数のコード生成を抑止する場合は、必ず、-ferase_static_function(-fESF)を使用してコード生成を抑止してください。
  (本オプション指定時に出力されたstatic関数の実体をCソースファイルから削除することはしないでください。)

- 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。

  例)
  ```
  void (*a[5])(void) = {f1,f2,f3,f4,f5};
  for(i = 0; i < 3; i++) (*a[i])();
  ```
  上記の例では、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。

-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。
本オプションを指定する場合は、必ず -finfo ”オプションも同時に指定してください。
エントリー版では、本オプションは指定できません。

-Wundefined_macro　-WUM

警告オプション

機　能：　未定義のマクロを、#ifの中で使用した場合に警告します。

-Wstop_at_link　-WSAL

警告オプション

機　能：　リンク時にワーニングが発生した場合、リンクを停止し、アブソリュートモジュールファイルの生成を抑止します。
また、戻り値 ”10 ”をホストOSに返します。

A.2.9　アセンブル / リンクオプション

【表A.9】にas308及びln308のオプションを指定する起動オプションを示します。

表A.10　アセンブル / リンクオプション

オプション	機　能
-as308△<*オプション*>	アセンブルコマンド as308 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。 エントリー版では、本オプションは指定できません。
-ln308△<*オプション*>	リンクコマンドln308 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。 エントリー版では、本オプションは指定できません。

-as308"*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as308 のオプションを指定します。
2個以上のオプションを指定する場合は、" (ダブルクオーテーション)で囲んでください。
エントリー版では、本オプションは指定できません。

書　式：　nc308△-as308△*"オプション1△オプション2"*△<C言語ソースファイル名>

実行例：　以下の例では、アセンブラリストファイルをコンパイル時に生成しています。

```
% nc308 -v -as308 " -l -s " sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

sample.c
cpp308 sample.c -o sample.i -DM16C -DNC308
ccom308 sample.i -o ./sample.a30
as308 -. -N -l -s sample.a30
% ls sample.*
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr     10508 Aug 17 15:43 sample.lst
-rw-r-----  1 toolusr       587 Aug 17 15:43 sample.r30
%
```

注意事項：　as308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。

-ln308"オプション"

アセンブル / リンクオプション

機　能：　リンクコマンドln308 のオプションを指定します。
リンクコマンドのオプションは、最大4個までオプションを指定することができます。2個以上のオプションを指定する場合は、 " (ダブルクオーテーション)で囲んでください。
エントリー版では、本オプションは指定できません。

書　式：　nc308△-ln308△"*オプション1*△*オプション2*"△<C言語ソースファイル名>

実行例：　以下の例では、マップファイルをコンパイル時に生成しています。

```
% nc308 -g -v -osample -ln308 -ms ncrt0.a30 sample.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
as308 -. -N --N ncrt0.a30

sample.c
cpp308 sample.c -o sample.i -DM16C -DNC308
ccom308 sample.i -o ./sample.a30 -g
as308 -. -N --N sample.a30


ln308 ncrt0.r30 sample.r30 -. -G -MS -ms -o sample
                :
             (省略)
                :
% ls sample.*
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr     44040 Aug 17 15:47 sample.x30
-rw-r-----  1 toolusr      8310 Aug 17 15:47 sample.map
%
```

注意事項：　ln308 の-.、-G、-O、-ORDER、-L、-T、-Vおよび@fileオプションは指定しないでください。

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

nc308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

nc308の起動時オプション中、

- -c：リロケータブルファイル(拡張子.r30)を作成して処理を終える
- -S：アセンブリ言語ソースファイル(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リ言語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

NC308は、M16C/80シリーズ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。

表B.1　拡張機能(1)

拡張機能	機能の内容
near / far修飾子	1.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( );
日本語文字のサポート	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。
関数のデフォルト引数宣言	1.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。
inline記憶クラスのサポート	1.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); ※インライン関数を使用する前に必ずインライン関数の実体定義を行ってください。

表B.2　拡張機能(2)

拡張機能	機能の内容
C++風コメント	1.C++言語風でのコメント"//"を記述できます。 記述例)　　//　以降はコメントです。
#pragma 拡張機能	C言語から、M16C/80シリーズハードウェア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	アセンブラ命令の一部をC言語の関数として記述することができます。 記述例)　char　dadd_b(char　val1,　char　val2); 記述例)　int　　dadd_w(char　val1,　char　val2);

B.1　near / far修飾子

M16C/80シリーズは、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。NC308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。

●near修飾子 ..... 000000H～00FFFFHの領域

●far修飾子 ..... 000000H～0FFFFFFH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、NC308は属性を以下のように解釈します。

●変数の配置 ..... near属性
●const修飾された定数の配置 .....far属性
●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。

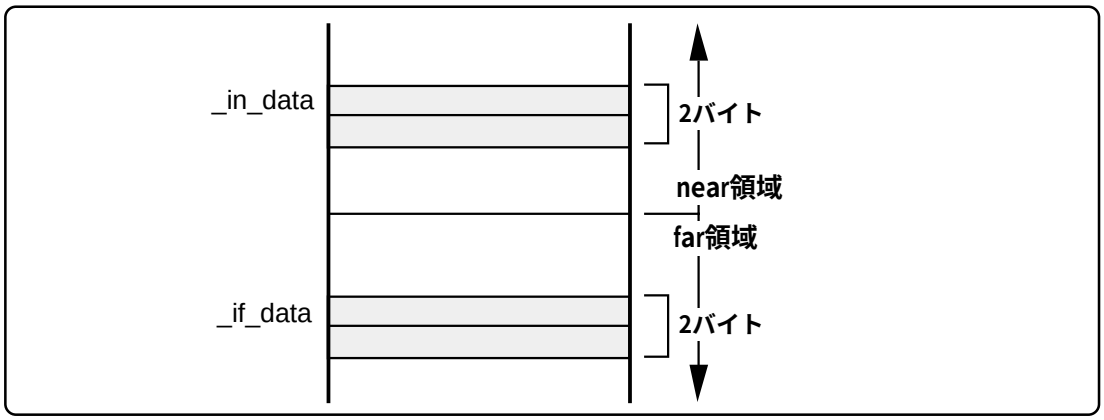
```
型指定子△near又はfar△変数;
```

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;

func()
{
      (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far型 (4byte)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)

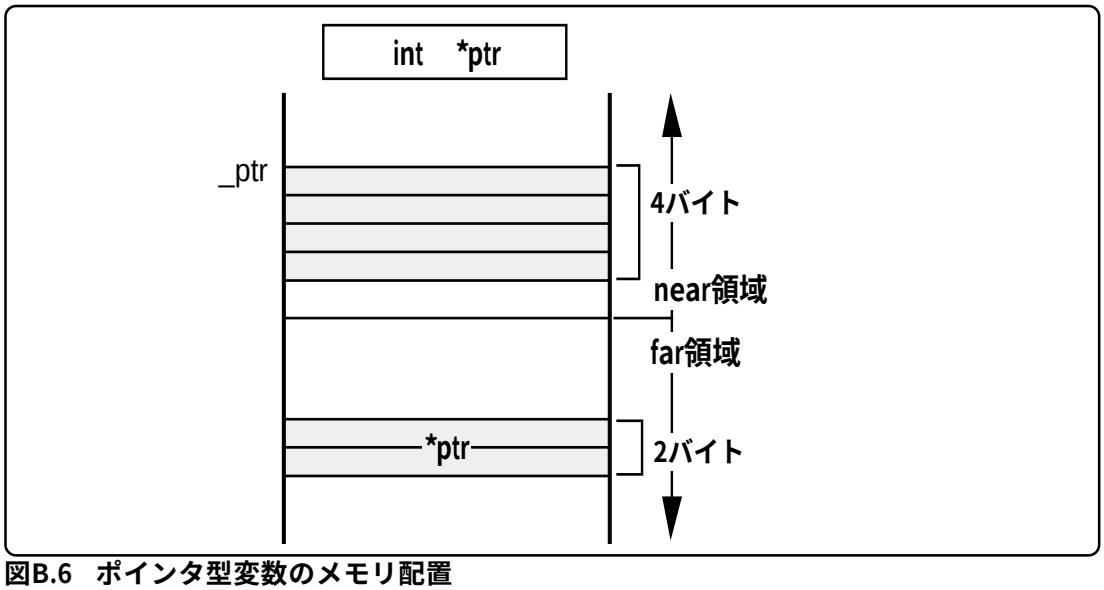
変数の配置は near 、ポインタ変数の型はfar 型となるため、【図B.4】の記述は、【図B.5】のように解釈されます。

```
●例
  int  far   *   near   ptr;
```

図B.5　ポインタ型変数の宣言例(2)

変数ptr は、far領域にあるint型変数を指し示す 4byte 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 を指定した場合には、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(1)

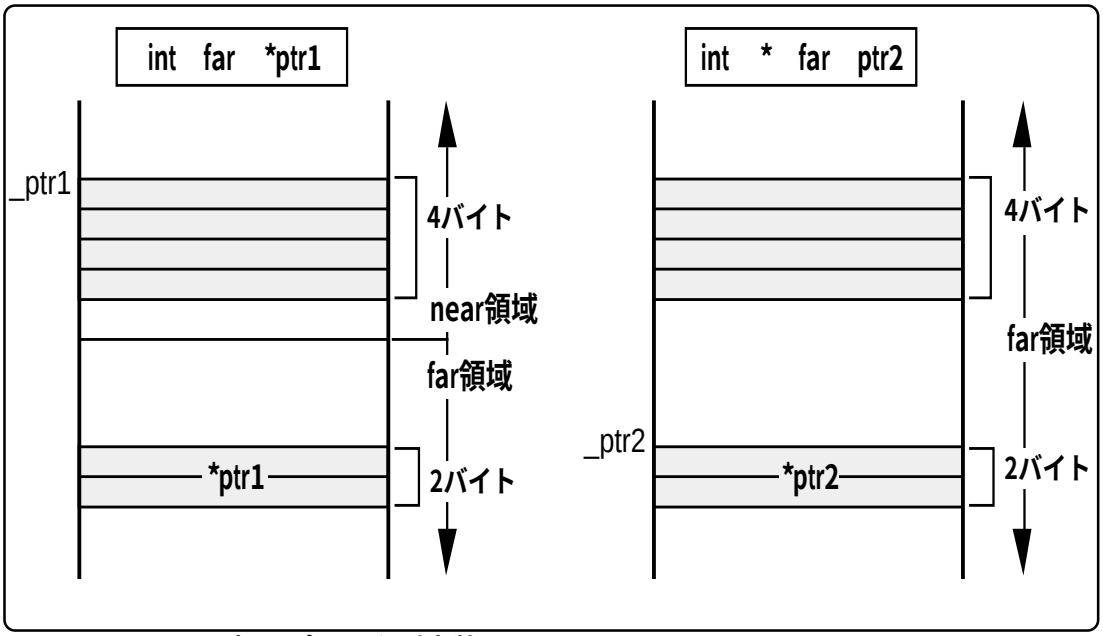
先にも説明したように near / far の指定がない場合には、変数の配置を“ near ”、変数の型を“ far ”として扱います。したがって、例１、例２はそれぞれ、【図B.8】のように解釈されます。

```
●例1
  int   far    *   near    ptr1;

●例2
  int  far    *   far       ptr2;
```

図B.8　アドレスを扱うポインタ型変数の宣言例(2)

例１では、変数ptr1 は far領域にある int型変数を指し示す 4byte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 far領域にある int型変数を指し示す 4byte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

関数のnear / far配置属性は、常にfarです。関数の宣言にnear属性を指定するとワーニングメッセージ(function must be far)を出力し、nearの宣言を無視します。

B.1.5　nc308の起動オプションによるnear / farの制御

NC308では、near / far属性を指定しない場合、関数はfar属性、変数(データ)はnear属性として扱われます。NC308の起動オプションには、変数(データ)のデフォルトを変更するオプションを用意しています(【表B.1】)。

表B.1　nc308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。

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;

main()
{
    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プロトタイプ宣言した */
                          /* 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に型変換される際、0(ゼロ)を上位アドレスとして拡張されます。

B.1.7　farからnearポインタへの代入の検査機能

コンパイル時に、【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ワ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;

main()
{
    n_ptr = f_ptr;      /* farポインタをnearポインタに代入 */
        :
      (省略)
        :
    func ( f_pyr );     /* 引数にnearポインタを持つ関数としてプロトタイプ宣言した */
                        /* 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ワーニングメッセージ(far pointer (implicity) casted by near pointer)を出力します。

B.1.8　関数の宣言

本コンパイラでは関数は必ず、far領域に配置されます。従って、関数には、nearの宣言を行なわないでください。

関数に対してnear属性の宣言を行なった時には、NC308はワーニングを出力した後、関数の属性をfarとして処理を続けます。関数に対してnear宣言を行なったときの表示例を【図B.12】に示します。

```
%nc308 -S smp.c
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

smp.c
[Warning(ccom):smp.c,line 3] function must be far
   ===> {
func
%
```

図B.12　関数の宣言例

B.1.9　複数の宣言でnear / farの確定を行う機能

【図B.13】に示すように同一の変数に対して複数の宣言を行った場合、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;

func()
{
     (以下省略)
        :
```

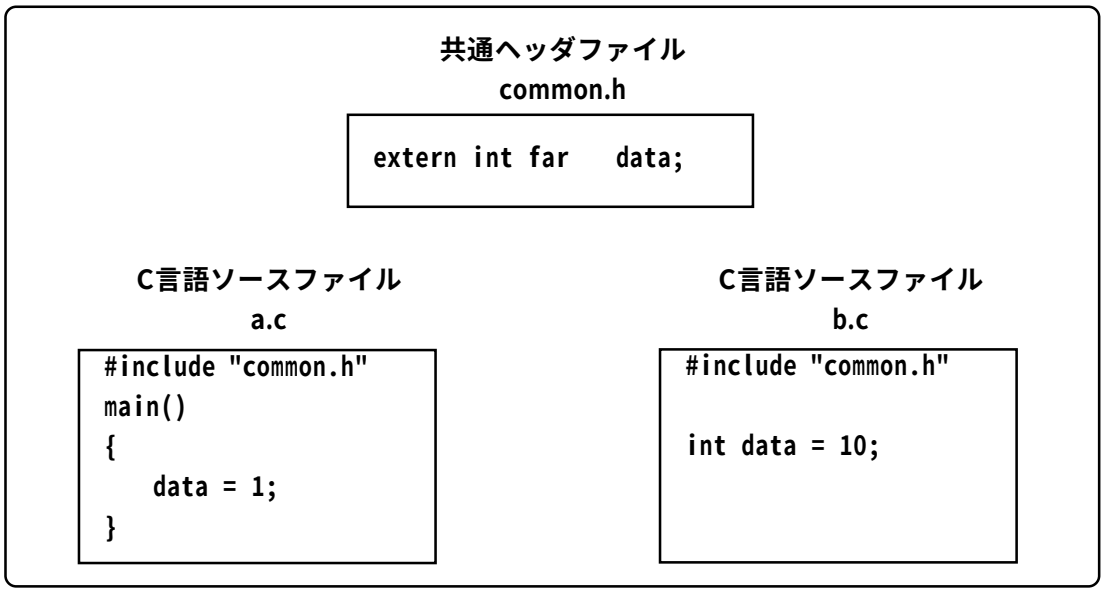
この宣言は、以下の宣言として解釈されます。

```
extern int far idata = 10;

func()
{
     (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

B.1.10　near / far属性に関する注意事項

a. 関数のnear / far属性に関する注意事項

関数は必ずfar属性になります。関数はnearで宣言しないでください。関数に対してnear属性の宣言を行なった場合NC308は警告を出力します。

b.　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、以下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

B.2　asm関数

本コンパイラでは、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AS308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
{
          :
      (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言語の変数名で指定できます。
- ●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言語の変数名で指定できます。
- ●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。

●asm関数内では、レジスタの内容を破壊しないでください。

コンパイラは、asm関数内のチェックを行っていません。
レジスタを破壊する場合には、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。

※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

C言語で記述した記憶クラス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関数中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　　R1, $$[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()
{
        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()
{
        int idata = 1;                    ←auto変数(FBオフセットは-2)
        asm("       MOV.W           $$[FB], R0", idata);
        asm("       CMP.W           #00001H   ,R0");

   (以下省略)
       :
}
```

●アセンブリ言語ソースファイル(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          ←FBオフセット-2の内容をR0レジスタに転送
        ._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関数中で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);
}
```

●アセンブリ言語ソースファイル(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'
         .align
         ._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には、ビット操作命令で参照可能な範囲(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はユーザ側で確認してください。

B.2.3　レジスタ変数のレジスタ名の指定

C言語で記述した記憶クラス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関数中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　　$$",　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;          ← i はレジスタ変数

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

}
```

●アセンブリ言語ソースファイル(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'
      .align
      ._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関数中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

C言語で記述した記憶クラス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関数中で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 R1,　$$", 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）

```
int idata;
int a[3];
struct TAG{
         int  i;
         int  k;
} s;
void func()
{
                :
        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;                    ←extern変数

func()
{
        static int s_val;              ←static変数

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

●アセンブリ言語ソースファイル(コンパイル結果)

```
    .glb _func
_func:
    ._line     7
;## # C_SRC :          asm("   mov.w   #01H,$$",ext_val);
;#### ASM START
   mov.w   #01H,_ext_val           ←シンボル_ext_valへの転送
    ._line     8
;## # C_SRC :          asm("   mov.w   #01H,$$",s_val);
   mov.w   #01H,___S0_s_val        ←シンボル___S0_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関数中で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[FB]",  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:

    .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には、直接1ビット操作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ユーザ側で確認してください。

B.2.5　記憶クラスに依存しない指定

C言語で記述した変数を、その変数の記憶クラス（auto変数、レジスタ変数、extern変数、static変数）に依存することなく、asm関数中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、asm関数中でC言語で記述した変数を使用することができます。※1

```
asm("　　オペコード　　R0, $@", 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1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;        ←extern変数

void func(void)
{
      int          f_val;      ←auto変数
      register int   r_val;    ←レジスタ変数 *2
      static int     s_val;    ←static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);
}
```

●アセンブリ言語ソースファイル(コンパイル結果)

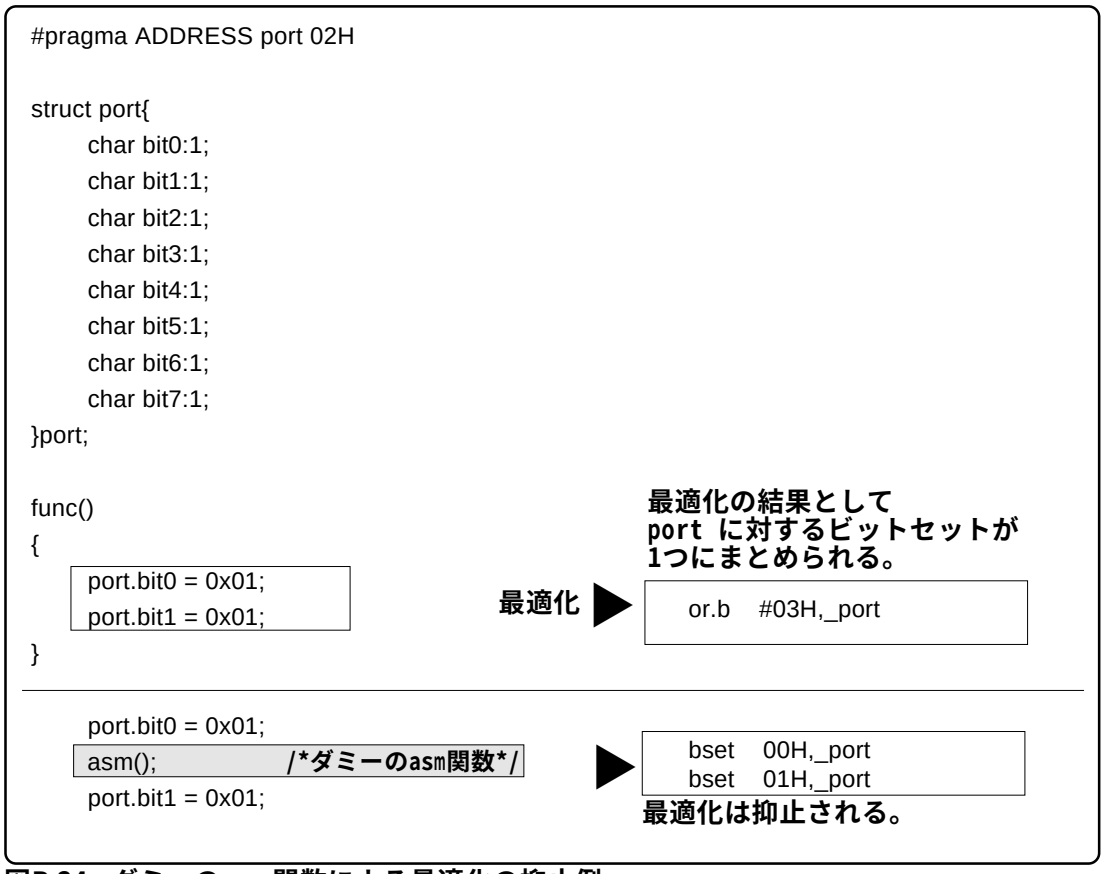
```
    .glb  _func
_func:
    enter #02H
    pushm R1
    ._line     9
;## # C_SRC :          asm("   mov.w   #1, $@", e_val);
;#### ASM START
   mov.w   #1, _e_val:16          ←-------------------------  extern変数の参照
    ._line     10
;## # C_SRC :          asm("   mov.w   #2, $@", f_val);
   mov.w   #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 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。
※2 register修飾子を指定しても、レジスタに割り当てられるとは、かぎりません。

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。

● 記憶クラスがautoの変数、引数もしくは、1ビットのビットフィールドの場合

記憶クラスがautoの変数、引数もしくは、1ビットのビットフィールド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。autoの変数、もしくは引数を指定する場合には、【図B.35】に示す書式で記述してください。

```
asm("MOV.W      #01H,$$[FB]", i);←記憶クラスautoの変数を参照する書式です。
asm("BSET       $$[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)

● 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関数で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関数で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。

● asm関数で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、asm関数の中で直接、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。

正しく参照できない例：

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

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。そのため、引数を正しく参照できなくなります。

● 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。

b.　レジスタについて

● asm関数内でレジスタを破壊しないでください。破壊する場合は、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。

● 本コンパイラでは、SBレジスタをスタートアッププログラムで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.37】に示すSBレジスタを元に戻す記述を行なってください。

```
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.37　変更したスタティックベースレジスタの復帰方法

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。

●　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ルソースファイルでは、【図B.38】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
例）A1:
     C9830:
●_(アンダースコア)で始まる２文字以上のラベル名
例）__LABEL:
    ___START:
```

図B.38　asm関数で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。NC308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

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、以下の設定となっています。

UNIX版	NCKIN,NCKOUT 共に EUC
MS-Windows版	NCKIN,NCKOUT 共に SJIS

- ●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.39】に示します。

```
[UNIXの場合]
入力をEUCコード、出力をシフトJISコードに設定するときの例です。
% setenv NCKIN EUC
% setenv NCKOUT SJIS

[MS-Windowsの場合]
autoexec.batの中に以下の内容を記述します。
set NCKIN=SJIS
set NCKOUT=SJIS
```

図B.39　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.40】に示します。

```
L"漢字文字列"
```

図B.40　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。

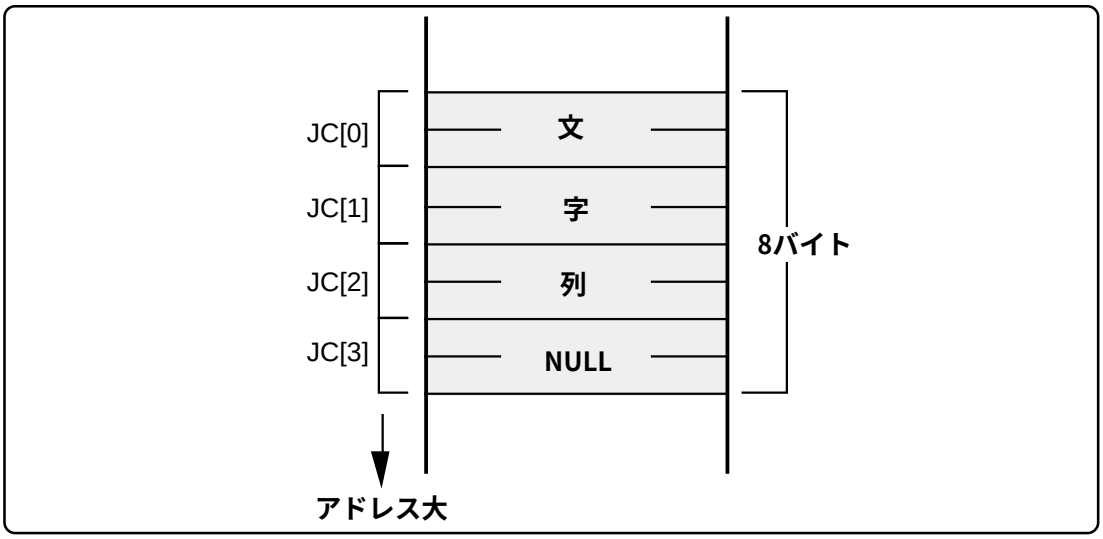
2バイト文字として扱う場合は、文字列の先頭にLを付加してwchar_t型へのポインタ型として使用します。wchar_t型は、標準ヘッダファイルstdlib.hの中でunsigned short型にtypedefしています。

【図B.41】に日本語の文字列の記述例を示します。

```
#include <stdlib.h>

void  func()
{
        wchar_t JC[4] = L"文字列";    ←①

          (以下省略)
              :
```

図B.41　日本語文字列の記述例

【図B.41】中の①の初期化された文字列のメモリ配置を【図B.42】に示します。



図B.42　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.43】に示します。

```
        L'漢'
```

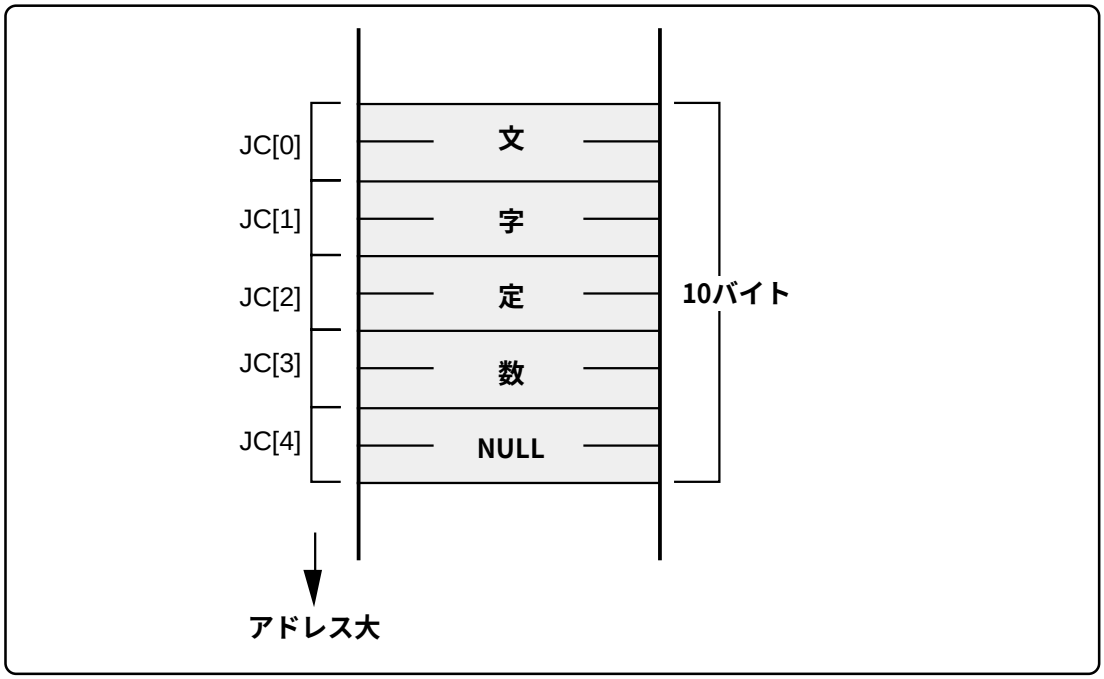
図B.43　文字列中の漢字コード記述書式

文字列と同様に、文字定数の前にLを付加した場合、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.44】に日本語の文字定数の記述例を示します。

```
#include <stdlib.h>

void func()
{
        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.44　日本語文字定数の記述例

【図B.44】中の文字定数を代入した配列のメモリ配置を【図B.45】に示します。



図B.45　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

NC308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。

B.4.1　関数のデフォルト引数宣言の概要

NC308では、関数のプロトタイプ宣言時に、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.46】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.46　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.47】に、【図B.47】で示すサンプルプログラムのコンパイル結果を【図B.48】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.47 関数のデフォルト引数の宣言例(smp1.c)

```
_main:
    ._line5
;## # C_SRC :           func();
    push.w#0002H                ←第2引数：2
    mov.w #0001H,R0             ←第1引数：1
    jsr   $func
    add.l #02H,SP
    ._line6
;## # C_SRC :           func(3);
    push.w#0002H                ←第2引数：2
    mov.w #0003H,R0             ←第1引数：3
    jsr   $func
    add.l #02H,SP
    ._line7
;## # C_SRC :           func(3,5);
    push.w#0005H                ←第2引数：5
    mov.w #0003H,R0             ←第1引数：3
    jsr   $func
    add.l #02H,SP
    ._line8
;## # C_SRC :   }
    rts
        :
      (省略)
        :
```

注：NC308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。また、この例では引数をレジスタ渡しで処理しています。

図B.48 smp1.cのコンパイル結果(smp1.a30)

関数の引数には、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.49】に、【図B.49】で示しましたサンプルプログラムのコンパイル結果を【図B.50】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.49　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.50 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き数の宣言を行なう時には、以下の点に注意してください。

●複数の引数にデフォルト値を指定する時

関数の引き数が複数ある場合にデフォルト値を指定する時には、必ず引き数の後ろから埋めてください。【図B.51】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.51 プロトタイプ宣言の記述例

●変数を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には、指定する変数の宣言を行なった後に、関数のプロトタイプ宣言を行なってください。もし、関数のプロトタイプ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には、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。

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.52】に宣言時の書式を示します。

```
inline△型指定子△関数;
```

図B.52　inline記憶クラスの宣言書式

関数の宣言例を【図B.53】に、コンパイル結果【図B.54】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()
{
      int s;
      s = func(s);                        ←インライン関数呼び出し部
}
```

図B.53　インライン関数のサンプルプログラム(smp.c)

```
    ._LANG     'C','X.XX.XX','REV.X'

;## NC308 C Compiler   OUTPUT
;## ccom308 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,
2000,2001,2002,2003

;## COMMAND_LINE: ccom308  D:\MTOOL\nc308wa5\TMP\sss.i -o .\smp.a30 -dS

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

    .GLB  __SB__
    .SB   __SB__
    .FB   0

;## #    FUNCTION func

;## #    FUNCTION main
;## #    FRAME AUTO  (       s) size  2,   offset -4
;## #    FRAME AUTO  (       i) size  2,   offset -2
;## #    ARG Size(0) Auto Size(4)     Context Size(8)


    .SECTION   program,CODE,ALIGN
    ._file     'smp.c'
    .align
    ._line     7
;## # C_SRC :  {
    .glb  _main
_main:
    enter #04H
    pushm R1
    ._line     9
;## # C_SRC :        s = func(s);
    mov.w -4[FB],R0   ;  s
    ._line     2
;## # C_SRC :  {
    mov.w R0,-2[FB]   ;  i
    ._line     3
;## # C_SRC :        return i++;
    mov.w R0,R1
    add.w #0001H,R0
    ._line     9
;## # C_SRC :        s = func(s);
    mov.w R1,-4[FB]   ;  s
    ._line     10
;## # C_SRC :  }
    popm  R1
    exitd
E1:
    .END

;## Compile End Time Tue Jul 16 13:12:00 20xx
```

←インライン関数が埋め込まれている

図B.54 サンプルプログラムのコンパイル結果(smp.a30)

B.5.3　inline記憶クラスの規定事項

inline記憶クラス指定時には、以下の点に注意してください。

●インライン関数の引数について

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。

●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、コンパイルエラーとなります。

●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、コンパイルエラーとなります。

●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時には、宣言の記述の後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。【図B.55】の記述は本コンパイラでは、エラーとして処理します。

```
inline void func(int i);

void main( void )
{
      func(1);
}
```

【エラーメッセージ】

```
[Error(ccom):smp.c,line 5] inline function's body is not declared previously
  ===>         func(1);
Sorry, compilation terminated because of these errors in main().
```

図B.55 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には、inlineの指定は無効になりすべてstaticな関数として扱います。この時、本コンパイラでは警告を発します。(【図B.56】)

```
int func(int i);

void main( void )
{
      func(1);
}

inline int func(int i)
{
      return i;
}
```

【ワーニングメッセージ】

```
[Warning(ccom):smp.c,line 9] inline function is called as normal function before
,change to static function.
```

図B.56 インライン関数の不適切な記述例(2)

●インライン関数のアドレスについて

インライン関数は、関数自身はアドレスを持ちません。そのため、インライン関数に対して&演算子を使用した場合には、エラーになります。(【図B.57】)

```
inline int func(int i)
{
        return i;
}

main()
{
        int (*f)(int);

        f = &func;
}
```

【エラーメッセージ】

```
[Error(ccom):smp.c,line 10] can't get inline function's address by '&' operator
  ===>          f = &func;
Sorry, compilation terminated because of these errors in main().
```

図B.57 インライン関数の不適切な記述例(3)

● 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、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。

そのため、複数のファイルにまたがったインライン関数ではアクセスする領域が異なります。

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宣言をした場合には、警告を発します。 また、インライン関数内のstatic宣言は、推奨しません。(【図B.58】)

```
inline int func( int j)
{
        static int i = 0;

        i++;
        return i + j;
}
```

【ワーニングメッセージ】

```
[Warning(ccom):smp.c,line 3] static valuable in inline function
  ===>          static int i = 0;
```

図B.58 インライン関数の不適切な記述例(4)

●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。

従いまして、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　コメント "//"の概要

NC308では、"/*"と"*/"で記述するコメント以外にC++言語風のコメント"//"を記述することができます。

B.6.1　コメント "//"の概要

C言語でコメントは、"/*"と"*/"の間に記述する必要があります。C++言語でのコメントは、同一行で"//""以降の記述は、すべてコメントになります。

B.6.2　コメント "//"の書式

行中で"//"を記述した場合同一行の以降の記述は、コメントとして扱われます。
【図B.59】に書式を示します。

```
// コメント
```

図B.59　コメントの記述書式

コメントの記述例を【図B.60】に示します。

```
void
func(void)
{
    int i;     /* コメントです。*/
    int j;     // コメントです。
     :
     :
}
```

図B.60　コメントの記述例

B.6.3　"//" と "/*"の優先順序

"//"と"/*" ～ "/*"の優先順序は、「先に出現した方」が優先されます。
したがって、"//" から改行コードの間に記述された"/*" は、コメント開始の意味を持ちません。また、"/*" ～ "*/" の間に記述された "//" も、コメント開始の意味を持ちません。

B.7　#pragma　拡張機能

B.7.1　#pragma　拡張機能の一覧

#pragma*1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.3　メモリ配置に関する拡張機能

拡張機能	機能の内容
#pragma ROM	指定した変数をromセクションに配置します。 記述形式)　#pragma ROM△変数名 記述例)　#pragma ROM val ※本機能はNC77,NC79との互換のためにあります。通常はconst修飾子を用いてromセクションに配置してください。
#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 SB16DATA	SB相対16ビットディスプレースメント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 SB16DATA △変数名 記述例)　#pragma SB16DATA　sym_data
#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

b.　組み込み機器に関する拡張機能の使用方法

表B.4　組み込み機器に使用するための拡張機能 (1/3)

拡張機能	機能の内容
#pragma ADDRESS (#pragma EQU)	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

※1.#pragma に続く拡張機能指定語(ADDRESS,INTERRUPT,SBDATA,ASM 等)を小文字でも記述できます。

表B.4　組み込み機器に使用するための拡張機能 (2/3)

拡張機能	機能の内容
#pragma DMAC	外部変数に対して、DMACレジスタを割り付けます。 記述形式)　#pragma DMAC△変数名△DMACレジスタ名 記述例)　#pragma DMAC dma0 DMA0
#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。 スイッチ[/C]は、宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。 記述形式 1) #pragma INTCALL△[/C]△ 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) #pragma INTCALL /C 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() #pragma INTCALL /C 25 func() ※本宣言を行う前に、必ず関数のプロトタイプ宣言を行ってください。
#pragma INTERRUPT (#pragma INTF)	C言語で記述した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|/F]△割り込み処理関数名 #pragma INTERRUPT△[/B|/E|/F]△割り込みベクタ番号△割り込み処理関数名 #pragma INTERRUPT△[/B|/E|/F]△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) #pragma INTERRUPT /F int_func (vect=20) ※C77との互換のため#pragma INTFも使用できます。
#pragma PARAMETER	アセンブラ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。 スイッチ[/C]は、宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。 記述形式) #pragma PARAMETER△[/C]△関数名(レジスタ名) 記述例) #pragma PARAMETER asm_func(R0, R1) #pragma PARAMETER /C asm_func(R0, R1) ※本宣言を行う前に、必ず関数のプロトタイプ宣言を行ってください。

表B.4　組み込み機器に使用するための拡張機能 (3/3)

拡張機能	機能の内容
#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。スイッチ[/C]は、宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。 記述形式) #pragma SPECIAL△[/C]△番号△関数名() #pragma SPECIAL△[/ C]△関数名(vect=呼び出し番号) 記述例) #pragma SPECIAL 30 func() #pragma SPECIAL /C 30 func() #pragma SPECIAL func() (vect=30) #pragma SPECIAL /C func() (vect=30)

c. MR308に関する拡張機能の使用方法

表B.5　MR308サポートに関する拡張機能

拡張機能	機能の内容
#pragma ALMHANDLER	MR308のアラームハンドラ名を宣言します。 記述形式)　#pragma　ALMHANDLER△関数名 記述例)　#pragma　ALMHANDLER　alm_func
#pragma CYCHANDLER	MR308の周期起動ハンドラ名を宣言します。 記述形式)　#pragma　CYCHANDLER△関数名 記述例)　#pragma　CYCHANDLER　cyc_func
#pragma INTHANDLER #pragma HANDLER	MR308の割り込みハンドラ名を宣言します。 記述形式1)　#pragma　INTHANDLER△関数名 #pragma　INTHANDLER△[/E]△関数名 記述形式2)　#pragma　HANDLER△関数名 #pragma　HANDLER△[/E]△関数名 記述例)　#pragma　INTHANDLER　int_func
#pragma TASK	MR308のタスクの開始関数名を宣言します。 記述形式)　#pragma　TASK△タスクの開始関数名 記述例)　#pragma　TASK　task1

補足: 上記の拡張機能は、通常、MR308付属のコンフィグレータが生成するので、ユーザーは、記述する必要はありません。

d. その他の拡張機能の使用方法

表B.6　その他の拡張機能

拡張機能	機能の内容
#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　R1,02H #pragma ENDASM
#pragma JSRA	JSR命令をJSR.A命令に固定して関数を呼び出します。 記述形式)　#pragma　JSRA △関数名 記述例)　#pragma　JSRA　func
#pragma JSRW	JSR命令をJSR.W命令に固定して関数を呼び出します。 記述形式)　#pragma　JSRW△関数名 記述例)　#pragma　JSRW　func
#pragma PAGE	アセンブラリスティング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
#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)

B.7.2　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 ROM

romセクションへの配置機能

[機　能] 指定データ(変数)をromセクションに配置します。

[書　式] #pragma ROM△変数名

[解　説] この拡張機能は、以下の条件のどちらか一方を満たす変数に対する#pragma ROMのみ有効となります。

①関数外で定義されたextern宣言されていない(実体を定義した)変数
②関数内でstatic宣言された変数

[規　定] 1.変数名以外を指定した場合、無効となります。
2.#pragma ROM宣言の二重定義はエラーとなりません。
3.初期化式を記述しない場合は、初期値を0としてromセクションに配置されます。

[使用例]

```
[C言語ソースプログラム]
#pragma ROM i
unsigned int    i;          ←①の条件を満たす変数i

void func()
{
    static int i = 20;      ←②の条件を満たす変数i
        :
    (以下省略)


[アセンブリ言語ソースプログラム]
        .SECTION        rom_NE,ROMDATA
___S0_i:                ;### C's name is i    ←②の条件を満たす変数i
        .word   0014H
        .glb    _i
_i:                                           ←①の条件を満たす変数i
        .byte   00H
        .byte   00H
```

図B.61　#pragma ROM宣言の使用例

[備　考]　本機能はNC77,NC79との互換のためにあります。通常はconst修飾子を用いてromセクションに配置してください。

#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/80シリーズ等では、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、アセンブラの疑似命令.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. 同一変数に対して #pragma SBDATAと #pragma SB16DATA を同時に指定できません。
6. 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。※1

[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;

func( void )
{
        sym_data.bit1 = 0;
            :
          (省略)
            :
```

図B.62　#pragma SBDATA宣言の使用例

[補　足]　NC308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。

※1.ROMデータに対して、#pragma SBDATAの宣言を行なわないでください。

#pragma SB16DATA

SB相対16ビットディスプレースメントアドレッシング使用変数宣言機能

[機　能] SB相対16ビットディスプレースメント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 SB16DATA△変数名

[解　説] M16C/60シリーズ等では、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。
SB相対アドレッシングでアクセスするセクションを far領域に配置した場合、#pragma SB16DATAを使用する事により、変数のデータ参照時に、SB相対アドレッシングを16ビットディスプレースメントで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 SB16DATAを使用する場合、SB相対アドレッシングでアクセスするセクションを far領域に配置する必要があります。
   そのためスタートアップファイルでのセクション配置の指定を変更する必要があります。
   本機能を使用するためのセクション配置については、”第２章　コンパイラの基本的な使い方　2.2.2 スタートアッププログラムのカスタマイズ” および、”第２章　コンパイラの基本的な使い方　2.2.3 メモリ配置のカスタマイズ” を参照してください。
2. 同一変数に対して #pragma SBDATAと #pragma SB16DATA を同時に指定できません。
3. 変数名以外を指定した場合、無効となります。
4. 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。
5. #pragma SB16DATAを宣言した変数は、領域確保の際にSB16DATA属性のセクションに配置されます。
6. ROMデータに対して#pragma SB16DATAを宣言した場合、無効となります。

[使用例]

```
#pragma SB16DATA sym_data

int far sym_data;

func( void )
{
        sym_data = 1;
}
```

図B.63　#pragma SB16DATA宣言の使用例

[補　足]　NC308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。
また、この宣言をデータ(data、bss 及び rom)セクションに対して行った場合は、そのファイル中で実体を定義した全てのデータ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。

- ●programセクション、dataセクションは、同一ファイル内で複数回セクション名を変更することができます。
- ●その他のセクションは、複数回セクション名を変更することができません。

[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)

[アセンブリ言語ソースプログラム]
;###    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;     ←data1_NEセクションに配置
func()
{
    (省略)
 }
#pragma SECTION data data2
int j;     ←data2_NEセクションに配置
sub()
{
    (省略)
 }
```

図B.64　#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。

#pragma SECTION

セクション名変更機能

[注意事項] NC308WA V3.10 以前では、data及び romセクションはbssセクションと同様に、ファイル単位でしかセクション名を変更できませんでした。
このため、NC308WA V3.10 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。

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。

#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

[解　説] 本コンパイラでは、構造体はパックされます。例えば、【図B.65】に示す構造体のメンバは、宣言された順にパディング(透き間)を入れずに配置されます。

[使用例]

```
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.65　構造体メンバの配置例(1)

①パックの禁止

本コンパイラでは、拡張機能として構造体メンバの配置を制御することができます【図B.65】に示した構造体を#pragma STRUCTでパックを禁止したときのメンバの配置例を【図B.66】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.66　構造体メンバの配置例(2)

【図B.66】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

[解　説] ②メンバ配置の並び換え

本コンパイラではでは、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.66】に示した構造体を#pragma STRUCTで配置を並び替えたときの配置例を【図B.67】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.67 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。

[使用例]

```
#pragma STRUCT  TAG   unpack
struct TAG {
        int     i;
        char    c;
} s1;
```

図B.68　#pragma STRUCT宣言の使用例

B.7.3　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(#pragma EQU)

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ファイルに展開され、疑似命令.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. #pragma ADDRESS宣言を行う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. #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ワーニングとなり、記述した初期化式は無効となります。
7. #pragma ADDRESS又は#pragma EQU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。

[注意事項] 下記のように、変数が、#pragma ADDRESS の指定より先に使用されている場合には、#pragma ADDRESS の指定は無効です。

```
char port;
void func()
{
    port = 0;    // #pragma ADDRESS の指定の前に変数を使用
}
#pragma ADDRESS port 100H
```

#pragma ADDRESS(#pragma EQU)

入出力変数の絶対アドレス割り付け機能

[使用例]

```
#pragma ADDRESS io      24H
int         io;

void func(void)
{
          io = 10;
}
```

図B.69　#pragma ADDRESS宣言

[備　考] C77 V.2.10以前のバージョンとの互換性のために#pragma EQUの記述もできます。この書式の絶対アドレスはC言語の数値表現で記述します。

#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] 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。

[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ファイルに展開され、疑似命令.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型の変数のみ、変数名に指定することができます。_Bool型以外の変数を指定した場合には、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. #pragma BITADDRESS宣言を行う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せんが、後に宣言したアドレスが有効になります。
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。

[使用例]

```
#pragma BITADDRESS io 1,100H
_Bool       io;

void func(void)
{
        io = 1;
}
```

図B.70　#pragma BITADDRESS宣言

#pragma DMAC

外部変数のDMACレジスタ割り付け機能

[機　能]　指定した外部変数に対して、CPU内部のDMACレジスタを割り付けます。

[書　式]　#pragma DMAC△変数名△DMACレジスタ名

[規　定]

1. #pragma　DMAC の記述前に、指定する変数を宣言しておく必要があります。
2. 指定できる DMAC レジスタ、および変数の型は以下の通りです。

	16bit レジスタ	24bit レジスタ
レジスタ名	DMD0,　DMD1, DCT0,　DCT1, DRC0,　DRC1,	DMA0,　DMA1, DSA0,　DSA1, DRA0,　DRA1,
使用できる型	unsigned int, unsigned short	任意の型への far ポインタ ただし、関数へのポインタは不可

3. 同一 DMAC レジスタに複数の #pragma DMAC を記述する事はできません。
4. #pragma DMAC で指定された変数に対して,”＆”（アドレス演算子）、”（　）“（関数呼び出し演算子）、”［　］“（配列添字演算子）、”-> “（間接メンバ演算子）を指定する事はできません。
5. #pragma　DMAC で指定された変数は、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。

[使用例]

```
void _far *dma0
#pragma DMAC dma0 DMA0

func()
{
        unsigned char buff[10];
        dma0 = buff;
}
```

図B.71　#pragma DMAC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△[/C]△INT番号△アセンブラ関数名(レジスタ名,レジスタ名, ...)
②#pragma INTCALL△[/C]△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。
また、宣言時に以下のスイッチを指定できます。
[/C]
宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。

[規　定] ①アセンブラ関数を宣言する場合
1. #pragma INTCALL宣言を行う前には、必ずアセンブラ関数のプロトタイプ宣言を行ってください。プロトタイプ宣言を行わない場合は警告を出力し、この宣言を無効にし　ます。
2. プロトタイプ宣言では、以下の項目を守ってください。
a. 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。
b. 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 long型
c. 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. 呼び出し時には、引き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}(24ビットレジスタ)
R2R0, R3R1,A1,A0
●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。
4. INT番号は、10進数でのみ記述可能です。

②C言語で実体を記述した関数を宣言する場合
1. #pragma INTCALL宣言を行う前には、必ず関数のプロトタイプ宣言を行ってくださいプロトタイプ宣言を行わない場合は警告を出力し、この宣言を無効にします。
2. #pragma INTCALL宣言を行う関数の引数にレジスタ名を指定できません。
3. プロトタイプ宣言では、以下の項目を守ってください。
a.プロトタイプ宣言では、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。
b. 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
4. INT番号は、10進数でのみ記述可能です。

#pragma INTCALL

ソフトウェア割り込み関数宣言機能

[使用例]

```
int asm_func(unsigned long, unsigned int); ←アセンブラ関数のプロトタイプ宣言
#pragma INTCALL 25 asm_func(R2R0, R1)

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.72　#pragma INTCALL宣言の使用例（①アセンブラ関数を宣言する場合）

```
int c_func(unsigned int, unsigned int)    ←C言語関数のプロトタイプ宣言
#pragma INTCALL 25 c_func();              ←引数のレジスタ名を指定しないでください。

void main()
{
     int     i, j;
     i = 0x7FFD;
     j = 0x007F;

     c_func( i, j );                      ←C言語関数の呼び出し

}
```

図B.73　#pragma INTCALL宣言の使用例（②C言語で実体を記述した関数を宣言する場合）

※付属のスタートアップファイルをご使用になる場合には、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、「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(#pragma INTF)

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] #pragma INTERRUPT△[/B ¦ /E ¦ /F]△割り込み処理関数名
#pragma INTERRUPT△[/B ¦ /E ¦ /F]△割り込みベクタ番号△割り込み処理関数名
#pragma INTERRUPT△[/B ¦ /E ¦ /F]△割り込み処理関数名(vect=割り込みベクタ番号)

[解　説] 1. C言語で記述した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。
[/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。
[/F]
出口処理で、FREIT でリターンします。

3. 宣言時に割り込みベクタ番号を指定できます。
割り込みベクタ番号を指定し、コンパイルオプション -fmake_vector_table(-fMVT)を指定してコンパイルすることで、可変ベクタテーブルを自動的に生成することができます。
なお、-fmake_vector_table(-fMVT)オプション指定時には、スタートアッププログラム中の vectorセクションに割り込み関数名を指定する必要はありません。

[規 定]
1. 引数を持つ割り込み処理関数を記述した場合、コンパイル時に警告を出力します。
2. 戻り値を返す割り込み処理関数を記述した場合、コンパイル時に警告を出力します。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は同時に指定できません。
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。

例）

```
#pragma INTTERUPT intr(vect=10)
#pragma INTTERUPT intr(vect=20)   // 割り込みベクタ番号20が有効
```

次ページ→

#pragma INTERRUPT(#pragma INTF)

割り込み関数の記述機能

[規 定]　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、コンパイル時にワーニングとなります。

- ●#pragma ALMHANDLER
- ●#pragma INTHANDLER
- ●#pragma HANDLER
- ●#pragma CYCHANDLER
- ●#pragma TASK

[使用例]

```
#pragma INTERRUPT /B i_func

void i_func()
{
        int_counter += 1;
}
```

図B.74　#pragma INTERRUPT宣言の使用例

[備 考] C77 V.2.10以前のバージョンとの互換性のために#pragma INTFの記述もできます。
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破壊されないように注意してください。

#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△[/ C]△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}(24ビットレジスタ)
　　A0，A1，R2R0，R3R1
●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。

また、宣言時に以下のスイッチを指定できます。

[/C]
　宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。

[規　定]
1. #pragma PARAMETER宣言を行う前には、必ずアセンブラ関数のプロトタイプ宣言を行ってください。プロトタイプ宣言を行わない場合は警告を出力し、この宣言を無効にします。
2. プロトタイプ宣言では、以下の項目を守ってください。
   a.プロトタイプ宣言の引数の数と#pragma PARAMETER宣言の引数の数が一致している必要があります。
   b.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 long型
   c. 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で指定した関数の実体をC言語で記述した時はエラーとなります。

[使用例]

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

void main()
{
     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] #pragma SPECIAL△[/ C]△呼び出し番号△関数名()
#pragma SPECIAL△[/ C]△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各テーブルに設定した番地に 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.宣言時に以下のスイッチを指定できます。
[/C]
宣言した関数の呼び出し時に、退避が必要なレジスタを退避するためのコードを生成します。
3. 宣言時に呼び出し番号を指定できます。
呼び出し番号を指定し、コンパイルオプション -fmake_special_table(-fMST)を指定してコンパイルすることで、スペシャルページベクタテーブルを自動的に生成することができます。

[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0F0000Hから 0FFFFFH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. #pragma SPECIALで宣言した関数の先頭アドレスには、ラベルとして"__SPECIAL_呼び出し番号:"が出力されます。スタートアップファイルで、スペシャルページサブルーチンテーブルにこのラベルを設定してください。※1
なお、-fmake_special_table(-fMST) オプションを指定した場合は、上記設定は、必要ありません。
4.同じ関数に、異なる呼び出し番号を記述した場合は、後に宣言された呼び出し番号が有効になります。
例）
#pragma SPECIAL func(vect=20)
#pragma SPECIAL func(vect=30)　　// 呼び出し番号30が有効
5.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。

[使用例]

```
#pragma SPECIAL 20 func()
void func(unsigned int, unsigned int);
void main()
{
     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.76　#pragma SPECIAL宣言の使用例

※1.付属のスタートアップファイルをご使用になる場合には、fvectorセクションの内容を変更して下さい。変更方法の詳細は、「2.2 スタートアッププログラムの準備」を参照してください。

B.7.4　MR308に関する拡張機能の使用方法

本コンパイラは、以下に示すリアルタイムOS　MR308 のサポートに関する拡張機能を持っています。

#pragma ALMHANDLER

アラームハンドラの記述機能

[機　能] MR308のアラームハンドラ関数を宣言します。

[書　式] #pragma ALMHANDLER△アラームハンドラ名

[解　説] C言語で記述したアラームハンドラを上記の書式で宣言することにより、関数の出入口でアラームハンドラの処理を行うコードを生成します。

●システムクロック割り込みからはJSR命令で呼び出します。復帰はRTS命令あるいはEXITD命令でリターンします。

[規　定]
1. 引数を持つアラームハンドラを記述することはできません。
2. アラームハンドラの戻り値がvoid型であるように宣言してください。
3. #pragma ALMHANDLER宣言以降に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、無効となります。
5. #pragma ALMHANDLER宣言の二重定義はエラーとなりません。
6. #pragma ALMHANDLER宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にワーニングとなります。
   - ●#pragma INTERRUPT
   - ●#pragma INTHANDLER
   - ●#pragma HANDLER
   - ●#pragma CYCHANDLER
   - ●#pragma TASK

[使用例]

```
#include   <mrXXX.h>
#include   "id.h"

#pragma ALMHANDLER      alm

void alm(void)          ←必ずvoid型で宣言します。
{
          :
          (省略)
          :
}
```

図B.77　#pragma ALMHANDLER宣言の使用例

#pragma CYCHANDLER

周期起動ハンドラ関数の記述機能

[機　能] MR308の周期起動ハンドラを宣言します。

[書　式] #pragma CYCHANDLER△周期起動ハンドラ名

[解　説] C言語で記述した周期起動ハンドラを上記の書式で宣言することにより、関数の出入口で周期起動ハンドラの処理を行うコードを生成します。

●システムクロック割り込みからはJSR命令で呼び出します。復帰はRTS命令あるいはEXITD命令でリターンします。

[規　定]
1. 引数を持つ周期起動ハンドラを記述することはできません。
2. 周期起動ハンドラの戻り値がvoid型であるように宣言してください。
3. #pragma CYCHANDLER宣言以降に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、無効となります。
5. #pragma CYCHANDLER宣言の二重定義はエラーとなりません。
6. #pragma CYCHANDLER宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にワーニングとなります。
   - ●#pragma INTERRUPT
   - ●#pragma INTHANDLER
   - ●#pragma HANDLER
   - ●#pragma ALMHANDLER
   - ●#pragma TASK

[使用例]

```
#include   <mrXXX.h>
#include   "id.h"

#pragma CYCHANDLER      cyc

void cyc(void)          ←必ずvoid型で宣言します。
{
          :
          (省略)
          :
}
```

図B.78　#pragma CYCHANDLER宣言の使用例

#pragma INTHANDLER(#pragma HANDLER)

割り込みハンドラ関数の記述機能

[機　能] MR308のOS依存割り込みハンドラを宣言します。

[書　式] ①#pragma INTHANDLER△[/E]△割り込みハンドラ名
②#pragma HANDLER△[/E]△割り込みハンドラ名

[解　説] 1. C言語で記述した割り込みハンドラ関数を上記の書式で宣言することにより、関数の出入口で以下の処理を行うコードを生成します。

- 入口処理
  レジスタを現在のスタックへ退避します。
- 出口処理
  ret_intシステムコールによる割り込みからの復帰を行います。また、関数の途中に記述されたreturn文によってリターンする場合も、ret_intシステムコールにより復帰します。

2. 宣言時に以下のスイッチを指定できます。
   [/E]
   本機能で宣言した割り込みハンドラに制御が切り替わった直後に、多重割り込みを許可します。

3. OS独立割り込みハンドラは#pragma INTERRUPTで宣言してください。

[規　定]
1. 引数を持つ割り込みハンドラを記述することはできません。
2. 割り込みハンドラの戻り値がvoid型であるように宣言してください。
3. C言語からret_intシステムコールを使用しないでください。
4. #pragma INTHANDLER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。
5. 関数名以外を指定した場合、無効となります。
6. #pragma INTHANDLER宣言の二重定義はエラーとなりません。
7. #pragma INTHANDLER宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にワーニングとなります。
   - ●#pragma INTERRUPT
   - ●#pragma HANDLER
   - ●#pragma ALMHANDLER
   - ●#pragma CYCHANDLER
   - ●#pragma TASK

[使用例]

```
#include   <mrXXX.h>
#include   "id.h"

#pragma INTHANDLER      hand

void hand(void)
{
        :
        (省略)
        :
    /* ret_int(); */
}
```

図B.79　#pragma INTHANDLER宣言の使用例

#pragma TASK

タスク開始関数の記述機能

[機　能] MR308のタスク開始関数を宣言します。

[書　式] #pragma TASK△タスクの開始関数名

[解　説] C言語で記述したタスクの開始関数を上記の書式で宣言することにより、関数の出口でタスク専用の処理を行うコードを生成します。

●出口処理

ext_tskシステムコールで終了します。また、関数の途中に記述されたreturn文によってリターンする場合も、ext_tskシステムコールにより復帰します。

[規　定]
1. タスクからのリターンにext_tskシステムコールを記述する必要はありません。
2. タスクの戻り値がvoid型であるように宣言してください。
3. #pragma TASK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、無効となります。
5. #pragma TASK宣言の二重定義はエラーとなりません。
6. #pragma TASK宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にワーニングとなります。
   - ●#pragma INTERRUPT
   - ●#pragma INTHANDLER
   - ●#pragma HANDLER
   - ●#pragma ALMHANDLER
   - ●#pragma CYCHANDLER

[使用例]

```
#include   <mrXXX.h>
#include   "id.h"

#pragma TASK      main
#pragma TASK      tsk1

void main(void)            ←必ずvoid型で宣言します。
{
          :
         (省略)
          :
       sta_tsk(ID_idle);
       sta_tsk(ID_tsk1);
       /* ext_tsk(); */    ←ext_tskシステムコールを記述する必要はありません。
}

void tsk1()
          :
       (以下省略)
```

図B.80　#pragma TASK宣言の使用例

B.7.5　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 ASM(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、アセンブリ言語ファイルにそのまま出力します。

[規　定] #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がない場合にはNC308は処理を中断します。

[注意事項]●アセンブリ言語記述において、レジスタの内容を破壊する記述をしないでください。レジスタの内容を破壊する記述をする場合には、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。
● '#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。
● '#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。

[使用例]

```
void func()
{
    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リ言語ファイルにそのまま出力します。

図B.81　#pragma ASM(ENDASM)記述例

[補　足] #pragma ASM から#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 JSRA

関数呼び出し命令指定機能

[機　能] 関数をJSR.A命令により呼び出します。

[書　式] #pragmaJSRA△関数名

[解　説] #pragma JSRA で宣言した関数は、JSR.A命令固定で関数呼び出しを行います。
オプション-fJSRWで生成したコードで、リンク時にエラーとなる関数は、 #pragma JSRA を指定する事によりエラーを回避できます。

[規　定] -fJSRWを指定しない場合には、効果がありません。

[使用例]

```
extern void func(int i);
#pragma JSRA func()

void main(void)
{

    func(1);


}
```

図B.82　#pragma JSRA 記述例

#pragma JSRW

関数呼び出し命令指定機能

[機　能] 関数をJSR.W命令により呼び出します。

[書　式] #pragma JSRW△関数名

[解　説] 任意の関数から、同一ファイルに実体の定義がない関数を呼び出す場合、通常ではJSR.A命令で呼び出します。#pragma JSRW で宣言した関数は、JSR.W命令固定で関数呼び出しを行います。この拡張機能によりROM容量を削減できます。

[規　定]
1. static関数には、#pragma JSRWを指定しないでください。
2. 関数呼び出し時に#pragma JSRW で宣言した関数に届かない時は、リンク時にエラーとなります。この場合には、宣言を行わないでください。

[使用例]

```
extern void func(int i);
#pragma JSRW func()

void main(void)
{

    func(1);



}
```

図B.83　#pragma JSRW 記述例

[備　考] #pragma JSRWは、関数の直接呼び出しの時にのみ有効です。間接呼び出しの時には、効果がありません。

#pragma PAGE

.PAGE疑似命令出力機能

[機　能] アセンブラで出力するリストファイルでの改行位置を宣言します。

[書　式] #pragma PAGE

[解　説] C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リ言語ファイルに、.PAGE疑似命令を出力します。アセンブラによりアセンブラリスティング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. アセンブラ疑似命令.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。

[使用例]

```
void func()
{
    int     i, j;

   for(i=0; i < 10;i++){
        func2();
   }

#pragma PAGE

   i++;

}
```

図B.84　#pragma PAGE 記述例

#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.本機能による宣言を行う前に、関数のプロトタイプ宣言を行ってください。アセンブラマクロ関数は、必ず static 宣言してください。
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。

char 型, _Bool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)

[使用例]

```
static   long   mul( int, int );     /* 必ず static にしてください。 */
#pragma   __ASMMACRO mul( R0, R2 )
#pragma   ASM
_mul .macro
mul.w     R2,R0       ;戻り値はR2R0 で返されます。
.endm
#pragma   ENDASM
long   l;
void   test_func( void )
{
    l  =  mul( 2, 3 );
}
```

図B.85　#pragma __ASMMACRO 記述例

B.8　アセンブラマクロ関数

B.8.1　アセンブラマクロ関数の概要

本コンパイラでは、アセンブラ命令の一部を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.8.2　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、下記のように、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];

func()
{
    l = rmpa_b(0,19,a,b); /* アセンブラマクロ関数(rmpa命令) */
}
```

図B.86　アセンブラマクロ関数の記述例

B.8.3　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ラ命令と、アセンブラマクロ関数としての機能と書式を示します

DADD

[機　能] val1 とval2 の10進加算の結果を返します。

[書　式]
```
#include <asmmacro.h>
static char dadd_b(char val1, char val2);     /* 8bit での演算の場合 */
static int  dadd_w(int  val1, int  val2);     /* 16bit 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 char dadc_b(char val1, char val2);     /* 8bit での演算の場合 */
static int  dadc_w(int  val1, int  val2);     /* 16bit での演算の場合 */
```

DSUB

[機　能] val1 からval2 の 10進減算の結果を返します。

[書　式]
```
#include <asmmacro.h>
static char dsub_b(char val1, char val2);  /* 8bit での演算の場合 */
static int  dsub_w(int  val1, int  val2);  /* 16bit での演算の場合 */
```

DSBB

[機　能] val1 からval2 の ボロー付き10進減算の結果を返します。

[書　式]
```
#include <asmmacro.h>
static char dsbb_b(char val1, char val2);  /* 8bit での演算の場合 */
static int  dsbb_w(int  val1, int  val2);  /* 16bit での演算の場合 */
```

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 long rmpa_b(long init, int count, char _far *p1, char _far *p2);
                                                    /* 8bit での演算の場合 */
static long rmpa_w(long init, int count, int  _far *p1, int  _far *p2);
                                                    /* 16bit での演算の場合 */
static long long rmpa_lw(long init, int count, int _far *p1, int  _far *p2);
                                                    /* 48bit での演算の場合 */
```

MAX

[機　能] val1 とval2 を比較して大きい方の値を返します。

[書　式]
```
static char max_b(char val1, char val2);  /* 8bit での演算の場合 */
static int  max_w(int  val1, int  val2);  /* 16bit での演算の場合 */
```

MIN

[機　能] val1 とval2 を比較して小さい方の値を返します。

[書　式]
```
static char min_b(char val1, char val2);  /* 8bit での演算の場合 */
static int  min_w(int  val1, int  val2);  /* 16bit 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(char _far *p1, char _far *p2, unsigned int count);
                                                    /* 8bitでの演算の場合 */
static void smovb_w(int  _far *p1, int  _far *p2, unsigned int count);
                                                    /* 16bit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(char _far *p1, char _far *p2, unsigned int count);
                                                    /* 8bit での演算の場合 */
static void smovf_w(int  _far *p1, int  _far *p2, unsigned int count);
                                                    /* 16bit での演算の場合 */
```

SMOVU

[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、アドレスの加算方向へ零が検出されるまで、ストリング転送を行います。戻り値はありません。

[書　式]
```
static void smovu_b(char _far *p1, char _far *p2); /* 8bit での演算の場合 */
static void smovu_w(int  _far *p1, int  _far *p2); /* 16bit での演算の場合 */
```

SIN

[機　能] p1 で示される固定の転送元番地から、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]
```
static void sin_b(char _far *p1, char _far *p2, unsigned int count);
                                                    /* 8bit での演算の場合 */
static void sin_w(int  _far *p1, int  _far *p2, unsigned int count);
                                                    /* 16bit での演算の場合 */
```

SOUT

[機　能] p1 で示される転送元番地からアドレスの加算方向へ、p2 で示される転送先番地にcount 回数分ストリング転送を行います。戻り値はありません。

[書　式]
```
static void sout_b(char _far *p1, char _far *p2, unsigned int count);
                                                        /* 8bit での演算の場合 */
static void sout_w(int  _far *p1, int  _far *p2, unsigned int count);
                                                        /* 16bit 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>
sfafic void sstr_b(char val, char _far *p, unsigned int count);
                                                        /* 8bit での演算の場合 */
static void sstr_w(int  val, int  _far *p, unsigned int count);
                                                        /* 16bit 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);  /* 8bit での演算の場合 */
static unsigned int  rolc_w(unsigned int  val);  /* 16bit 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);  /* 8bit での演算の場合 */
static unsigned int  rorc_w(unsigned int  val);  /* 16bit 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);
                                                    /* 8bitでの演算の場合 */
static unsigned int  rot_w(signed char count, unsigned int val);
                                                    /* 16bit 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);
                                                    /* 8bit での演算の場合 */
static unsigned int  sha_w(signed char count, unsigned int  val);
                                                    /* 16bit での演算の場合 */
static unsigned long sha_l(signed char count, unsigned long val);
                                                    /* 32bit 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);
                                                    /* 8bit での演算の場合 */
static unsigned int  shl_w(signed char count, unsigned int  val);
                                                    /* 16bit での演算の場合 */
static unsigned long shl_l(signed char count, unsigned long val);
                                                    /* 32bit での演算の場合 */
```

DIV

[機　能] val1 とval2 の除算を行ない符号付き除算した商を求めます。

[書　式] #include <asmmacro.h>

```
static signed char div_b(signed char val1, signed int  val2);
                                                /*符号付きの16bit/8bitの演算の場合 */
static signed int  div_w(signed int  val1, signed long  val2);
                                                /*符号付きの32bit/16bitの演算の場合 */
```

DIVU

[機　能] val1 とval2 の除算を行ない符号なし除算した商を求めます。

[書　式]
```
#include <asmmacro.h>
static unsigned char divu_b(unsugned char val1, unsigned int  val2);
                                                    /*符号なしの16bit/8bitの演算の場合*/
static unsigned int  divu_w(unsigned int  val1, unsigned long val2);
                                                    /*符号なしの32bit/16bitの演算の場合*/
```

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);  /* 8bit での演算の場合 */
static signed int  abs_w(signed int  val);  /* 16bit 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 */
```

付録C
C言語仕様概要

C言語仕様は、標準的なC言語の仕様に加え、ROM化を容易に行うための拡張機能を持っています。

C.1　性能仕様

C.1.1　標準仕様概要

本コンパイラは、M16C/80シリーズをターゲットにしたクロス環境のCコンパイラです。言語仕様的には、以下のM16C/80シリーズのハードウェア的な仕様、及びROM化を容易にするために用意した拡張機能を除けば、標準的なフルセットのC言語とほぼ同様の仕様を持っています。

●ROM化のための拡張機能(near/far修飾子、asm関数等)
●標準ライブラリ関数の中で浮動小数点ライブラリやホストマシンに依存した内容

C.1.2　性能概要

以下に本コンパイラの性能の概要を示します。

a.　測定環境

性能測定時のEWSの標準環境を【表C.1】、PCの標準環境を【表C.2】、Linuxの標準環境を【表C.3】に示す条件として想定しています。

表C.1　EWS標準環境

項　目	EWSの種類	UNIXのバージョン
EWSの環境	SPARCstation	日本語Solaris2.5
	HP 9000 700シリーズ	HP-UX V.10.20
スワップ領域の空き容量	100Mバイト以上	

表C.2　PC標準環境

項　目	PCの種類	OSのバージョン
ﾊﾟｰｿﾅﾙｺﾝﾋﾟｭｰﾀの環境	IBM PC/AT互換機	Windows ME Windows 2000
搭載CPUの種類	Pentium II	
メモリ容量	128Mバイト以上	

表C.3　Linux標準環境

項　目	PCの種類	OSのバージョン
ﾊﾟｰｿﾅﾙｺﾝﾋﾟｭｰﾀの環境	IBM PC/AT互換機	Turbo Linux 7.0
搭載CPUの種類	Pentium II	
メモリ容量	128Mバイト以上	

b.　C言語ソースファイル記述仕様

【表C.4】に本コンパイラのC言語ソースファイル記述に関する仕様を示します。なお、実測が不可能な一部の仕様は計算による予想値を示しています。

表C.4　C言語ソースファイル記述仕様

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

c. 仕様

【表C.5】に本コンパイラの仕様を示します。なお、実測が不可能な一部の仕様は計算による予想値を示しています。

表C.5 仕様

項　　目	仕 様
指定可能なファイル数	メモリの容量に依存します
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	メモリの容量に依存します
起動オプション-Iで指定可能なディレクトリ数	最大16
起動オプション-as30で引き渡し可能なパラメータ数	メモリの容量に依存します
起動オプション-ln30で引き渡し可能なパラメータ数	メモリの容量に依存します
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	メモリの容量に依存します
条件コンパイルにおけるネスト数	メモリの容量に依存します
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	メモリの容量に依存します
関数定義の数	メモリの容量に依存します
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	メモリの容量に依存します
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	メモリの容量に依存します
マクロ名の置き換えの数	メモリの容量に依存します
入力プログラムにおける論理ソース行の数	メモリの容量に依存します
#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)	メモリの容量に依存します
#if、#elif文で指定できる演算子、被演算子の合計数	メモリの容量に依存します
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
#pragma ADDRESSで定義可能な変数の数	メモリの容量に依存します
括弧のネスト数	メモリの容量に依存します
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	メモリの容量に依存します
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	メモリの容量に依存します
１つの外部識別子で意味をもつ文字数	メモリの容量に依存します
１ソースファイル中の外部識別子の数	メモリの容量に依存します
１ブロックでブロックスコープを持つ識別子	メモリの容量に依存します
１ソースファイル中のマクロ数	メモリの容量に依存します
１つの関数、関数呼び出しのパラメータ数	メモリの容量に依存します
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	メモリの容量に依存します
１つのオブジェクトサイズ(バイト数)	メモリの容量に依存します
１つの構造体／共用体のメンバ数	メモリの容量に依存します
１つの列挙中の列挙定数の数	メモリの容量に依存します
１つのstruct宣言リスト中の構造体／共用体のネスト数	メモリの容量に依存します
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	メモリの容量に依存します

C.2　基本言語仕様

本コンパイラの言語仕様を基本的な言語仕様と併せて説明します。

C.2.1　文法

文法の字句要素について説明します。本コンパイラは以下のものを字句(トークン)として処理します。

●キーワード
●識別子
●定数
●文字リテラル
●演算子
●区切り子
●注釈

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。

表C.6　キーワード一覧表

_asm	default	int	switch
_far	do	long	typedef
_near	double	near	union
asm	else	register	unsigned
auto	enum	restrict	void
_Bool	extern	return	volatile
break	far	short	while
case	float	signed	inline
char	for	sizeof	
const	goto	static	
continue	if	struct	

エントリー版では、キーワードの内、
near　far　inline　asm
を、キーワードとして扱いません。これらのキーワードを使用する場合には、
各キーワードの前に、”_”(アンダースコア)を付加してください。
_near　_far　_inline　_asm

b.　識別子

識別子は、以下の要素で構成されます。

●1文字目は英字又はアンダースコア (A～Z 、 a～z 、 __)
●2文字目以降は英数字又はアンダースコア (A～Z 、a～z 、0～9 、 __)

識別子名は最大31文字まで記述できます。ただし、日本語文字は識別子として記述できません。

c. 定数

定数は以下に示す3種類のタイプがあります。

●整数定数
●浮動小数点定数
●文字定数

(1)整数定数

整数定数は、10進数のほか、8進数、16進数を指定することができます。各進数の書式を【表C.7】に示します。

表C.7　整数定数の記述法

進　数	記述法	構　成	記述例
10進数	なにも付けない	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	0123456789ABCDEF	0XF又は0xf
		0123456789abcdef	

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　signed int 型→unsigned int 型→signed long 型→unsigned long 型
→signed long long 型→unsigned long long 型
●10進数　signed int 型→signed long 型→signed long long 型

また、接尾詞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

②long 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　signed long 型→unsigned long 型→signed long long 型
→unsigned long long 型
●10進数　signed long 型→signed long long 型

③long 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　signed long long 型→unsigned long long 型
●10進数　signed long long 型

(2)浮動小数点定数

浮動小数点定数は、定数値の後になにも付加しない場合、double 型として扱われます。float 型として扱う場合は、定数値の後にF又はfを付加して記述します。また、L又はlを付加した場合は、long double型として扱われます。

(3)文字定数

文字定数は通常、 '文字'のようにシングルクォーテーションの中に文字を記述して表現します。文字の中には以下のような拡張表記(エスケープ系列／トライグラフ系列)が使用できます。16進数と8進数の表現は、\xの後に16進数を続けると16進数に、\の後に8進数を続けると8進数となります。

表C.8　拡張表記一覧表

表　記	内容(エスケープ系列)	表　記	内容(トライグラフ系列)
\'	シングルクォーテーション	\定数値	8進数
\"	ダブルクォーテーション	\x定数値	16進数
\\	逆スラッシュ	??(	文字[を表します。
\?	疑問符	??/	文字\を表します。
\a	ベル文字	??)	文字]を表します。
\b	バックスペース	??'	文字^を表します。
\f	改ページ	??<	文字{を表します。
\n	改行	??!	文字|を表します。
\r	復帰	??>	文字}を表します。
\t	水平タブ	??－	文字~ を表します。
\v	垂直タブ	??=	文字#を表します。

d. 文字リテラル

文字リテラルは、"文字列"のようにダブルクォーテーションの中に文字列を記述して表現します。文字リテラルにも【表C.8】に示した文字定数と同じ拡張表記が使用できます。

e. 演算子

本コンパイラは、【表C.9】に示すものを演算子として解釈します。

表C.9　演算子一覧表

単項演算子	＋＋	論理演算子	&&
	－－		||
	－		!
二項演算子	＋	条件演算子	?:
	–	カンマ演算子	,
	＊	アドレス演算子	&
	/	ポインタ演算子	＊
	%	ビット演算子	<<
代入演算子	=		>>
	＋=		&
	－=		|
	＊=		^
	/=		~
	%=		&=
関係演算子	>		|=
	<		^=
	>=		<<=
	<=		>>=
	==	sizeof演算子	sizeof
	!=		

f. 区切り子

本コンパイラは、以下に示すものを区切り子として解釈します。

- {
- }
- :
- ;
- ,

g. 注釈

- 注釈は、/＊で始まり＊/で終了します。注釈のネストはできません。
- 注釈は、// で始まり行末で終了します。

C.2.2　型

a.　データ型

本コンパイラでは、以下に示すデータ型をサポートしています。

- ●文字型
- ●整数型
- ●構造体
- ●共用体
- ●列挙型
- ●void型
- ●浮動小数点型

b.　型修飾子

本コンパイラでは、以下に示すものを型修飾子として解釈します。

- ●const
- ●volatile
- ●restrict
- ●near
- ●far

c.　データ型とサイズ

各データ型に対応するビットサイズを【表C.10】に示します。

表C.10　データ型とビットサイズ

型　名	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	なし	8	0～255
unsigned char			
signed char	有り	8	-128～127
int	有り	16	-32768～32767
short			
signed int			
signed short			
unsigned int	なし	16	0～65535
unsigned short			
long	有り	32	-2147483648～2147483647
signed long			
unsigned long	なし	32	0～4294967295
long long	有り	64	-9223372036854775808 ～ 9223372036854775807
signed long long			
unsigned long long	なし	64	18446744073709551615
float	有り	32	1.17549435e-38F～3.40282347e+38F
double	有り	64	2.2250738585072014e-308～ 1.7976931348623157e+308
long double			
nearポインタ	なし	16	0～0xFFFF
farポインタ	なし	32	0～0xFFFFFFFF

●_Bool型は符号指定は、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して解釈します。
●long long型のビットフィールドは、使用できません。

C.2.3　式

【表C.11】、【表C.12】に式の種類と式の構成要素の関係を示します。

表C.11　式の種類と構成要素(1)

式の種類	式の構成要素
一次式	識別子
	定数
	文字リテラル
	(式)
	一次式
後置式	後置式[式]
	後置式(引数の並び，...)
	後置式．識別子
	後置式－>識別子
	後置式＋＋
	後置式－－
	後置式
単項式	＋＋単項式
	－－単項式
	単項演算子　キャスト式
	sizeof　単項式
	sizeof(型名)
	単項式
キャスト式	(型名)キャスト式
	キャスト式
式	式 * 式
	式 / 式
	式 % 式
加減式	式 ＋ 式
	式 － 式
ビット単位のシフト式	式 << 式
	式 >> 式

表C.12　式の種類と構成要素(2)

式の種類	式の構成要素
関係式	式
	式 < 式
	式 > 式
	式 <= 式
	式 >= 式
等価式	式 == 式
	式 != 式
ビット単位のAND式	式 & 式
ビット単位の排他的OR式	式 ^ 式
ビット単位のOR式	式 ¦ 式
論理AND式	式 && 式
論理OR式	式 ¦¦ 式
条件式	式 ? 式 : 式
代入式	単項式 += 式
	単項式 -= 式
	単項式 ∗= 式
	単項式 /= 式
	単項式 %= 式
	単項式 <<= 式
	単項式 >>= 式
	単項式 &= 式
	単項式 ¦= 式
	単項式 ^= 式
	代入式
コンマ演算子	式 , 単項式

C.2.4　宣言

宣言には以下に示す2種類のタイプがあります。

●変数宣言
●関数宣言

a.　変数宣言

変数の宣言は、【図C.1】に示す書式で記述します。

記憶クラス指定子△型宣言子△宣言指定子△初期化式;

図C.1　変数の宣言書式

(1)記憶クラス指定子

本コンパイラでは、以下の記憶クラス指定子をサポートしています。

● extern　● auto　● typedef
● static　● register

(2)型宣言子

本コンパイラでは、以下の型宣言子をサポートしています。

● _Bool　● char　● int　● short　● long
● long long　● float　● double　● unsigned　● signed
● struct　● union　● enum

(3)宣言指定子

本コンパイラでは、【図C.2】に示す書式で宣言指定子を記述します。

宣言子　：　ポインタ$_{opt}$ 宣言子2 宣言子2　：　識別子(宣言子) 宣言子2[定数式$_{opt}$] 宣言子2(仮引数の並び$_{opt}$) ※配列の個数を示す定数式は最初の配列のみ省略できます。 ※optはオプション部であることを示します。

図C.2　宣言指定子の書式

(4)初期化式

本コンパイラでは、初期化式に【図C.3】に示す初期値を記述できます。

整数型	：	定数
整数型配列	：	定数、定数．．．
文字型	：	定数
文字型配列	：	文字リテラル定数、定数．．．
ポインタ型	：	文字リテラル
ポインタ配列	：	文字リテラル、文字リテラル．．．

図C.3　初期化式に記述できる初期値

b.　関数宣言

関数の宣言は、【図C.4】に示す書式で記述します。

●関数宣言（定義）
記憶クラス指定子△型宣言子△宣言指定子△プログラム本体

●関数宣言（プロトタイプ宣言）
記憶クラス指定子△型宣言子△宣言指定子；

図C.4　関数の宣言書式

(1)記憶クラス指定子

本コンパイラでは、以下の記憶クラス指定子をサポートしています。

- extern
- static

(2)型宣言子

本コンパイラは、以下の型宣言子をサポートしています。

● _Bool	● char	● int	● short	● long
● long long	● float	● double	● signed	● unsigned
● struct	● union	● enum		

(3)宣言指定子

本コンパイラでは、【図C.5】に示す書式で宣言指定子を記述します。

```
宣言子　　：　　ポインタopt 宣言子2
宣言子2　：　　識別子( 仮引数の並びopt )
　　　　　　( 宣言子 )
　　　　　　宣言子[ 定数式opt ]
　　　　　　宣言子( 仮引数の並びopt )

※配列の個数を示す定数式は最初の配列のみ省略できます。
※optはオプション部であることを示します。
※プロトタイプ宣言の場合は仮引数の並びでなく、型宣言子の並びとなります。
```

図C.5　宣言指定子の書式

(4)プログラム本体

プログラム本体は、【図C.6】に示す書式で記述します。

```
変数宣言子の並びopt 複文

※プロトタイプ宣言の場合はプログラム本体は無く、セミコロンで終了します。
※optはオプション部であることを示します。
```

図C.6　プログラム本体の書式

C.2.5　文

本コンパイラでは、以下の文をサポートしています。

- ●名札付き文
- ●複文
- ●式 / 空文
- ●選択文
- ●繰り返し文
- ●分岐文
- ●アセンブリ言語記述文

a.　名札付き文

名札付き文は、【図C.7】に示す書式で記述します。

```
識別子　　：　　文
case定数　：　　文
default　　：　　文
```

図C.7　名札付き文の書式

b. 複文

複文は、【図C.8】に示す書式で記述します。

{ 宣言の並び$_{opt}$　文の並び$_{opt}$ }

※optはオプション部であることを示します。

図C.8　複文の書式

c. 式 / 空文

式及び空文は、【図C.9】に示す書式で記述します。

式：
式；
空文：
；

図C.9　式 / 空文の書式

d. 選択文

選択文は、【図C.10】に示す書式で記述します。

if(式)文
if(式)文 else 文
switch(式)文

図C.10　選択文の書式

e. 繰り返し文

繰り返し文は、【図C.11】に示す書式で記述します。

while(式)文；
do 文 while (式)；
for(式$_{opt}$；式$_{opt}$；式$_{opt}$)文；

※optはオプション部であることを示します。

図C.11　繰り返し文の書式

f. 分岐文

分岐文は、【図C.12】に示す書式で記述します。

```
goto 識別子；
continue;
break;
return 式opt;

※optはオプション部であることを示します。
```

図C.12　分岐文の書式

g. アセンブリ言語記述文

アセンブリ言語記述文は、【図C.13】に示す書式で記述します。

```
asm( "文字列" );

文字列　　：　　アセンブリ言語ステートメント
```

図C.13　アセンブリ言語記述文の書式

C.3　プリプロセスコマンド

プリプロセスコマンドは、#で始まるコマンドでプリプロセッサcpp308で処理されます。プリプロセスコマンドの仕様を説明します。

C.3.1　プリプロセスコマンドの機能別一覧

本コンパイラは、【表C.13】に示すプリプロセスコマンドを用意しています。

表C.13　プリプロセスコマンド一覧表

コマンド名	機　能
#define	マクロの定義を行います。
#undef	マクロを未定義にします。
#include	指定したファイルの取り込みを行います。
#error	メッセージを標準出力に出力し処理を中断します。
#line	ファイルの行番号を指定します。
#assert	定数式が偽のときに警告を出力します。
#pragma	NC308の拡張機能の処理を指示します。
#if	条件コンパイルを行います。
#ifdef	条件コンパイルを行います。
#ifndef	条件コンパイルを行います。
#elif	条件コンパイルを行います。
#else	条件コンパイルを行います。
#endif	条件コンパイルを行います。

C.3.2　プリプロセスコマンドリファレンス

以降に本コンパイラのプリプロセスコマンドの詳細仕様を説明します。

#define

[機　能]　マクロの定義を行います。

[書　式]　①#define△識別子△字句列
②#define△識別子(識別子の並び)△字句列

[解　説]　①識別子をマクロとして定義します。
②識別子をマクロとして定義します。この書式では、最初の識別子と左括弧'('との間にスペース及びタブを入れないでください。

●下記の記述をした場合、識別子は空白文字に置換されます。

```
#define SYMBOL
```

●マクロで関数を定義した場合、定義文中に逆スラッシュ又は' \ 'を挿入することにより複数行にわたる記述が可能になります。
●以下の4つの識別子はコンパイラの予約語です。

```
__FILE__ ..... ソースファイルの名前
__LINE__ ..... 現在のソースファイルの行番号
__DATE__ ..... コンパイルの日付(形式:mm dd yyyy)
__TIME__ ..... コンパイルの時間(形式:hh:mm:ss)
```

また、NC308では予め以下のマクロ(プリデファインドマクロ)が定義されています。

```
M16C80 （ -M82 オプション使用時は、M32C80 が、代わりに定義されます）
NC308
```

●字句列に対して、文字列化演算子'#'及び字句結合演算子'##'を下記のように使用できます。

```
#define  debug(s,t)  printf("x"#s" = %d x"#t" = %d",x ## s,x ## t)
この識別子に引数をdebug(1, 2)と与えたときは以下のように解釈されます。

#define  debug(s,t)  printf("x1 = %d x2 = %d", x1,x2)
```

#define

●マクロの定義は下記のようにネスト(入れ子)することができます。ネストレベルは最大20です。

```
#define XYZ1      100
#define XYZ2       XYZ1
      :
   (省略)
      :
#define XYZ20     XYZ19
```

#undef

[機　能]　マクロを未定義にします。

[書　式]　#undef△識別子

[解　説]　●マクロとして定義された識別子を無効にします。

●以下の4つの識別子はコンパイラの予約語です。これらの識別子は常に有効にしておく必要がありますので、絶対に#undefで無効にしないでください。

```
__FILE__ ..... ソースファイルの名前
__LINE__ ..... 現在のソースファイルの行番号
__DATE__ ..... コンパイルの日付(形式:mm dd yyyy)
__TIME__ ..... コンパイルの時間(形式:hh:mm:ss)
```

#include

[機　能]　指定したファイルの取り込みを行います。

[書　式]　①#include△<ファイル名>
②#include△"ファイル名"
③#include△識別子

[解　説]　①nc308の起動オプション-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INC308により設定された標準ディレクトリ
②カレント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順番に検索します。
1. 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INC308により設定された標準ディレクトリ
③マクロ展開された識別子が、<ファイル名>又は"ファイル名"であるとき、そのファイルを①又は②の検索規則に準じてディレクトリから取り込みます。

●ネストレベルは最大40です。
●該当するファイルが存在しないときはインクルードエラーとなります。

#error

[機　能]　メッセージを標準出力に出力し処理を中断します。

[書　式]　#error△文字列

[解　説]　●コンパイルを中断します。
●字句列がある場合、その文字列を標準出力に出力します。

#line

[機　能]　ファイル中の行番号を付け換えます。

[書　式]　#line△整数△"ファイル名"

[解　説]　●ファイルの行番号及びファイル名を設定します。
●ソースファイル名及び行番号を変更することができます。

#assert

[機　能]　定数式が偽のときに警告を出力します。

[書　式]　#assert△定数式

[解　説]　●定数式の結果が0(ゼロ)の場合、以下の警告を発します。ただし、コンパイルはそのまま続行されます。

```
[Warning(cpp308.82):x.c, line xx]assertion warning
```

#pragma

[機　能]　NC308の拡張機能の処理を指示します。

[書　式]　[1]　#pragma ROM△変数名
[2]　#pragma SBDATA△変数名
[3]　#pragma SB16DATA△変数名
[4]　#pragma SECTION△既定セクションベース名△変更セクションベース名
[5]　#pragma STRUCT△構造体のタグ名△unpack
[5]　#pragma STRUCT△構造体のタグ名△arrange
[6]　#pragma ADDRESS△変数名△絶対アドレス
[6]　#pragma EQU△変数名 = 絶対アドレス
[7]　#pragma BITADDRESS△関数名△ビット位置,絶対アドレス
[8]　#pragma DMAC△変数名△DMACレジスタ名
[9]　#pragma INTCALL△[/C]△INT番号△アセンブラ関数名(レジスタ名,レジスタ名, ...)
[9]　#pragma INTCALL△[/C]△INT番号△C言語関数名( )
[10] #pragma INTERRUPT△[/B ¦/E¦/F△割り込みベクタ番号]△割り込み処理関数名
[10] #pragma INTF△割り込み処理関数名
[11] #pragma PARAMETER△[/C]△アセンブラ関数名(レジスタ名, レジスタ名, ..)
[12] #pragma SPECIAL△[/C]△呼び出し番号△関数名( )
[13] #pragma ALMHANDLER△アラームハンドラ関数名
[14] #pragma CYCHANDLER△周期起動ハンドラ関数名
[15] #pragma INTHANDLER△割り込みハンドラ関数名
[15] #pragma HANDLER△割り込みハンドラ関数名
[16] #pragma TASK△タスクの開始関数名
[17] #pragma ASM
[17] #pragma ENDASM
[18] #pragma JSRA△ 関数名
[19] #pragma JSRW△関数名
[20] #pragma PAGE
[21] #pragma __ASMMACRO△関数名(レジスタ名)

[解　説]　[1]　romセクションへの配置機能
[2]　SB相対アドレッシング使用変数記述機能
[3]　SB相対16ビットディスプレースメントアドレッシング使用変数宣言機能
[4]　セクションベース名変更機能
[5]　構造体配列制御機能
[6]　入出力変数の絶対アドレス指定機能
[7]　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
[8]　外部変数のDMACレジスタ割り付け機能
[9]　ソフトウェア割り込み使用関数宣言機能
[10] 割り込み関数の記述機能
[11] レジスタ渡しのアセンブラ関数宣言機能
[12] スペシャルページ分岐関数宣言機能
[13] アラームハンドラ関数の記述機能
[14] 周期起動ハンドラ関数の記述機能
[15] 割り込みハンドラ関数の記述機能
[16] タスク開始関数の記述機能
[17] インラインアセンブラ記述機能
[18] 関数呼び出し命令指定機能
[19] 関数呼び出し命令指定機能
[20] 改ページ指定機能
[21] アセンブラマクロ関数宣言機能

次ページへ　→

●#pragmaでは上記21種類の処理機能のみを指定することができます。
#pragmaに続けて上記以外の文字列、識別子を記述した場合、その処理指定は無視されます。

●サポートしていない#pragmaを使用した場合、デフォルトでは警告を出力しません。nc308の起動オプション-Wunknown_pragma(-WUP)を指定したときのみ警告を出力します。

#if～#elif～#else～#endif

[機　能]　条件コンパイルを行います(式が真であるか否かを調べます)。

[書　式]　#if△定数式
　　：
#elif△定数式
　　：
#else
　　：
#endif

[解　説]　●#ifと#elifは、定数の値が真(0でない)の場合、後に続くプログラムを処理します。
●#elifは#if、#ifdef、#ifndefと対で使用します。
●#elseは#ifと対で使用します。#elseと改行の間に字句列を記述してはいけません。ただし、コメントを記述することはできます。
●#endifは#ifで制御する範囲の終了を示します。#ifコマンドを使用する場合には必ずこのコマンドを記述してください 。
●#if～#elif～#else～#endifの組み合せはネストすることができます。ネストレベルは特に制限はありません(ただし、メモリ容量に依存します)。
●定数式の中にsizeof演算子、cast演算子、変数を使用することはできません。

#ifdef～#elif～#else～#endif

[機　能]　条件コンパイルを行います(マクロが定義されているか否かを調べます)。

[書　式]　#ifdef△識別子
　　：
#elif△定数式
　　：
#else
　　：
#endif

[解　説]　●#ifdefは、識別子が定義されている場合、後に続くプログラムを処理します。また以下のように記述することもできます。

```
#if△defined△識別子
#if△defined△(識別子)
```

●#elseは#ifdefと対で使用します。#elseと改行の間に字句列を記述してはいけません。ただし、コメントを記述することはできます。
●#elifは#if、#ifdef、#ifndefと対で使用します。
●#endifは#ifdefで制御する範囲の終了を示します。#ifdefコマンドを使用する場合には必ずこのコマンドを記述してください 。
●#ifdef～#else～#endifの組み合せはネストすることができます。ネストレベルは特に制限はありません(ただし、メモリ容量に依存します)。

#ifndef～#elif～#else～#endif

[機　能]　条件コンパイルを行います(マクロが定義されているか否かを調べます)。

[書　式]　#ifndef△識別子
　　:
#elif△定数式
　　:
#else
　　:
#endif

[解　説]　●#ifndefは、識別子が定義されていない場合、後に続くプログラムを処理します。また以下のように記述することもできます。

```
#if△!defined△識別子
#if△!defined△(識別子)
```

●#elseは#ifndefと対で使用します。#elseと改行の間に字句列を記述することはできません。ただし、コメントを記述することはできます。
●#elifは#if、#ifdef、#ifndefと対で使用します。
●#endifは#ifndefで制御する範囲の終了を示します。#ifndefコマンドを使用する場合には必ずこのコマンドを記述してください 。
●#ifndef～#else～#endifの組み合せはネストすることができます。ネストレベルは特に制限はありません(ただし、メモリ容量に依存します)。

C.3.3　プリデファインドマクロ

NC308では、予め以下のマクロが定義されています。

- M16C80（ -M82 オプション使用時は、M32C80 が、代わりに定義されます）
- NC308

C.3.4　プリデファインドマクロの使用方法

プリデファインドマクロを使用して、NC308以外のC言語プログラム中でマシン依存部をプリプロセスコマンドを使用して切り換えるとき等に使用します。

```
#ifdef  NC308
#pragma ADDRESS       port0     2H
#pragma ADDRESS       port1     3H

#else
#pragma AD             portA = 0x5F
#pragma AD             portA = 0x60

#endif
```

図C.14　プリデファインドマクロの使用例

付録D
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　データの内部表現

D.1.1　整数型

【表D.1】に整数型のデータが使用するバイト数を示します。

表D.1　整数型のデータサイズ

型　名	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	なし	8	0～255
unsigned char			
signed char	有り	8	-128～127
int	有り	16	-32768～32767
short			
signed int			
signed short			
unsigned int	なし	16	0～65535
unsigned short			
long	有り	32	-2147483648～2147483647
signed long			
unsigned long	なし	32	0～4294967295
long long	有り	64	-9223372036854775808 ～ 9223372036854775807
signed long long			
unsigned long long	なし	64	18446744073709551615
float	有り	32	1.17549435e-38F～3.40282347e+38F
double	有り	64	2.2250738585072014e-308～ 1.7976931348623157e+308
long double			
nearポインタ	なし	16	0～0xFFFF
farポインタ	なし	32	0～0xFFFFFFFF

●_Bool型は符号指定は、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して解釈します。
●long long型のビットフィールドは、使用できません。

D.1.2　浮動小数点型

【表D.2】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。

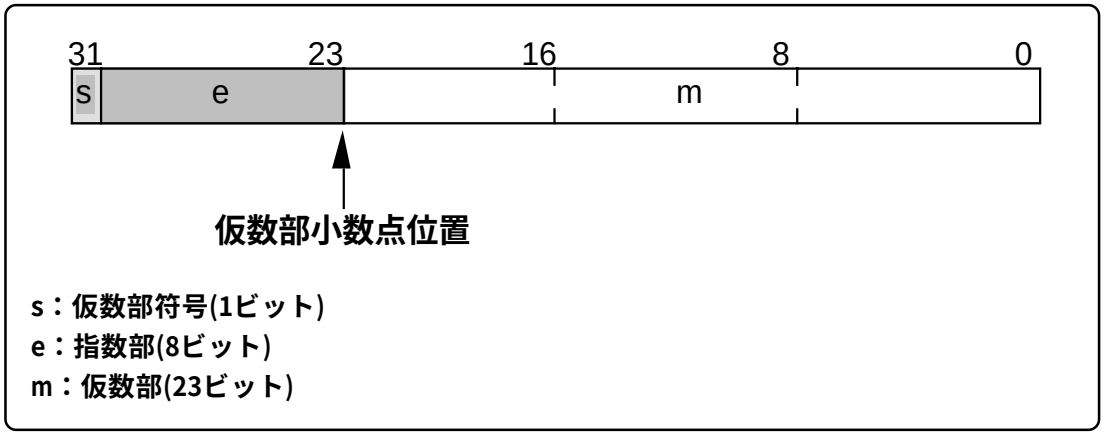
表D.2　浮動小数点型のデータサイズ

型　名	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	有り	64	2.2250738585072014e-308～
long double			1.7976931348623157e+308

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

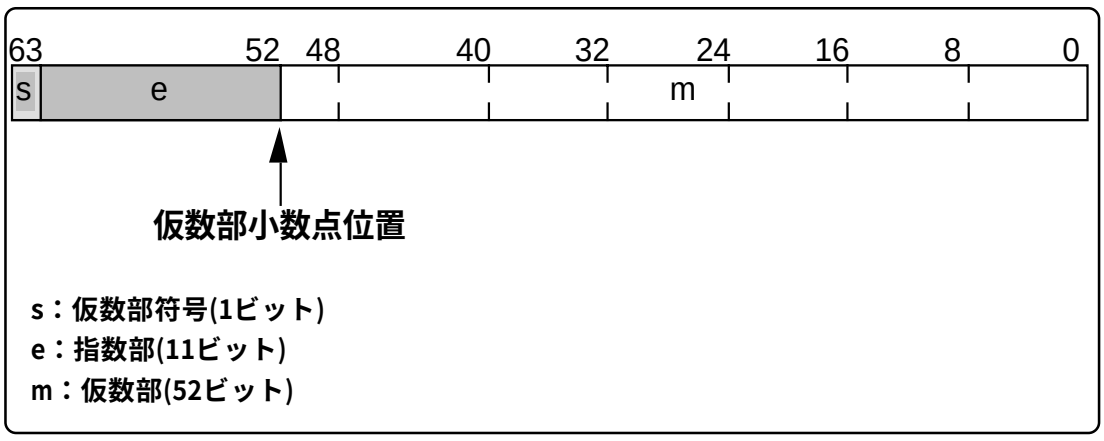
【図D.1】に示すデータ形式で2進数の浮動小数点(float)データを表現します。



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.1.3　列挙型

列挙型は、unsigned 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。

また、本コンパイラの起動オプション-fchar_enumerator(-fCE)を使用することにより列挙型をunsigned char型と同じ内部表現にできます。

D.1.4　ポインタ型

【表D.3】にポインタ型のデータが使用するバイト数を示します。

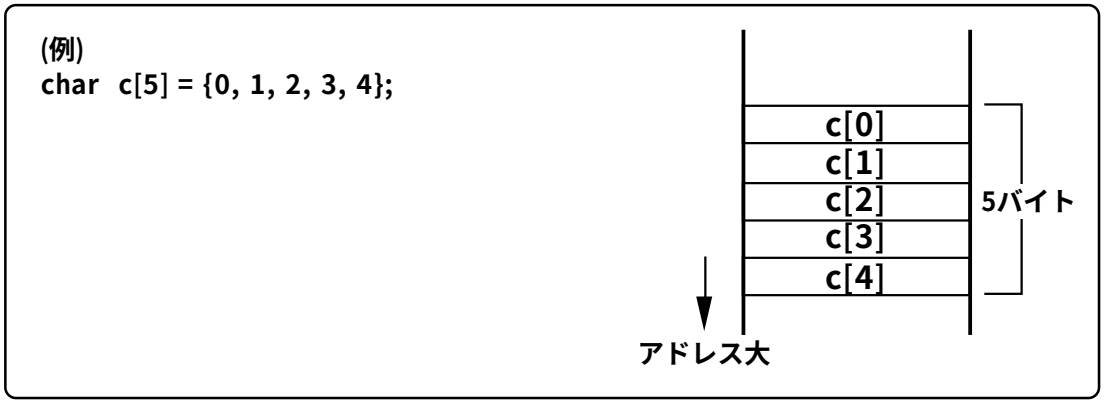
表D.3　ポインタ型のデータサイズ

型　名	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

なお、farポインタは、32ビットの内、下位の24ビットを有効ビットとして使用します。

D.1.5　配列型

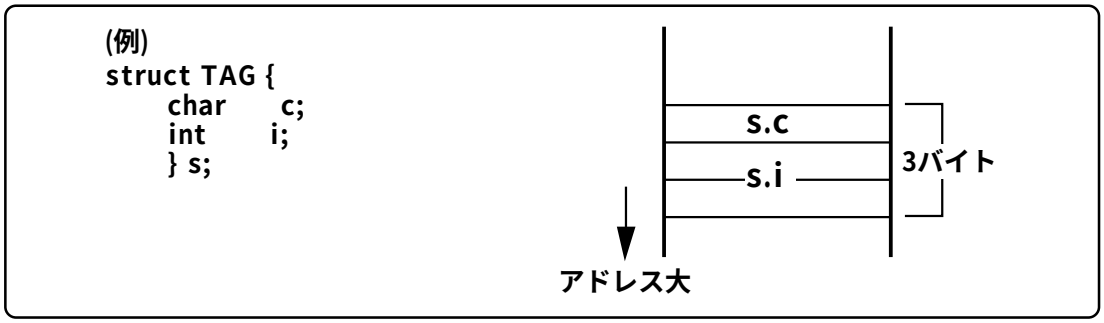
配列型は、要素のサイズ(バイト数)と要素数との積で表す領域に連続して配置されます。要素の出現順にメモリに配置されます。【図16.3】に配置例を示します。



図D.3　配列の配置例

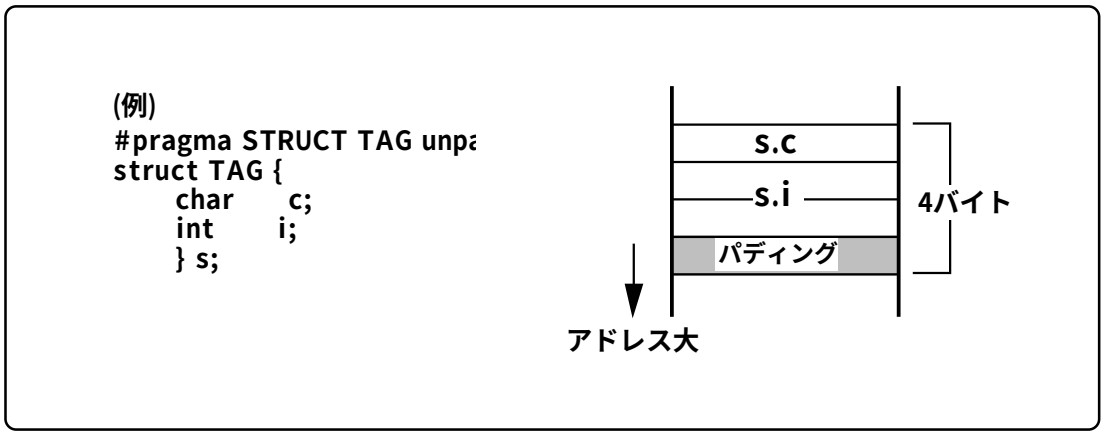
D.1.6　構造体型

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)

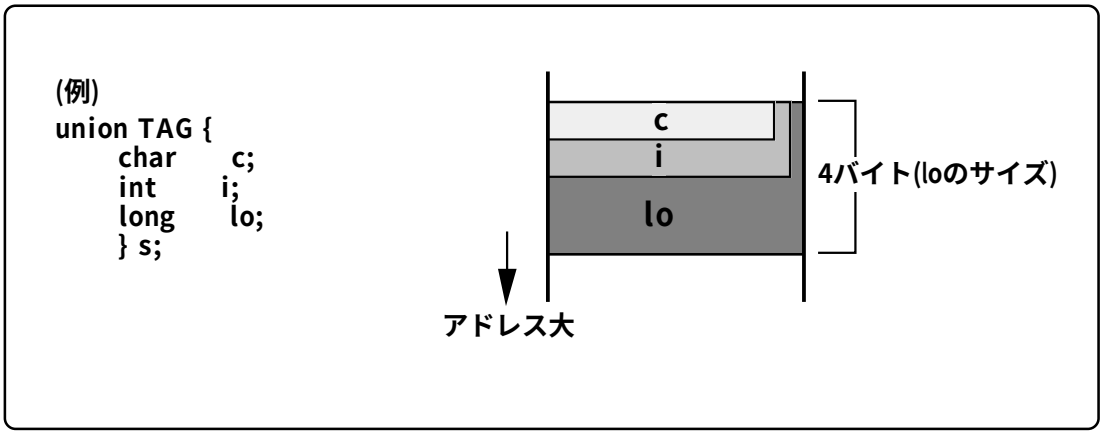
構造体は通常の場合、ワードアライメントを行いません。複数の構造体のメンバは連続して配置されます。ワードアライメントを行う場合は、拡張機能の#pragma STRUCTを使用します。#pragma STRUCTで宣言することにより、メンバのサイズの合計が奇数バイトであるときに1バイトのパディングを付加します。【図D.5】に配置例を示します。



図D.5　構造体の配置例(2)

D.1.7　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。

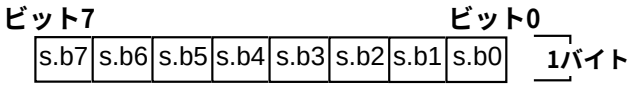


図D.6　共用体の配置例

D.1.8　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。

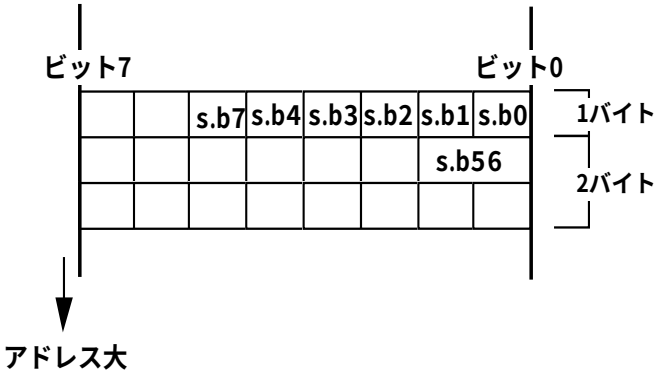
```
(例)
struct BTAG {
     char  b0 : 1;
     char  b1 : 1;
     char  b2 : 1;
     char  b3 : 1;
     char  b4 : 1;
     char  b5 : 1;
     char  b6 : 1;
     char  b7 : 1;
     } s;
```



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。このような場合、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：

- 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。
- long long 型のビットフィールドは、宣言できません。

D.2　符号拡張規則

ANSI規格等で定められた標準のC言語仕様では、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。この仕様は、【図D.9】に示すようなchar型の演算を行うときに、演算の途中でchar型で表現できる最大値をオーバフローして結果が予期しない値になることを防ぐためです。

```
func()
{
    char  c1, c2, c3;

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

図D.9　C言語のサンプルプログラム例

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型をint型に符号拡張を行いません。この仕様は、コンパイルドライバの起動オプション-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言語と同様の符号拡張を行います。

-fansi又は-fextend_to_int(-fETI)オプションを使用しないで【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ときは、char型で表現できる最小値及び最大値※1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。

※1.本コンパイラでは、char型で表現できる値の範囲は以下のとおりです。

- unsigned　char型 ..... 0～255
- signed　char型 ..... -128～127

D.3　関数呼び出し規則

D.3.1　戻り値に関する規則

関数から戻り値を返す場合、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します。【表D.4】に戻り値に関する呼び出し規則を示します。

表D.4　戻り値に関する呼び出し規則

戻り値の型	規　則
char型 _Bool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	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。

D.3.2　引き数渡しに関する規則

本コンパイラでは、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通りの方法を使用します。

(1)引き数のレジスタ渡し

以下に示す条件を満たす場合、【表 D.5】、【表 D.6】中で対応する「使用するレジスタ」を用いて引き数を渡します。

●関数のプロトタイプ宣言※1を行ない、関数呼びだし時に引数の型が確定している。
●プロトタイプ宣言に可変引数"..."を使用していない。
●関数の引数の型として、【表 D.5】、【表 D.6】の引数と引数の型が一致している。

表 D.5 レジスタ引数渡しの規則 (NC308)

引数	引数の型	使用するレジスタ
第1引数	char型，_Bool型	R0Lレジスタ
	int型 nearポインタ型	R0レジスタ

表 D.6 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	_Bool型 char型	R1Lレジスタ
	int型 nearポインタ型	R1レジスタ
第2引数	int型 nearポインタ型	R2レジスタ

(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。
引き数の渡し方をまとめると、【表 D.7】【表 D.8】の様になります

表D.7　関数の引き数渡しの規則 (NC308)

引数の型	第1引数	第2引数	第3引数以降
char型，_Bool型	R0Lレジスタ	スタック	スタック
int型 nearポインタ型	R0レジスタ	スタック	スタック
その他の型	スタック	スタック	スタック

表D.8　関数の引き数渡しの規則 (NC30)

引数の型	第1引数	第2引数	第3引数以降
_Bool型 char型	R1Lレジスタ	スタック	スタック
int型 nearポインタ型	R1レジスタ	R2レジスタ	スタック
その他の型	スタック	スタック	スタック

※1.本コンパイラでは、プロトタイプ宣言を行なった時(新しい形式の記述の時)のみ、レジスタ渡しを適応します。つまり、K&R形式の記述(旧形式の記述)を行なった場合には、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、関数に対してプロトタイプ宣言を行なう記述形式(新しい形式)とK&R形式(旧形式)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。
本コンパイラでは、上記の理由によって、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。

D.3.3　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_(アンダーバー)あるいは$(ダラー)を付加したもの、もしくは、関数名それ自身になります。付加文字列と文字列が付加される条件を【表 D.9】に示します。

表 D.9 関数に文字列の付加される条件

付加文字列	条件
$(ダラー)	一つでも引数がレジスタ渡しとなる関数
_(アンダーバー)	上記条件以外の関数※1

【図 D.10】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                                                 ④
main(void)
{
    int sum;
    sum = func_proto(1,2,3);    ←⑤
    sum = func_no_proto(1,2,3);←⑥
}
```

①関数func_protoのプロトタイプ宣言です。
②関数func_protoの実体です。(プロトタイプ宣言を行なっています。(新しい形式))
③関数func_no_protoの実体です。(K&R形式(旧形式)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.10 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.11】に、関数func_no_protoの定義(③の部分)を【図D.12】に、関数func_protoと関数func_no_protoの呼び出し(④の部分)を【図D.13】に示します。

※1.ただし、#pragma INTCALLで指定した関数は関数名を出力しません。

```
;## #    FUNCTION func_proto
;## #    FRAME    AUTO    (       i)    size  2,    offset -2
;## #  FRAME   ARG (       j)  size   2,    offset 8        ←⑧
;## #  FRAME   ARG (       k)  size   2,    offset 10       ←⑦
;## #  REGISTER ARG    (       i)   size   2,   REGISTER R0  ←⑨
;## #    ARG Size(4)    Auto Size(2)    Context Size(8)


    .SECTION    program,CODE,ALIGN
    ._file    'sample.c'
    .align
    ._line    4
;## # C_SRC :    {
    .glb    $func_proto
$func_proto:                ←⑩
    enter    #02H
    mov.w    R0,-2[FB]    ;  i  i
    ._line    5
;## # C_SRC :        return i + j + k;
    mov.w    -2[FB],R0    ;  i
    add.w    8[FB],R0    ;  j
    add.w    10[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.11 サンプルプログラム(sample.c)のコンパイル結果(1)

図D.10 のサンプルプログラム(sample.c)のコンパイル結果(1)では、関数func_protoは、プロトタイプ宣言を行なっているので第1引数をレジスタ渡しにしています。第2、3引数はレジスタ渡しの対象とはならないので、スタック渡しとなっています。

また、関数の引数がレジスタ渡しとなっているため、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$(ダラー)を付加して"$func_proto"としています。

```
;## #    FUNCTION func_no_proto                          ⑪
;## #  FRAME   ARG (       i)  size   2,    offset 8
;## #  FRAME   ARG (       j)  size   2,    offset 10
;## #  FRAME   ARG (       k)  size   2,    offset 12
;## #    ARG Size(6)    Auto Size(0)    Context Size(8)

    .align
    ._line    11
;## # C_SRC :    {
    .glb    _func_no_proto    ←⑫
_func_no_proto:
    enter    #00H
    ._line    12
;## # C_SRC :        return i + j + k;
    mov.w    8[FB],R0    ;  i
    add.w    10[FB],R0    ;  j
    add.w    12[FB],R0    ;  k
    exitd
E2:
```

⑪すべての引き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.12 サンプルプログラム(sample.c)のコンパイル結果(2)

図D.10 のサンプルプログラム(sample.c)のコンパイル結果(2)では、関数func_no_protoは、K&R形式の記述を行なっているのですべての引数をスタック渡しにしています。

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_(アンダーバー)を付加して"_func_no_proto"としています。

```
;## #    FUNCTION main
;## #    FRAME    AUTO    (     sum)    size  2,    offset -2
;## #    ARG Size(0)    Auto Size(2)    Context Size(8)

    .align
    ._line    16
;## # C_SRC :    {
    .glb    _main
_main:
    enter    #02H
    ._line    18
;## # C_SRC :        sum = func_proto(1,2,3);          ⑪
    push.w    #0003H
    push.w    #0002H
    mov.w    #0001H,R0
    jsr    $func_proto
    add.l    #04H,SP
    mov.w    R0,-2[FB]    ; sum
    ._line    19
;## # C_SRC :        sum = func_no_proto(1,2,3);       ⑫
    push.w    #0003H
    push.w    #0002H
    push.w    #0001H
    jsr    _func_no_proto
    add.l    #06H,SP
    mov.w    R0,-2[FB]    ; sum
    ._line    20
;## # C_SRC :    }
    exitd
E3:
    .END
```

図D.13 サンプルプログラム(sample.c)のコンパイル結果(3)

【図D.13】において、⑪の部分はfunc_protoの呼び出しを、⑪の部分はfunc_no_protoの呼び出しをおこなっています。

D.3.4　関数間のインターフェース

【図D.14】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.15】は、【図D.14】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 = x + y + z ;

        return  sum;  ←mainへの戻り値
}
```

図D.14　C言語サンプルプログラム

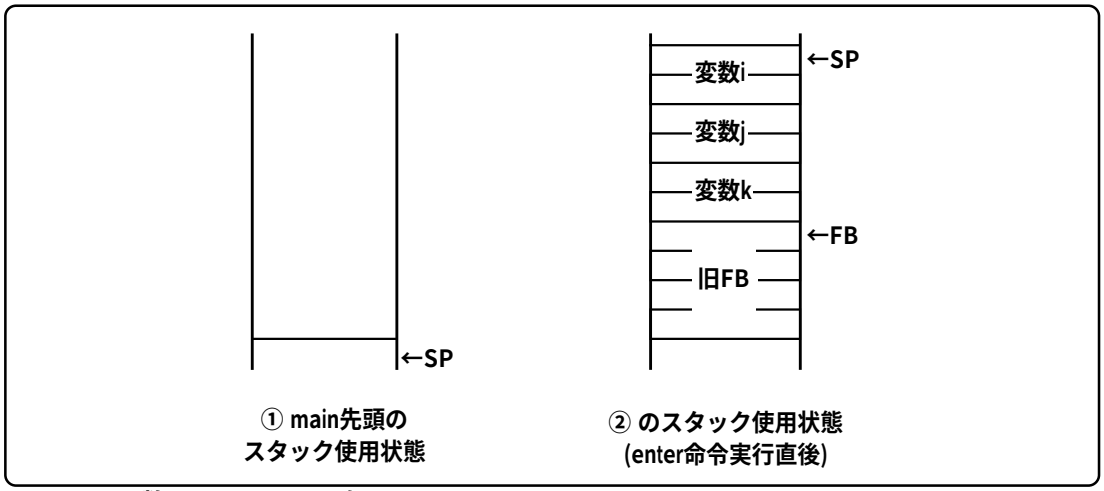
```
;## #    FUNCTION main
;## #    FRAME    AUTO    (       i)    size  2,    offset -6
;## #    FRAME    AUTO    (       j)    size  2,    offset -4
;## #    FRAME    AUTO    (       k)    size  2,    offset -2
;## #    ARG Size(0)    Auto Size(6)    Context Size(8)


    .SECTION    program,CODE,ALIGN
    ._file    'proto2.c'
    .align
    ._line    3
;## # C_SRC :    {
    .glb    _main
_main:                                  ←①
    enter    #06H                       ←②
    ._line    4
;## # C_SRC :         int i = 0x1234;
    mov.w    #1234H,-6[FB]     ;  i
    ._line    5
;## # C_SRC :         int j = 0x5678;
    mov.w    #5678H,-4[FB]     ;  j
    ._line    6
;## # C_SRC :         int k = 0x9abc;
    mov.w    #9abcH,-2[FB]     ;  k
    ._line    7
;## # C_SRC :         k = func(i,j,k);
    push.w    -2[FB]     ;  k           ←③
    push.w    -4[FB]     ;  j           ←④
    mov.w    -6[FB],R0     ;  i         ←⑤
    jsr    $func                        ←⑥
    add.l    #04H,SP                    ←⑩
    mov.w    R0,-2[FB]     ;  k         ←⑪
    ._line    8
;## # C_SRC :    }
    exitd
E1:
```

図D.15 アセンブリ言語サンプルプログラム(1)

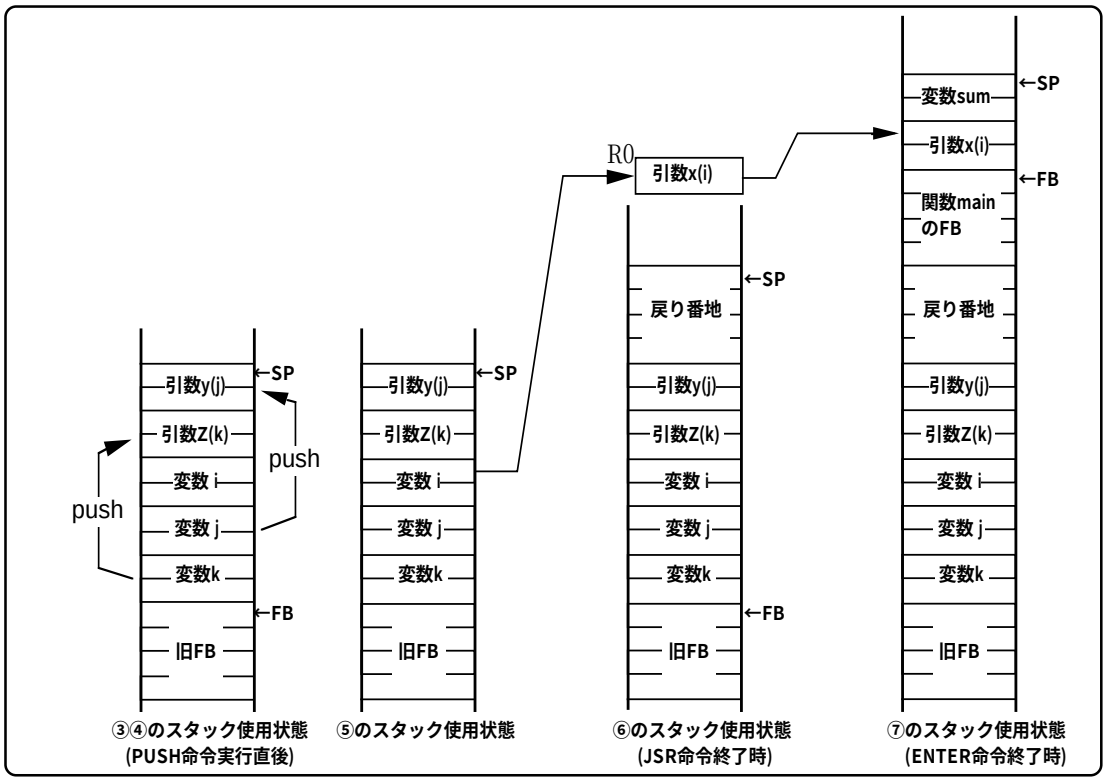
```
;## #    FUNCTION func
;## #    FRAME    AUTO    (       x)    size  2,    offset -2
;## #    FRAME    AUTO    (     sum)    size  2,    offset -2
;## #  FRAME   ARG (       y)  size   2,    offset 8
;## #  FRAME   ARG (       z)  size   2,    offset 10
;## #  REGISTER ARG    (       x)   size   2,   REGISTER R0
;## #    ARG Size(4)    Auto Size(2)    Context Size(8)

    .align
    ._line    11
;## # C_SRC :    {
    .glb    $func
$func:
    enter    #02H                ←⑦
    mov.w    R0,-2[FB]    ;  x  x
    ._line    13
;## # C_SRC :         sum = x + y + z;
    mov.w    -2[FB],R0    ;  x
    add.w    8[FB],R0    ;  y
    add.w    10[FB],R0    ;  z
    mov.w    R0,-2[FB]    ;  sum
    ._line    15
;## # C_SRC :         return sum;
    mov.w    -2[FB],R0    ;  sum ←⑧
    exitd                        ←⑨
E2:
    .END
```

図D.16 アセンブリ言語サンプルプログラム(2)

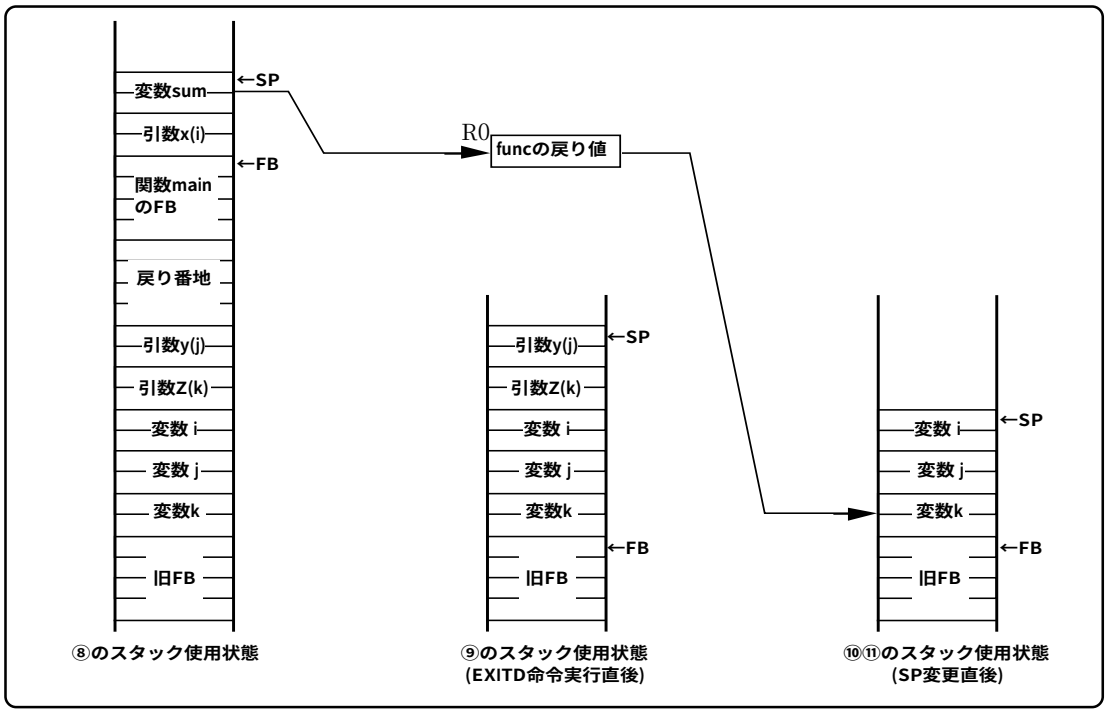
【図D.15】中の①→②の処理(関数mainの入り口処理)を【図D.17】に、③→④→⑤→⑥→⑦の処理(関数funcの呼び出し及び関数funcで使用するスタックフレームの構築処理)を【図D.18】に、⑧→⑨→⑩→⑪の処理(関数funcから関数mainへの戻り処理)を【図D.19】に、各々のスタック及びレジスタの遷移を示します。



図D.17　関数 mainの入り口処理



図D.18　関数funcの呼び出し及び、入り口処理



図D.19　関数funcの出口処理

D.4　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.24】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。

```
func()
{
    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.24　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.24】をコンパイルして生成されたアセンブリ言語ソースファイルを【図D.25】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①②③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.25　アセンブリ言語ソースプログラム例

D.5　レジスタの退避

C言語の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

1. R0 レジスタ
2. 呼び出す関数の戻り値に使用する、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

R0 および、戻り値に使用するレジスタ以外で、関数内で使用されるレジスタ。

付録E
標準ライブラリ

E.1　標準ヘッダファイル

　標準ライブラリを使用する場合、その関数の定義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】に示すように15個の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
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	FILE構造体の定義
	ストリーム名の定義
	入出力関数のプロトタイプ宣言
stdlib.h	メモリ管理関数、終了関数のプロトタイプ宣言
string.h	文字列操作関数、メモリ操作関数のプロトタイプ宣言
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

　以降に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)

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。プロトタイプ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

[機　能]　数学関数のプロトタイプを宣言しています。プロトタイプ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
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

setjmp.h

[機　能]　分岐関数のプロトタイプ宣言と、その関数で使用する構造体を定義しています。プロトタイプ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ルーチンを定義しています。

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、入出力関数のプロトタイプを宣言しています。プロトタイプ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	init	マイコンの入出力の初期化
	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、終了関数のプロトタイプを宣言しています。プロトタイプ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プロトタイプを宣言しています。プロトタイプ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119個の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11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、監視タイマに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.1】に示します。

表E.1　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	文字列を変換(ロケール情報を使用)	○

※標準関数の幾つかは、その関数専用の大域変数を使用しています。関数が呼び出されて実行している最中に割り込みが入り、割り込み処理プログラム中で同じ関数が呼び出された場合、最初に呼び出された関数で使用していた大域変数の内容を書き換えることがあります。

リエントラント性がある関数(表中では○)は、このような大域変数の書き換えは発生しません。リエントラント性がない関数(表中では×)を割り込み処理プログラムで使用するときは注意してください。

b.　文字操作関数

文字操作関数の一覧を【表E.2】に示します。

表E.2　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.3】に示します。

表E.3　入出力関数

種別	関数名	機　能	ﾘｴﾝﾄﾗﾝﾄ性
初期化	init	マイコンの入出力の初期化	×
	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.4】に示します。

表E.4　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.5】に示します。

表E.5　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.6】に示します。

表E.6　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の設定	○

g.　数学関数

数学関数の一覧表を【表E.7】に示します。

表E.7　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.8】に示します。

表E.8　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	疑似乱数にシードを与える	○

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.9】に示します。

表E.9　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	○
strtoul	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.10】に示します。

表E.10　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.11】に示します。

表E.11　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]に記述しています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。

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

[変　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

[変　数]　int n;..... 整数

[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

[変　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

[変　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

[変　数]　double x;..... 実数

[戻り値]　●-π/2からπ/2ラジアンの範囲の値を返します。

atan2

数学関数

[機　能]　"x"と"y"の商の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

[変　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

[変　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返還した値を返します。

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

[変　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

[変　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。

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

[変　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]　●戻り値はありません。

[解　説]　●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]　配列内のバイナリサーチを行います。

[書　式]　#include　<stdlib.h>

void _far *bsearch(key, base, nelem, size, cmp);

[実現方法]　関数

[変　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。

[注　意]　●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。

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。

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。

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。

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が示す変数に格納します。
●argumentは各変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。

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。

init

入出力関数

[機　能]　ストリームの初期化を行います。

[書　式]　#include　<stdio.h>

void　init(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]　●ストリームの初期化を行います。また関数内でspeed、init_prnを呼び出して、UART、及びセントロニクス出力装置の初期設定を行います。
●initは通常、スタートアッププログラムから呼び出して使用します。

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; ..... 判定する文字

[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

文字操作関数

[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]　英大文字(A～Z)を判定します。

[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。

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);
struct lconv *localeconv(void);　[NC308のみ]

[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。

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には戻り値としてNULLを返します。

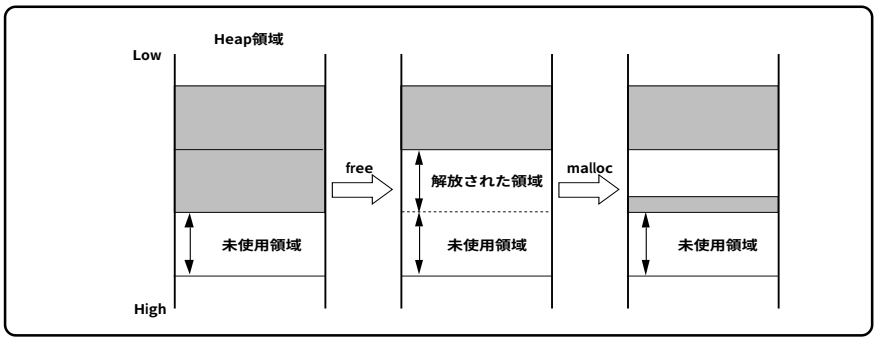
[解　説]　●動的なメモリ領域を割り当てます。

[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

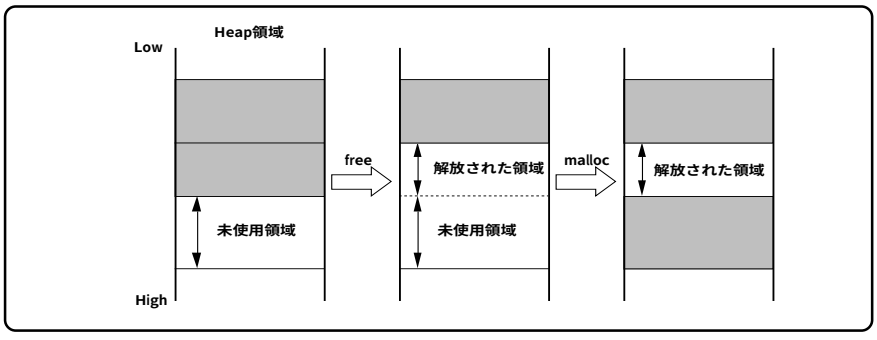
(1-1) 確保するサイズがfreeで解放された領域より小さい場合

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。



(1-2) 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。

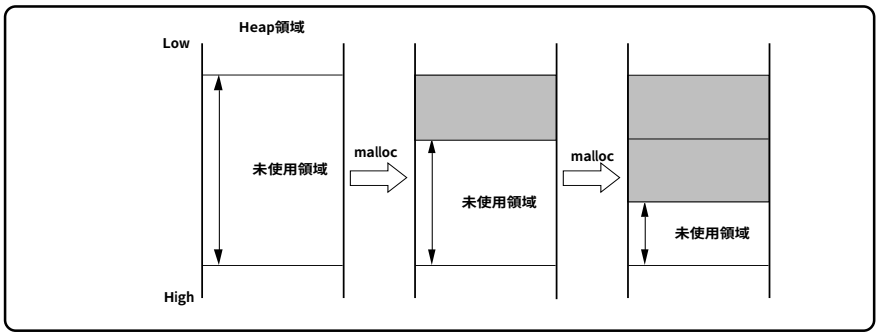


malloc

(2) freeで解放された領域がない場合

(2-1) 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。



(2-2) 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] 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。
●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には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及び-OS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及び-OS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]　マクロ、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]　●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には、#include <string.h> の記述の後に、 #undef memcpy と記述してください。

（例）

```
#include <string.h>
#undef memcpy;  ← #undef memcpy の記述により、ライブラリ関数を有効にする。
static int a = 3;
static int b;

void func(void)
{
    void * result;
    result = memcpy((void *)&a, (const void *)&b, 1);
}
```

●"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及び-OS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及び-OS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。
●オプション-O[3～5],-OR及び-OS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]　マクロ、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]　●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には、#include <string.h> の記述の後に、 #undef memcpy と記述してください。

```
（例）
#include <string.h>
#undef memset;  ← #undef memset の記述により、ライブラリ関数を有効にする。
static int a[10];

void func(void)
{
    char * result;
    result = memset((void *)a, 'a ',10);
}
```

●"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及び-OS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 _far ＊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。

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。書式の各指定については次のページで説明します。

書式： %[フラグ][最小フィールド幅][精度][修飾文字(l、L、又はh)]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。

printf—formatの書式指定

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。※1

●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。

●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。

※1.製品付属の標準ライブラリでは、文字列のポインタはfarポインタになっています。

printf—formatの書式指定

2.フラグ

●－

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。

3.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。

4.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

a. 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。

b. 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。

c. 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。

d. デフォルト値は1です。

e. ゼロを最小桁数0で変換した場合、何も出力しません。

printf—formatの書式指定

●s変換の場合

a. 最大文字数を表します。

b. 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。

c. デフォルトには文字数制限がありません。

d. 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。

e. 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。

5.l、L、又はh

●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。

●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。

●l、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。※1

※1.標準のC言語の仕様では、Lの場合、e、E、f、g変換をlong doubleの引数に対して行います。本コンパイラでは、long doubleはdoubleとして扱いますので、Lを指定した場合doubleの引数として扱います。

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]　マクロ

[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。

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。

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, nbytes);

[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には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"nbytes"で変更する大きさを指定します。

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

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。
書式の各指定については次のページで説明します。
書式： %[＊][最大フィールド幅][修飾子(l、L、又はh)]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。

scanf—formatの書式指定

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。

●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。

scanf—formatの書式指定

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.l、L、又はh

- l の場合、d、i、o、u、x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。
- h の場合、d、i、o、u、xの変換結果をshort int又はunsigned short intとして格納します。
- l 、h を d、i、o、u、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 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]　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

[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。※1
●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及び-OS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"を含まない場合には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及び-OS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。

※1.s1はs1とs2が充分に入る大きさが必要です。

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には、#include <string.h> の記述の後に、 #undef strcmp と記述してください。

（例）

```
#include <string.h>
#undef strcmp;  ← #undef strcmp の記述により、ライブラリ関数を有効にする。
static char *a = “ macro ”;
static char *b = “ function ”;

void func(void)
{
    int result;

    result = strcmp(b,a);
}
```

●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及び-OS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及び-OS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]　マクロ、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]　●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には、#include <string.h> の記述の後に、 #undef strcpy と記述してください。

（例）

```
#include <string.h>
#undef strcpy;  ← #undef strcpy の記述により、ライブラリ関数を有効にする。
static char _far *a = " macro ";

void func(void)
{
    char _far * result;
    char _far b[5];

    result = strcpy(b,a);
}
```

●"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及び-OS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。

[注　意]　●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及び-OS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及び-OS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"文字より短い場合には、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及び-OS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..... 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。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及び-OS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及び-OS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"を含まない場合には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及び-OS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及び-OS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及び-OS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

文字列数値変換関数

[機　能]　文字列を倍精度浮動小数に変換します。

[書　式]　#include　<string.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及び-OS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("s1"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及び-OS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　<string.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の場合に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及び-OS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　<string.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の場合に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及び-OS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及び-OS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。

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]　#include　<ctype.h>

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。

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。

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
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。

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 _far 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

[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]、-OS、-OR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時には、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.12】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.12　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：farポインタ 第2引数：文字列定数	strcpy(str, "sample");
memcpy	第1引数：far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。

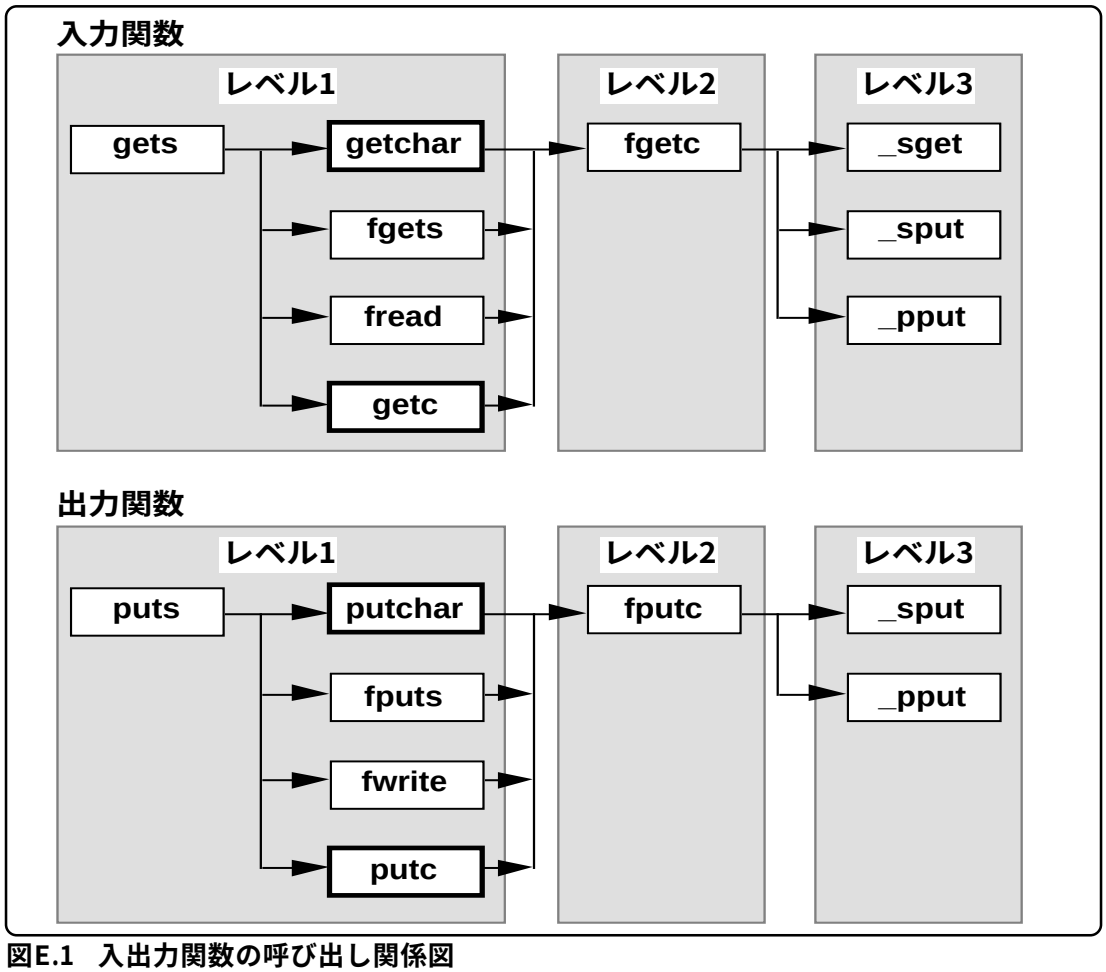
アプリケーションプログラムにおいて、組み込むシステムのハードウェアによって入出力関数を変更する場合があります。このような場合、製品に添付しています標準関数ライブラリのソースファイルを変更することで、対処できます。

エントリー版には、標準関数ライブラリのソースファイルは付属していません。従ってカスタマイズすることはできません。

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。

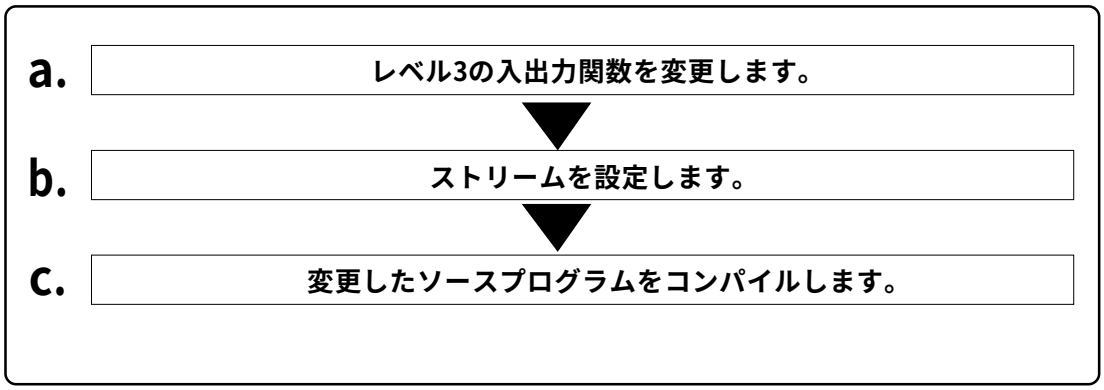
マイコンのハードウェア(入出力ポート)に依存しているのは最下位のレベル3の関数のみです。入出力関数をアプリケーションプログラムで使用する場合、必要に応じてレベル3の関数のソースファイルを書き換えることで、システムに順応した関数に変更できます。



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/80シリーズ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。

[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

[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(init.c)中で設定されています。

レベル3の入出力関数は、C言語で記述されたライブラリのソースファイルdevice.cに記述しています。レベル3の関数の仕様を【表E.13】に示します。

表E.13　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
_sput		
_pput		

出力関数	引数(int型)	戻り値(int型)
_sput	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。
_pput		

シリアル通信は、M16C/80シリーズが持つ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 ..... -DUART0

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。

①device.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 EQU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。
④speed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(init.c)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

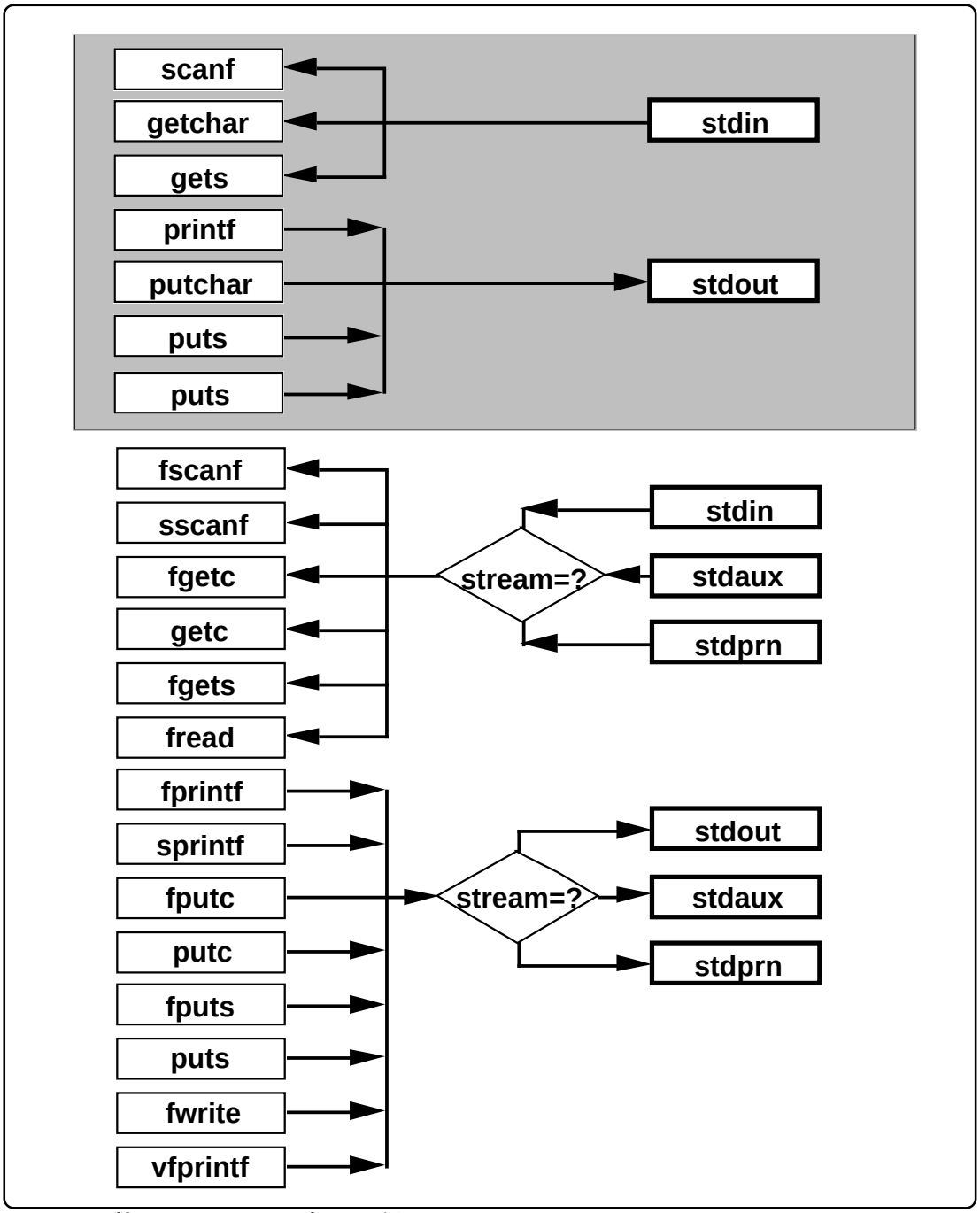
本コンパイラの標準ライブラリはstdin、stdout、stderr、stdaux、stdprnの5種類のストリーム情報を外部構造体として持っています。この外部構造体は標準ヘッダファイルstdio.h内で定義されており、各ストリームのモード情報(入力ストリームか出力ストリームを表すフラグ)やステータス情報(エラー又はEOFを表すフラグ)等を管理しています。

表E.14　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 auxialiary input output */
#define  stdprn  (&_iob[3])         /* Fundamental printer output */

#define  stderr  stdout

/*********************************************************
*********************************************************/
#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】に示しました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。

①char _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。

②int _cnt

_buffのデータ数を格納します(0もしくは1)。

③int _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。

④int _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。

⑤int (＊_func_in)()

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。

⑥int (＊_func_out)()

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。NC308の製品に含まれる標準ライブラリファイルでは、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 = stdout->_cnt = stdaux->_cnt = 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
    speed(_96, _B8, _PN, _S2);
#endif
    init_prn();
}
```

図E.5　init関数のソースファイル(init.c)

M16C/80シリーズの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。

①UART0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。
④speed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)
          :
    speed0(_96, _B8, _PN, _S2);         ←④(UART0用のspeed関数を指定)
    speed(_96, _B8, _PN, _S2);
    init_prn();
}
```

※図中の②～④は、上記の設定手順の項番に対応しています。

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

変更したソースファイルをシステムに組み込む方法として、以下の2通りの方法があります。

①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ます。
②製品に同封の実行手順ファイル(makefile（Windows版（PC版）はmakefile.dos）
を使用してライブラリファイルを更新します。

手順①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。
①の方法を【図E.7】に、②の方法を【図E.8】にそれぞれ示します。

```
% nc308 -c -g -osample  ncrt0.a30 device.r30 init.r30 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

```
% make <RET>
```

図E.8　変更したソースプログラムを基にライブラリを更新する方法

付録F
エラーメッセージ一覧表

NC308が出力するすべてのエラーメッセージとワーニングメッセージ、そのメッセージに対する対処方法を説明します。

F.1　メッセージの出力形式

NC308は、処理中にエラーを検出すると、エラーメッセージを画面に表示した後コンパイルを中止します。
以下にエラーメッセージとワーニングメッセージの出力形式を示します。

```
nc308:[エラーメッセージ]
```

図F.1　コンパイルドライバnc308のエラー出力形式

```
[Error(cpp308.エラー番号):ファイル名,行番号]エラーメッセージ
[Error(ccom):ファイル名,行番号]エラーメッセージ
[Fatal(ccom):ファイル名,行番号]エラーメッセージ          ←※1
```

図F.2　各コマンドのエラー出力形式

```
[Warning(cpp308.ワーニング番号):ファイル名,行番号]ワーニングメッセージ
[Warning(ccom):ファイル名,行番号]ワーニングメッセージ
```

図F.3　各コマンドのワーニング出力形式

次項から各コマンドのエラーメッセージとワーニングメッセージの内容と対処策を示します。cpp308のメッセージは番号順で、その他のコマンドのメッセージはアルファベット順(記号、英字)で掲載しております。

※1. 致命的エラーの場合のメッセージです。
通常、このようなエラーメッセージは出力されませんが、万一発生した場合は表示内容をご連絡ください。

F.2　nc308エラーメッセージ

【表F.1】と【表F.2】にコンパイルドライバnc308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F.1　nc308エラーメッセージ一覧表(1)

エラーメッセージ	エラー内容と対策
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NC308の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、-vオプションで確認することができます。
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。
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。
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。
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。
Ignore option '-?'	●NC308で使用できないオプション-?を使用しています。 ⇒正しいオプション指定してください。
illegal option	●-as308 や-ln308 などの各処理系に指示するオプションが100文字を越えました。 ⇒各処理系に指示するオプションは99文字までにしてください。
Invalid argument	●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。
Invalid option '-?'	●-?オプションに必要なパラメータがありません。 ⇒-?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。
	●-?オプションと必要なパラメータの間にスペースがあります。 ⇒-?オプションとパラメータ間のスペースを削除してください。
Invalid option '-o'	●-oオプションの次に出力ファイル名がありません。 ⇒出力ファイル名を指定してください。ファイル名は拡張子を指定しないでください。
Invalid suffix '.xxx'	●NC308が認識できないファイル拡張子 (.c,.i,.a30,.r30,.x30 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。

表F.2　nc308エラーメッセージ一覧表(2)

エラーメッセージ	エラー内容と対策
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。
Not enough core	[UNIX版] ●スワップ領域が不足しています。 ⇒セカンダリスワップを追加するなど、スワップ領域を増やしてください。 [MS-Windows 版] ●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。
Permission denied	●各処理系が実行できません。 ⇒各処理系のパーミッション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。
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
too many options	●指定されたオプションの数が100を超えています。 ⇒オプションは99文字までにしてください。
Result too large	●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。
Too many open files	●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。

F.3　cpp308エラーメッセージ

【表F.3】～【表F.6】にプリプロセッサcpp308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F.3　cpp308エラーメッセージ一覧表(1)

番号	エラーメッセージ	エラー内容と対策
1	illegal command option	●入力ファイル名を2回指定しています。 ⇒入力ファイル名の指定を1つにしてください。
		●入力ファイル名と出力ファイル名が同じ名前です。 ⇒出力ファイル名は入力ファイル名と違う名前を指定してください。
		●出力ファイル名を2回指定しています。 ⇒出力ファイル名の指定を1つにしてください。
		●コマンドラインが-oオプションで終了しています。 ⇒-oオプションの後に出力ファイル名を指定してください。
		●インクルードファイルのパスを指定する-Iオプションが制限値を越えました。 ⇒-Iオプションを8個以下にしてください。
		●コマンドラインが-Iオプションで終了しています。 ⇒-Iオプションの後に出力ファイル名を指定してください。
		●-Dオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は__)でありません。不当なマクロ名定義を行っています。 ⇒正しいマクロ名、正しいマクロ定義を指定してください。
		●コマンドラインが-Dオプションで終了しています。 ⇒-Dオプションの後に出力ファイル名を指定してください。
		●-Uオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は__)でありません。 ⇒正しいマクロ名定義を指定してください。
		●cpp308で使用できないオプションを指定しています。 ⇒正しいオプションを指定してください。
11	cannot open input file	●入力ファイルが見つかりません。 ⇒正しいファイル名を指定してください。
12	cannot close input file	●入力ファイルをクローズできません。 ⇒入力ファイルを確認してください。
14	cannot open output file.	●出力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。
15	cannot close output file	●出力ファイルをクローズできません。 ⇒ディスクの空き容量を確認してください。

表F.4　cpp308エラーメッセージ一覧表(2)

番号	エラーメッセージ	エラー内容と対策
16	cannot write output file	●ファイルの書き込み中にエラーが発生しました。 ⇒ディスクの空き容量を確認してください。
17	input file name buffer overflow	●入力ファイル名のバッファがオーバーフローしました。ファイル名は、ディレクトリパス名を含みますので注意してください。 ⇒ファイル名、パス名（標準ディレクトリ、-Iオプション指定）を短くしてください。
18	not enough memory for macro identifier	●マクロ名、綴り文字を登録するためのメモリが足りません。 [UNIX版] ⇒セカンダリスワップを追加するなど、スワップ領域を増やしてください。 [MS-Windows 版] ⇒スワップ領域を増やしてください。
21	include file not found	●インクルードファイルがオープンできません。 ⇒インクルードファイルは、カレントディレクトリ、-Iオプションや環境変数で指定したディレクトリに存在します。これらのディレクトリを確認してください。
22	illegal file name error	●ファイル名の指定が誤っています。 ⇒ファイル名の指定を正しく行ってください。
23	include file nesting over	●インクルードファイルのネスティングの上限（40）を越えました。 ⇒インクルードファイルのネスティングを40までにしてください。
25	illegal identifier	●#defineの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
26	illegal operation	●プリプロセスコマンド#if～#elseif～#assertの演算式中に誤りがあります。 ⇒演算式を正しく記述してください。
27	macro argument error	●マクロ展開時のマクロ引数の数に誤りがあります。 ⇒マクロ定義及び参照を確認して正しく記述してください。
28	input buffer over flow	●ソースファイルリード中に入力行バッファがオーバーフローしました。又は、マクロ変換中にバッファがオーバーフローしました。 ⇒ソースファイルの1行を1023文字以下にしてください。マクロ変換を予想される場合は、変換結果が1023文字以下になるように変更してください。
29	EOF in comment	●コメント中にファイルが終了しています。 ⇒ソースファイルを正しく記述してください。

表F.5　cpp308エラーメッセージ一覧表(3)

番号	エラーメッセージ	エラー内容と対策
31	EOF in preprocess command	●プリプロセスコマンド中にファイルが終了しています。 ⇒ソースファイルを正しく記述してください。
32	unknown preprocess command	●不当なプリプロセスコマンドを使用しています。 ⇒cpp308で使用できるプリプロセスコマンドは、次のコマンドのみです。 #include、#define、#undef、#if、#ifdef、#ifndef、#else,#endif、#elseif、#line、#assert、#pragma、#error
33	new_line in string	●文字定数又は、文字列定数中に改行が含まれています。 ⇒プログラムを正しく記述しなおしてください。
34	string literal out of range 509 characters	●文字列が509文字をこえました。 ⇒文字列は509文字以下にてください。
35	macro replace nesting over	●マクロのネスティングが制限値(20)を越えました。 ⇒制限値を越えないようにしてください。
41	include file error	●#include命令の記述に誤りがあります。 ⇒正しく記述してください。
43	illegal id name	●#defineコマンドの以下のマクロ名又は引数の記述に誤りがあります。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒ソースファイルを正しく記述してください。
44	token buffer over flow	●#defineの綴り文字のバッファがオーバーフローしました。 ⇒綴り文字を短くしてください。
45	illegal undef command usage	●#undefの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
46	undef id not found	●#undefで未定義にしようとしている以下のマクロ名が定義されていません。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒マクロ名を確認してください。
52	illegal ifdef / ifndef command usage	●#ifdefの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
53	elseif / else sequence error	●#if～#ifdef～#ifndefがないのに#elseif又は#elseを使用しています。 ⇒#elseif又は#elseは#if～#ifdef～#ifndefの後に使してください。
54	endif not exist	●#if～#ifdef～#ifndefに対応した#endifがありません。 ⇒#endifをソースファイル中に記述してください。
55	endif sequence error	●#if～#ifdef～#ifndefがないのに#endifを使用しています。 ⇒#endifは#if～#ifdef～#ifndefの後に使してください。

表F.6　cpp308エラーメッセージ一覧表(4)

番号	エラーメッセージ	エラー内容と対策
61	illegal line command usage	●#lineの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。

F.4　cpp308ワーニングメッセージ

【表F.7】にプリプロセッサcpp308が出力するワーニングメッセージとその内容及び対処方法を示します。

表F.7　cpp308ワーニングメッセージ一覧表

番号	ワーニングメッセージ	ワーニング内容と対策
81	reserved id used	●cpp308が予約済みの以下のマクロ名を定義又は未定義にしようとしています。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒他のマクロ名を使用してください。
82	assertion warning	●#assertの演算式の結果が0になりました。 ⇒演算式を確認してください。
83	garbage argument	●プリプロセスコマンドの後にコメント以外の文字があります。 ⇒プリプロセスコマンドの後の文字はコメント(/＊．　. . . . ＊/)で記述してください。
84	escape sequence out of range for character	●文字定数、文字列定数に含まれるエスケープ文字が255越えました。 ⇒エスケープ文字は255までにしてください。
85	redefined	●一度定義したマクロを以前定義したときと異なる内容で再度定義しています。 ⇒以前定義した内容と比較して確認してください。
87	/* within comment	●コメント内に/*を記述しています。 ⇒ネストしないようにコメントを記述してください。
88	Environment variable ‘ NCKIN ’ must be“ SJIS ”or“ EUC ”	●環境変数 NCKIN に誤りがあります。 ⇒NCKIN には、’SJIS ”、’EUC ”のいずれかを設定してください。
90	‘ マクロ名 ’ in #if is not defined, so it ’s treated as 0	●#if 文で未定義のマクロが使用されています。 ⇒マクロ定義を確認してください。

F.5　ccom308エラーメッセージ

【表F.8】～【表F.19】にコンパイラccom308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F.8　ccom308エラーメッセージ一覧表(1)

エラーメッセージ	エラー内容と対策
#pragma プラグマ名 関数名 re-defined	●#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。
#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。
#pragma プラグマ名 & function prototype mismatched	●#pragma プラグマ名 で指定した関数とプロトタイプ宣言の引数の内容が異なっています。 ⇒プロトタイプ宣言の引数と合わせてください。
#pragma プラグマ名 function argument is struct or union	●#pragma プラグマ名 で指定した関数のプロトタイプ宣言で struct / union型 を指定しています。 ⇒プロトタイプ宣言で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。
#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ます。 ⇒関数の呼び出しを行う前に宣言してください。
#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。
#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き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には引き数にはレジスタ渡しとなる型を指定してください。
#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。
'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。
'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。
'far' is duplicate	●farを2回以上記述しています。 ⇒farを正しく記述してください。

表F.9　ccom308エラーメッセージ一覧表(2)

エラーメッセージ	エラー内容と対策
'near' is duplicate	●near を2回以上記述しています。 ⇒nearを正しく記述してください。
'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。
'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。
(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。
(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。
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。
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。
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。
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。
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回にしてください。
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。
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。
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。
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。

表F.10　ccom308エラーメッセージ一覧(3)

エラーメッセージ	エラー内容と対策
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。
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。
can't get void value	●代入式の右辺がvoid型等のように、void型のデータを取り出そうとしています。 ⇒データの型を確認してください。
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パーミッションを確認してください。
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。
can't set argument	●プロトタイプ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。
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。
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。
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。
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。
default function argument conflict	●プロトタイプ宣言において、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。
default: is duplicated	●defaultの値を重複して使用しています。 ⇒1つのswitch文で、defaultは1つにしてください。
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。
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。
duplicate frame position defind 変数名	●auto変数が2回以上記述しています。 ⇒正しく記述してください。
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。

表F.11　ccom308エラーメッセージ一覧(4)

エラーメッセージ	エラー内容と対策
float and double not have sign	●floatやdoubleにsigned/unsignedを記述しています。 ⇒型指定子を正しく記述してください。
floating point value overflow	●浮動小数点の即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。
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。
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。
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。
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。
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。
function returning a function declared	●関数の宣言においてリターン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。
function returning an array	●関数の宣言においてリターン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。
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。
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。
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。
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。
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。
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。
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。

表F.12　ccom308エラーメッセージ一覧(5)

エラーメッセージ	エラー内容と対策
incomplete return type	●不完全な型を戻り値に記述しています。 ⇒戻り値を確認してください。
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。
incomplete struct member	●不完全な構造体をメンバとして記述しています。 ⇒完全な構造体を記述してください。
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。
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。
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。
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。
inline function have invalid argument or return code	●インライン関数に、不正な引き数または、不正な戻り値があります。 ⇒正しい、引き数、戻り値を指定してください。
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。
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。
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。
inline function (関数名) is recursion	●インライン関数の再帰呼び出しはできません。 ⇒再帰呼び出しをしない様に記述してください。
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。
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。

表F.13　ccom308エラーメッセージ一覧(6)

エラーメッセージ	エラー内容と対策
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プロトタイプ宣言と、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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。

表F.14　ccom308エラーメッセージ一覧(7)

エラーメッセージ	エラー内容と対策
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
invalid -> used	● ->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。
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。
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。
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。
Invalid #pragma SPECIAL special page number	●#pragma SPECIAL で記述した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。

表F.15　ccom308エラーメッセージ一覧(8)

エラーメッセージ	エラー内容と対策
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。
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。
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。
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。
invalid case 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。
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。
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。
invalid continue statements	●continue文を記述できないところで使用しています。 ⇒while, do-while, forのなかで記述してください。
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。
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。
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。
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。
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。
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。
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。
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。

表F.16　ccom308エラーメッセージ一覧(9)

エラーメッセージ	エラー内容と対策
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。
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。
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。
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。
invalid lvalue	●代入文の左辺が、lvalueではありません。 ⇒左辺式に代入可能な式を記述してください。
invalid lvalue at '=' operator	●代入文の左辺が、lvalueではありません。 ⇒左辺式に代入可能な式を記述してください。
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。
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。
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。
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。
invalid sign specifier	● signed/unsignedを2回以上記述しています。 ⇒型指定子を正しく記述してください。
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。
invalid struct or union type	●列挙型のデータに対して構造体,共用体のメンバ参照しています。 ⇒正しく記述してください。
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。
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。
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。
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。
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。

表F.17　ccom308エラーメッセージ一覧表(10)

エラーメッセージ	エラー内容と対策
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。
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。
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。
invalid void type	●void型指定子にlongやsignedの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。
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。
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。
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。
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。
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。
mismatch prototyped parameter type	●プロトタイプ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。
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。
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。
Not enough memory	[UNIX版] ●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。 [MS-Windows 95,98 / NT版] ●メモリ空間が不足しています。 ⇒メモリを増やすか、Windows 95 ,98/ NT のスワップ空間を増やしてください。
not have 'long char'	●longとcharを同時に記述しています。 ⇒型指定子を正しく記述してください。
not have 'long float'	●longとfloatを同時に記述しています。 ⇒型指定子を正しく記述してください。

表F.18　ccom308エラーメッセージ一覧表(11)

エラーメッセージ	エラー内容と対策
not have 'long short'	●longとshortを同時に記述しています。 ⇒型指定子を正しく記述してください。
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。
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。
redeclare of 変数名	●変数名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。
redeclare of 列挙子	●列挙子が重複して定義されています。 ⇒どちらかの列挙子の名前を変更してください。
redefine function 関数名	●関数名で示される関数が2度定義されています。 ⇒関数は1度しか定義できません。どちらかの関数名を変更してください。
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。
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。
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。
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。
‘ restrict ’is duplicate	●restrictの宣言が重複しています。 ⇒restrictの宣言は1回にしてください。
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。
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。 ⇒構造体、共用体を先に定義してください。
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。
Sorry stack frame memory exhaust, max. 128 bytes but now nnn bytes	●スタックフレーム上に確保可能な引数は最大128バイトまでです。現在 nnnバイト使用しています。 ⇒引数のサイズあるいは引数の個数減らしてください。

表F.19　ccom308エラーメッセージ一覧表(12)

エラーメッセージ	エラー内容と対策
Sorry stack frame memory exhaust, max. 64(or 255) bytes but now nnn bytes	●スタックフレームは最大 64バイト (NC79)、 255バイト(NC30、NC77、および、 NC79で起動オプション -fDPO8 使用時) までです。 現在nnnバイト使用しています。 ⇒auto変数、引数などのスタックフレーム領域に確保される変数を減らしてください。
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。
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。
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。
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。
struct or union,enum does not have long or sign	●struct/union/enum型指定子にlongやsignedの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。
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。
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。
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。
syntax error	●文法エラーです。 ⇒正しく記述してください。
System Error...	●通常は発生しません(内部エラーです)。 本エラーの発生より以前に、発生したエラーに伴い発生する場合があります。 ⇒本エラー発生以前のエラーを全て取り除いても、本エラーが発生する場合には、メッセージの内容を弊社までご連絡ください。
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。

表F.20　ccom308エラーメッセージ一覧表(13)

エラーメッセージ	エラー内容と対策
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。
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回以上指定しないでください。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。
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。

表F.21　ccom308エラーメッセージ一覧表(14)

エラーメッセージ	エラー内容と対策
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。
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。
void value can't return	●voidでキャストされ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。
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。
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。
multiple #pragma EXT4MPTR's pointer, ignored (NC30 のみ)	●#pragma EXT4MPTR が２個以上宣言されています。 ⇒宣言を一つにしてください。
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。
'関数名' is recursion, then inline is ignored	●インライン宣言された '関数名 'が再帰呼び出しされています。 インライン宣言を無視します。 ⇒再帰呼び出しをしない様に記述してください。

F.6　ccom308ワーニングメッセージ

【表F.22】～【表F.31】にコンパイラccom308が出力するワーニングメッセージとその内容及び対処方法を示します。

表F.22　ccom308ワーニングメッセージ一覧表(1)

ワーニングメッセージ	ワーニング内容と対策
#pragma プラグマ名 & HANDLER 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。
#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。
#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。
#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。 ⇒正しく記述してください。
#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。 ⇒正しく記述してください。
#pragma プラグマ名 not function, ignored	●#pragma プラグマ名 において関数でない名前を記述しています。 ⇒関数名で記述してください。
#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、予めプロトタイプ宣言を行ってください。
#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、プロトタイプ宣言されていません。 ⇒#pragma プラグマ名で指定する関数は、予めプロトタイプ宣言を行ってください。
#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リターン値の型が不当です。 ⇒リターン値の型は、struct, union, double型以外を指定してください。
#pragma プラグマ名 unknown switch, ignored	●#pragmaプラグマ名で不正な switch を記述しています。 ⇒正しい switch を指定してください。

表F.23　ccom308ワーニングメッセージ一覧表(2)

ワーニングメッセージ	ワーニング内容と対策
#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。
#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バイト以内で記述してください。
#pragma directive conflict	●一つの関数に対して、異なる機能の#pragma を指定しています。 ⇒正しく記述してください。
#pragma DP[n]DATA　format error, ignored (NC79のみ)	●#pragma DP[n]DATA と -fDPO8オプションを併用しています。 ⇒#pragma DP[n]DATA と -fDPO8オプションを併用した場合、#pragma DP[n]DATA は無効となります。-fDPO8オプションを削除してください。
	●#pragma DP[n]DATA のフォーマットが間違っています。 ⇒正しく記述してください。
#pragma DMAC duplicate	●#pragma DMAC を２回以上定義しています。 ⇒#pragma DMAC を正しく定義してください。
#pragma DMAC variable must be far pointer for 変数名,ignored	●#pragma DMAC 宣言された変数は、far ポインタである必要があります。DMAC宣言を無視します。 ⇒#pragma DMAC を正しく定義してください。
#pragma DMAC variable must be unsigned int for 変数名,ignored	●#pragma DMAC 宣言された変数は、unsigned　int 型である必要があります。DMAC宣言を無視します。 ⇒#pragma DMAC を正しく定義してください。
#pragma DMAC 's variable must be pre-declared,ignored	●#pragma DMAC 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 DMAC を正しく定義してください。
#pragma DMAC, register conflict	●#pragma DMAC宣言 において同一レジスタに複数割り当てようとしています。 ⇒#pragma DMAC を正しく定義してください。
#pragma DMAC, unknown register name used	●#pragma DMAC宣言 において不明なレジスタが使用されています。 ⇒#pragma DMAC を正しく定義してください。
#pragma JSRA illegal location ,ignored (NC30, NC308 のみ)	●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。
#pragma JSRW illegal location ,ignored (NC30, NC308 のみ)	●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。
#pragma PARAMETER function's address used	●#pragma PARAMETERで指定された関数のアドレスをポインタ変数に代入しています。 ⇒代入しないで、正しく記述してください。

表F.24　ccom308ワーニングメッセージ一覧表(3)

ワーニングメッセージ	ワーニング内容と対策
#pragma control for function duplicate, ignored (NC30, NC308 のみ)	●#pragmaで同じ関数に対して、INTERRUPT, TASK, HANDLER, CYCHANDLER, ALMHANDLER を重複して指定しています。 ⇒INTERRUPT,TASK,HANDLER,CYCHANDLER, ALMHANDLERのうち１つを指定してください。
#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。
'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。
'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。
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。
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。バンクアドレスを無効にします。 ⇒データの型、near / far を確認してください。
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。記述が正しい場合には、このワーニングは無視してください。
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。記述が正しい場合には、このワーニングは無視してください。
assignment in comparison statement	●比較式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。
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。
can't get address from register storage class variable	●register記憶域クラスの変数に&演算子を記述しています。 ⇒register記憶域クラスの変数に&演算子を記述しないでください。
can't get size of bitfield	●sizeof演算子のオペランドにビットフィールド型を記述しています。 ⇒sizeof演算子のオペランドを正しく記述してください。

表F.25　ccom308ワーニングメッセージ一覧表(4)

ワーニングメッセージ	ワーニング内容と対策
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。 ⇒sizeof演算子のオペランドを正しく記述してください。
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。
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。
case value is out of range	●caseの値がswitchの引数の範囲を越えてます。 ⇒正しく記述してください。
character buffer overflow	●文字列のサイズが512文字を超えました。 ⇒511文字以下で記述してください。
character constant too long	●文字定数（シングルクォートに囲まれた文字）の文字数が多すぎます。 ⇒正しく記述してください。
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。 ⇒代入先の宣言部を確認してください。
cyclic or alarm handler always Bank 0 (NC77, NC79 のみ)	●#pragma CYCHANDLER又はALMHANDLERで指定した関数は、常にバンク0（アドレスが10000H未満）の領域にコンパイルされます。 ⇒ありません。
cyclic or alarm handler always load DT (NC77, NC79 のみ)	●#pragma CYCHANDLER又はALMHANDLERで指定した関数は、#pragma LOADDT をする必要はありません。 ⇒#pragma LOADDT を削除してください。
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。
enumerator value overflow size of unsigned char	●-fCEオプション使用時において、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。
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。
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。
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。

表F.26　ccom308ワーニングメッセージ一覧表(5)

ワーニングメッセージ	ワーニング内容と対策
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。
function must be far	●関数をnear型で宣言しています。 ⇒正しく記述してください。
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。
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。
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。
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。
hex character is out of range	●文字定数においてHEX文字が長すぎます。また、\の後に16進数以外の文字が入っています。 ⇒HEX文字を短くしてください。
identifier (メンバ名) is dupilicated, 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。
identifier (変数名) is dupilicated	●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。
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。
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。
incomplete array access	●不完全型の多次元配列を参照してます。 ⇒多次元配列のサイズを明記してください。
incompatible pointer types	●ポインタの示すオブジェクトの型が異なります。 ⇒ポインタの型を確認してください。
incomplete return type	●不完全な型を戻り値に記述しています。 ⇒戻り値を確認してください。
incomplete struct member	●不完全な構造体をメンバとして記述しています。 ⇒完全な構造体を記述してください。

表F.27　ccom308ワーニングメッセージ一覧表(6)

ワーニングメッセージ	ワーニング内容と対策
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。
inline function is called as normal funciton before, change to static function	●記憶クラスinline で宣言された関数が通常の関数として呼び出されています。 ⇒inline関数は使用する前に必ず定義を行ってください。
integer constant is out of range	●整数定数の値がunsigned longで表現できる値を超えました。 ⇒定数の値をunsigned longで表現できる値で記述してください。
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。
interrupt function can't return value	●#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。
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。
invalid #pragma EQU	●#pragma EQUの記述に誤りがあります。この行を無視します。 ⇒正しく記述してください。
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, interruptです。この行を無視します。 ⇒正しく記述してください。
invalid #pragma operand,ignored	●#pragmaのオペランドの記述に誤りがあります。この行を無視します。 ⇒正しく記述してください。
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。
invalid asm's M flag (NC77, NC79 のみ)	●asmステートメントにおいて、Mフラグに設定する値に誤りがあります。 ⇒整数型の定数 (0, 1, 2)で記述してください。
invalid asm's MX flag, ignored (NC77, NC79 のみ)	●asmステートメントにおいて、MXフラグに設定する値が整数値の定数ではありません。 ⇒整数型の定数 (0, 1, 2)で記述してください。
invalid asm's X flag (NC77, NC79 のみ)	●asmステートメントにおいて、Xフラグに設定する値に誤りがあります。 ⇒整数型の定数 (0, 1, 2)で記述してください。

表F.28　ccom308ワーニングメッセージ一覧表(7)

ワーニングメッセージ	ワーニング内容と対策
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。
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。
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記述しないでください。
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記述しないでください。
keyward (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。
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、このワーニングは無視してください。
mismatch prototyped parameter type	●プロトタイプ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。
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。
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。
mismatch function pointer assign-ment	●レジスタ引数の関数のアドレスを、レジスタ引数でない(プロトタイプされていない)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プロトタイプ形式にしてください。
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。
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。
No hex digit	●16進数の定数に16進数で使用できない文字が含まれています。 ⇒16進数の定数は 0 から 9、A から F、a から f で記述してください。

表F.29　ccom308ワーニングメッセージ一覧表(8)

ワーニングメッセージ	ワーニング内容と対策
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。
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。
non-initialized variable '変数名' is used	●初期化していない変数を参照している可能性があります。 ⇒記述を確認してください。このワーニングは、関数の最後の行で発生することもあります。この場合、関数内のauto変数等の記述を確認してください。記述が正しい場合は、このワーニングは無視してください。
non-prototyped function used	●プロトタイプ宣言していない関数を呼び出しています。-Wnon_prototypeオプションを指定した時のみ出力されます。 ⇒プロトタイプ宣言を行うか、-Wnon_prototypeオプションを削除してください。
non-propototyped function declared	●定義されている関数のプロトタイプ宣言が存在しません(-WNPオプション指定時のみ表示)。 ⇒プロトタイプ宣言を行ってください。
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。
octal_character 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。
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。
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。
prototype function is defined as non-prototyped function before	●プロトタイプ宣言していない関数を再度プロトタイプ宣言しています。 ⇒関数の宣言方法を統一してください。
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。
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。

表F.30　ccom308ワーニングメッセージ一覧表(9)

ワーニングメッセージ	ワーニング内容と対策
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プロトタイプ宣言を行ってください。
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。
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。
sorry, get stack's address, but DT not 0 (NC77, NC79 のみ)	●オプション -bank を指定した時に発生します。auto変数のアドレスをポインタに代入し、そのポインタでオブジェクトを参照した場合、DTがバンク0以外を示しているため、バンク0を参照できません。 ⇒ポインタを far で宣言してください。
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。 ⇒構造体、共用体を先に定義してください。
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。
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。
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。
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。
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。
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。
struct (or union) member's address can't has no near far information	●構造体(もしくは共用体)のメンバー(変数)の配置位置情報として near, far を指定しています。 ⇒メンバーについては near, far は指定しないでください。

表F.29　ccom308ワーニングメッセージ一覧表(10)

ワーニングメッセージ	ワーニング内容と対策
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。
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。
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。
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。
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。
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。
this function used before with non-default argument	●使用されたことのある関数を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。
this interrupt function is called as normal function before	●使用したことのある関数を #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。
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。
too few parameters	●プロトタイプ宣言で宣言した時より引数が足りません。 ⇒引数の数を確認してください。
too many parameters	●プロトタイプ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。
unknown #pragma STRUCT xxx	●#pragma STRUCTxxxは、処理できません。この行を無視します。 ⇒正しく記述してください。
Unknown debug option (-dx)	●オプション-dxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown function option (-Wxxx)	●オプション-Wxxxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown function option (-fx)	●オプション-fxは、指定できません。 ⇒オプションを正しく指定してください。

表F.29　ccom308ワーニングメッセージ一覧表(11)

ワーニングメッセージ	ワーニング内容と対策
Unknown function option (-gx)	●オプション-gxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown optimize option (-mx)	●オプション-mxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown optimize option (-Ox)	●オプション-Oxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown option (-x)	●オプション-xは、指定できません。 ⇒オプションを正しく指定してください。
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。 ※この警告は -Wunknown_pragma (-WUP)、 および、-Wallオプション指定時のみ表示されます。
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。
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。
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。
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。
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。
Code generaion for static functions (func) can be suppressed by using -ferase_ststic_function(-fESF)option.	●参照されない static 関数があります。 ⇒-ferase_ststic_function オプションにより、static 関数（関数名）のコード生成を抑止することができます。

付録G

SBDATA宣言&SPECIALページ関数宣言ユーティリティ(utl308)

SBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 の起動方法と起動オプションの機能を説明します。(本ユーティリティは、エントリー版には含まれません。)

G.1　utl308 の概要

G.1.1　utl308 の処理概要

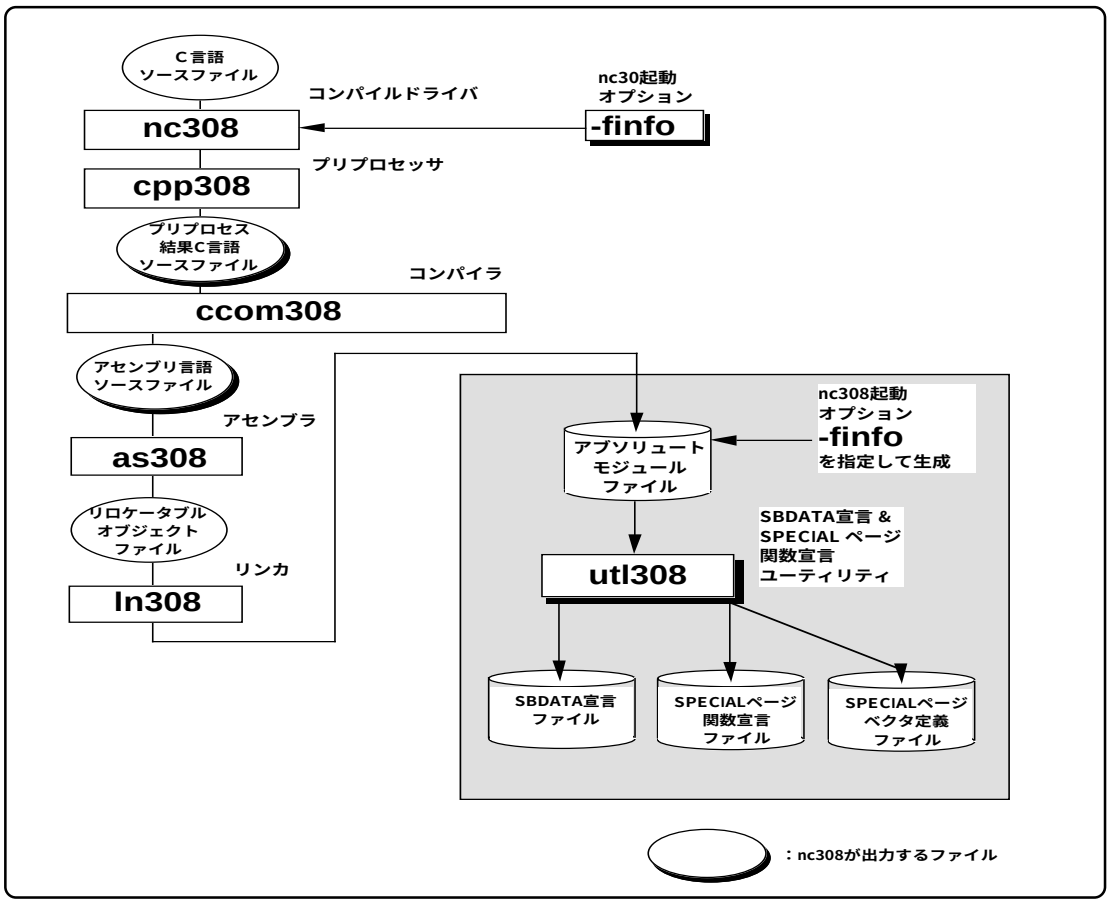
SBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 は、アブソリュートモジュールファイル(拡張子 .x30)を処理して、

1. SBDATA宣言
   使用頻度の高い変数からSB領域に割り当てるための宣言
   (#pragma SBDATA)
2. SPECIALページ関数宣言
   使用頻度の高い変数からスペシャルページ領域に割り当てるための宣言
   (#pragma SPECIAL)

を出力します。

utl308 を使用するには、コンパイル時に、コンパイルドライバに起動オプション ”-finfo ”を指定してアブソリュートモジュールファイル(拡張子 .x30)を生成してください。

NC308の処理フローを【図G.1】に示します。



図G.1　NC308の処理フロー

G.2　utl308の起動方法

G.2.1　入力書式

utl308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。

```
% utl308△[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。
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　utl308 コマンドの入力書式

utl308 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
●デバッグ情報の出力 ..... -gオプション

の両方を指定して、**アブソリュートモジュールファイル(拡張子 .x30)**を生成してください。

以下に入力例を示します。入力例では utl308 に、以下のオプションを指定しています。

●出力情報のファイルへ出力 ..... -oオプション
（デフォルトでは、標準出力への出力となっています。）

●アブソリュートモジュールファイルの生成

```
%nc308 ncrt0.a30 -finfo sample.c<RET>
M32C/80,M16C/80 Series NC308 COMPILER V.X.XX Release X
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

ncrt0.a30
sample.c


%
```

●SBDATA宣言の出力

```
%utl308 -sb308 ncrt0.x30 -o sample <RET>
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED



%
```

●SPECIALページ関数宣言の出力

```
%utl308 -sp308 ncrt0.x30 -o sample <RET>
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED



%
```

<RET>：リターンキーの入力を示します。

図G.3　utl308 コマンドの入力例

G.2.2　出力情報の切り替え

utl308 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。なお、どちらのオプションも指定されない場合には、utl308 は、エラーとなります。

1. SBDATA宣言を出力
   ●オプション“ -sb308 ”

2. SPECIALページ関数宣言を出力
   ●オプション“ -sp308 ”

オプションの指定例は、図G.3 を参照してください。

G.2.3　オプションリファレンス

utl308 を起動するためには、以下の表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。

【表G.1】にutl308 の起動オプションを示します。

表G.1　utl308の起動オプション

オプション	短縮形	機　能
-sb308 -sp308	なし	-sb308 → SBDATA宣言を出力します。 -sp308 → SPECIALページ関数宣言を出力します。 utl308 を使用する場合には、どちらかを必ず指定してください。どちらも指定されない場合にはエラーとなります。
-o	なし	SBDATA宣言、またはSPECIALページ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、ホストマシン(ＥＷＳ又はパーソナルコンピュータ)の標準出力に出力します。
-fover_write	-fOW	-oオプションで指定された出力ファイルに対して、強制的に上書きします。
-all	なし	[-sb308 オプション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。 [-sp308 オプション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。
-Wstdout	なし	エラー及びワーニングメッセージを標準出力へ出力します。
-sp=番号 -sp=番号,番号,...　（複数指定） -sp=番号-番号（複数範囲指定）	なし	指定された番号をスペシャルページ関数番号として割り当てません。 -sp308 オプションと同時に使用してください。
-fsection	なし	処理の対象として、#pragma SECTION で指定された変数および関数も、含めます。

-sb308

SBDATA宣言の出力

機　能：　SBDATA宣言を出力します。指定が無い場合には、エラーとなります。

実行例：

```
% utl308 -sb308 ncrt0.x30 -o sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED


%
```

-sp308

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。指定が無い場合には、エラーとなります。

実行例：

```
% utl308 -sp30 ncrt0.x30 -o  sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED


%
```

-o

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。
指定が無い場合には表示をホストマシン(EWS又はパーソナルコンピュータ)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には、標準出力に出力します。

実行例：

●SBDATA宣言の出力

```
% utl308 -sb308 ncrt0.x30 -o sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h

/*
 *    #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma  SBDATA  z                /* size = (2) / ref = [2] */
   (省略)
#pragma  SBDATA  vx               /* size = (2) / ref = [1] */


%
```

●SPECIALページ関数宣言の出力

```
% utl308 -sp308 ncrt0.x30 -o sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h

/*
 *    #pragma SPECIAL PAGE Utility
 */

#pragma  SPECIAL  255 func()                /* size = (200) / ref = [2]
*/
   (省略)
#pragma  SPECIAL  254 func1()               /* size = (200) / ref = [1]
*/


%
```

-all

全変数のSBDATA宣言 or 全関数へのSPECIALページ関数宣言の出力

機　能：　[-sb308 オプション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。

[-sp308 オプション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。

実行例：

●SBDATA宣言の出力

```
% utl308 -sb308 -all ncrt0.x30 -o sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h

/*
 *    #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma  SBDATA  z                  /* size = (2) / ref = [2] */
   (省略)
//#pragma  SBDATA  vx                 /* size = (2) / ref = [1] */

%
```

●SPECIALページ関数宣言の出力

```
% utl308 -sp308 -all ncrt0.x30 -o sample
M16C/80 UTILITY UTL308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

%type sample.h

/*
 *    #pragma SPECIAL PAGE Utility
 */

#pragma  SPECIAL  255 func()                /* size = (200) / ref = [2]
*/
   (省略)
#pragma  SPECIAL  254 func1()               /* size = (200) / ref = [1]
*/

%
```

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。

-Wstdout

エラーメッセージの標準出力への表示

機　能：　エラー、及び、ワ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。

実行例：

```
% utl308 -o sample ncrt0.x30 -Wstdout
M16C/80 UTILITY utl308 for M16C/80 V.X.XX.XX
COPYRIGHT(C) XXXX(XXXX) RENESAS TECHNOLOGY CORPORATION ALL RIGHTS RESERVED
AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS RESERVED

warning:cannot open file ‘ ncrt0.x30 ’

%
```

-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。
-sp308 オプションと同時に使用してください。

実行例：

●番号を一つだけ指定する場合
　　-SP= 番号
　　例）%utl308 -sp308 -sp=255 ncrt0.x30

●番号を複数指定する場合
　　-SP= 番号,番号,...
　　例）%utl308 -sp308 -sp=255,254 ncrt0.x30

●番号を範囲で指定する場合
　　-SP= 番号-番号
　　例）%utl308 -sp308 -sp=255-250 ncrt0.x30

-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には、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。

-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：　-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、上書きします。
-o オプションと同時に指定してください。

G.3　制限事項

SBDATA宣言を出力する場合、utl308 では、アセンブラで記述されたファイル中で宣言されている.sbsymは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.sbsymがある場合には、utl308 実行後生成された結果に対してSB領域内に入るように調整する必要があります。

SPECIALページ関数宣言を出力する場合、utl308では、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ページ関数は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には、ut308 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　utl308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

utl308 の処理対象となる変数は、以下の型をもつ外部変数に対してのみ処理を行います。

- ●_Bool
- ●unsigned char, signed char
- ●unsigned short, signed short
- ●unsigned int, signed int
- ●unsigned long, signed long
- ●unsigned long long, signed long lomg

なお、上記の型をもつ変数でも下記の条件を満たす場合は、処理の対象外となります。

- ●#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
- ●#pragma ROMで定義された変数

※　プログラム中で既に#pragma SBDATAを用いて宣言された変数が存在する場合は、utl308 で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ていきます。

G.4.2　SPECIALページ関数宣言が処理対象とする関数

utl308 の処理対象となる関数は、以下の外部関数に対してのみ処理を行います。

- ●staticで宣言されていない関数
- ●3回以上呼ばれている関数

なお、上記の関数でも下記の条件を満たす場合は、処理の対象外となります。

- ●#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

※　プログラム中で既に#pragma SPECIALを用いて宣言された関数が存在する場合は、utl308 で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ていきます。

G.5　utl308 の使用例

G.5.1　SBDATA宣言の場合

a.　SBDATA宣言ファイルの出力

SBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 にアブソリュートモジュールファイル(コンパイル時にオプション -finfo を使用)を処理させることにより、SBDATA宣言ファイルを出力することができます。【図G.4】にutl308の入力例を、【図G.5】にSBDATA宣言ファイル例を示します。

```
% utl308 ncrt0.x30 -osbdata<RET>

  %                    ：プロンプトを示します。
  ncrt0.x30            ：アブソリュートモジュールファイル名です。
```

図G.4　utl308 コマンドの入力例

```
/*
 *    #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma  SBDATA  data3                    /* size = (4) / ref = [     2] */
#pragma  SBDATA  data2                    /* size = (1) / ref = [     1] */
#pragma  SBDATA  data1                    /* size = (2) / ref = [     1] */
                                                 (1)           (2)
/*
 *    End of File
 */

                                        (1)データのサイズを示します。
                                        (2)データの使用頻度を示します。
```

図G.5　SBDATA宣言ファイル(sbdata.h)

上記方法により生成したSBDATA宣言ファイルをプログラム中にヘッダファイルとしてインクルードします。SBDATAファイルの設定例を【図G.6】に示します。

```
#include “ sbdata.h ”

func()
{
     (省略)
```

図G.6　SBDATA宣言ファイルの設定例

b.　アセンブラでSB宣言がある場合の調整

アセンブラルーチン中で.sbsym宣言によりSB領域を使用している場合は、utl308 により生成されたファイルを調整する必要が有ります。

```
[アセンブラルーチン]
      .sbsym        _sym
    (省略)
      .glb          _sym
_sym:
      .blkb         2



[utl308 生成ファイル]

/*
 *    #pragma SBDATA Utility
 */
/* SBDATA Size [255] */
#pragma  SBDATA  data3              /* size = (4) / ref = [     2] */
#pragma  SBDATA  data2              /* size = (1) / ref = [     1] */
                 •
                 •
              (省略)
                 •
                 •
#pragma  SBDATA  data1              /* size = (2) / ref = [     1] */
/*
 *    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utl308 生成ファイルから２バイト分のSBDATA宣言を削除します。

例)
                 •
                 •
//#pragma  SBDATA  data1              /* size = (2) / ref = [     1] */
  /*コメントアウト*/
```

図G.7　utl308 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

SBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 にアブソリュートモジュールファイル(コンパイル時にオプション -finfo を使用)を処理させることにより、SPECIALページ関数宣言ファイルおよびSPECIALページベクタ定義ファイルを出力することができます。【図 G.8】にutl308 の入力例、【図 G.9】にSPECIALページ関数宣言ファイルの例、【図 G.10】にSPECIALページベクタ定義ファイルの例を示します。

```
% utl308 -sp308 ncrt0.x30 -o special<RET>

  %              ：プロンプトを示します。
  ncrt0.x30      ：アブソリュートモジュールファイル名です。
```

図 G.8　utl308 コマンドの入力例

```
/*
 *    #pragma SPECIAL PAGE Utility
 */
/* SBDATA Size [255] */
#pragma  SPECIAL  255 func1                    /* size = (100) / ref = [    10] */
#pragma  SPECIAL  254 func2                    /* size = (100) / ref = [     7] */
#pragma  SPECIAL  253 func3                    /* size = (100) / ref = [     5] */
                                                  (1)            (2)
/*
 *    End of File
 */
                                        (1)関数のサイズを示します。
                                        (2)関数の参照頻度を示します。
```

図 G.9　SPECIALページ関数宣言ファイル(special.h)

```
;
;    #pragma SPECIAL PAGE Utility
;
;    special page definition
;
SPECIAL .macro NUM
        .org    OFFFFEH-(NUM*2)
        .glb    __SPECIAL_@NUM
        .word   __SPECIAL_@NUM & OFFFFH
.endm

        SPECIAL  255
        SPECIAL  254
        SPECIAL  253
;
;    End of File
;
```

図 G.10　SPECIALページベクタ定義ファイル(special.inc)

上記方法により生成したSPECIALページ関数宣言ファイルをプログラム中にヘッダファイルとしてインクルードします。

SPECIALページ関数宣言ファイルの設定例を【図 G.11】に示します。

```
#include “special.h”

func()
{
    (省略)
```

図 G.11　SPECIALページ関数宣言ファイルの設定例

また、SPECIALページベクタ定義ファイルをスタートアップ中にインクルードファイルとしてインクルードします。SPECIALページベクタ定義ファイルの設定例を【図 G.12】に示します。

```
(省略)

.section      vector
.include      “special.inc”

(省略)
```

図 G.12　SPECIALページベクタ定義ファイルの sect308.inc への設定例

また、コンパイラオプション -fmake_special_table(-fMST) オプション を使用する事により、スタートアップファイルへのSPECIALページベクタ定義ファイルの設定を省略する事ができます。詳しくは、’付録A コマンドオプションリファレンス”、’付録 B 拡張機能リファレンス” の　’#pragma SPECIAL” の項を参照してください。

G.6　utl308 のエラーメッセージ

G.6.1　エラーメッセージ

【表G.2、表G.3】にSBDATA宣言&SPECIALページ関数宣言ユーティリティutl308 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.2　utl308 エラーメッセージ一覧表

エラーメッセージ	エラー内容と対策
ignore option ’-? ’	● utl308 で使用できないオプションを指定していま　す。 ⇒正しいオプションを指定してください。
Illegal file extension ’.XXX ’	●ファイル拡張子が間違っています。 ⇒正しい拡張子を入力してください。
No input ’x30 ’ file specified	●アブソリュートモジュールファイルの指定がありません。 ⇒アブソリュートモジュールファイルを指定してください。
cannot open ’x30 ’ file ’ファイル名 ’	●アブソリュートモジュールファイルがオープンできません。 ⇒アブソリュートモジュールファイルを確認してください。
cannot close file ’ファイル名 ’	●ファイルがクローズできません。 ⇒ファイルを確認してください。
cannot open output file ’ファイル名 ’	●出力ファイルがオープンできません。 ⇒出力ファイルを確認してください。
not enough memory	●メモリが足りません。 ⇒メモリを増やしてください。
since ’ファイル名 ’ file exist, it makes a standard output	● -o で指定されたファイル名が、既に存在します ⇒出力ファイル名を確認してください。 -fover_write を同時に指定すると上書きすることができます。

G.6.2　ワーニングメッセージ

【表G.4】にSBDATA宣言ユーティリティutl308 が出力するワーニングメッセージとその内容及び対処方法を示します。

表G.4　utl308 ワーニングメッセージ一覧表

ワーニングメッセージ	ワーニング内容と対策
conflict declare of 変数名	●該当する変数が複数ファイル間で、異なる記憶域クラス、型、等で宣言されています。 ⇒変数の宣言を確認して下さい。
conflict declare of 関数名	●該当する関数が複数ファイル間で、異なる記憶域クラス、型、等で宣言されています。 ⇒関数の宣言を確認して下さい。

MEMO

M32C/90,80,M16C/80,70**シリーズ用**
Ｃコンパイラパッケージ V.5.20
Ｃコンパイラユーザーズマニュアル

発行年月日　2005**年**03**月**01**日**　　Rev.2.00

発行　　**株式会社 ルネサス テクノロジ　営業企画統括部**
〒100-0004　**東京都千代田区大手町**2-6-2

編集　　**株式会社 ルネサス ソリューションズ　ツール開発部**

M32C/90, 80, M16C/80, 70 シリーズ用
Cコンパイラパッケージ V.5.20
Cコンパイラユーザーズマニュアル

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668

RJJ10J0885-0100