User's Manual

M32C シリーズ用
C コンパイラパッケージ V.5.42
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　2010.04

ご注意書き

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

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

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

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

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

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

はじめに

NC308 は、ルネサス 32/16 ビットマイクロコンピュータ M32C 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 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
NC308	本コンパイラに含まれるコンパイラシステムを意味します。
nc308	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。
AS308	本コンパイラに含まれるアセンブラシステムを意味します。
as308	リロケータブルアセンブラ、又はその実行ファイルを意味します。
High- performance Embedded Workshop	付属の統合環境を意味します

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
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。

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 9 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) utl308 …… SBDATA 宣言& SPECIAL ページ関数宣言ユーティリティ
(6) MapViewer …… マップビューワ
(7) gensni …… Call Walker 用.sni ファイル作成ツール
(8) genmap …… High-performance Embedded Workshop マップ機能用 .map ファイル作成ツール

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。

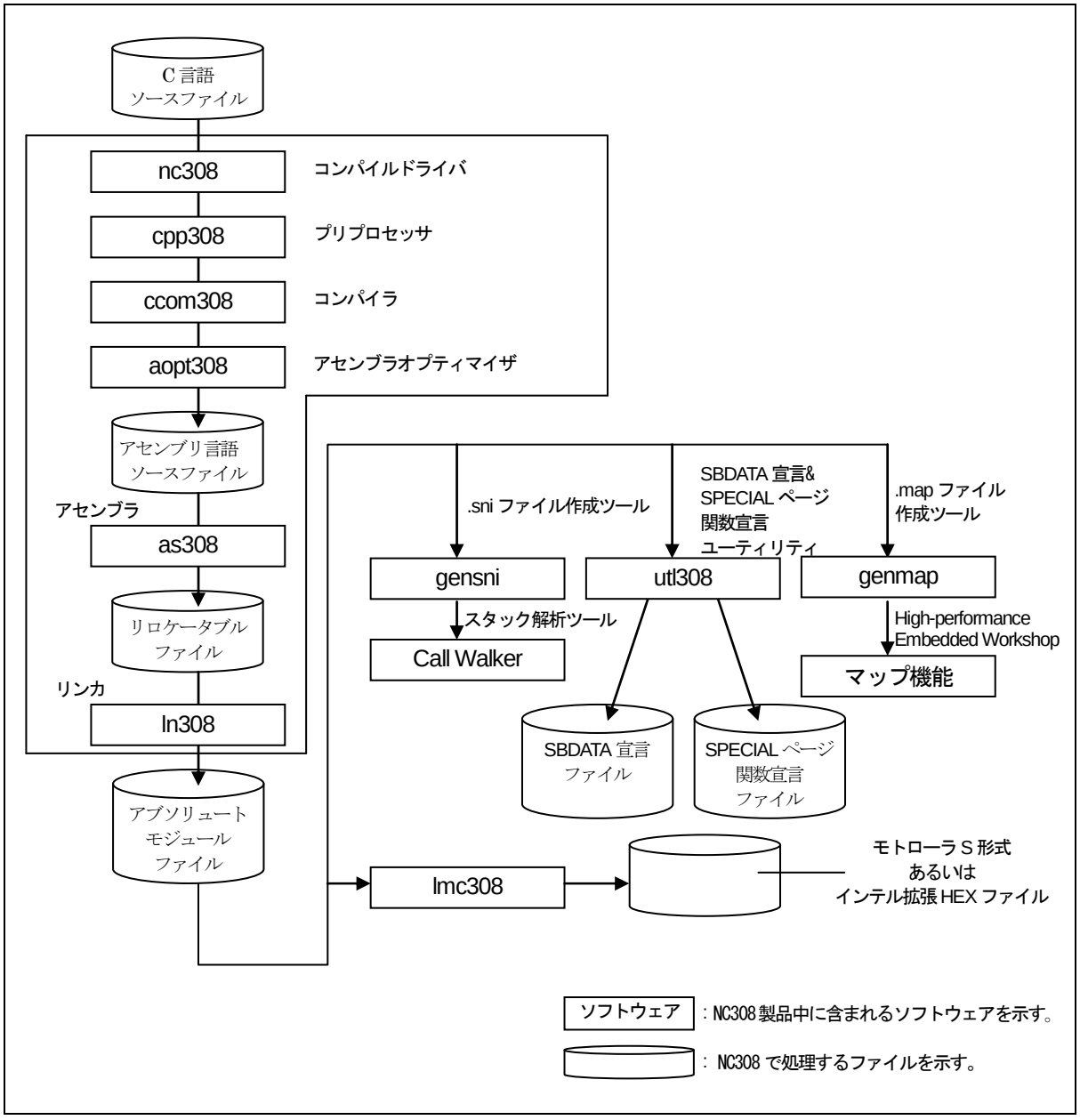


図1.1　NC308 の処理フロー

1.2.1 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。

1.2.5 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。

1.2.6 Call Walker & gensni

Call Walker 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です。また、gensni は、Call Walker で必要な情報の解析を行うユーティリティです。
Call Walker は、gensni が出力したスタック情報ファイル(.sni) を読み込んでスタックサイズを表示します。また、スタック情報ファイルに出力できないアセンブリプログラムのスタックサイズは、編集機能を用いて情報を追加・修正することが可能であり、システム全体のスタック使用量を求めることもできます。編集したスタック使用量に関する情報は、呼び出し情報ファイル(*.cal)として保存・読込み可能です。
Call Walker & gensni 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。

1.2.7 High-performance Embedded Workshop マップ機能 & genmap

High-performance Embedded Workshop のマップ機能により、リンケージエディタのセクション設定、およびセクション情報とシンボル情報を表示することができます。また、gensni は、High-performance Embedded Workshop のマップ機能を利用できるようにするためのユーティリティあり、マップ機能利用時に自動実行します。
High-performance Embedded Workshop のマップ機能は、genmap が出力した.map ファイルを読み込んでリンク後のメモリ配置をグラフィカルに表示します。
High-performance Embedded Workshop のマップ機能を使用するには、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。

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 エレクトロニクス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変更を行った場合は再度アプリケーションプログラムの動作評価を必ず行ってください。

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。

【図 1.2】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。

```
#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 */
}
```

図1.2 SFR 領域に対する C ソースコード記述

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 1.3】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 1.3】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。

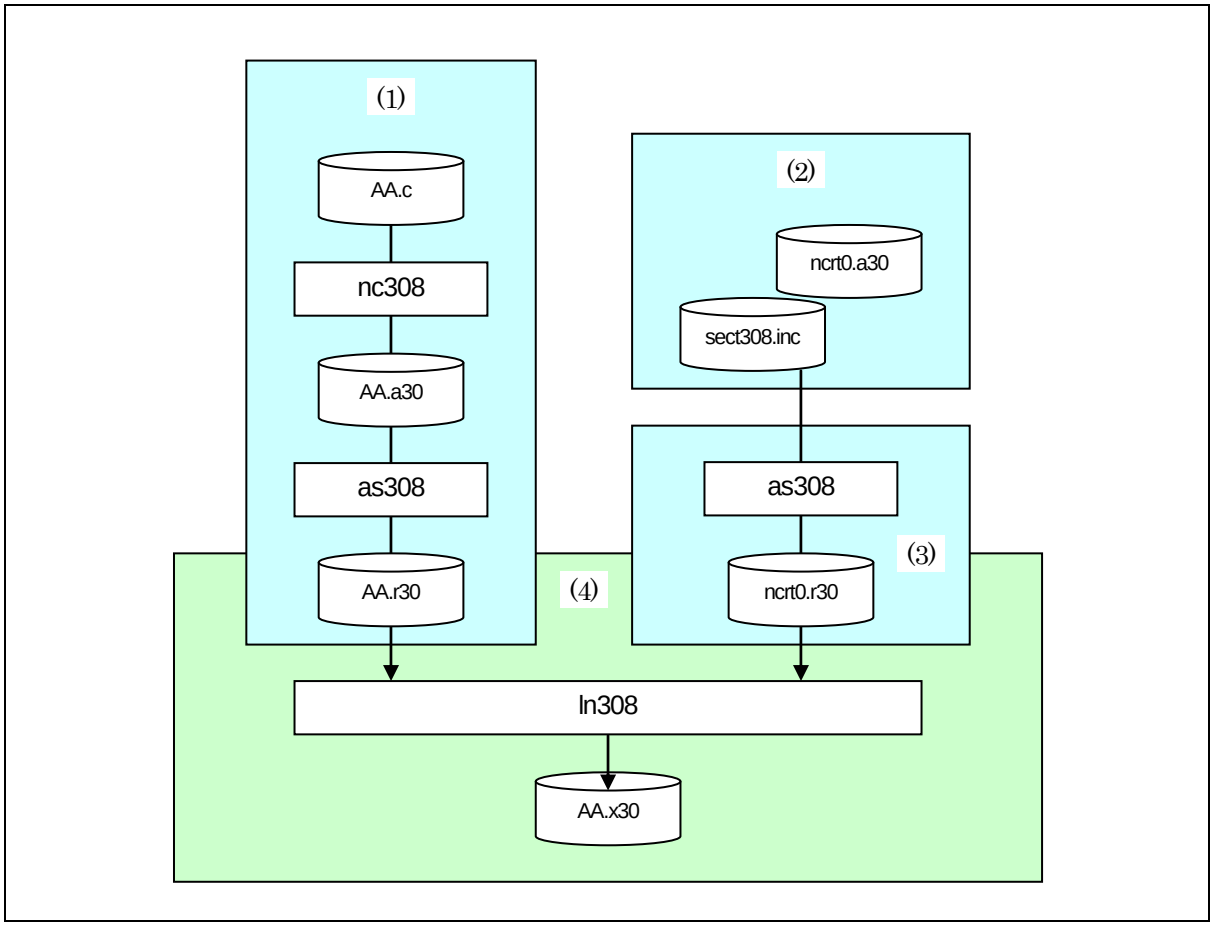


図1.3　プログラム開発フロー

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

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

ncrt0.r30 : ncrt0.a30
        as308 ncrt0.a30

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

図1.4　実行手順ファイル(makefile)の記述例

また、コンパイルドライバnc308 では、【図 1.4】と同様の処理をコマンドラインから【図 1.5】に示すように入力できます。

```
% 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1.5　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 sample.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 1.6】に示すファイルを出力します。次項から【図 1.7】に示すC言語ソースファイルsample.cをコンパイル/アセンブル/リンクした結果、出力された個々のファイル例と表示内容を説明します。

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「アセンブラのユーザーズマニュアル」を参照してください。

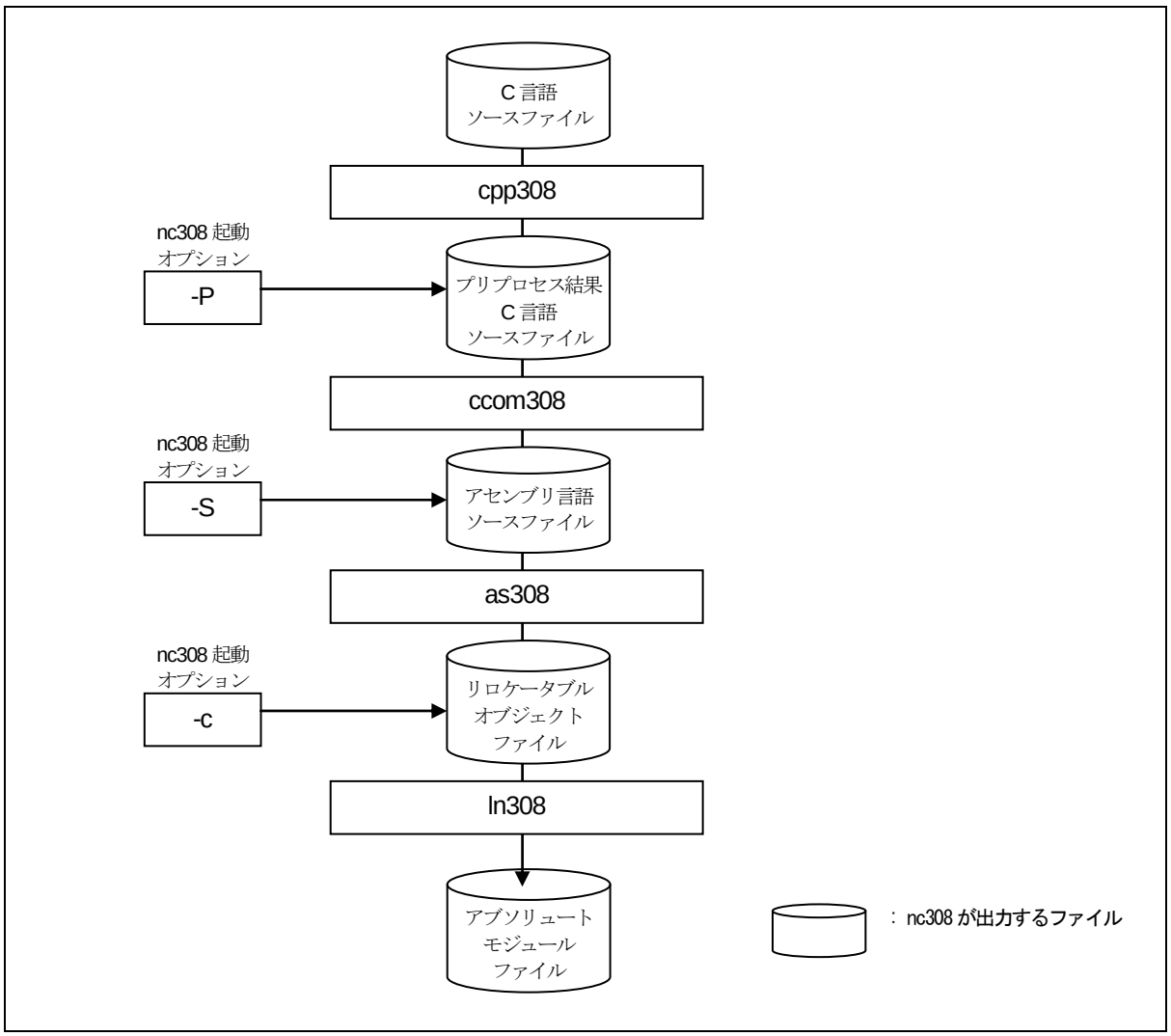


図1.6　nc308 起動オプションと出力ファイルの相互関係図

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

void        main(void)
{
            int         flag;

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

図1.7　C 言語ソースファイル例 (sample.c)

1.5.2　プリプロセス結果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 1.8】、【図 1.9】に示します。

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

typedef     unsigned int            size_t;

extern FILE _iob[];
```

図1.8　プリプロセス結果 C 言語ソースファイル例 (1)

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

```
void          main(void)                                                            (2)
{
              int           flag;

              flag = 0 ;                                    ← (3)

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

}
```

図1.9　プリプロセス結果 C 言語ソースファイル例 (2)

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 1.8】、【図 1.9】中の(1)～(4)にそれぞれ対応しています。

(1)　#include で指定したヘッダファイル stdio.h の展開部を示します。
(2)　マクロを展開した結果の C 言語ソースプログラムを示します。
(3)　#define で指定された CLR を 0 として展開していることを示します。
(4)　#define で指定された PRN が 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 1.10】、【図 1.11】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.XXX','REV.X'

;## NC308 C Compiler OUTPUT
;## ccom308 Version X.XX.XX.XXX
;## Copyright(C) XXXX(XXXX). Renesas Electronics Corp.
;## and Renesas Solutions Corp., All Rights Reserved.
;## Compile Start Time XXX XXX XX XX:XX:XX XXXX

;## COMMAND_LINE: ccom308   C:¥Renesas¥nc308wa¥v520r02¥TMP¥sample.i -o .¥sample.a30 -dS -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      'sample.c'
                .align
                ._line      6
;## # C_SRC :   {
                .glb        _main
_main:
                enter       #02H
                ._line      9
;## # C_SRC :                flag = CLR;
                mov.w       #0000H,-2[FB]           ;   flag
                ._line      11
;## # C_SRC :                printf( "flag = %d¥n", flag );                ← (2)
                push.w      -2[FB]      ;   flag
                push.l      #___T0
                jsr         _printf
                add.l       #06H,SP
                ._line      13
;## # C_SRC :   }
                exitd
                    :
                (省略)
                    :
                .glb        _puts
                .glb        $ungetc
                .glb        _printf
                .glb        _fprintf
                .glb        _sprintf
                    :
                (省略)
                    :
```

図1.10　アセンブリ言語ソースファイル例 (1) (sample.a30)

```
        .SECTION rom_FO,ROMDATA
___T0:
        .byte   66H     ;  'f'
        .byte   6cH     ;  'l'
        .byte   61H     ;  'a'
        .byte   67H     ;  'g'
        .byte   20H     ;  ' '
        .byte   3dH     ;  '='
        .byte   20H     ;  ' '
        .byte   25H     ;  '%'
        .byte   64H     ;  'd'
        .byte   0aH
        .byte   00H
        .END

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

図1.11　アセンブリ言語ソースファイル例 (2) (sample.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 1.10】中の(1)～(2)に対応しています。

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

(2) nc308 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.r30 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、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>
<CR> : 改行を示します。        -g  -as308  -l<CR>
                              -o<CR>
                              sample<CR>
```

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

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 →
```

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は"-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。
-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(-dS)"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。
-E	プリプロセスコマンドのみを処理し結果を標準出力に出力します。
-I *ディレクトリ名*	プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。ディレクトリは最大 256 個まで指定可能です。
-P	プリプロセスコマンドのみを処理し、ファイル（拡張子".i"）を作成します。
-S	アセンブ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。
-silent	起動時のコピーライトメッセージを出力しません。
-U *プリデファインドマクロ名*	指定したプリデファインドマクロを未定義にします。

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
-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*	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。

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)。

[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。

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は、必ずこのオプションを指定してください。
-gno_reg	レジスタ変数に関するデバッグ情報の出力を抑止します。

e. 最適化オプション

【表 2.5】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。

表2.5　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 容量ともに効果がある最適化を行います。
-OR	なし	ROM 容量を重視した最適化を行います。
-OS	なし	速度を重視した最適化を行います。
-OR_MAX	-ORM	ROM サイズを優先する最大限の最適化を行います。
-OS_MAX	-OSM	速度を優先する最大限の最適化を行います。
-Ocompare_byte_to_word	-OCBTW	連続した領域のバイト単位の比較をワード単位で行います。
-Oconst	-OC	const 修飾子で宣言した外部変数の参照を定数に置き換える最適化を行います。
-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。
-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。
-Oloop_unroll[=ループ回数]	-OLU	ループ文を回さずに、ループ回数分コードを展開します。"ループ回数"は省略可能、省略時は最大 5 回のループ文が対象となります。
-Ono_asmopt	-ONA	アセンブラオプティマイザ"aopt308" による最適化を抑止します。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_logical_or_combine	-ONLOC	論理 OR をまとめる最適化を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。 これにより ROM 容量を削減することができます。 ただし、使用するスタック量が多くなる可能性があります。
-Ostatic_to_inline	-OSTI	static 宣言された関数を、inline 宣言扱いにします。
-O5OA	なし	最適化オプション"-O5"選択時におけるビット操作命令(BTSTC、BTSTS)を使用したコード生成を抑止します。

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

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。
-fchar_enumerator	-fCE	enumerator(列挙子)の型を int 型ではなく unsigned char 型で扱います。
-fconst_not_ROM	-fCNR	const で指定した型を ROM データとして扱いません。
-fdouble_32	-fD32	double 型を float 型として処理します。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fextend_to_int	-fETI	char型データをint型に拡張し演算します(ANSI規格で定められた拡張を行います)。[2]
-ffar_RAM	-fFRAM	RAM データのデフォルト属性を far にします。
-finfo	なし	インスペクタ、"Stk Viewer"、"Map Viewer"、"utl308"に必要な情報を出力します。
-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。
-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。
-fnear_ROM	-fNROM	ROM データのデフォルト属性を near にします。
-fno_align	-fNA	関数の先頭アドレスのアライメントを行いません。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべて odd 属性のセクションに配置します。
-fno_switch_table	-fNST	switch 文に対し、比較を行ってから分岐するコードを生成します。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#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。
-fnot_reserve_asm	-fNRA	asm を予約語にしません("_asm"のみ有効になります)。
-fnot_reserve_far_and_near	-fNRFAN	far、near を予約語にしません(_far、_near のみ有効になります)。
-fnot_reserve_inline	-fNRI	inline を予約語にしません(_inline のみ予約語となります) 。
-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。
-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。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。
-M82	なし	M32C/80 シリーズに対応したコードを生成します。
-M90	なし	M32C/90 シリーズに対応したコードを生成します。
-fsizet_16	-fS16	型定義 size_t を unsigned long 型から unsinged int 型に変更します。
-fptrdifft_16	-fP16	型定義 ptrdiff_t を signed long 型から singed int 型に変更します。
-fuse_strings	-fUS	ストリングス命令を使用したコードを生成します。
-fuse_product_sum	-fUPS	積和演算命令を使用したコードを生成します。

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

g. ライブラリ指定オプション

【表 2.7】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2.7　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 2.8】に本コンパイラの言語仕様に関する記述の間違いに対して警告(ウォーニングメッセージ)を出力する起動オプションを示します。

表2.8　警告オプション

オプション	短縮形	機能
-Wall	なし	検出可能な警告("-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。
-Wccom_max_warnings =*ウォーニング回数*	-WCMW	ccom308 の出力するウォーニングの回数の上限を指定できます。
-Werror_file<*ファイル名*>	-WEF	タグファイルを出力します。
-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、ウォーニングを出力します。
-Wmake_tagfile	-WMT	error および warning が発生した場合はファイル毎にタグファイルを出力します。
-Wnesting_comment	-WNC	コメント中に"/*"を記述した場合に警告を出します。
-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。
-Wno_used_argument	-WNUA	引数を持つ関数を定義した場合に、使用していない引数に対してウォーニングを出力します。
-Wno_used_function	-WNUF	リンク時に未使用のグローバル関数を表示します。
-Wno_used_static_function	-WNUSF	コード生成が不要な static 関数名を表示します。
-Wno_warning_stdlib	-WNWS	"-Wnon_prototype"指定時や"-Wall"指定時に本オプションを指定すると、「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。
-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。
-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。
-Wstop_at_link	-WSAL	リンク時にウォーニングが発生した場合、アブソリュートモジュールファイルの生成を抑止します。
-Wstop_at_warning	-WSAW	ウォーニング発生時にコンパイル処理を停止します。
-Wundefined_macro	-WUM	#if の中で未定義のマクロを使用した場合に警告します。
-Wuninitialize_variable	-WUV	初期化されていない auto 変数に対してウォーニングを出力します。
-Wunknown_pragma	-WUP	サポートしていない #pragma を使用した場合、警告を出します。

i. アセンブル / リンクオプション

【表 2.9】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2.9　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。
-ln308△< オプション>	リンクコマンド ln308 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。

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。

- 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。
- 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 2.11】に示します。

```
;***********************************************************************
;
;       C COMPILER for M16C/80
; Copyright(C) XXXX(XXXX-XXXX). Renesas Electronics Corp.
; and Renesas Solutions Corp., All rights reserved.
;
;
;       ncrt0.a30 : NC308 startup program
;
;       This program is applicable when using the basic I/O library
;
;       $Id: ncrt0.a30,v X.XX.X.X XXXX/XX/XX XX:XX:XX XXXXX 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 ncrt0.a30 リスト (1)

```
;------------------------------------------------------------------------
; 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
```

(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) sect308. 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 ncrt0.a30 リスト (2)

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

;======================================================================
; 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
;
```

図2.8　スタートアッププログラム ncrt0.a30 リスト (3)

```
;=====================================================================
; 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

;=====================================================================
; 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
```

(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。
(9) near 領域の bss セクションのゼロクリア処理を行います。
(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。

図2.9　スタートアッププログラム ncrt0.a30 リスト (4)

```
;=====================================================================
; 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    __mnext
                .glb    __msize
                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
;=====================================================================
; Call main() function                                          ← (15)
;---------------------------------------------------------------------
                ldc     #0h,fb      ; for debuger

                .glb    _main
                jsr.a   _main
```

(11) far 領域の bss セクションのゼロクリア処理を行います。
(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。
(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。
(15) main 関数の呼び出しを行います。
※ main 関数呼び出し時には、割り込みは禁止になっています。
　割り込みを使用する際には"FSET"命令により割り込みを許可してください。

図2.10　スタートアッププログラム ncrt0.a30 リスト (5)

```
;=====================================================================
; 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 Electronics Corp.
; and Renesas Solutions Corp., All rights reserved.
;
;
;*************************************************************************
```

(16) exit 関数部です。
(17) ダミーの割り込み処理関数です。

図2.11　スタートアッププログラム ncrt0.a30 リスト (6)

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

(1) ncrt0.a30 について

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

- 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
  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
  上記(1)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
  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
  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 2.12】に示します。

図2.12　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[3]は、標準入出力関数ライブラリの入出力の初期化を行います。init関数は、ncrt0.a30 内でmain関数を呼び出す前に呼び出されます。【図 2.13】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2.13　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。

[3] init 関数は、標準入出力関数のためのマイコン（ハードウエア）の初期化も行っています。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 * __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。

(2)　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 2.14】にncrt0.a30 内の初期化部を示します。

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

図2.14　メモリ管理関数を使用するときの初期化部(ncrt0.a30)

メモリ管理関数を使用しない場合は、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 項に注意してください。

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

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。[4]

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。

[4] 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。

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2.15　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 2.16】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2.16　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を設定しています。通常は、設定値を変更する必要はありません。

g. プロセッサモードレジスタの設定

ncrt0.a30 中の【図 2.17】で示す部分において 04H番地(プロセッサモードレジスタ)に、組み込むシステムに合わせたプロセッサ動作モードを設定します。

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

図2.17　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、【図 2.18】で構成されます。

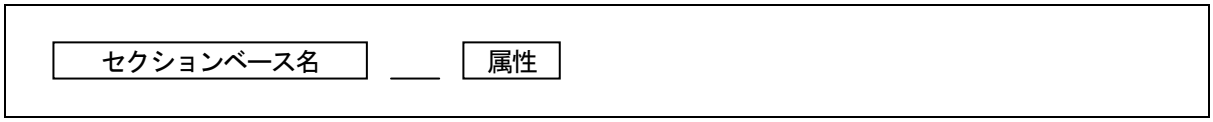


図2.18　セクション名

セクション名を【表 2.10】に属性を【表 2.11】に示します。

表2.10　セクションベース名

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

表2.11　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[5]	data, bss, rom
	F	far 属性	
	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	

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 2.12】に示します。

表2.12　セクションの名称

セクション名	内容
fvector	マイコンの固定ベクタの内容を格納します。
heap	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。
program	program
program_S	#pragma SPECIAL で指定したプログラムを格納します。
stack	スタックとして使用する領域です。アドレス 0400H から 0FFFFH の間に配置してください。
switch_table	switch 文に対するテーブルコードを格納します。 このセクションは、コンパイルオプション"-fswitch_other_section(-fSOS)"を使用した場合のみ生成されます。
vector	マイコンの割り込みベクタテーブルの内容を格納します。割り込みベクタテーブルは intb レジスタ相対によりマイコンの全メモリ空間に任意に配置できます。詳しくはマイコンのユーザーズマニュアルを参照してください。

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 2.19】に示します。

[5] 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

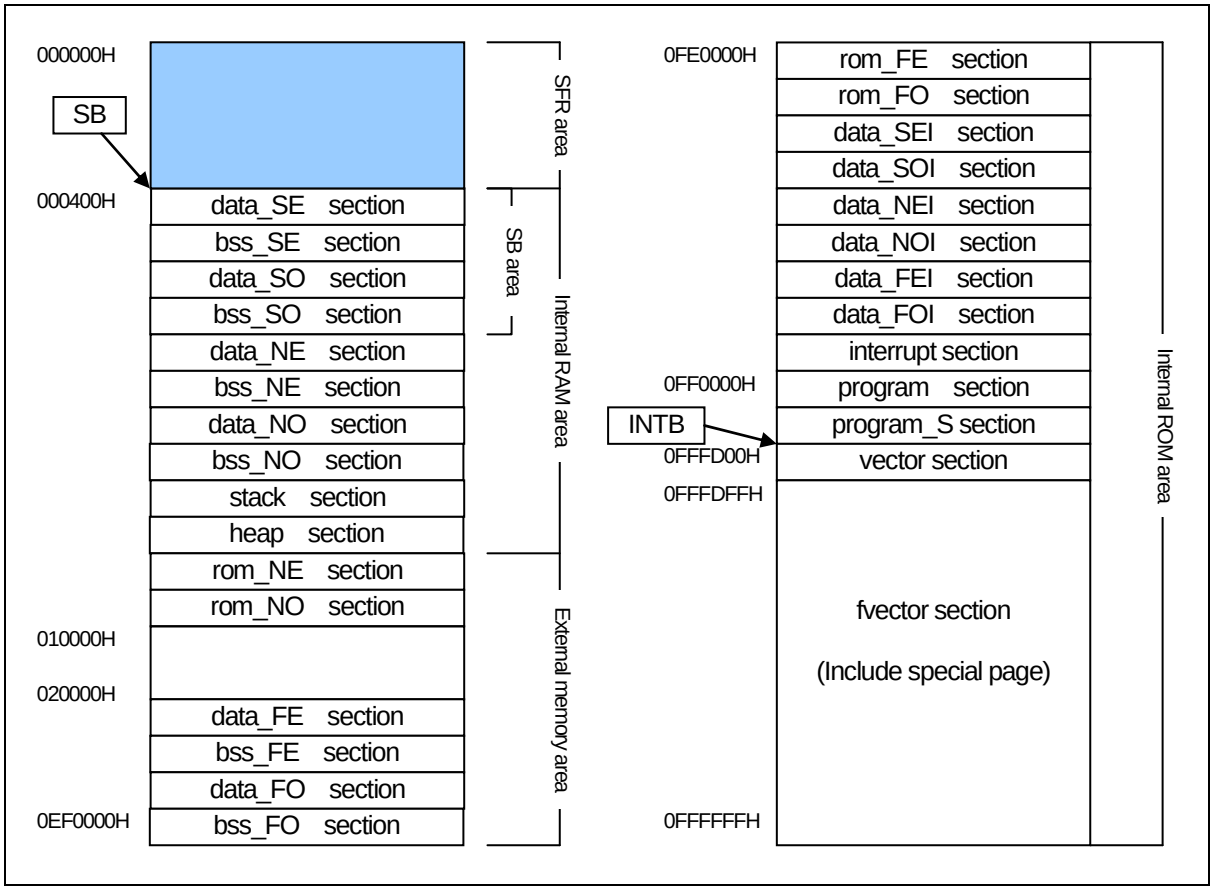


図2.19　セクション配置例 (1)

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 2.20】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。

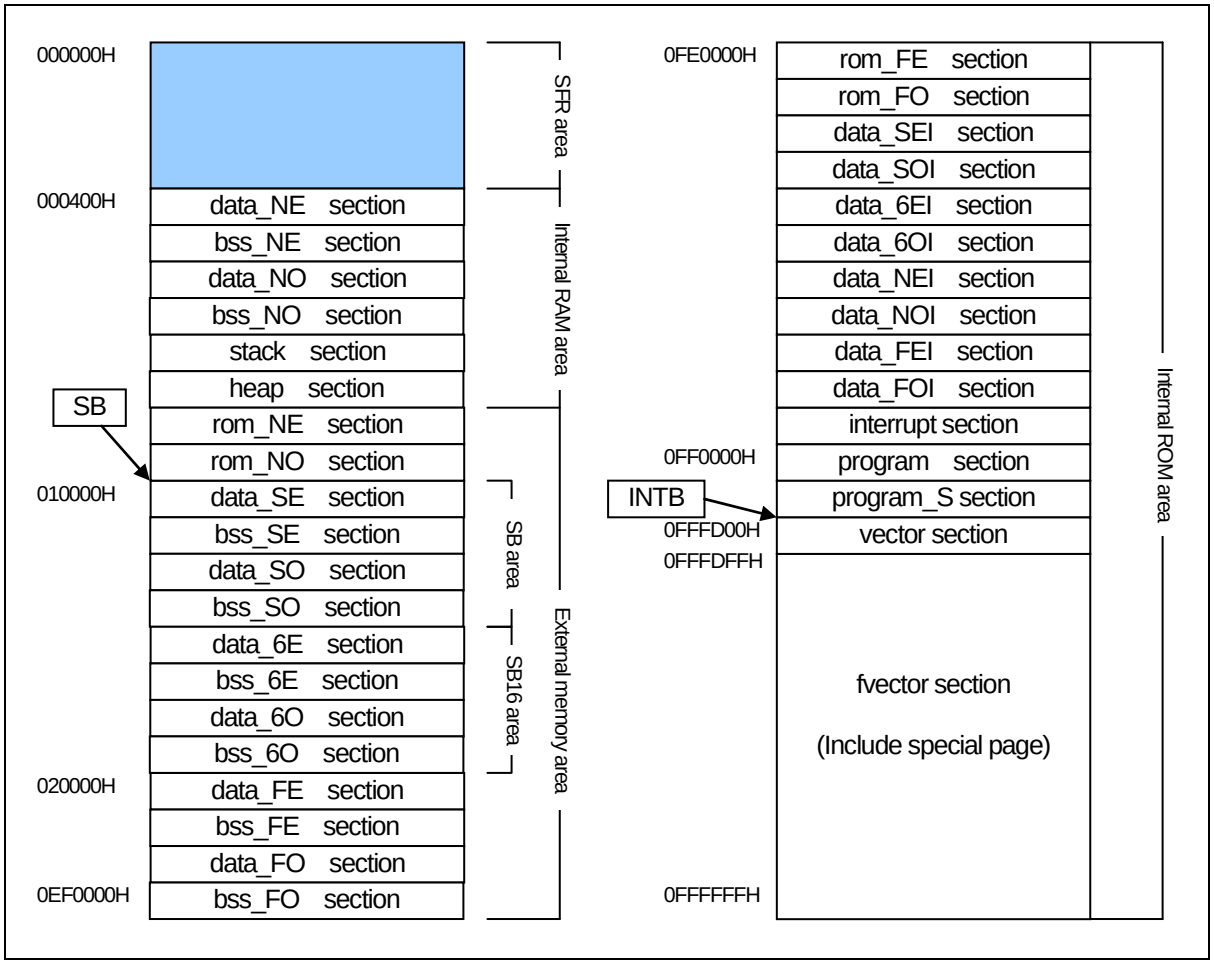


図2.20　セクション配置例 (2)

b.　メモリ配置設定用ファイルの概要

(1)　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 2.21】に示します。

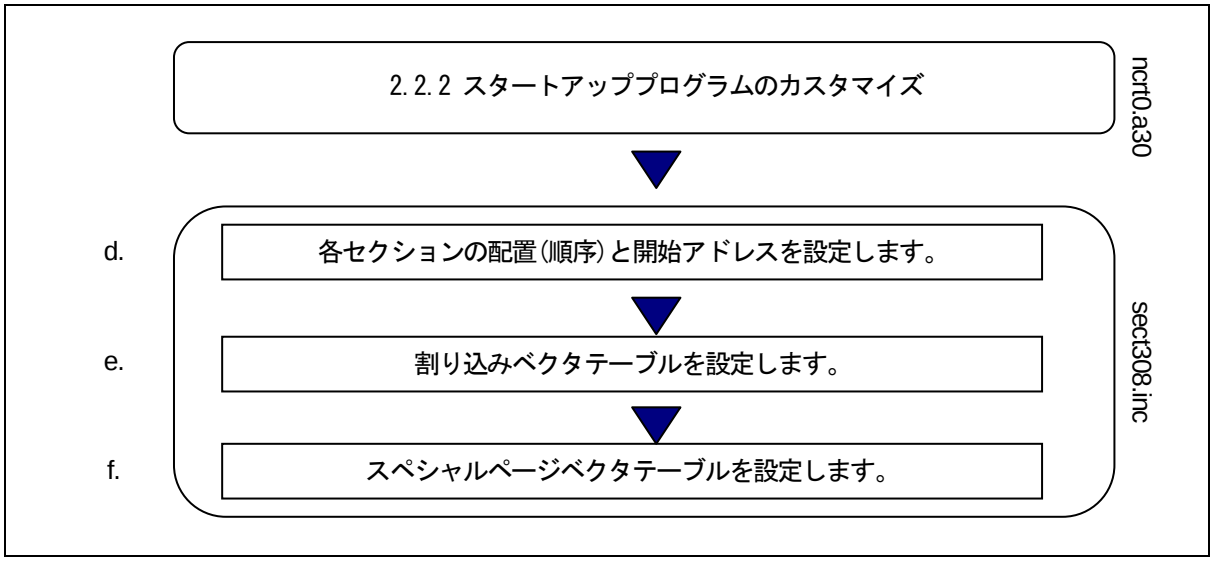


図2.21　メモリ配置の変更手順例

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 の指示命令".ORG"を用いて指定します。【図 2.22】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2.22　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。セクションの配置は、以下の規則に従ってください。

(1)　RAM 領域に配置するセクション

- stack セクション
- data_SE セクション
- data_NE セクション
- data_6E セクション
- bss_SE セクション
- bss_NE セクション
- bss_FE セクション
- bss_6E セクション
- heap セクション
- data_SO セクション
- data_NO セクション
- data_6O セクション
- bss_SO セクション
- bss_NO セクション
- bss_FO セクション
- bss_6O セクション

(2)　ROM に配置するセクション

- program セクション
- fvector セクション
- rom_NO セクション
- rom_FO セクション
- data_SOI セクション
- data_NOI セクション
- data_FOI セクション
- data_6O1 セクション
- interrupt セクション
- rom_NE セクション
- rom_FE セクション
- data_SEI セクション
- data_NEI セクション
- data_FEI セクション
- data_6E1 セクション

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

(1)　0H～0FFFFH(near 領域)にのみ配置できるセクション

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

(2)　0FF0000H～0FFFFFFH にのみ配置できるセクション

- program_S セクション
- fvector セクション

(3)　M32C シリーズの全メモリ空間に配置できるセクション

- program セクション
- data_NEI セクション
- data_FE セクション
- data_FEI セクション
- data_SEI セクション
- data_6E セクション
- data_6O セクション
- bss_FE セクション
- bss_6E セクション
- rom_FE セクション
- vector セクション
- data_NOI セクション
- data_FO セクション
- data_FOI セクション
- data_SOI セクション
- data_6EI セクション
- data_6OI セクション
- bss_FO セクション
- bss_6O セクション
- rom_FO セクション

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

- data_SE セクション
- data_SO セクション
- data_NE セクション
- data_NO セクション
- data_FE セクション
- data_FO セクション
- data_6E セクション
- data_6O セクション
- bss_NE セクション
- bss_FE セクション
- bss_SE セクション
- bss_6E セクション
- rom_NE セクション
- rom _FE セクション
- data_MON[n]_E セクション
- bss_MON[n]_E セクション
- data_MON[n]_EI 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_6EI セクション
- data_6OI セクション
- bss_NO セクション
- bss_FO セクション
- bss_SO セクション
- bss_6O セクション
- rom _NO セクション
- rom_FO セクション
- data_MON[n]_O セクション
- bss_MON[n]_O セクション
- data_MON[n]_OI セクション

(n は、1～4 の数字)

(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 2.23】から【図 2.26】に示します。

```
;***********************************************************************
;
;       C Compiler for M16C/80
; Copyright(C) XXXX(XXXX-XXXX). Renesas Electronics Corp.
; and Renesas Solutions Corp., All rights reserved.
;                       :
                (省略)
                        :
;       $Id: sect308.inc,v X.XX.X.X XXXX/XX/XX XX:XX:XX XXXXX 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:
```

図2.23　シングルチップモードにおける sect308.inc リスト(1)

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

;------------------------------------------------------------------------
; 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                    XXXX0H
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。

図2.24　シングルチップモードにおける sect308.inc リスト(2)

```
;------------------------------------------------------------------------
; 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:

                .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
                    :
                (省略)
                    :
                .lword          dummy_int               ; software int 63
.endif          ; __MVT__
```

図2.25　シングルチップモードにおける sect308.inc リスト(3)

```
.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
                    :
                (省略)
                    :
;               SPECIAL 18
;
.endif          ; __MST__
;=====================================================================
; 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 Electronics Corp.
; and Renesas Solutions Corp., All rights reserved.
;
;
;******************************************************************************
```

図2.26　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)　割り込み関数宣言時にベクタ番号付きで宣言すると、可変ベクタテーブルを自動生成します。

(2)　割り込み関数をベクタ番号付きで宣言しない場合は、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)　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 2.27】に割り込みベクタテーブル例を示します。

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

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

※ dummy_int はダミーの割り込み処理関数です

図2.27　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指示命令.GLB で外部参照宣言します。

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

(3) 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 2.28】にUART1 送信割り込み処理関数uarttrnの設定例を示します。

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

(以下省略)
```

図2.28　割り込みベクタテーブルの設定例

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 エレクトロニクス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場合は再度アプリケーションプログラムの動作評価を必ず行ってください。

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。

【図 3.1】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。

```
#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 */
}
```

図3.1　SFR 領域に対する C ソースコード記述

3.1.3　最適化について

a.　常に行われる最適化

最適化オプションの有無に関わらず、以下のものは最適化されます。

(1)　意味のない変数アクセス

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

```
extern int  port;

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

図3.2　意味のない変数アクセス例(最適化される)

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

```
extern int volatile     port;

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

図3.3 意味のない変数アクセス例(最適化を抑止)

(2) 意味のない比較

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

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

図3.4 意味のない比較

この例の場合、変数 c は char と記述されていますので、本コンパイラでは unsigned char 型として取り扱います。unsigned char 型で表現できる数値の範囲は 0 から 255 までですので、変数 c は-1 の値を持つことはありません。

このため、このような論理的に意味のない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3.5 実行されることのないプログラム

(4) 定数間の演算

定数間の演算はコンパイル時に演算されます。

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

            return i;
}
```

図3.6　定数間の演算

(5) 最適命令の選択

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

b. volatile 修飾子について

volatile 修飾子を使用することにより、変数の参照や参照順序、参照回数等に対して最適化の影響が無いようにすることができます。

ただし、以下の図に示すような、解釈が曖昧になるような記述は行わないでください。

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

a = b = c;              /* a = c なのか、a = b なのか? */
a = ++b;                /* a = b なのか、a = (b+1) なのか? */
```

図3.7　volatile 修飾子の解釈が曖昧になる例

3.1.4 register変数の使用に関する注意事項

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

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

この機能は、最適化に頼らずに生成コードを改良するためのものです。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 3.8】に示すようにfar型配列のextern宣言においてサイズを明記するか、もしくはオプション"-fsmall_array(-fSA)"[2]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3.8　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] コンパイルオプション"-fsmall_array(-fSA)"は、サイズ不明の配列を 64K バイト以内と仮定してコード生成を行います。

c. プロトタイプ宣言の活用

本コンパイラでは、関数のプロトタイプ宣言を行なうことにより、効率の良い関数呼び出しを行なうことができます。

すなわち、本コンパイラでは関数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は、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。

d. SB レジスタの活用

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

```
#pragma SBDATA   val

int              val;
```

図3.9　SB レジスタを用いたアドレッシングモードを使用する変数の宣言例

e. コンパイルオプション"-fJSRW"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は、"JSR.A"命令で呼出しを行ないます。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。

このような場合にコンパイルオプション"-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。

※コンパイルオプション"-OGJ"を使用すると、リンク時に最適な jmp 命令を選択します。

f. 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] 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。

3.2.2　スタートアップ処理を高速化する方法

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

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

```
static int      i;
```

図3.10　初期値を持たない変数の宣言例

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_SE_top,bss_SE
;               BZERO     bss_SO_top,bss_SO
;               BZERO     bss_6E_top,bss_6E
;               BZERO     bss_6O_top,bss_6O
```

図3.11　bss 領域のクリア処理のコメントアウト例

[4] 初期値を持たない RAM 上の外部変数のことを"bss"と呼びます。

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 3.12】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3.12　引数がない場合のアセンブラ関数の呼び出し例(sample1.c)

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

図3.13　sample1.c のコンパイル結果(抜粋)(sample1.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、32 ビット汎用レジスタ(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 3.14】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(void)
{
            int         i = 0x02;
            int         j = 0x05;

            asm_func(i, j);
}
```

図3.14　引数がある場合のアセンブラ関数の呼び出し例(sample2.c)

```
            .SECTION  program,CODE,ALIGN
            ._file    'sample2.c'
            .align
            ._line    5
;## # C_SRC :          {
            .glb      _main
_main:
            enter     #04H
            pushm     R1
            ._line    6
;## # C_SRC :                  int         i = 0x02;
            mov.w     #0002H,-4[FB]       ;   i
            ._line    7
;## # C_SRC :                  int         j = 0x05;
            mov.w     #0005H,-2[FB]       ;   j
            ._line    9
;## # 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
            ._line    10
;## # C_SRC :          }
            popm      R1
            exitd
E1:
            .glb      _asm_func         ← アセンブラ関数の呼び出し('_'を付加しています)
            .END                          #pragma PARAMETER で宣言した関数は常に'_'が付加されます
```

図3.15 sample2.c のコンパイル結果(抜粋)(sample2.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
- 64 ビット整数型(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) アセンブラの指示命令".SECTION"でセクション名を指定します。
(2) 関数名ラベルをアセンブラの指示命令".GLB"でグローバル指定します。
(3) 関数名に_(アンダースコア)を付加して、ラベルとして記述します。
(4) 関数内でB及びUフラグを変更する場合は、フラグレジスタをスタック上に退避してください。[5]
(5) 関数内で破壊されるレジスタを退避してください。[6]

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。
(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 3.16】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
```

図3.16 アセンブラ関数の記述例

[5] 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。
[6] 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 3.17】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.l       #01A000H, R2R0
        rts
        .END
```

図3.17　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指示命令.GLB で外部参照宣言する必要があります。

【図 3.18】にC言語プログラムの大域変数counter をアセンブラ関数asm_func内で参照する例を示します。

```
C 言語プログラム:

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

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


アセンブラ関数:

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

図3.18　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 3.19】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3.19　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。

3.3.3 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 3.20】に示す(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3.20 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
- 64 ビット整数型(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「NCxx」と言語仕様レベル(拡張機能を含む)で基本的に互換性を有しています。ただし、以下の点について異なりますのでご注意ください。

a. near / far のデフォルトの違い

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

表3.3 NC シリーズの near / far デフォルト

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

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

関数呼び出し時のレジスタの退避は、NC30 では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 において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) (1)で退避したレジスタを関数の出口で復帰

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]△
                          [リロケータブルモジュールファイル名]△[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	リロケータブルモジュールファイル(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。
-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(-dS)"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。
-E	プリプロセスコマンドのみを処理し結果を標準出力に出力します。
-I *ディレクトリ名*	プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。ディレクトリは最大 256 個まで指定可能です。
-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。
-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。
-silent	起動時のコピーライトメッセージを出力しません。
-U *プリデファインドマクロ名*	指定したプリデファインドマクロを未定義にします。

-c

コンパイルドライバの制御

機　　能:　リロケータブルモジュールファイル(拡張子.r30)を作成し、処理を終了します。

注意事項:　このオプションを選択したときは、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。

[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。

-D *識別子名*

コンパイルドライバの制御

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

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。

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

-dsource　　-dS

コメントオプション

機　　能: 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: (1) -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。
(2) 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(-dS)"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。

-E

コンパイルドライバの制御

機　　能:　プリプロセスコマンドのみを処理し結果を標準出力に出力します。

注意事項:　このオプションを選択したときは、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。

-I *ディレクトリ名*

コンパイルドライバの制御

機　　能:　プリプロセスコマンドの#include で参照するファイルを検索するディレクトリ名を指定します。
最大 256 個のディレクトリを指定できます。

補足説明 :　-I オプションに複数のディレクトリ dir1 と dir2 を指定する例を以下に示します。
% nc308 -Idir1 -Idir2 sample.c<RET>
%: プロンプトを示します。
<RET>:リターンキーの入力を示します。

書　　式:　nc308△-I ディレクトリ名△<C 言語ソースファイル名>

注意事項:　指定できるディレクトリ名の数は、使用しているホストマシンの 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。

-P

コンパイルドライバの制御

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

注意事項:
(1)　このオプションを選択したときは、アセンブリ言語ソースファイル(拡張子.a30)、リロケータブルモジュールファイル(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は、-E オプションを選択し、リダイレクトしてください。

-S

コンパイルドライバの制御

機　　能:　アセンブ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。

注意事項:　このオプションを選択したときは、リロケータブルモジュールファイル(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。

-silent

コンパイルドライバの制御

機　　能:　起動時のコピーライトメッセージを出力しません。

-U *プリデファインドマクロ名*

コンパイルドライバの制御

機　　能:　プリデファインドマクロ定数を未定義にします。

書　　式:　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>

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。

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
-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*	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*

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:　nc308△-dir ディレクトリ名

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

-o *ファイル名*

出力ファイル指定

機　　能:　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>

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。

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。

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

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

A.2.4　デバッグ用オプション

【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は、必ずこのオプションを指定してください。
-gno_reg	レジスタ変数に関するデバッグ情報の出力を抑止します。

-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。

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

-genter

enter 命令の出力

機　　能:　関数呼び出し時に必ず enter 命令を出力します。

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

-gno_reg

レジスタ変数に対するデバッグ情報の抑止

機　　能:　レジスタ変数に対するデバッグ情報の出力を抑止します。

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

A.2.5　最適化オプション

【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。

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	ROM 容量を重視した最適化を行います。
-OS	なし	速度を重視した最適化を行います。
-OR_MAX	-ORM	ROM サイズを優先する最大限の最適化を行います。
-OS_MAX	-OSM	速度を優先する最大限の最適化を行います。
-Ocompare_byte_to_word	-OCBTW	連続した領域のバイト単位の比較をワード単位で行います。
-Oconst	-OC	const 修飾子で宣言した外部変数の参照を定数に置き換える最適化を行います。
-Oforward_function_to_inline	-OFFTI	全てのインライン関数に対して、インライン展開を行います。
-Oglb_jmp	-OGJ	外部分岐の最適化を行います。
-Oloop_unroll[=ループ回数]	-OLU	ループ文を回さずに、ループ回数分コードを展開します。"ループ回数"は省略可能、省略時は最大 5 回のループ文が対象となります。
-Ono_asmopt	-ONA	アセンブラオプティマイザ"aopt308" による最適化を抑止します。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_logical_or_combine	-ONLOC	論理 OR をまとめる最適化を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。 これにより ROM 容量を削減することができます。 ただし、使用するスタック量が多くなる可能性があります。
-Ostatic_to_inline	-OSTI	static 宣言された関数を、inline 宣言扱いにします。
-O5OA	なし	最適化オプション"-O5"選択時におけるビット操作命令(BTSTC、BTSTS)を使用したコード生成を抑止します。

主な最適化オプションの効果を【表A.6】に示します

表A.6　最適化オプション効果一覧表

効果	-O	-OR	-OS	-OSA
速度	良	悪	良	良
ROM サイズ	良	良	悪	良
消費スタック	良	同	同	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。

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

-O2: -O1 と同じです

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

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

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

但し以下の条件を満たす場合、正常なコードを出力できない可能性があります。
- 異なるポインタ変数が同時に同じメモリ位置を指す場合。

例:

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

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

実行結果:

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

-O[1-5]

最適化

注意事項:　M32C シリーズの 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。

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 */
}
```

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

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

-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。

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。

-OR_MAX　　-ORM

最適化

機　　能:　ROM サイズを優先する最大限の最適化を行います。

説明詳細:

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

(2) 統合開発環境 High-performance Embedded Workshop 上で選択する場合は、「Renesas M32C Standard Toolchain」の「C」タブの「Size or speed:」を有効にし、「ROM size to the minimum」を選択します。

注意事項:

(1) M32C シリーズの一部のマイコンシリーズは、SFR 領域のレジスタへの書き込み、読み出しにビット操作命令(BTSTC、BTSTS)を使用することができません。コンパイルオプション”-OR_MAX”または"-O5"を選択した場合は、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。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を確認した場合は、以下のいずれかの方法で回避してください。また、いずれの場合も生成コードに問題が無いことを必ず確認してください。
- コンパイルオプションオプション"-OR_MAX"または"-O5"以外を選択する。
- asm 関数を使用してプログラム中に直接命令を記述する。

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

(3) 必ずリンクオプション"-JOPT"を指定してください。

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

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

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

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

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

-OS_MAX　　-OSM

最適化

機　　能:　サイクル数を優先する最適化を行います。

説明詳細:

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

(2) 統合開発環境 High-performance Embedded Workshop 上で選択する場合は、「Renesas M32C Standard Toolchain」の「C」タブの「Size or speed:」を有効にし、「ROM size to the minimum」を選択します。

注意事項:

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

-Ocompare_byte_to_word　　-OCBTW

最適化

機　　能:　連続した領域のバイト単位の比較をワードで行います。

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

-Oconst　　-OC

最適化

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

補足説明:　以下の条件を同時に満たした場合に最適化を行います。

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

-Oforward_function_to_inline　　-OFFTI

最適化

機　　能:　全てのインライン関数に対して、インライン展開を行います。

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

注意事項:
(1) インライン関数の宣言と、インライン関数の実体定義は、同一ファイル内に記述してください。
(2) インライン関数の引数には、構造体や共用体を使用する事はできません。これらを使用した場合、コンパイルエラーとなります。
(3) インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合、コンパイルエラーとなります。
(4) インライン関数の再帰呼出しをすることはできません。再帰呼出しの記述を行った場合、コンパイルエラーとなります。

-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。

注意事項:　本機能を使用する場合は、必ず、リンクオプション ”JOPT” を指定してください。

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

ループの展開

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

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

注意事項:　for 文を展開するため、ROM サイズは増加します。

-Ono_asmopt　　-ONA

アセンブラオプティマイザの抑止

機　　能:　アセンブラオプティマイザ"aopt308"による最適化を抑止します。

-Ono_bit　　-ONB

最適化の抑止

機　　能:　ビット操作をまとめる最適化を抑止します。

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

注意事項:　-O[3～5](もしくは-OR、-OS)オプションを選択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。

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] オプションを選択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。

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_logical_or_combine　　-ONLOC

最適化の抑止

機　　能:　論理 OR をまとめる最適化を抑止します。

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

例:

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

この場合、変数 a に対して最大 3 回の参照が行われますが、最適化後は 1 回の参照になります。
しかし、変数 a が、I/O 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ります。この場合は、本オプションを選択して論理 OR をまとめる最適化を抑止してください。
なお、変数に volatile 宣言がされている場合は、論理 OR をまとめる最適化は行われません。

-Ono_stdlib　　-ONS

最適化の抑止

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

補足説明: 本オプションは、以下の最適化を抑止します。

- strcpy()、memcpy() 等の標準ライブラリ関数を SMOVF 命令等に置き換える、最適化。
- 引数の near / far に応じたライブラリ関数に変更する最適化。

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

-Osp_adjust　　-OSA

スタック補正コードをまとめる

機　　能: 関数呼び出し後のスタック補正コードをまとめる最適化を行います。

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

例:

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

```
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。

-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。

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

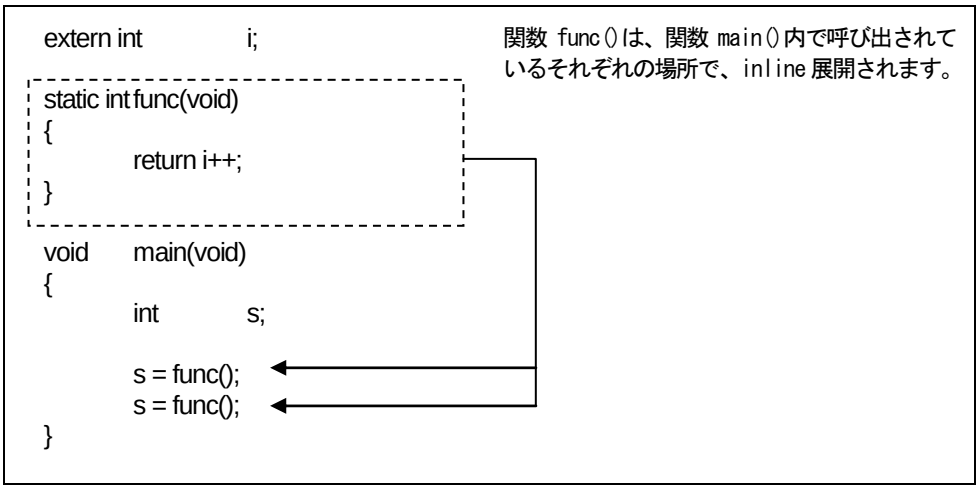
(1) 関数呼び出しの前に、実体が記述されている static 関数を対象とします。
- 関数の呼び出しと、その関数の実体が、同じソースファイル内に記述されていなければなりません。
- "-Oforward_function_to_inline"オプションを選択した場合は、本条件を無視してください。

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

(3) 対象となる static 関数を再帰呼び出ししていない場合。

(4) コンパイラのアセンブ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"オプションを選択した場合は、本条件を無視してください。

以下に、inline 展開される static 関数の記述例を示します。



注意事項: (1) inline 関数扱いになった static 関数の実体の記述に対するアセンブラコードは、常に生成されます。

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

-O5OA

最適化の抑止

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

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

A.2.6　生成コード変更オプション

【表A.7】にnc308 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.7　生成コード変更オプション(1/2)

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。
-fchar_enumerator	-fCE	enumerator(列挙子)の型を int 型ではなく unsigned char 型で扱います。
-fconst_not_ROM	-fCNR	const で指定した型を ROM データとして扱いません。
-fdouble_32	-fD32	double 型を float 型として処理します。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fextend_to_int	-fETI	char型データをint型に拡張し演算します(ANSI規格で定められた拡張を行います)。[2]
-ffar_RAM	-fFRAM	RAM データのデフォルト属性を far にします。
-finfo	なし	"Call Walker"、"Map Viewer"、"utl308"に必要な情報を出力します。
-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。
-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。
-fnear_ROM	-fNROM	ROM データのデフォルト属性を near にします。
-fno_align	-fNA	関数の先頭アドレスのアライメントを行いません。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべて odd 属性のセクションに配置します。
-fno_switch_table	-fNST	switch 文に対し、比較を行ってから分岐するコードを生成します。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#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。
-fnot_reserve_asm	-fNRA	asm を予約語にしません("_asm"のみ有効になります)。
-fnot_reserve_far_and_near	-fNRFAN	far、near を予約語にしません(_far、_near のみ有効になります)。
-fnot_reserve_inline	-fNRI	inline を予約語にしません(_inline のみ予約語となります) 。
-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。
-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。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。
-M82	なし	M32C/80 シリーズに対応したコードを生成します。
-M90	なし	M32C/90 シリーズに対応したコードを生成します。
-fsizet_16	-fS16	型定義 size_t を unsigned long 型から unsinged int 型に変更します。

[2] ANSI 規格では char 型データ又は signed char 型データを評価する時に必ず int 型に拡張します。
これは 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.8　生成コード変更オプション（2/2）

オプション	短縮形	機能
-fptrdifft_16	-fP16	型定義 ptrdiff_t を signed long 型から singed int 型に変更します。
-fuse_strings	-fUS	ストリングス命令を使用したコードを生成します。
-fuse_product_sum	-fUPS	積和演算命令を使用したコードを生成します。

-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。

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

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

-fconst_not_ROM　　-fCNR

生成コードの変更

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

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

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

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

-fdouble_32　　-fD32

生成コードの変更

機　　能:　double 型を float 型として処理します。

補足説明:
(1) 本オプション指定場合は、必ず、関数のプロトタイプ宣言を行なってください。プロトタイプ宣言がない場合は、不正なコードを生成する場合があります。
(2) 本オプションを選択した場合の double 型に対するデバッグ情報は float 型として扱われます。このため、デバッガ、シミュレータの C ウォッチウィンドウ、グローバルウィンドウ等では float 型として表示されます。

-fenable_register　　-fER

生成コードの変更

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

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

- 整数型変数
- ポインタ変数

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

-fextend_to_int　　-fETI

生成コードの変更

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

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

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

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

この場合「c2 * 2」の演算で char 型をオーバーフローし、正しい結果を求められません。
本オプションを選択することにより、正しい結果を求めることができます。
デフォルトの設定を int 型への拡張を行わないようにしているのは、ROM 効率を少しでも良くするためです。

-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。

-finfo

生成コードの変更

機　　能: "Call Walker"、"Map Viewer"、"utl308"に必要な情報を出力します。

補足説明: "Call Walker"、"Map Viewer"、"utl308"を使用する時は、このオプションの選択により生成されたアブソリュートファイル(.x30)が必要です。

注意事項:
(1) asm 関数内でのグローバル変数の使用はチェックされません。このため、utl308 でも asm 関数の使用は無視されます。
(2) -finfo は-g を含みます。
(3) 本オプションを選択しても、コンパイラの生成コードには影響を与えません。

-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に JSR.A 命令を用いて呼び出します。本オプションを選択した場合、JSR.A 命令を JSR.W 命令に変更することができるため、生成コードサイズを圧縮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 ROM 圧縮をしたい場合に有効です。

注意事項: 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 JSR.W 命令で呼び出すと、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)。

-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。本オプションを選択した場合にポインタ型のデータは 16 ビットサイズで扱います。

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 32 ビット(実質は 24 ビット)サイズです。これを 16 ビットサイズに変更するときに本オプションを使用します。本オプションを選択することにより、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。

注意事項: 本オプションを選択した場合は、near/far の制御を厳密に行う必要があります。near/far の制御は、near / far 修飾子を用いず、const 修飾子を用いることを推奨します。

-fnear_ROM　　-fNROM

生成コードの変更

機　　能: ROM データのデフォルト属性を near 属性にします。

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

-fno_align　　-fNA

生成コードの変更

機　　能:　関数の先頭アドレスのアライメントを行いません。

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

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

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

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

-fno_switch_table　　-fNST

生成コードの変更

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

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

注意事項:　1 関数のコードサイズが 32K バイトを超えるような大きな関数では、switch 文に対してジャンプテーブルを用いたコードを生成すると、正しいアドレスに分岐できない場合があります。この場合、必ず本オプションを指定してください。
なお、本オプションを指定せずに、正しく分岐できないコードを生成した場合に、コンパイラ、アセンブラ、およびリンカでは正しく分岐できないことに関する警告およびエラーメッセージは出力されませんので、ご注意ください。

-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。

注意事項: 通常の用途では、本オプションを使用する必要はありません。

-fnot_reserve_asm　-fNRA

生成コードの変更

機　　能: asm を予約語として扱いません。

補足説明: 同じ機能の_asm は予約語として扱われます。

-fnot_reserve_far_and_near　-fNRFAN

生成コードの変更

機　　能: far、near を予約語として扱いません。

補足説明: 同じ機能の_far、_near は予約語として扱われます。

-fnot_reserve_inline　-fNRI

生成コードの変更

機　　能: inline を予約語として扱いません。

補足説明: 同じ機能の_inline は予約語として扱われます。

-fsmall_array　　-fSA

生成コードの変更

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

補足説明:　デフォルトでは far 型配列の要素を参照する場合に、配列のサイズが不明であれば添字を 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。

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

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

-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。

注意事項:　本オプションは通常、使用する必要はありません。

-fuse_DIV　　-fUD

生成コードの変更

機　　能:　除算に対する生成コードを変更します。

補足説明:　除算時に、被除数が 4 バイト値、除数が 2 バイト値で、かつ、演算結果が 2 バイト値の場合や、被除数がバイト値、除数が 1 バイト値で、かつ、演算結果が 1 バイト値の様な演算を行う場合に M32C/90,80,M16C/80,70 シリーズの"div.w(divu.w)"及び"div.b(divu.b)"命令を生成します。

注意事項:
(1) 本オプションを選択した場合に、除算結果がオーバーフローすると ANSI の規定とは異なる動作になります。
(2) M32C シリーズの div 命令は演算結果がオーバーフローした場合、結果は不定になります。このため"-fuse_DIV(-fUD)”を選択せずにコンパイルを行った場合は、被除数が 4 バイト、除数が 2 バイトで、かつ、結果が 2 バイトのような場合もランタイムライブラリを呼び出します。

-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。

注意事項:
(1) コンパイル・アセンブル時にオプション-M82 を選択した場合は、リンク時に標準関数ライブラリ nc382lib.lib（コンパイルオプション-fsizet_16、-fptrdifft_16 選択時は、nc382_16.lib）を使用してください。
(2) M32C80 をプリディファインドします。
(3) M16C/80,70 シリーズ、および M32C/90 シリーズ用の C ソースプログラムの場合は、本オプションを選択しないでください。

-M90

生成コードの変更

機　　能:　M32C/90 シリーズに対応したコードを生成します。

注意事項:
(1) コンパイル・アセンブル時にオプション-M90 を選択した場合は、リンク時に標準関数ライブラリ nc390lib.lib（コンパイルオプション-fsizet_16、-fptrdifft_16 選択時は nc390_16.lib）を使用してください。
(2) M32C90 をプリデファインドします。
(3) M16C/80,70 シリーズ、および M32C/80 シリーズ用の C ソースプログラムの場合は、本オプションを選択しないでください。

-fsizet_16　　-fS16

型定義のビットサイズ変更

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

注意事項:
(1) 本オプションを選択した場合は、リンク時に以下の標準関数ライブラリを使用してください。
- M32C/90 シリーズ
  nc390_16.lib
- M32C/80 シリーズ
  nc382_16.lib
- M16C/80, /70 シリーズ
  nc308_16.lib

(2) 本オプションを選択する場合は、全ての C ソースファイルに対して本オプションを選択してください。
(3) 本オプションを選択する場合は、コンパイルオプション-fptrdifft_16(-fP16)も選択してください。

-fptrdifft_16　　-fP16

生成コードの変更

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

注意事項:　(1) 本オプションを選択した場合は、リンク時に以下の標準関数ライブラリを使用してください。
- M32C/90 シリーズ
  nc390_16.lib
- M32C/80 シリーズ
  nc382_16.lib
- M16C/80, /70 シリーズ
  nc308_16.lib

(2) 本オプションを選択する場合は、全ての C ソースファイルに対して本オプションを選択してください。
(3) 本オプションを選択する場合は、コンパイルオプション-fsizet_16(-fS16)も選択してください。

-fuse_strings　　-fUS

生成コードの変更

機　　能:　ストリングス命令を使用したコードを生成します。

注意事項:　本オプションは、割り込み制御レジスタの設定内容を把握した上で選択してください。

-fuse_product_sum　　-fUPS

生成コードの変更

機　　能:　積和演算命令を使用したコードを生成します。

注意事項:　本オプションは、割り込み制御レジスタの設定内容を把握した上で選択してください。

A.2.7　ライブラリ指定オプション

【表A.9】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.9　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>

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(コンパイルオプション"-M82"指定時は"nc382lib.lib", "-M90"指定時は"nc390lib.lib"をリンクします)。
(4)　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.10】にnc308 の言語仕様に関する記述の間違いに対して警告(ウォーニングメッセージ)を出力する起動オプションを示します。

表A.10 警告オプション

オプション	短縮形	機能
-Wall	なし	検出可能な警告("-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。
-Wccom_max_warnings =*ウォーニング回数*	-WCMW	ccom308 の出力するウォーニングの回数の上限を指定できます。
-Werror_file<*ファイル名*>	-WEF	タグファイルを出力します。
-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、ウォーニングを出力します。
-Wmake_tagfile	-WMT	error および warning が発生した場合はイル毎にタグファイルを出力します。
-Wnesting_comment	-WNC	コメント中に"/*"を記述した場合に警告を出します。
-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。
-Wno_used_argument	-WNUA	引数を持つ関数を定義した場合に、使用していない引数に対してウォーニングを出力します。
-Wno_used_function	-WNUF	リンク時に未使用のグローバル関数を表示します。
-Wno_used_static_function	-WNUSF	コード生成が不要な static 関数名を表示します。
-Wno_warning_stdlib	-WNWS	"-Wnon_prototype"指定時や"-Wall"指定時に本オプションを指定すると、「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。
-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。
-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。
-Wstop_at_link	-WSAL	リンク時にウォーニングが発生した場合、アブソリュートモジュールファイルの生成を抑止します。
-Wstop_at_warning	-WSAW	ウォーニング発生時にコンパイル処理を停止します。
-Wundefined_macro	-WUM	#if の中で未定義のマクロを使用した場合に警告します。
-Wuninitialize_variable	-WUV	初期化されていない auto 変数に対してウォーニングを出力します。
-Wunknown_pragma	-WUP	サポートしていない#pragma を使用した場合、警告を出します。

-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。

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

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

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

警告オプション

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

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

注意事項: ウォーニングの出力の上限回数は、0 回以上で指定してください。また、指定回数の省略はできません。0 回を指定するとウォーニング出力を完全に抑止します。

-Werror_file <*ファイル名*>

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>

注意事項: 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。

-Wlarge_to_small　　-WLTS

警告オプション

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

補足説明:　各型の負数の境界値では、型に収まる数値であっても、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、signed int 型に収まる値ですが、"-"と"32768"に分解すると、"32768"は signed int 型に収まらないため、"signed long 型"になります。従って即値"-32768"は、signed long 型です。
このため、「 int i = -32768; 」のような記述に対して、警告が出力されます。

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

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

-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:　本オプションを選択した場合に"-Werror_file <ファイル名> (-WEF) "を同時に指定した場合は、エラーとなります。

-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。

-Wno_stop　　-WNS

警告オプション

機　　能:　エラーが発生してもコンパイル作業を停止しません。

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

注意事項:　記述によるエラーが原因で System Error が発生する場合があります。その場合は、本オプションを使用している場合であってもコンパイル作業が停止します。

-Wno_used_argument　　-WNUA

警告オプション

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

-Wno_used_function　　-WNUF

警告オプション

機　　能:　未使用のグローバル関数を、リンク時に表示します。

注意事項:　本オプションを選択する場合は、必ず"-finfo"オプションも同時に指定してください。
リンク用に-U オプションを指定している場合は、本オプションは必要ありません。

-Wno_used_static_function　　-WNUSF

警告オプション

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

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

注意事項:　(1)　下記に示すように配列の初期化子に関数名を記述した場合は、プログラム動作時に参照されない関数であってもコンパイラは参照されるものとして処理します。下記の例では、関数 f4 と f5 は参照されませんが、コンパイラはこれらの関数を参照されるものとして処理します。

例:

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

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

-Wno_warning_stdlib　　-WNWS

警告オプション

機　　能:　"-Wnon_prototype"や、"-Wall"と同時に本オプションを選択すると、「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。

-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。

-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。

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標準出力に出力されます。

-Wstop_at_link　　-WSAL

警告オプション

機　　能: リンク時にウォ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。

-Wstop_at_warning　　-WSAW

警告オプション

機　　能: コンパイル時にウォーニングが発生した場合、コンパイルを停止し、コンパイラの終了コード "10"を返します。

補足説明: デフォルトでは、コンパイル時にウォーニングが発生した場合、コンパイルの終了コードは、"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、ウォ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。

-Wundefined_macro　　-WUM

警告オプション

機　　能: #if の中で未定義のマクロを使用した場合に警告します。

-Wuninitialize_variable　　-WUV

警告オプション

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

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

-Wunknown_pragma　　-WUP

警告オプション

機　　能:　サポートしていない＃pragma を使用した場合、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。

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

A.2.9 アセンブル / リンクオプション

【表A.11】にas308 及びln308 のオプションを選択する起動オプションを示します。

表A.11 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。
-ln308△< オプション>	リンクコマンド ln308 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。

-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 のオプションを選択します。
2 個以上のオプションを選択する場合は、"(ダブルクオーテーション)で囲んでください。

書　　式: 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>

注意事項: as308 の-.、-C、-M、-O、--mode60p、-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 のオプションを選択します。
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選択することができます。2 個以上のオプションを選択する場合は、 "(ダブルクオーテーション)で囲んでください。

書　　式: 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>

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、M32C 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 (1/2)

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

asm 関数	(1) C 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。 ● 日本語のカタカナ半角文字は使用できません。

関数のデフォルト引数宣言	関数の引数にデフォルト値を定義できます。
	記述例 1:
	extern int func(int=1, char=0);
	記述例 2:
	extern int func(int=a, char=0);
	● デフォルト値として変数を記述する時は、関数を宣言するよりも前にデフォルト値として使用する変数の宣言を行ってください。
	● デフォルト値は引数の後ろから順に埋めてください。

表B.2　拡張機能 (2/2)

拡張機能	機能の内容
inline 記憶クラスのサポート	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。
C++風コメント	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、M32C シリーズの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修飾弧

M32C 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。

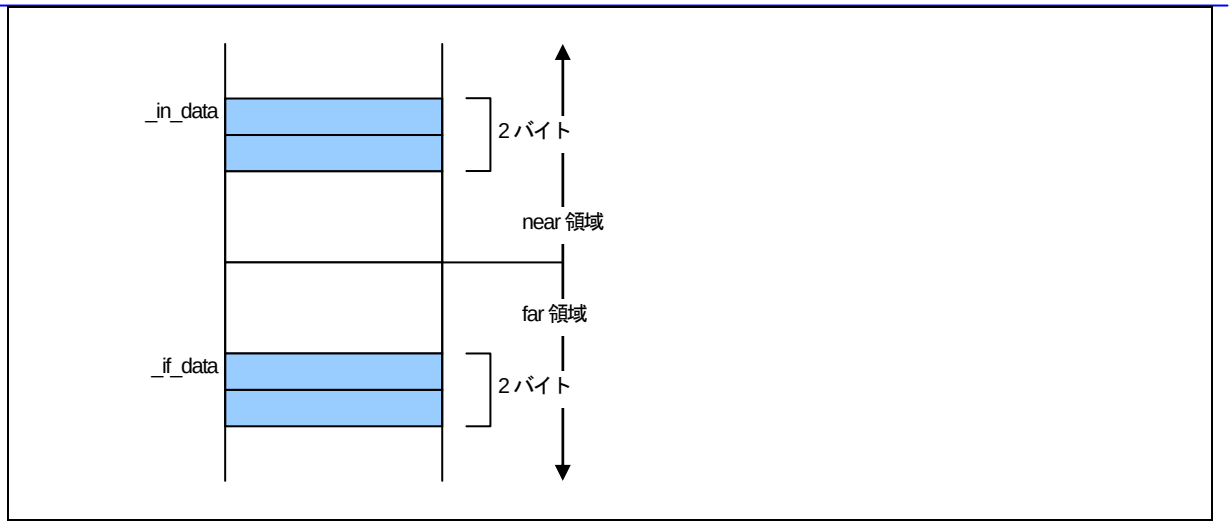
```
型指定子△near 又は far△変数;
```

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;

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

図B.2　変数の宣言例



図B.3　変数のメモリ配置

B.1.3　ポインタ型変数の宣言書式

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

```
例:
        int         * ptr;
```

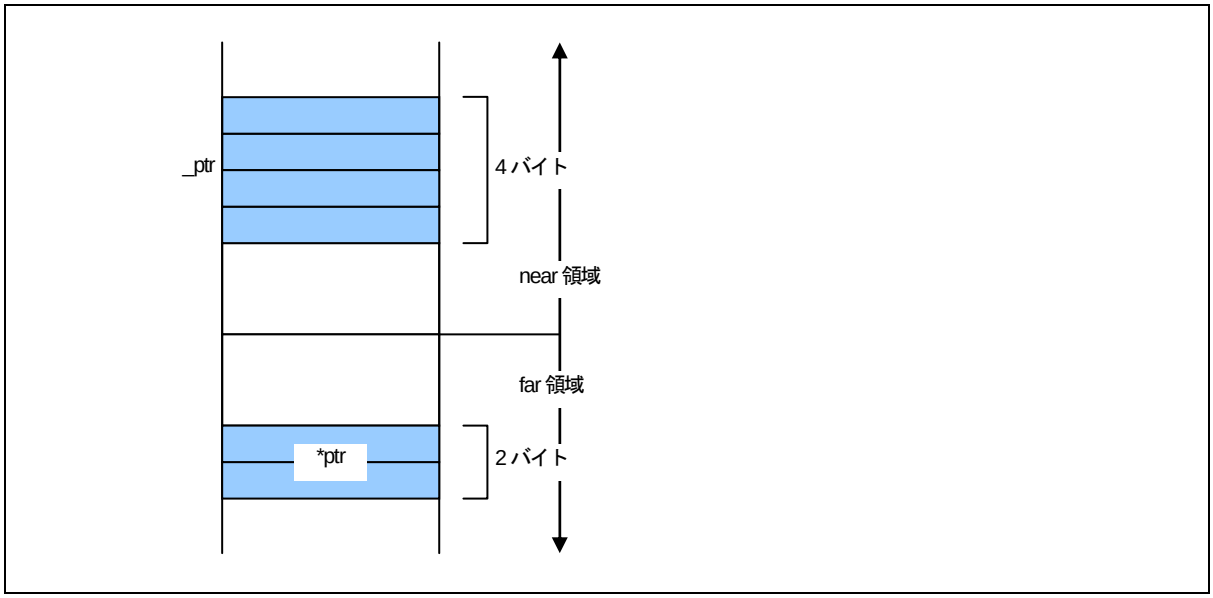
図B.4　ポインタ型変数の宣言例 (1)

変数の配置はnear、ポインタ変数の型は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 4 バイトの変数です。ptr自身はnear領域に配置されます。上記例のメモリ配置を【図B.6】に示します。



図B.6　ポインタ型変数のメモリ配置

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

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

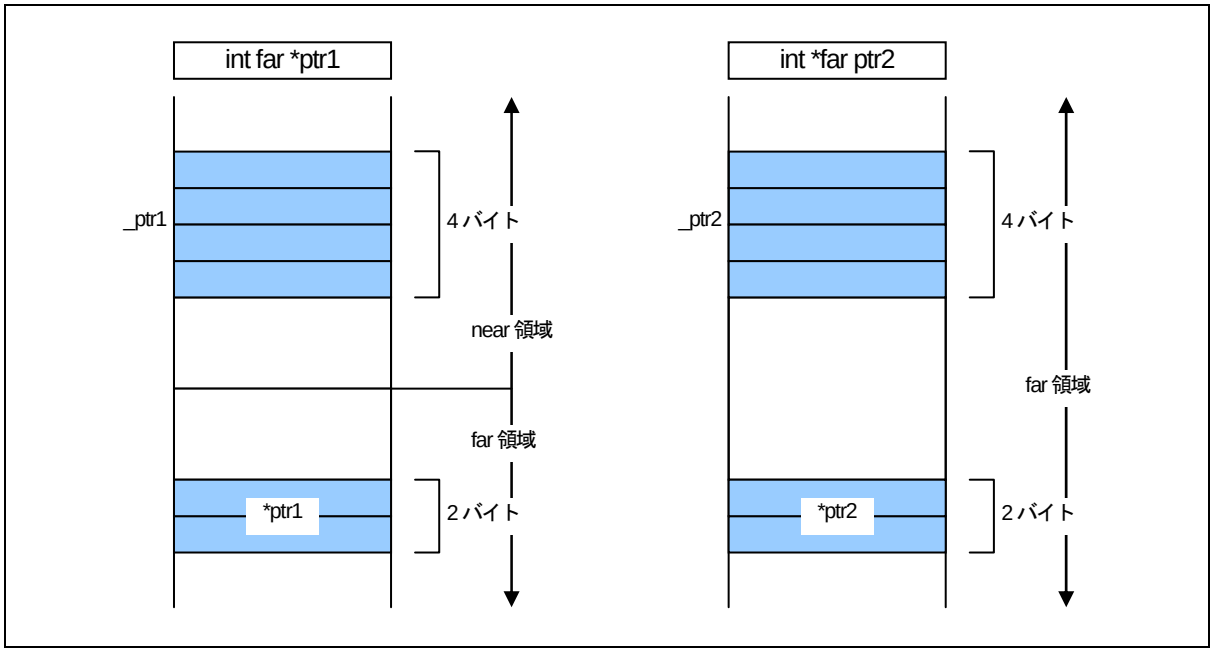
図B.7　アドレスを扱うポインタ型変数の宣言例 (1)

先にも説明したようにnear/farの指定がない場合は、変数の配置を"near"、変数の型を"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 4 バイト型の変数で、変数自身はnear領域に配置されます。例 2 では、変数ptr2 はfar領域にあるint型変数を指し示す 4 バイト型の変数で、変数自身は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 属性を指定するとウォ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

near/far属性を指定しない場合、NC308 では関数は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.3】)。

表B.3　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;

void        main(void)
{
            f_ptr = n_ptr;              /* near ポインタを far ポインタに代入 */
              :
            (省略)
              :
            func ( n_ptr );             /* 引数に far ポインタを持つ関数として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ウォーニングメッセージ(assign far pointer to near pointer,bank value ignored)を出力します。

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

void        main(void)
{
            n_ptr = f_ptr;              /* far ポインタを near ポインタに代入 */
              :
            (省略)
              :
            func ( f_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ウォ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

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場合は、NC308 はウォ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 Series Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Electronics Corp.
and Renesas Solutions Corp., All rights reserved.
smp.c
[Warning(ccom):smp.c,line 3] function must be far
===> {
func
%
```

図B.12　メッセージの表示例

B.1.9　複数の宣言でnear/farの確定を行う機能

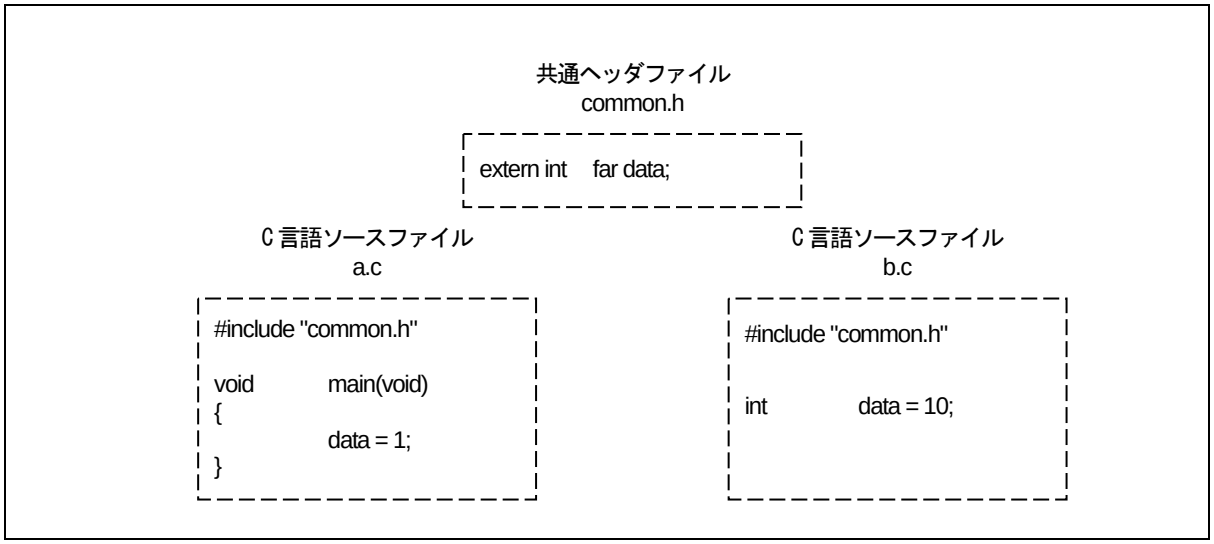
【図B.13】に示すように同一の変数に対して複数の宣言を行った場合、変数の型の情報が結合された型として解釈されます。

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

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


この宣言は、以下の宣言として解釈されます。

extern int    far idata = 10;

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

図B.13　変数の宣言の結合機能

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

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



図B.14　共通ヘッダファイルの宣言例

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修飾子と同じように扱われます。したがって、【図B.15】に示す記述はエラーとなります。

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

図B.15　変数の宣言例

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

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

図B.16　asm 関数の記述例 (1)

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

```
asm( );
```

図B.17　asm 関数の記述例 (2)

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

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

図B.19 FB オフセット指定の記述例

auto変数の参照例とコンパイル結果を【図B.20】に示します。

C 言語ソースファイル:

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

            asm("       MOV.W     $$[FB], R0", idata);
            asm("       CMP.W     #00001H ,R0");
            (以下省略)
                  :
}
```

アセンブ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
            ._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を確認してください。

B.2.3　レジスタ変数のレジスタ名の指定

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。[2]

```
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;

            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。

[2] 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(void)
{
                :
            asm("       MOV.W     R0, $$", idata);
                :
            asm("       MOV.W     R0, $$", a[2]);
                :
            asm("       MOV.W     R0, $$", s.i);
            (以下省略)
                :
}
```

図B.27　シンボル名指定の記述例

extern変数及びstatic変数の参照例を【図B.28】示します。

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

void          func(void)
{
              static int      s_val;

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


アセンブリ言語ソースファイル(コンパイル結果):
_func:
              ._line 7
;## # C_SRC : asm("    mov.w       #01H,$$",ext_val);
;#### ASM START
              mov.w       #01H,_ext_val                    ← シンボル_ext_val への転送
              ._line 8
;## # C_SRC : asm(" mov.w          #01H,$$",s_val);
              mov.w       #01H,___S0_s_val                 ← シンボル_ext_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[3]、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、C言語で記述した変数をasm関数中で使用することができます。[4]

```
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;

void          func(void)
{
              int           f_val;
              register int  r_val;
              static int    s_val;

              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):

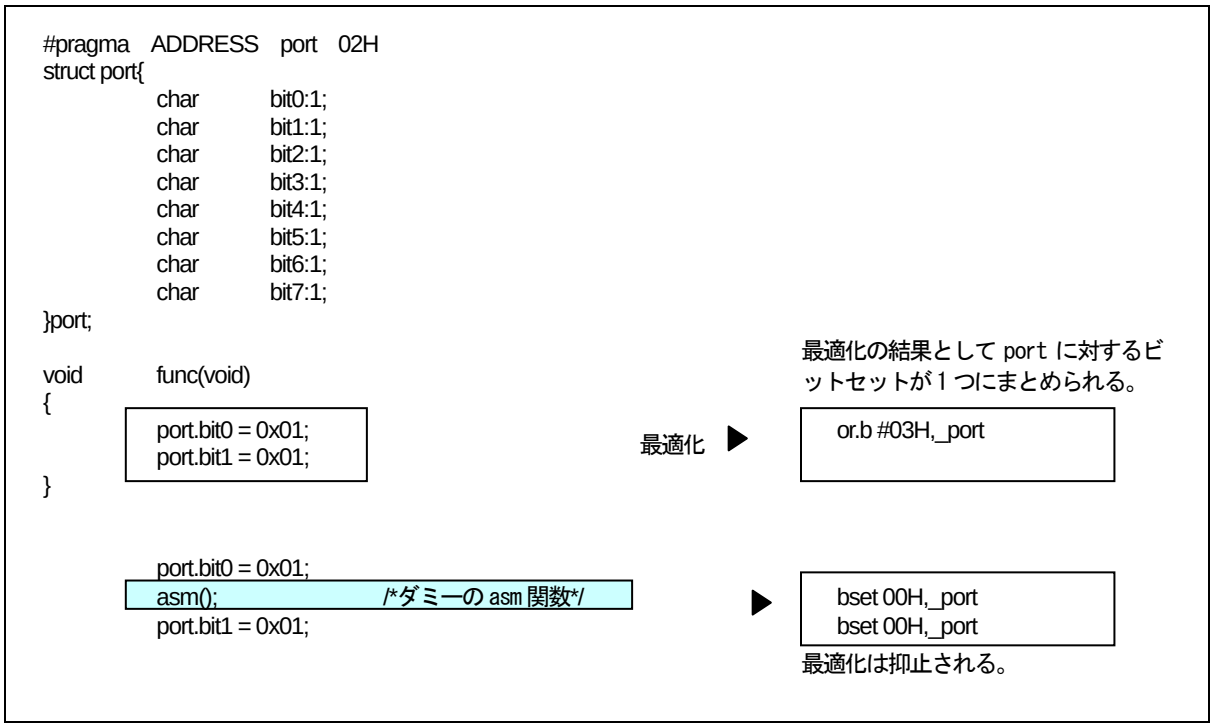
```
              .glb        _func
_func:
              enter       #02H
              pushm       R1
              ._line 9
;## # C_SRC : asm("   mov.w       #1, $@", e_val);
;#### ASM START
              mov.w       #1, _e_val:16                         ← extern 変数の参照
              ._line 10
;## # 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

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

B.2.6 最適化の部分的な抑止方法

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



図B.34 ダミーの asm 関数による最適化の抑止例

B.2.7 asm関数に関する注意事項

a. asm 関数の拡張機能

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

(1) 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)

(2) register 記憶クラスの指定

本コンパイラでは、register記憶クラスを指定することができます。 register記憶クラスで指定した変数でかつオプション-fenable_register(-fER)を指定してコンパイルした場合にasm関数で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。

(3) 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、【図B.37】のように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。

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

図B.37　正しく参照することができない例

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

(4) asm 関数内でのブランチについて

本コンパイラでは、レジスタの生存区間、変数の生存区間についてプログラムフローを解析して処理を行っているため、asm 関数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。

c. 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。
- 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.38】に示すSBレジスタを元に戻す記述を行なってください。また、変更している間に呼び出す関数や、その間に発生する割り込み処理について十分考慮してください。

```
asm("      .SB      0);
asm("      LDC      #0H, SB");                    ← SB 変更
asm("      MOV.W    R0, _port[SB]");
             :
           (省略)
             :
asm("      .SB      __SB__);
asm("      LDC      #__SB__,SB");                 ← SB を元に戻す
```

図B.38　変更したスタティックベースレジスタの復帰方法

- フレームベースレジスタ(FB)は、スタックフレームポインタとして使用しますので、asm 関数で変更しないでください。

d.　ラベルの記述に関する注意事項

本コンパイラが生成するアセンブラソースファイルでは、【図B.39】に示す形式の内部ラベルが出力されます。したがって、asm関数を用いてこの規定と重複する可能性のあるラベルを記述しないでください。

```
アルファベットの大文字 1 文字と数字で構成されるラベル名:

        A1:
        C9830:

_(アンダースコア)で始まる 2 文字以上のラベル名

        __LABEL:
        ___START:
```

図B.39　asm 関数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、NCKIN、NCKOUT 共に SJIS となっています。

- 入力コード系指定環境変数..... NCKIN
- 出力コード系指定環境変数..... NCKOUT

環境変数の設定例を【図B.40】に示します。

```
autoexec.bat の中に以下の内容を記述します。

set NCKIN=SJIS
set NCKOUT=SJIS
```

図B.40　環境変数 NCKIN と NCKOUT の設定例

本コンパイラでは入力漢字コードをプリプロセッサで処理します。プリプロセッサで EUC コードに変換し、その後コンパイラ内の字句解析部終段で、環境変数を基に変換し出力します。

B.3.3　文字列中の日本語文字

文字列に日本語文字を記述するときの書式を【図B.41】に示します。

```
L"漢字文字列"
```

図B.41　文字列中の漢字コード記述書式

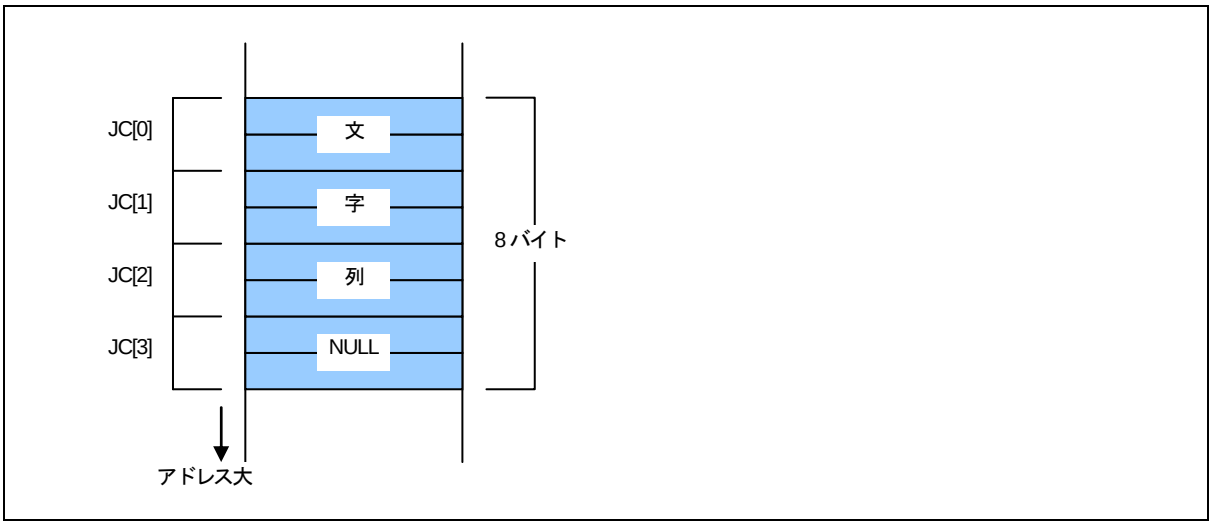
日本語を通常の文字列と同様に"漢字文字列"と記述した場合、文字列の操作では char 型へのポインタ型として扱われます。したがって、2 バイト文字として操作することはできません。

2 バイト文字として扱う場合は、文字列の先頭に"L"を付加してwchar_t型へのポインタ型として使用します。wchar_t型は、標準ヘッダファイルstdlib.hの中でunsigned short型にtypedefしています。【図B.42】に日本語の文字列の記述例を示します。

```
#include <stdlib.h>

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

            (以下省略)
                :
```

図B.42　日本語文字列の記述例

【図B.42】中の(1)の初期化された文字列のメモリ配置を【図B.43】に示します。



図B.43　wchar_t 型文字列のメモリ配置

B.3.4　文字定数としての日本語文字

文字定数として日本語文字を記述するときの書式を【図 B.44】に示します。

```
L'漢'
```

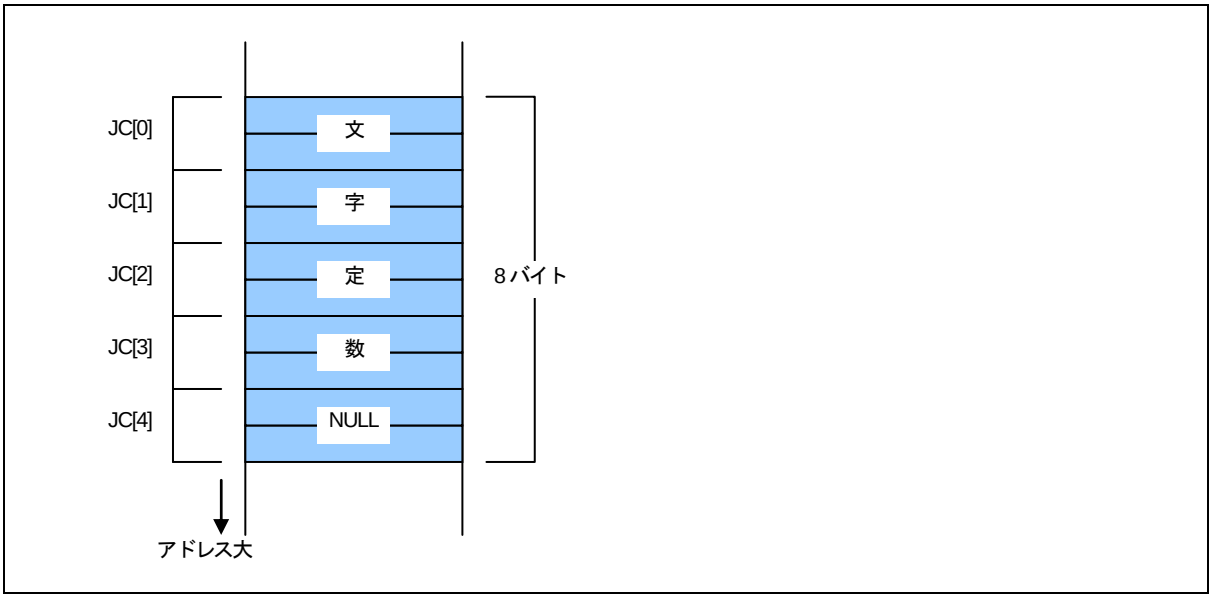
図B.44　文字列中の漢字コード記述書式

文字列と同様に、文字定数の前に"L"を付加した場合は、wchar_t 型として扱われます。文字定数として'文字'のように複数の文字を記述した場合は、最初の文字「文」のみが文字定数として扱われます。
【図B.45】に日本語の文字定数の記述例を示します。

```
#include <stdlib.h>

void        func(void)
{
            wchar_t     JC[5];

            JC[0] = L'文';
            JC[1] = L'字';
            JC[2] = L'定';
            JC[3] = L'数';

            (以下省略)
                :
```

図B.45　日本語文字定数の記述例

【図B.45】中の文字定数を代入した配列のメモリ配置を【図B.46】に示します。



図B.46　配列に代入した wchar_t 型文字定数のメモリ配置

B.4 関数のデフォルト引数宣言

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.47】に関数のデフォルト引数宣言時の書式を示します。

```
記憶クラス指定子△型宣言子△宣言子([仮引数[=デフォルト値あるいは変数],...]);
```

図B.47　関数のデフォルト引数宣言書式

デフォルト引数宣言の例を【図B.48】に、【図B.48】で示すサンプルプログラムのコンパイル結果を【図B.49】に示します。

```
int     func( int i=1 , int j=2 );      ← 関数 func に仮引数のデフォルト値を第 1 引数 : 1、
                                          第 2 引数 : 2 に宣言しています。

void    main(void)
{
        func();                         ← 実引数は第 1 引数 : 1、第 2 引数 : 2 になります。
        func(3);                        ← 実引数は第 1 引数 : 3、第 2 引数 : 2 になります。
        func(3,5);                      ← 実引数は第 1 引数 : 3、第 2 引数 : 5 になります。
}
```

図B.48　関数のデフォルト引数の宣言例 (sample.c)

```
;### C_SRC :          {
                .glb        _main
_main:
                ._line      5
;### C_SRC :                      func();
                push.w      #0002H                  ← 第 2 引数:2
                mov.w       #0001H,R0               ← 第 1 引数:1
                jsr         $func
                add.l       #02H,SP
                ._line      6
;### 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
                ._line      7
;### 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
                ._line      8
;### C_SRC :          }
                rts
                    :
                (省略)
                    :

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。
この例では引数をレジスタ渡しで処理しています。
```

図B.49　smp1.c のコンパイル結果 (sample.a30)

関数の引数に変数を記述することができます。デフォルト引数の変数指定の例を【図B.50】に、【図B.50】で示すサンプルプログラムのコンパイル結果を【図B.51】に示します。

```
int         near sym ;
int         func( int i = sym);             ← デフォルトの引数を変数で指定しています。

void        main(void)
{
            func();                         ← 変数(sym)を引数として関数呼び出しを行ないます。
}
                :
            (省略)
                :
```

図B.50　デフォルト引数の変数指定例 (smp2.c)

```
_main:
        ._line 6
        mov.w   _sym,R1          ← 変数(sym)を引数として関数呼び出しを行ないます。
        jsr     $func
        ._line 7
        rts
```

図B.51　smp2.c のコンパイル結果 (smp2.a30)

B.4.3　関数のデフォルト引数宣言の規定事項

関数のデフォルト引き数の宣言を行なう場合は、以下の点に注意してください。

a.　複数の引数にデフォルト値を指定する時

関数の引き数が複数ある場合にデフォルト値を指定する場合は、必ず引き数の後ろから埋めてください。【図B.52】に不適切な記述の例を示します。

```
void    func1(int i, int j=1, int k=2);     /* 正しい記述です*/
void    func2(int i, int j, int k=2);       /* 正しい記述です*/
void    func3(int i = 0, int j, int k);     /* 誤った記述です*/
void    func4(int i = 0, int j, int k = 1); /* 誤った記述です*/
```

図B.52　プロトタイプ宣言の記述例

e.　変数のデフォルト値を指定する時

デフォルト値として変数を指定する場合は、指定する変数の宣言を行なった後に関数のプロトタイプ宣言を行なってください。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.53】に宣言時の書式を示します。

```
inline△型指定子△関数;
```

図B.53 inline 記憶クラスの宣言書式

関数の宣言例を【図B.54】に、コンパイル結果【図B.55】を示します。

```
inline int    func(int i)                    ← インライン関数の宣言及び定義部
{
              return i++;
}

void          main(void)
{
              int           s;

              s = func(s);                   ← インライン関数呼び出し部
}
```

図B.54 インライン関数のサンプルプログラム (sample.c)

```
                .SECTION  program,CODE,ALIGN
                ._file    'sample.c'
                .align
                ._line    7
  ;### C_SRC :  {
                .glb      _main
  _main:
                enter     #02H
                pushm     R1
                ._line    10
  ;### C_SRC :                    s = func(s);
                mov.w     -2[FB],R0  ;  s
                ._line    3
  ;### C_SRC :                    return i++;
                mov.w     R0,R1
                add.w     #0001H,R0
                ._line    10
  ;### C_SRC :                    s = func(s);
                mov.w     R1,-2[FB]  ;  s
                ._line    11
  ;### C_SRC :  }
                popm      R1
                exitd
  E1:
                .END
```

← インライン関数が埋め込まれている

図B.55　サンプルプログラムのコンパイル結果 (sample.a30)

B.5.3　inline記憶クラスの規定事項

inline 記憶クラス指定場合は、以下の点に注意してください。

(1)　インライン関数の引数について

インライン関数の引数に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は、コンパイルエラーになります。

(2)　インライン関数の間接呼び出しについて

インライン関数の間接呼び出しをすることはできません。間接呼び出しの記述を行った場合は、コンパイルエラーになります。

(3)　インライン関数の再帰呼出しについて

インライン関数の再帰呼出しをすることはできません。再帰呼出しの記述を行った場合、コンパイルエラーになります。

(4)　インライン関数の定義について

関数に対してinline記憶クラスを指定する場合は、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。本コンパイラでは、【図B.56】の記述はエラーになります。

```
inline void   func(int i);

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

エラーメッセージ:
[Error(ccom):sample.c,line 5] inline function's body is not declared previously
===> func(1);
Sorry, compilation terminated because of these errors in main().

図B.56　インライン関数の不適切な記述例 (1)

また、ある関数を通常の関数として使用した後に、その関数をインライン関数として定義した時は、本コンパイラではエラーなります(【図B.57】)。

```
int           func(int i);

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

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

エラーメッセージ:
[Error(ccom):in.c,line 9] inline function is called as normal function before
===>{

図B.57　インライン関数の不適切な記述例 (2)

(5)　インライン関数のアドレスについて

インライン関数は、アドレスを持ちません。このため、インライン関数に対して&演算子を使用した場合は、エラーになります(【図B.58】)。

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

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

                f = &func;
}
```

エラーメッセージ:
[Error(ccom):sample.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.58　インライン関数の不適切な記述例 (3)

(6)　static データの宣言

インライン関数内で static データを宣言した場合は、宣言した static データの実体はファイル単位で確保されます。

このため、複数のファイルにまたがったインライン関数ではデータのアクセス領域が異なります。インライン関数内で使用する static データは関数外で宣言してください。

本コンパイラでは、インライン関数内でstatic宣言をした場合は、警告になります。また、インライン関数内のstatic宣言は推奨しません(【図B.59】)。

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

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

ウォーニングメッセージ:
[Warning(ccom):smp.c,line 3] static valuable in inline function
===> static int   i = 0;

図B.59　インライン関数の不適切な記述例 (4)

(7)　デバッグ情報について

本コンパイラではインライン関数に対する C 言語レベルのデバッグ情報を出力しません。このため、インライン関数のデバッグはアセンブリ言語レベルで行なうことになります。

B.6 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"//""以降の記述が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.60】に書式を示します。

```
// コメント
```

図B.60 コメントの記述書式

コメントの記述例を【図B.61】に示します。

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

図B.61 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 に関する拡張機能の内容と規定を一覧表で示します。

a. メモリ配置に関する拡張機能

表B.4 メモリ配置に関する拡張機能 (1/2)

拡張機能	機能の内容
#pragma 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 NC79、NC77 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 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 SBDATA	SB 相対アドレッシングを使用するデータであることを宣言します。 記述形式: #pragma SBDATA△変数名 記述例: #pragma SBDATA sym
#pragma SECTION	本コンパイラが生成するセクション名を変更します。 記述形式: #pragma SECTION△既定ｾｸｼｮﾝ名△変更ｾｸｼｮﾝ名 記述例: #pragma SECTION bss nonval_data
#pragma STRUCT	(1) 指定したタグを持つ構造体のパックを禁止します。 記述形式: #pragma STRUCT△構造体のタグ名△unpack 記述例: #pragma STRUCT TAG1 unpack (2) 指定したタグを持つ構造体のメンバの並べ替えを行い、偶数サイズのメンバを先に配置します。 記述形式: #pragma STRUCT△構造体のタグ名△arrange 記述例: #pragma STRUCT TAG1 arrange

表B.5　メモリ配置に関する拡張機能 (2/2)

拡張機能	機能の内容
#pragma MONITOR[n]	指定された外部変数をRAMモニタ領域専用のセクションに配置することを宣言します。 記述形式: #pragma MONITOR[n]△外部変数名 (n は、1～4 の数字) 記述例: #pragma MONITOR1　i #pragma MONITOR2　c int　i; char　c;

b. 組み込み機器に関する拡張機能

表B.6　組み込み機器に使用するための拡張機能 (1/2)

拡張機能	機能の内容
#pragma ADDRESS	変数を絶対アドレスに割り付けます。 記述形式: #pragma ADDRESS△変数名△絶対アドレス 記述例: #pragma ADDRESS port0 2H
#pragma BITADDRESS	変数を指定した絶対アドレスの指定したビット位置に、割り付けます。 記述形式: #pragma BITADDRESS△変数名△ビット位置, 絶対アドレス 記述例: #pragma BITADDRESS io 1, 100H
#pragma 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	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¦/V]△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) #pragma INTERRUPT /V int_func()

表B.7　組み込み機器に使用するための拡張機能 (2/2)

拡張機能	機能の内容
#pragma PARAMETER	アセンブ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。
#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

表B.8　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

d. その他の拡張機能

表B.9　その他の拡張機能

拡張機能	機能の内容
#pragma __ASMMACRO	アセンブラのマクロで定義した関数を宣言します。 記述形式: #pragma __ASMMACRO△関数名（レジスタ名） 記述例: #pragma __ASMMACRO mul(R0,R2)
#pragma ASM #pragma ENDASM	アセンブリ言語で記述を行う領域を指定します。 記述形式: #pragma△ASM #pragma△ENDASM 記述例: #pragma ASM mov.w R0,R1 add.w 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

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。

- 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
- 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(void)
{
            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.62　#pragma ROM 宣言の使用例

備　考: 本機能は NC79,NC77 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 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: M32C シリーズでは、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。[5]

使 用 例:

```
#pragma   SB16DATA   sym_data
int           far sym_data;

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

図B.63　#pragma SB16DATA 宣言の使用例

備　考: 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。

[5] ROM データに対して"#pragma SBDATA"宣言を行なわないでください。

#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: M32C シリーズでは、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。

使 用 例:

```
#pragma   SBDATA   sym_data
struct sym_data{
                char        bit0:1;
                char        bit1:1;
                char        bit2:1;
                char        bit3:1;
                char        bit4:1;
                char        bit5:1;
                char        bit6:1;
                char        bit7:1;
}sym_data;

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

図B.64　#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。

#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。

- program セクション、data セクション、rom セクションおよび bss 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

void      func(void)
{
          (省略)
}

#pragma   SECTION   data   data2
int       j;                                  ← data2_NE セクションに配置

void      sub(void)
{
          (省略)
}
```

図B.65　#pragma SECTION 宣言の使用例

#pragma SECTION

セクション名変更機能

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。

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、ファイル単位でのみ、セクション名を変更できました。
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.66】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。

```
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.66　構造体メンバの配置例 (1)

規　定:　(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.66】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.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
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.67　構造体メンバの配置例 (2)

【図B.67】に示したように、構造体メンバのサイズの合計が奇数バイトのとき、#pragma STRUCTを用いることにより最後のメンバ配置位置の後に、1 バイトのパディングが入ります。したがって、#pragma STRUCTでパックを禁止したときの構造体は、すべて偶数バイトのサイズとなります。

#pragma STRUCT

構造体配列制御機能

規　　定:　(2)　メンバ配置の並び換え

本コンパイラではでは、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.67】に示した構造体を#pragmaSTRUCTで配置を並び替えたときの配置例を【図B.68】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.68　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.69　#pragma STRUCT 宣言の使用例

#pragma MONITOR[n](n は 1～4)

RAM モニタ領域の配置指定機能

機　能:　指定された外部変数を RAM モニタ領域専用のセクションに配置することを宣言します。

書　式:　#pragma MONITOR[n]△外部変数名
(n は、1～4 の数字)

規　定

(1) 指定できる変数は外部変数および外部 static 変数のみです。
(2) #pragma MONITOR[n]で宣言された外部変数の領域は、下記のセクションに割り当てられます。
- data_MON[n]_E (初期値を持つ偶数サイズの外部変数を配置)
- data_MON[n]_O (初期値を持つ奇数サイズの外部変数を配置)
- bss_MON[n]_E (初期値を持たない偶数サイズの外部変数を配置)
- bss_MON[n]_O (初期値を持たない奇数サイズの外部変数を配置)
- data_MON[n]_EI (初期値を持つ偶数サイズの外部変数の初期値を配置)
- data_MON[n]_OI (初期値を持つ奇数サイズの外部変数の初期値を配置)

(3) #pragma MONITOR[n]の宣言は外部変数の定義よりも前に行う必要があります。
(4) #pragma MONITOR[n]で宣言した外部変数は他の#pragma 拡張機能と併用することはできません。ただし、#pragma SBDATA と#pragma MONITOR[n]が同時に指定された場合は#pragma SBDATA が優先されます。この際、ウォーニングは出力されません。

注　意

(1) #pragma MONITOR[n]はコンパイラが生成するオペコードに影響を与えません。変数の near/far 属性に注意してください。
(2) RAM モニタ領域専用のセクションに near/far 属性が異なる外部変数が混在してもエラー・警告にはなりません。変数の near/far 属性に注意してください。
(3) RAM モニタ領域専用のセクションにサイズ制限はありません。
(4) #pragma MONITOR[n]により割り当てられるセクションの配置アドレス、および外部変数の初期値設定処理はスタートアッププログラムで記述してください。
(5) 同一外部変数に対して#pragma MONITOR[n]を複数回指定した場合は、最初に指定された#pragma MONITOR[n]が有効になります。
(6) コンパイルオプション-fno_even[-fNE]を指定している場合は、奇数サイズ属性のセクション(例：data_MON1_O)に配置されます。
(7) #pragma SECTION の影響は受けません。
(8) #pragma MONITOR[n]の n が 1～4 以外の場合は無効になります。コンパイルオプション-Wunknown_pragma[-WUP]または-Wall を指定した場合はウォーニングを出力します。
(9) ROM 属性を持つ外部変数は#pragma MONITOR[n]の対象外になります。ただし、コンパイルオプション-fconst_not_ROM[-fCNR]を指定している場合は#pragma MONITOR[n]対象になります。

```
#pramga   MONITOR1i
const          int          i;   <===== 無効
```

(10) #pragma MONITOR[n]はコンパイルオプション-M82 および-M90 指定の影響を受けません。

使 用 例:

```
#pragma MONITOR1        i
#pragma MONITOR2        c
int         i;
char        c;
```

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

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、指示命令.EQU を用いて定義されます。したがいまして、数値の記述形式はアセンブラに依存します。アセンブラの数値表現を以下に示します。

- 2 進数数値の最後に'B'又は'b'を付けます。
- 8 進数数値の最後に'O'又は'o'を付けます。
- 10 進数整数のみで記述します。
- 16 進数数値の最後に'H'又は'h'を付けます。先頭が英文字(A～F)で始まる場合は先頭に 0 を付けます。

規　定:

(1) #pragma ADDRESS により指定された変数に対する extern、static 等の記憶クラスはすべて無効になります。
(2) #pragma ADDRESS により指定された変数は、関数外で定義された変数に対してのみ有効になります。
(3) #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ウォーニングとなり、記述した初期化式は無効となります。
(7) #pragma ADDRESS は通常、I/O 変数に対して使用するため、volatile 指定が無くても、volatile 指定がされているものとして処理されます。

使 用 例:

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

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

図B.70　#pragma ADDRESS 宣言

#pragma ADDRESS

入出力変数の絶対アドレス割り付け機能

注意事項:　【図B.71】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は、#pragma ADDRESS の指定は無効です。

```
char        port;

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

#pragma   ADDRESS   port   100H
```

図B.71　#pragma ADDRESS の指定が無効になる場合

#pragma BITADDRESS

入出力変数のビット位置指定付き絶対アドレス割り付け機能

機　能:　指定した絶対アドレスの指定したビット位置に、変数を割り付けます。

書　式:　#pragma BITADDRESS△変数名△ビット位置, 絶対アドレス

解　説:　この宣言により指定した絶対アドレスは、文字列としてアセンブラファイルに展開され、指示命令.BITEQU を用いて定義されます。したがいまして、数値の記述形式はアセンブラに依存します。アセンブラの数値表現を以下に示します。また、ビット位置の記述可能範囲も以下に示します。

(1) ビット位置
- 0～65535 の範囲で 10 進数のみ。

(2) アドレス
- 2 進数数値の最後に'B'又は'b'を付けます。
- 8 進数数値の最後に'O'又は'o'を付けます。
- 10 進数整数のみで記述します。
- 16 進数数値の最後に'H'又は'h'を付けます。先頭が英文字(A～F)で始まる場合は先頭に 0 を付けます。

規　定:
(1) _Bool 型の変数のみ、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.72　#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

解　説: #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。

	16 ビットレジスタ	24 ビットレジスタ
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

void        func(void)
{
            unsigned char           buff[10];

            dma0 = buff;
}
```

図B.73　#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:

- 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。
    (1) プロトタイプ宣言の引数の数と#pragma INTCALL 宣言の引数の数が一致している必要があります。
    (2) アセンブラ関数の引数に以下の型は宣言できません。
      - 構造体型
      - 共用体型
      - double 型
      - long long 型
    (3) アセンブラ関数のリターン値の型として以下の関数は宣言できません。
      - 構造体、共用体を返す関数
  (3) 呼び出し場合は、引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。
- C 言語で実体を記述した関数を宣言する場合
  (1) #pragma INTCALL 宣言を行う前には、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。
    (1) 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。
    (2) 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.74　#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(void)
{
        int     i, j;

        i = 0x7FFD;
        j = 0x007F;

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

図B.75　#pragma INTCALL 宣言の使用例((2) C 言語で実体を記述した関数を宣言する場合)

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

#pragma INTERRUPT

割り込み関数の記述機能

機　能:　割り込み処理関数の宣言を行います。

書　式:
(1) #pragma INTERRUPT△[/B ¦ /E ¦ /F ¦ /V]△割り込み処理関数名
(2) #pragma INTERRUPT△[/B ¦ /E ¦ /F]△割り込みベクタ番号△割り込み処理関数名
(3) #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。
- [/V]
  割込み関数のベクタテーブルを生成します。主に固定ベクタで使用します。

(3) 宣言時に割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た場合は、ベクタテーブルを自動生成します。

#pragma INTERRUPT

割り込み関数の記述機能

規　　定:

(1) 引数を持つ割り込み処理関数を記述した場合、コンパイル時に警告を出力します。
(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。
(8) /V とその他のスイッチは同時に使用できません。

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

図B.76　異なる割り込みベクタ番号の記述例

(9) #pragma INTTERUPT 宣言で以下に示す宣言と同じ関数を指定した場合、コンパイル時にウォーニングとなります。
- #pragma ALMHANDLER
- #pragma INTHANDLER
- #pragma HANDLER
- #pragma CYCHANDLER
- #pragma TASK

使 用 例:

```
extern int    int_counter;

#pragma   INTERRUPT   /B   i_func

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

図B.77　#pragma INTERRUPT 宣言の使用例

備　　考:

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

#pragma PARAMETER

レジスタ渡しのアセンブラ関数宣言機能

機　能:　引数をレジスタに格納して渡すアセンブラ関数を宣言します。

書　式:　#pragma PARAMETER△[/ 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(64 ビット整数型)、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。

(1) 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。

(2) アセンブラ関数の引数の型として、以下の型は宣言できません。

- 構造体型
- 共用体型
- double 型
- long long 型

(3) アセンブラ関数のリターン値の型として以下の関数は宣言できません。

- 構造体、共用体を返す関数

(3) #pragma PARAMETER で指定した関数は、常に先頭に '_' を付加したシンボル名になります。

#pragma PARAMETER

レジスタ渡しのアセンブラ関数宣言機能

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(void)
{
            int          i, j;

            i = 0x7FFD;
            j = 0x007F;

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

図B.78　#pragma PARAMETER 宣言の使用例

#pragma SPECIAL

スペシャルページサブルーチン呼び出し関数宣言機能

機　能:　スペシャルページサブルーチン呼び出し(JSRS 命令)の関数を宣言します

書　式:
(1) #pragma SPECIAL△[/ C]△呼び出し番号△関数名()
(2) #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、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。
(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。

```
#pragma   SPECIAL   func(vect=20)
#pragma   SPECIAL   func(vect=30)          /* 呼び出し番号 30 が有効 */
```

図B.79　異なる呼出し番号の記述例

(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(void)
{
              int i, j;

              i = 0x7FFD;
              j = 0x007F;

              func( i, j );              ← スペシャルページサブルーチン呼び出し
}
```

図B.80　#pragma SPECIAL 宣言の使用例

備　考:　付属のスタートアップファイルをご使用になる場合は、vector セクションの内容を変更して下さい。変更方法の詳細は、「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、コンパイル時にウォ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.81　#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、コンパイル時にウォ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.82　#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、コンパイル時にウォ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.83　#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、コンパイル時にウォ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)
{
                  :
              (省略)
                  :
              sta_tsk(ID_idle);
              sta_tsk(ID_tsk1);
              /* ext_tsk(); */
}

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

図B.84　#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 __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。

_Bool 型、char 型:	R0L	float 型:	R2R0
int、short 型:	R0	double 型:	R3R2R1R0
long 型:	R2R0	long long 型:	R3R1R2R0

(5) アセンブラマクロ内で内容が変更されるレジスタは、アセンブラマクロの先頭で退避して、復帰直前に復帰してください(戻り値の格納レジスタの退避・復帰は不要です)。

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

#pragma ASM～#pragma ENDASM

インラインアセンブラ指定機能

機　能:　アセンブリ言語で記述を行なう領域を指定します。

書　式:
#pragma ASM
アセンブリ言語記述
#pragma ENDASM

解　説:　#pragma ASM と#pragma ENDASM の間の領域を直接、アセンブリ言語ファイルにそのまま出力します。
#pragma ASM を記述する際は、必ず#pragma ENDASM を組み合わせて記述してください。#pragma ASM と対になる#pragma ENDASM がない場合は NC308 は処理を中断します。

規　定:
(1) アセンブリ言語記述において、レジスタの内容を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は、push 命令・pop 命令を使用して、レジスタ内容の退避・復帰を行ってください。
(2) "#pragma ASM～#pragma ENDASM"内では、引数および auto 変数を参照しないでください。
(3) "#pragma ASM～#pragma ENDASM"内でフローに影響を与えるようなブランチ(条件ブランチ含む)を記述しないようにしてください。

使 用 例:

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

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

#pragma   ASM
            FCLR        I
LOOP1:
            MOV.W       #0FFH,R0
                :
            (省略)
                :
            FSET        I
#pragma   ENDASM
}
```

この領域をアセンブリ言語ファイルにそのまま出力します。

図B.86　#pragma ASM(ENDASM)記述例

補　足:　#pragma ASM～#pragma ENDASM までに記述されたアセンブリ言語プログラムは、C 言語プリプロセッサの処理対象となります。

#pragma 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: #pragma JSRA△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。
コンパイルオプション"-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: コンパイルオプション"-fJSRW"を指定しない場合は、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.87　#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。この場合は、本宣言を行わないでください。

使 用 例:

```
#pragma   JSRW   func()

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

図B.88　#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。間接呼び出しの時は、効果がありません。

#pragma PAGE

. PAGE 指示令出力機能

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: 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、指示命令.PAGE を出力します。アセンブラによりアセンブラリス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) アセンブラ指示命令.PAGE のヘッダに指定する文字列の指定はできません。
(2) auto 変数の宣言中に#pragma PAGE を記述できません。

使 用 例:

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

            for(i=0; i < 10;i++){
                        func2();
            }
#pragma PAGE
            i++;
}
```

図B.89　#pragma PAGE 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【図B.90】のように、C言語プログラム中にC言語の関数と同じ書式で記述することができます。
アセンブラマクロ関数機能を使用する場合は、必ず asmmacro.h をインクルードしてください。

```
#include   <asmmacro.h>                  /* アセンブラマクロ関数の定義ファイルをインクルード */
long       l;
char       a[20];
char       b[20];

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

図B.90　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。

ABS

機　能: val の絶対値を返します。

書　式:
```
#include <asmmacro.h>

static signed char abs_b( signed char val );      /* 8 ビットでの演算の場合 */
static signed int abs_w( signed int val );        /* 6 ビットでの演算の場合 */
```

DADC

機　能: al1 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 );       /* 8 ビットでの演算の場合 */
static int dadc_w( int val1, int val2 );          /*16 ビットでの演算の場合 */
```

DADD

機　能: val1 と val2 の 10 進加算の結果を返します。

書　式:
```
#include <asmmacro.h>

s char dadd_b(char val1, char val2);      /* 8 ビットでの演算の場合 */
int dadd_w(int val1, int val2);           /*16 ビットでの演算の場合 */
```

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>

/*符号付きの 16 ビット/8 ビットの演算の場合 */
static signed char div_b(signed char val1, signed int val2);

/*符号付きの 32 ビット/16 ビットの演算の場合 */
static signed int div_w(signed int val1, signed long val2);

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>

/*符号なしの 16 ビット/8 ビットの演算の場合*/
static unsigned char divu_b(unsugned char val1, unsigned int val2);

/*符号なしの 32 ビット/16 ビットの演算の場合*/
static unsigned int divu_w(unsigned int val1, unsigned long val2);

DSBB

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

書　　式:　　#include <asmmacro.h>

static　char　dsbb_b(char val1, char val2);　　/* 8 ビットでの演算の場合 */
static　int　dsbb_w(int val1, int val2);　　/*16 ビットでの演算の場合 */

DSUB

機　　能:　　val2 から val1 の ボローなし 10 進減算の結果を返します。

書　　式:　　#include <asmmacro.h>

```
static   char   dsub_b( char val1, char val2 );      /* 8 ビットでの演算の場合 */
static   int    dsub_w( int val1, int val2 );        /*16 ビットでの演算の場合 */
```

MAX

機　　能:　　val1 と val2 を比較して小さい方の値を返します。

書　　式:　　#include <asmmacro.h>

```
static   char   min_b( char val1, char val2 );       /* 8 ビットでの演算の場合 */
static   int    min_w( int val1, int val2 );         /*16 ビットでの演算の場合 */
```

MIN

機　　能:　　val1 と val2 を比較して大きい方の値を返します。

書　　式:　　#include <asmmacro.h>

```
static   char   min_b( char val1, char val2 );       /* 8 ビットでの演算の場合 */
static   int    min_w( int val1, int val2 );         /*16 ビットでの演算の場合 */
```

MOVdir

機　能: val1 から val2 への 4 ビット転送を行ないます。

書　式:
```
#include <asmmacro.h>

/*下位 4 ビットから下位 4 ビットへの転送 */
static unsigned char movll( 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 movlh( 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 movhl( unsigned char val1, unsigned char val2 );

/*上位 4 ビットから上位 4 ビットへの転送 */
static unsigned char movhh( unsigned char val1, unsigned char val2 );
```

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>

/*8 ビットでの演算の場合 */
static long rmpa_b( long init, int count, char _far *p1, char _far *p2 );

/*16 ビットでの演算の場合 */
static long rmpa_w( long init, int count, int _far *p1, int _far *p2 );

/*48 ビットでの演算の場合 */
static long long rmpa_lw( long init, int count, int _far *p1, int _far *p2 );
```

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>

```
/* 8 ビットでの演算の場合 */
static unsigned char sha_b( signed char count, unsigned char val );

/*16 ビットでの演算の場合 */
static unsigned int sha_w( signed char count, unsigned int val );

/*32 ビットでの演算の場合 */
static unsigned long sha_l( signed char count, unsigned long val );
```

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>

```
/* 8 ビットでの演算の場合 */
static unsigned char shl_b( signed char count, unsigned char val );

/*16 ビットでの演算の場合 */
static unsigned int shl_w( signed char count, unsigned int val );

/*32 ビットでの演算の場合 */
static unsigned long shl_l( signed char count, unsigned long val );
```

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:　　#include <asmmacro.h>

```
/* 8 ビットでの演算の場合 */
static   void   sin_b( char _far *p1, char _far *p2, unsigned int count );

/*16 ビットでの演算の場合 */
static   void   sin_w( int _far *p1, int _far *p2, unsigned int count );
```

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>

/* 8 ビットでの演算の場合 */
static  void  smovb_b( char _far *p1, char _far *p2, unsigned int count );

/*16 ビットでの演算の場合 */
static  void  smovb_w( int _far *p1, int _far *p2, unsigned int count );
```

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:

```
#include <asmmacro.h>

static  void  smovu_b( char _far *p1, char _far *p2 );      /*8 ビットでの演算の場合*/
static  void  smovu_w( int _far *p1, int _far *p2 );       /*16 ビットでの演算の場合*/
```

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:

```
#include <asmmacro.h>

/*8 ビットでの演算の場合 */
static  void  sout_b( char _far *p1, char _far *p2, unsigned int count );

/*16 ビットでの演算の場合 */
static  void  sout_w( int _far *p1, int _far *p2, unsigned int count );
```

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>

/* 8 ビットでの演算の場合 */
sfafic void sstr_b( char val, char _far *p, unsigned int count );

/*16 ビットでの演算の場合 */
static   void   sstr_w( int val, int _far *p, unsigned int count );
```

ROLC

機　能: val を C フラグを含めて、1 ビット左へ回転した値を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char rolc_b( unsigned char val );    /* 8 ビットでの演算の場合 */
static unsigned int rolc_w( unsigned int val );      /*16 ビットでの演算の場合 */
```

RORC

機　能: val を C フラグを含めて、1 ビット右へ回転した値を返します。

書　式:
```
#include <asmmacro.h>

static unsigned char rorc_b( unsigned char val );    /* 8 ビットでの演算の場合 */
static unsigned int rorc_w( unsigned int val );      /*16 ビットでの演算の場合 */
```

ROT

機　　能:　　val を count 回数分、回転した値を返します。

書　　式:　　#include <asmmacro.h>

```
/* 8 ビットでの演算の場合 */
static unsigned char rot_b( signed char count, unsigned char val );

/*16 ビットでの演算の場合 */
static unsigned int rot_w( signed char count, unsigned int val );
```

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、M32C 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 M32C 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PCの標準環境を【表C.1】に示す条件として想定しています。

表C.1 PC 標準環境

項目	PC の種類	OS のバージョン
パーソナルコンピュータの環境	IBM PC/AT 互換機	Windows XP
搭載 CPU の種類	Pentium 4	
メモリ容量	512M バイト以上	

b. C 言語ソースファイル記述仕様

【表C.2】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.2 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.3】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.3 仕様

項目	仕様
指定可能なファイル数	制限なし
ファイル名の長さ	OS に依存します
起動オプション-D で指定可能なマクロ名の総数	制限なし
起動オプション-I で指定可能なディレクトリ数	最大 256
起動オプション-as308 で引き渡し可能なパラメータ数	制限なし
起動オプション-ln308 で引き渡し可能なパラメータ数	制限なし
複文、繰り返し制御構造、及び選択制御構造に対するネスト数	制限なし
条件コンパイルにおけるネスト数	制限なし
宣言中の基本型を修飾するポインタ、配列、及び関数宣言子の数	制限なし
関数定義の数	制限なし
1 つのブロック中におけるブロック有効範囲を持つ識別子の数	制限なし
1 つのソースファイル中で同時に定義され得るマクロ識別子の数	制限なし
マクロ名の置き換えの数	制限なし
入力プログラムにおける論理ソース行の数	制限なし
#include ファイルに対するネスト数	最大 40
1 つの switch 文中における case 名札の数 (switch 文のネストがない場合)	制限なし
#if、#elif 文で指定できる演算子、被演算子の合計数	制限なし
関数 1 個あたりで確保可能なスタックフレーム容量(バイト数)	最大 64K バイト
#pragma ADDRESS で定義可能な変数の数	制限なし
括弧のネスト数	制限なし
初期化式付きの変数定義を行う場合の定義可能な初期値の数	制限なし
修飾宣言子のネスト数	YACC スタックに依存
宣言子の括弧によるネスト数	YACC スタックに依存
演算子の括弧によるネスト数	YACC スタックに依存
1 つの内部識別子又はマクロ名で意味をもつ文字数	制限なし
1 つの外部識別子で意味をもつ文字数	制限なし
1 ソースファイル中の外部識別子の数	制限なし
1 ブロックでブロックスコープを持つ識別子	制限なし
1 ソースファイル中のマクロ数	制限なし
1 つの関数、関数呼び出しのパラメータ数	制限なし
1 つのマクロ定義、マクロ呼び出しのパラメータ数	最大 31
結合後の文字列リテラル内の文字数	制限なし
1 つのオブジェクトサイズ(バイト数)	制限なし
1 つの構造体／共用体のメンバ数	制限なし
1 つの列挙中の列挙定数の数	制限なし
1 つの struct 宣言リスト中の構造体/共用体のネスト数	制限なし
1 つの文字列の文字数	OS に依存します
1 ファイルの行数	制限なし

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
- 定数
- 演算子
- 注釈
- 識別子
- 文字リテラル
- 区切り子

a. キーワード

本コンパイラは【表C.4】のものをキーワードとして解釈します。

表C.4 キーワード一覧表

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

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 200 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、2 進数を指定することができます。各進数の書式を【表C.5】に示します。

表C.5　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 0123456789abcdef	0XF 又は 0xf
2 進数	0B 又は 0b で始まる	01	0B1 又は 0b0

整数定数の型は、その値の大小により以下の順で決定されます。

- 8 進数、16 進数、2 進数
  signed int 型→unsigned int 型→signed long 型→unsigned long 型→signed long long 型→unsigned long long 型
- 10 進数
  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

(2) long 型定数

long 型定数は、定数値の後に L 又は l を付加して記述します。型は値により以下の順で決定されます。

- 8 進数、16 進数、2 進数
  signed long 型→unsigned long 型→signed long long 型→unsigned long long 型
- 10 進数
  signed long 型→signed long long 型

(3) long long 型定数

long long 型定数は、定数値の後に LL 又は ll を付加して記述します。型は値により以下の順で決定されます。

- 8 進数、16 進数
  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.6　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.6】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.7】に示すものを演算子として解釈します。

表C.7　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.8】に示します。

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

- _Bool 型の符号指定はできません。
- char 型は、符号指定がない場合 unsigned char 型と解釈します。
- int 型、short 型は、符号指定がない場合 signed int 型、signed short 型と解釈します。
- long 型は、符号指定がない場合 signed long 型と解釈します。
- long long 型は、符号指定がない場合 signed long long 型と解釈します。
- 構造体のビットフィールドメンバで符号指定がない型は、符号なしと解釈します。
- long long 型のビットフィールドは使用できません。

C.2.3 式

【表C.9】、【表C.10】に式の種類と式の構成要素の関係を示します。

表C.9 式の種類と構成要素 (1/2)

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
	後置式ー>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
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

表C.10 式の種類と構成要素 (2/2)

式の種類	式の構成要素
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
- static
- typedef
- auto
- 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
- int
- long
- float
- unsigned
- struct
- enum
- char
- short
- long long
- double
- signed
- union

(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
- inline

(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
- int
- long
- float
- unsigned
- struct
- enum
- char
- short
- long long
- double
- signed
- union

(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
- 式/空文
- 繰り返し文
- アセンブリ言語記述文
- 複文
- 選択文
- 分岐文

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、#で始まるコマンドによりプリプロセッサ 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.11】に示すプリプロセスコマンドを用意しています。

表C.11 プリプロセスコマンド一覧表

コマンド名	機能
#assert	定数式が偽のときに警告を出力します
#define	マクロを定義します
#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
#error	メッセージを標準出力に出力し処理を中断します
#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
#include	指定したファイルを取り込みます
#line	ファイルの行番号を指定します
#pragma	NC308 の拡張機能の処理を指示します
#undef	マクロを未定義にします

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。

#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
```

#define

機　能: マクロを定義します。

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 (コンパイルオプション-M82 使用時は、M32C80, -M90 使用時は M32C90 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)
```

- 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
```

#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。

#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。

#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。

#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: (1) nc308 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。

#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。

#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
(6) #pragma STRUCT△構造体のタグ名△arrange
(7) #pragma ADDRESS△変数名△絶対アドレス
(8) #pragma BITADDRESS△関数名△ビット位置, 絶対アドレス
(9) #pragma DMAC△変数名△DMAC レジスタ名
(10) #pragma INTCALL△[/C]△INT 番号△アセンブラ関数名(レジスタ名,レジスタ名, ...)
(11) #pragma INTCALL△[/C]△INT 番号△C 言語関数名( )
(12) #pragma INTERRUPT△[/B ¦/E¦/F]△割り込みベクタ番号△割り込み処理関数名
(13) #pragma PARAMETER△[/C]△アセンブラ関数名(レジスタ名, レジスタ名, ..)
(14) #pragma SPECIAL△[/C]△呼び出し番号△関数名( )
(15) #pragma ALMHANDLER△アラームハンドラ関数名
(16) #pragma CYCHANDLER△周期起動ハンドラ関数名
(17) #pragma INTHANDLER△割り込みハンドラ関数名
(18) #pragma HANDLER△割り込みハンドラ関数名
(19) #pragma TASK△タスクの開始関数名
(20) #pragma ASM
(21) #pragma ENDASM
(22) #pragma JSRA△ 関数名
(23) #pragma JSRW△関数名
(24) #pragma PAGE
(25) #pragma __ASMMACRO△関数名(レジスタ名)
(26) #pragma MONITOR[n]△外部変数名

#pragma

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) 構造体配列制御機能
(7) 入出力変数の絶対アドレス指定機能
(8) 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
(9) 外部変数の DMAC レジスタ割り付け機能
(10) ソフトウェア割り込み使用関数宣言機能
(11) ソフトウェア割り込み使用関数宣言機能
(12) 割り込み関数の記述機能
(13) レジスタ渡しのアセンブラ関数宣言機能
(14) スペシャルページ分岐関数宣言機能
(15) アラームハンドラ関数の記述機能
(16) 周期起動ハンドラ関数の記述機能
(17) 割り込みハンドラ関数の記述機能
(18) 割り込みハンドラ関数の記述機能
(19) タスク開始関数の記述機能
(20) インラインアセンブラ記述機能
(21) インラインアセンブラ記述機能
(22) 関数呼び出し命令指定機能
(23) 関数呼び出し命令指定機能
(24) 改ページ指定機能
(25) アセンブラマクロ関数宣言機能
(26) RAM モニタ領域専用セクションへの配置機能

- #pragma では上記 26 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。

#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)
```

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 (コンパイルオプション-M82 使用時は M32C80、-M90 使用時は M32C90 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

【図C.14】のように、プリデファインドマクロはNC308 以外のC言語プログラム中でマシン依存部を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	符号の有無	ビットサイズ	表現できる数値
_Bool	なし	8	0、1
char unsigned char	なし	8	0～255
signed char	有り	8	-128～127
int short signed int signed short	有り	16	-32768～32767
unsigned int unsigned short	なし	16	0～65535
long signed long	有り	32	-2147483648～2147483647
unsigned long	なし	32	0～4294967295
long long signed long long	有り	64	-9223372036854775808～9223372036854775807
unsigned long long	なし	64	18446744073709551615
float	有り	32	1.17549435e-38F～3.40282347e+38F
double long double	有り	64	2.2250738585072014e-308 ～ 1.7976931348623157e+308
near ポインタ	なし	16	0～0xFFFF
far ポインタ	なし	32	0～0xFFFFFFFF

- _Bool 型の符号指定はできません。
- char 型は、符号指定がない場合、unsigned char 型と解釈します。
- int 型、short 型は、符号指定がない場合 signed int 型、signed short 型と解釈します。
- long 型は、符号指定がない場合 signed long 型と解釈します。
- long long 型は、符号指定がない場合 signed long long 型と解釈します。
- 構造体のビットフィールドメンバで符号指定がない型は、符号なしと解釈します。
- long long 型のビットフィールドは使用できません。

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。

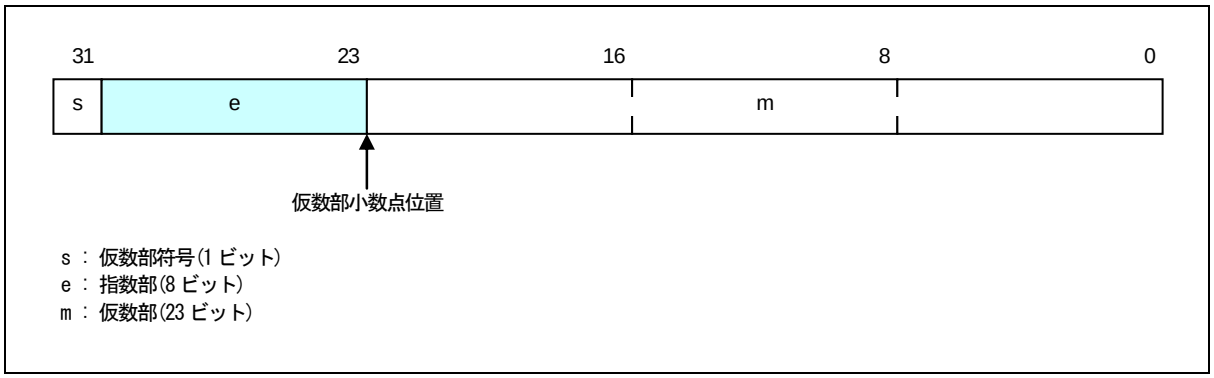
表D.2　浮動小数点型のデータサイズ

型	符号の有無	ビットサイズ	表現できる数値
float	有り	32	1.17549435e-38F～3.40282347e+38F
double long double	有り	64	2.2250738585072014e-308 ～ 1.7976931348623157e+308

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

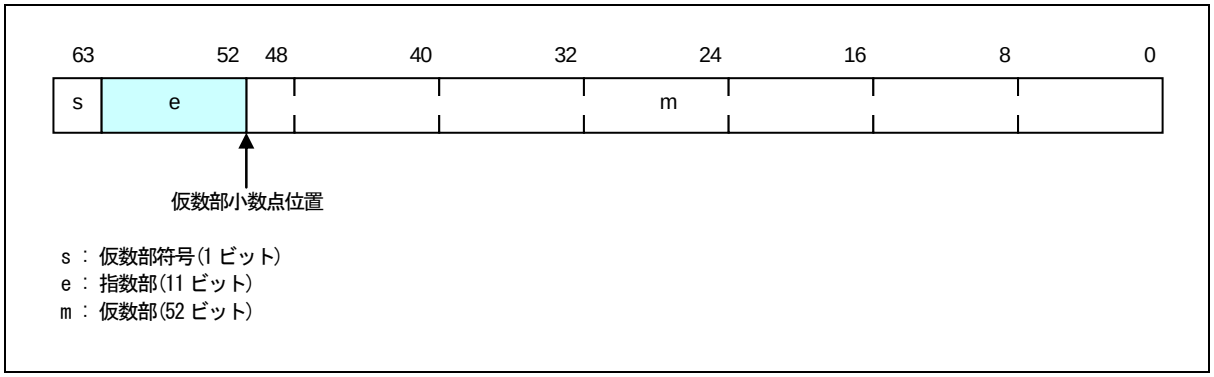
【図D.1】に示すデータ形式で 2 進数の浮動小数点(float)データを表現します。



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、コンパイルオプション"-fchar_enumerator(-fCE)"を使用することにより、列挙型を 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。

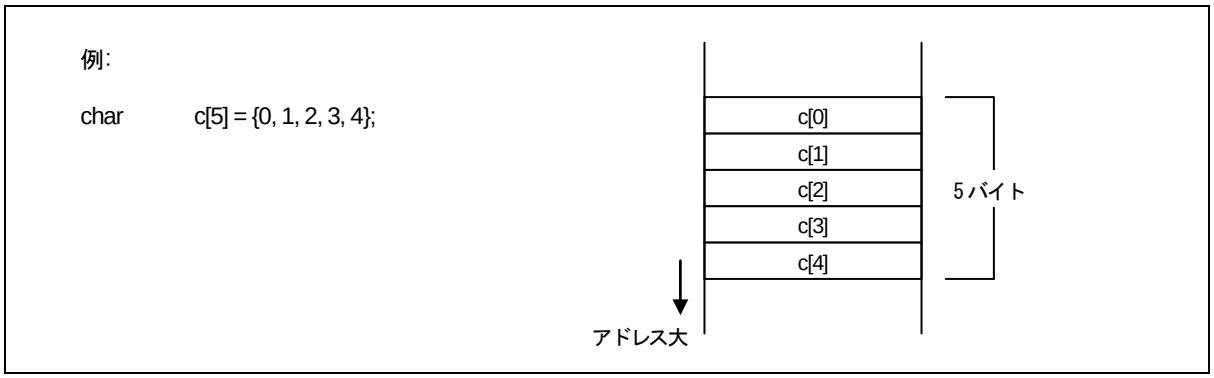
表D.3　ポインタ型のデータサイズ

型	符号の有無	ビットサイズ	表現できる数値
near ポインタ	なし	16	0～0xFFFF
far ポインタ	なし	32	0～0xFFFFFFFF

far ポインタは、32 ビット長の下位 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D.3】に配置例を示します。



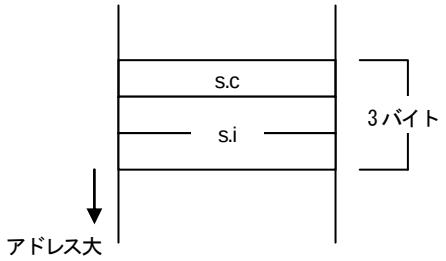
図D.3　配列の配置例

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。

例:

```
struct TAG {
        char    c;
        int     i;
} s;
```

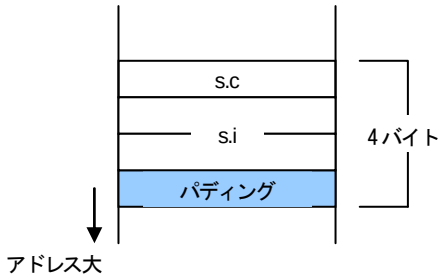


図D.4　構造体の配置例 (1)

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

例:

```
#pragma STRUCT   TAG   unpac

struct TAG {
        char    c;
        int     i;
} s;
```



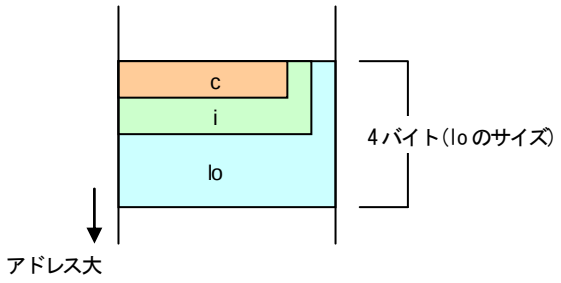
図D.5　構造体の配置例 (2)

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。

例:

```
union TAG {
        char    c;
        int     i;
        long    lo;
} s;
```



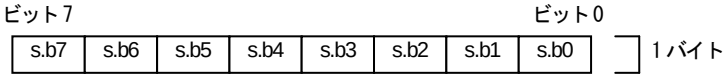
図D.6　共用体の配置例

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;
```

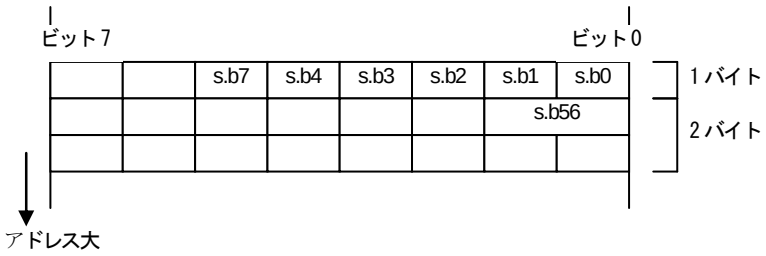


図D.7　ビットフィールドの配置例 (1)

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
  (1)　ビットフィールドのメンバの型は、符号指定が無い場合 unsigned 型とみなします。
  (2)　long long 型のビットフィールドは宣言できません。

D.2 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、演算の途中において、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。

```
void        func(void)
{
            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 型に符号拡張しません。この仕様は、コンパイルオプション"-fansi"又は"-fextend_to_int(-fETI)"を使用することにより無効となり、標準の C 言語と同様の符号拡張を行います。

コンパイルオプション"-fansi"又は"-fextend_to_int(-fETI)"を使用せず、【図D.9】のように演算結果をchar型に代入するような演算を記述する場合は、char型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、戻り値の型が整数型、ポインタ型、浮動小数点型の場合は、レジスタ渡しになります。【表D.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
_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。

D.3.2 引き数渡しに関する規則

本コンパイラは、関数への引数渡しの方法として、レジスタ渡しとスタック渡しの 2 通りがあります。

(1) 引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】中の「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[2]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】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	_Bool 型 char 型	R0L レジスタ
	int 型 near ポインタ型	R0 レジスタ

[2]本コンパイラでは、プロトタイプ宣言を行なった時のみ、レジスタ渡しを適応します。K&R 形式の記述を行なった場合は、すべての引数をスタック渡しで行ないます。
また、C 言語の言語仕様上、関数に対してプロトタイプ宣言を行なう記述形式と K&R 形式の記述を混在すると、引数が関数に正しく渡されない場合があることに注意してください。
上記理由により、プロトタイプ宣言を行なう記述形式に統一して C 言語ソースファイルを記述することを推奨しています。

(2) 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.6】の様になります。

表D.6　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
_Bool 型 char 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

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_(アンダースコア)あるいは$(ダラー)を付加したもの、または、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.7】に示します。

表D.7　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
_(アンダースコア)	上記条件以外の関数[3]

【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。

[3] #pragma INTCALL で指定した関数は関数名を出力しません。

```
int         func_proto( int , int , int);                          ← (1)

int         func_proto(int i, int j, int k)                        (2)
{
            return i + j + k;
}

int         func_no_proto( i, j, k)                                (3)
int         i;
int         j;
int         k;
{
            return i + j + k;
}

void        main(void)                                             (4)
{
            int         sum;

            sum = func_proto(1,2,3);                    ← (5)
            sum = func_no_proto(1,2,3);                 ← (6)
}
```

(1) 関数 func_proto の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）。
(3) 関数 func_no_proto の実体です（K&R 形式の記述です）。
(4) 関数 main の実体です。
(5) 関数 func_proto を呼んでいます。
(6) 関数 func_no_proto を呼んでいます。

図D.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。

```
;## #        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 をスタック渡しにしています。
(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ため第１引数をレジスタ渡しになります。第2、3引数はレジスタ渡しの対象とはならないため、スタック渡しになります。

また、関数の引数がレジスタ渡しになるため、関数の先頭アドレスのシンボル名は、C 言語ソースファイルに記述した" func_proto" の前に$ (ダラー) を付加した"$func_proto"になります。

```
;### #        FUNCTION func_no_proto
;### #  FRAME    ARG (        i)  size    2,      offset 8                    (11)
;### #  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       12
;### # C_SRC :  {
               .glb         _func_no_proto                          ← (12)
_func_no_proto:
               enter        #00H
               ._line       13
;### # 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ため、すべての引数がスタック渡しになります。

また、関数の引数にレジスタ渡しを含まないため、関数の先頭アドレスのシンボル名は、C 言語ソースファイルに記述した"func_no_proto"の前に_(アンダースコア)を付加した"_func_no_proto"になります。

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

                .align
                ._line      17
;### # C_SRC :  {
                .glb        _main
_main:
                enter       #02H
                ._line      20
;### # C_SRC :                sum = func_proto(1,2,3);
                push.w      #0003H                                          (13)
                push.w      #0002H
                mov.w       #0001H,R0
                jsr         $func_proto
                add.l       #04H,SP
                mov.w       R0,-2[FB]  ; sum
                ._line      21
;### # C_SRC :                sum = func_no_proto(1,2,3);
                push.w      #0003H                                          (14)
                push.w      #0002H
                push.w      #0001H
                jsr         _func_no_proto
                add.l       #06H,SP
                mov.w       R0,-2[FB]  ; sum
                ._line      22
;### # 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、(13) の部分はfunc_protoの呼び出しを、(14) の部分はfunc_no_protoの呼び出しを行っています。

D.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.17】～【図D.19】に示します。なお、【図D.15】と【図D.16】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

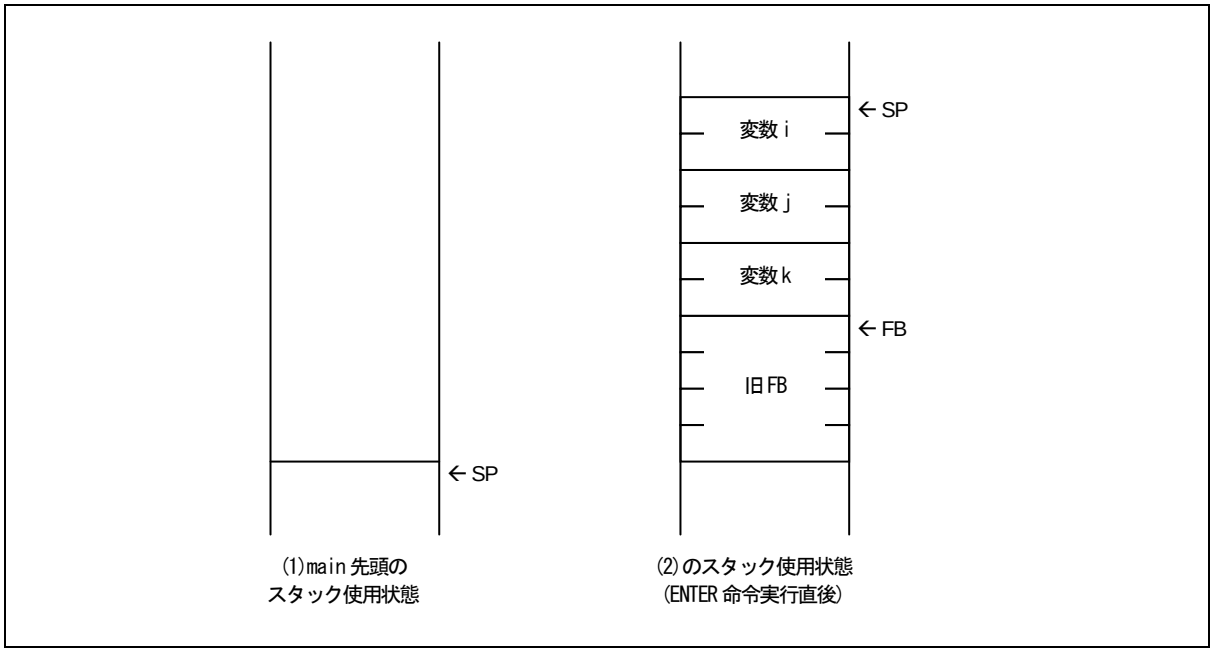
```
;## #       FUNCTION main
;## #       FRAME   AUTO    (       i)  size  2,    offset -6
;## #       FRAME   AUTO    (       j)  size  2,    offset -4
;## #       FRAME   AUTO    (       k)  size  2,    offset -2
;## #       ARG Size(0)             Auto Size(6)            Context Size(8)


            .SECTION  program,CODE,ALIGN
            ._file    'sample.c'
            .align
            ._line    4
;## # C_SRC :        {
            .glb      _main
_main:                                                      ← (1)
            enter     #06H                                  ← (2)
            ._line    5
;## # C_SRC :              int         i = 0x1234;
            mov.w     #1234H,-6[FB]       ;   i
            ._line    6
;## # C_SRC :              int         j = 0x5678;
            mov.w     #5678H,-4[FB]       ;   j
            ._line    7
;## # C_SRC :              int         k = 0x9abc;
            mov.w     #9abcH,-2[FB]       ;   k
            ._line    9
;## # 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    10
;## # C_SRC :        }
            exitd
E1:
```

図D.15　アセンブリ言語サンプルプログラム (1)

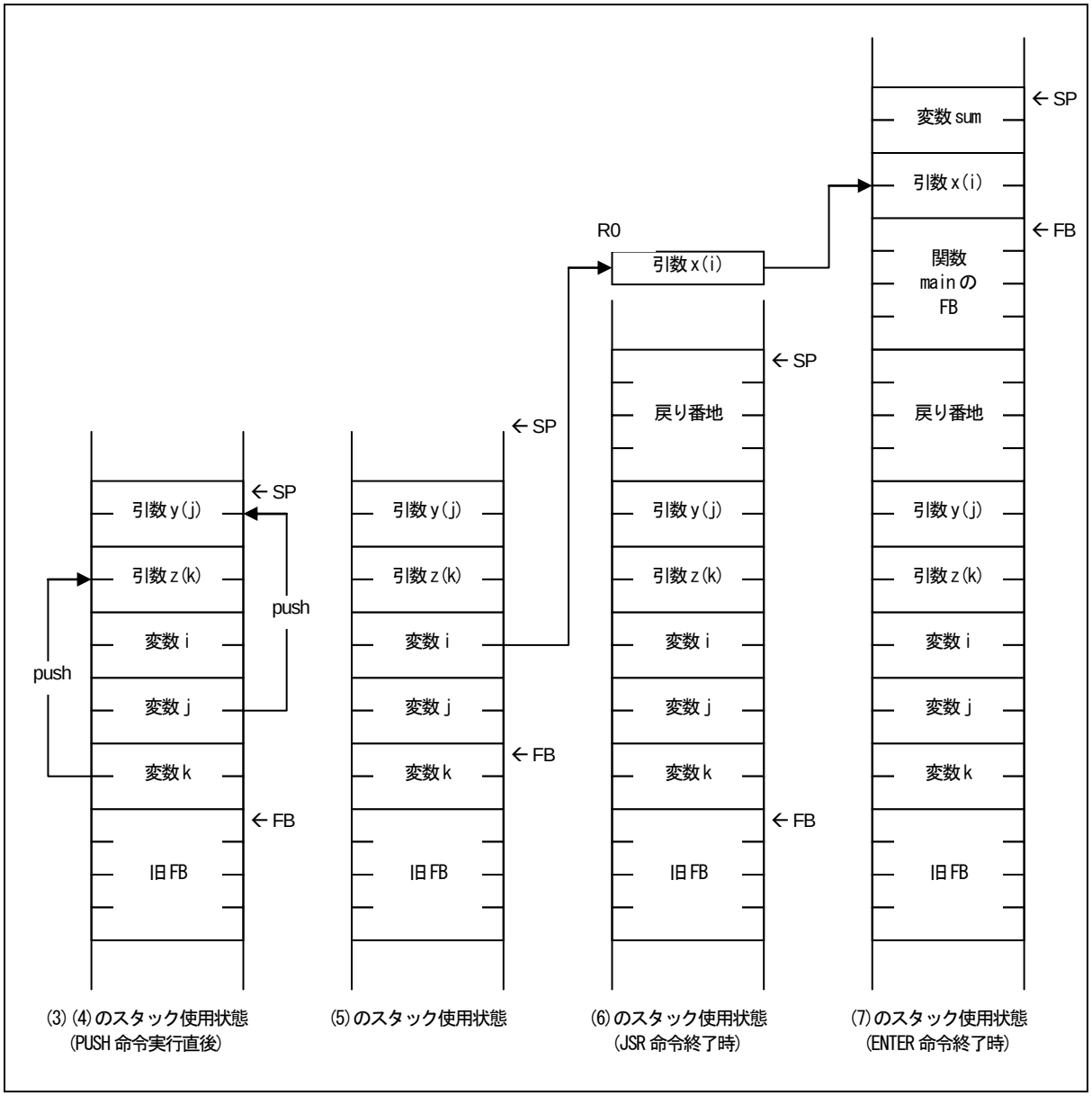
```
;## #        FUNCTION func
;## #        FRAME    AUTO      (          x) size   2,     offset -2
;## #        FRAME    AUTO      (        sum) size   2,     offset -2
;## #   FRAME    ARG (          y)   size    2,      offset 8
;## #   FRAME    ARG (          z)   size    2,      offset 10
;## #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

             .align
             ._line       13
;## # C_SRC :           {
             .glb         $func
$func:
             enter        #02H                                          ← (7)
             mov.w        R0,-2[FB]   ;   x   x
             ._line       16
;## # 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       17
;## # 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)

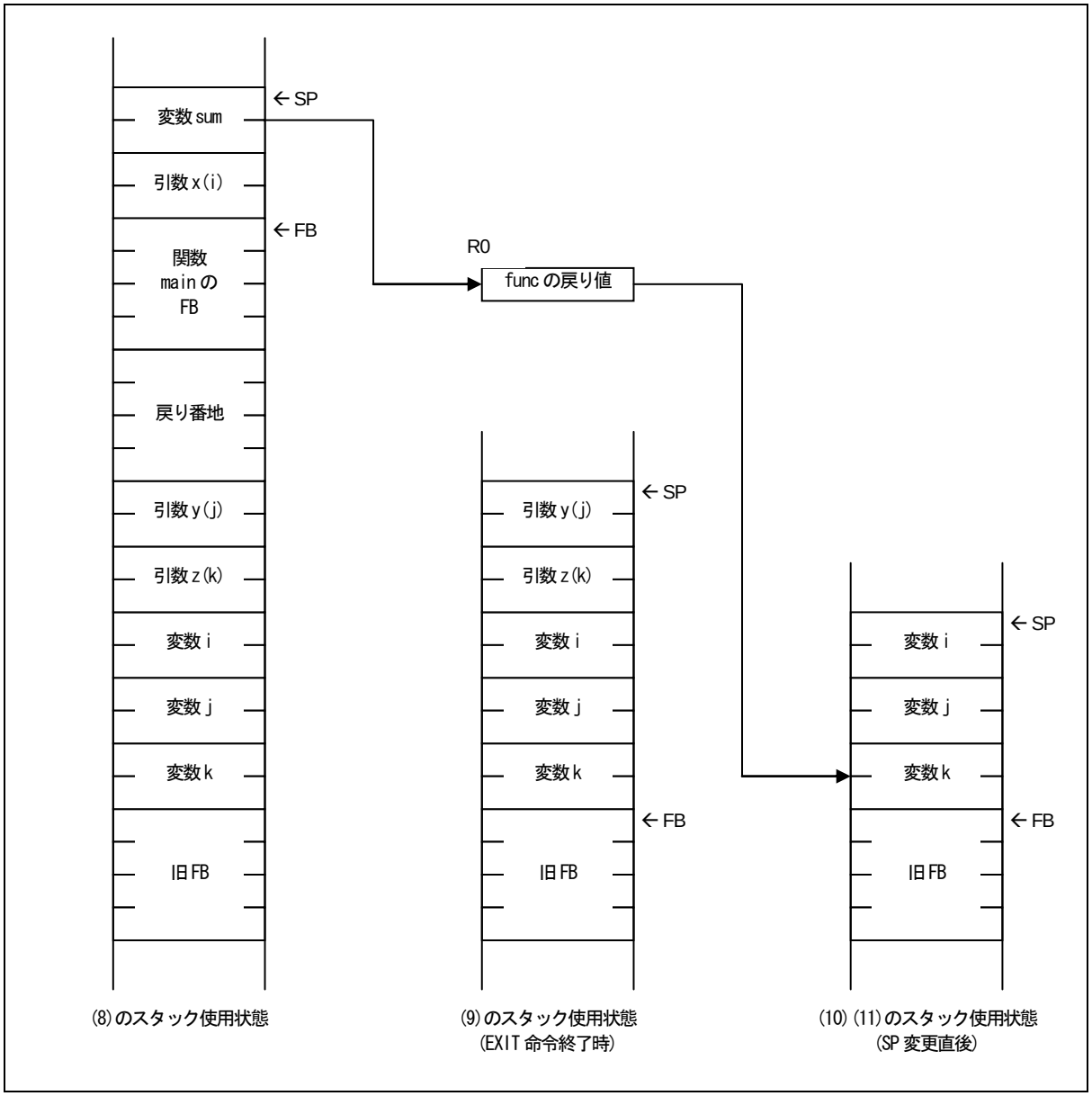
【図D.15】中における (1) → (2) の処理(関数mainの入り口処理)を【図D.17】に、(3) →(4) → (5) → (6) → (7) の処理(関数funcの呼び出し及び関数funcで使用するスタックフレームの構築処理)を【図D.18】に、【図D.16】中における(8) → (9) → (10) → (11)の処理(関数funcから関数mainへの戻り処理)を【図D.19】に、各々のスタック及びレジスタの遷移を示します。



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.20】に示すようなC言語ソースプログラムでは、記憶クラスautoの変数が有効となる領域が互いに重ならない場合、1 つの領域のみ確保を行い複数の変数でその領域を共有します。

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

        for( i=0 ; i<=0 ; i++ ){
                    処理                                    i の有効範囲
        }
                    :
                    (省略)
                    :
        for( j=0xFF ; j<=0 ; j-- ){
                    処理                                    j の有効範囲
        }
                    :
                    (省略)
                    :
        for( k=0 ; k<=0 ; k++ ){
                    処理                                    k の有効範囲
        }
}
```

図D.20　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.20】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.21】に示します。

```
;### FUNCTION      func
;###        FRAME AUTO      (       k)      size 2,     offset -2       ← (1)
;###        FRAME AUTO      (       j)      size 2,     offset -2       ← (2)
;###        FRAME AUTO      (       i)      size 2,     offset -2       ← (3)
           .section program
           ._file      'test1.c'
           ._line      3
           .glb        _func
_func:
           enter       #02H
                :
           (以下省略)
```

※3 つの auto 変数は(1)～(3)に示すように、FB オフセット-2 の領域を共有しています。

図D.21　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

- R0 レジスタ
- 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	(1) FILE 構造体の定義 (2) ストリーム名の定義 (3) 入出力関数のプロトタイプ宣言
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

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	英数字、ブランク以外の文字判定
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、監視タイマ使用時は十分注意してください。

(8) 整数算術関数
整数値に対しての演算を行う関数です。

(9) 文字列数値変換関数
文字列を数値に変換する関数です。

(10) 多バイト文字/多バイト文字列操作関数
多バイト文字/多バイト文字列を扱う関数です。

(11) 地域化関数
ロケールに関する関数です。

E.2.2 標準関数ライブラリ機能別一覧

a. 文字列操作関数

文字列操作関数の一覧を【表 E.2】に示します。

表E.2 文字列操作関数

種別	関数名	機能	リエントラント性
複写	strcpy	文字列の複写	○
	strncpy	文字列の複写(n 文字の複写)	○
連結	strcat	文字列の連結	○
	strncat	文字列の連結(n 文字の連結)	○
比較	strcmp	文字列の比較	○
	strcoll	文字列の比較(ロケール情報を使用)	○
	stricmp	文字列の比較(すべての英字は英大文字として扱う)	○
	strncmp	文字列の比較(n 文字の比較)	○
	strnicmp	文字列の比較(n 文字の比較、英字は英大文字として扱う)	○
検索	strchr	文字列の先頭より指定文字を検索	○
	strcspn	文字列より指定以外の文字列の長さを計算	○
	strpbrk	文字列より指定文字の検索	○
	strrchr	文字列の末尾より指定文字を検索	○
	strspn	文字列より指定文字列の長さを計算	○
	strstr	文字列より指定文字列の検索	○
	strtok	文字列より文字列を切り出す	×
長さ	strlen	文字列中の文字数を計算	○
変換	strerror	エラー番号を文字列に変換	×
	strxfrm	strxfrm	○

b. 文字操作関数

文字操作関数の一覧を【表 E.3】に示します。

表E.3 文字操作関数

関数名	機能	リエントラント性
isalnum	英数字の判定	○
isalpha	英字の判定	○
iscntrl	コントロール文字の判定	○
isdigit	数字の判定	○
isgraph	英数字、ブランク以外の文字判定	○
islower	英数字、ブランク以外の文字判定	○
isprint	ブランク文字を含む印字可能文字の判定	○
ispunct	区切り文字の判定	○
isspace	ブランク、タブ、改行の判定	○
isupper	英大文字の判定	○
isxdigit	16 進数字の判定	○
tolower	大文字から小文字への変換	○
toupper	小文字から大文字への変換	○

c. 入出力関数

入出力関数の一覧を【表 E.4】に示します。

表E.4 入出力関数

種別	関数名	機能	リエントラント性
初期化	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.5】に示します。

表E.5 メモリ管理関数

関数名	機能	リエントラント性
calloc	メモリの確保と 0(ゼロ)による初期化	×
free	メモリの解放	×
malloc	メモリの確保	×
realloc	確保済み領域の大きさを変更	×

e. メモリ操作関数

メモリ操作関数の一覧を【表 E.6】に示します。

表E.6 メモリ操作関数

種別	関数名	機能	リエントラント性
初期化	bzero	メモリ領域の初期化(ゼロクリア)	○
複写	bcopy	メモリ領域の複写	○
	memcpy	メモリ領域の複写(n 文字の複写)	○
	memset	メモリ領域の設定	○
比較	memcmp	メモリ量域の比較(n バイトの比較)	○
	memicmp	メモリ領域の比較(英文字は大文字として扱う)	○
移動	memmove	文字列の領域を移動	○
検索	memchr	メモリ領域より文字を検索	○

f. 実行制御関数

実行制御関数の一覧を【表 E.7】に示します。

表E.7 実行制御関数

関数名	機能	リエントラント性
abort	プログラムの実行を終了	○
longjmp	大域ジャンプ	○
setjmp	大域ジャンプのためのスタック環境を設定	○

g. 数学関数

数学関数の一覧表を【表 E.8】に示します。

表E.8 数学関数

関数名	機能	リエントラント性
acos	逆コサインを計算	○
asin	逆サインを計算	○
atan	逆タンジェントを計算	○
atan2	逆タンジェントを計算	○
ceil	整数繰り上げ値を計算	○
cos	コサインを計算	○
cosh	双曲線コサインを計算	○
exp	指数関数を計算	○
fabs	倍精度浮動小数の絶対値を計算	○
floor	整数繰り下げ値を計算	○
fmod	剰余計算	○
frexp	浮動小数を仮数部と指数部に分割	○
labs	long 型整数の絶対値を計算	○
ldexp	浮動小数の巾を計算	○
log	自然対数を計算	○
log10	常用対数を計算	○
modf	実数を仮数部と指数部に分割	○
pow	巾乗計算	○
sin	サインを計算	○
sinh	双曲線サインを計算	○
sqrt	数値の平方根を計算	○
tan	タンジェントを計算	○
tanh	双曲線タンジェントを計算	○

h. 整数算術関数

整数算術関数の一覧表を【表 E.9】に示します。

表E.9 整数算術関数

関数名	機能	リエントラント性
abs	整数の絶対値を計算	○
bsearch	配列内のバイナリサーチを行う	○
div	int 型整数の除算と剰余	○
labs	long 型整数の絶対値を計算	○
ldiv	long 型整数の除算と剰余	○
qsort	配列をソート	○
rand	疑似乱数を発生	○
srand	疑似乱数発生関数 rand にシード（種）を与える	○

i. 文字列数値変換関数

文字列数値変換関数の一覧表を【表 E.10】に示します。

表E.10 文字列数値変換関数

関数名	機能	リエントラント性
atof	文字列を double 型に変換	○
atoi	文字列を int 型に変換	○
atol	文字列を long 型に変換	○
strtod	文字列を double 型に変換	○
strtol	文字列を long 型に変換	○
strtou	文字列を unsigned long 型に変換	○

j. 多バイト文字/多バイト文字列操作関数

多バイト文字/多バイト文字列操作関数の一覧表を【表 E.11】に示します。

表E.11 多バイト文字/多バイト文字列操作関数

関数名	機能	リエントラント性
mblen	マルチバイト文字列の長さを計算	○
mbstowcs	マルチバイト文字列をワイド文字列に変換	○
mbtowc	マルチバイト文字をワイド文字に変換	○
wcstombs	ワイド文字列をマルチバイト文字列に変換	○
wctomb	ワイド文字をマルチバイト文字に変換	○

k. 地域化関数

地域化関数の一覧表を【表 E.12】に示します。

表E.12 地域化関数

関数名	機能	リエントラント性
localeconv	構造体 lconv を初期化	○
setlocale	プログラムのロケール情報の設定と検索	○

E.2.3 標準関数リファレンス

以降に本コンパイラが提供する標準関数の詳細仕様を説明します。関数は、アルファベット順に掲載しています。

なお、[書式]に記述の標準ヘッダファイル(拡張子.h)は、その関数を使用するときに必ずインクルードしてください。

A

abort

実行制御関数

機　　能: プログラムを異常終了します。

書　　式: #include <stdlib.h>

void abort(void);

実現方法: 関数

引　　数: 引数はありません。

戻 り 値: 戻り値はありません。

解　　説: プログラムを異常終了します。

注　　意: 実際には、abort プログラム内部で無限ループとなります。

abs

整数算術関数

機　　能: int 型整数の絶対値を計算します

書　　式: #include <stdlib.h>

int abs(n);

実現方法: 関数

引　　数: int n; 実数

戻 り 値: int 型整数 n の絶対値(0 からの距離)を返します。

acos

数学関数

機　　能: 逆コサインを計算します。

書　　式: #include <math.h>

double acos(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値:
- x の値が-1.0 から 1.0 の範囲外の場合はエラーとして 0 を返します。
- それ以外の場合は、0 からπラジアンの範囲の値を返します。

asin

数学関数

機　　能: 逆サインを計算します。

書　　式: #include <math.h>

double asin(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値:
- x の値が-1.0 から 1.0 の範囲外の場合はエラーとして 0 を返します。
- それ以外の場合は、-π/2 からπ/2 ラジアンの範囲の値を返します。

atan

数学関数

機　　能: 逆タンジェントを計算します。

書　　式: #include <math.h>

double atan(x);

実現方法: 関数

引　　数: double x;

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

引　　数:

double x;	実数
double y;	実数

戻 り 値: -πからπラジアンの範囲の値を返します。

atof

文字列数値変換関数

機　　能: 文字列を浮動小数に変換します。

書　　式: #include <stdlib.h>

double atof(s);

実現方法: 関数

引　　数:

const char _far *s;	変換文字列へのポインタ

戻 り 値: 文字列を倍精度浮動小数に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

引　　数:

const char _far *s;	変換文字列へのポインタ

戻 り 値: 文字列を int 型整数に変換した値を返します。

atol

文字列数値変換関数

機　　能: 文字列を long 型整数に変換します。

書　　式:
```
#include <stdlib.h>

long atol( s );
```

実現方法: 関数

引　　数: const char _far *s; 変換文字列へのポインタ

戻 り 値: 文字列を long 型整数に変換した値を返します。

B

bcopy

メモリ操作関数

機　　能: メモリ領域の複写を行います。

書　　式:
```
#include <string.h>

void bcopy( src, dtop, size );
```

実現方法: 関数

引　　数:

char _far *src;	複写元のメモリ領域の先頭アドレス
char _far *dtop;	複写先のメモリ領域の先頭アドレス
unsigned long size;	複写するバイト数

戻 り 値: dtop で示される領域に、src で示される領域の先頭から size で指定されたバイト数分の内容を複写します。

bsearch

整数算術変換関数

機　　能: 文字列を浮動小数に変換します

書　　式:
```
#include <stdlib.h>

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

実現方法: 関数

引　　数:

const void _far *key;	検索キー
const void _far *base;	配列開始アドレス
size_t nelem;	要素数
size_t size;	各要素の大きさ
int cmp();	比較関数

戻 り 値:
- 検索キーに等しい配列要素へのポインタを返します。
- 一致する要素がない場合は NULL ポインタを返します。

解　　説: 昇順にソート済みの配列内から指定された項目を検索します。

bzero

メモリ操作関数

機　　能: メモリ領域の初期化(ゼロクリア)を行います。

書　　式:
```
#include <string.h>

void bzero( top, size );
```

実現方法: 関数

引　　数:

char _far ＊top;	ゼロクリアするメモリ領域の先頭アドレス
unsigned long size;	ゼロクリアするバイト数

戻 り 値: 戻り値はありません。

解　　説: top で示される領域の先頭アドレスから size で示されるバイト数分の内容を 0 で初期化します。

C

calloc

メモリ管理関数

機　　能: メモリの割り当てとゼロクリアを行います。

書　　式:
```
#include <stdlib.h>

void _far * calloc( n, size );
```

実現方法: 関数

引　　数:

size_t n;	要素の数
size_t size;	要素の大きさをバイト数で表した値

戻 り 値: 指定した大きさの領域が確保できなかった場合、戻り値として NULL を返します。

解　　説:
- 指定されたメモリを割り当てた後、ゼロクリアを行います。
- メモリ領域の大きさは 2 つの引数の積になります。

規　　則: メモリの確保規則については、malloc と同様です。

ceil

数学関数

機　　能: 整数繰り上げ値を返します。

書　　式:
```
#include <math.h>

double ceil( x );
```

実現方法: 関数

引　　数:

double x;	実数

戻 り 値: x より大きい整数の中から最小の整数値を double 型で返します。

clearerr

入出力関数

機　　能: ストリームのエラー状態指示子をクリアします。

書　　式:
```
#include <stdio.h>
void clearerr( stream );
```

実現方法: 関数

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値: 戻り値はありません。

解　　説: エラー状態指示子とファイル終端状態指示子を正常値にリセットします。

cos

数学関数

機　　能: コサインを計算します。

書　　式:
```
#include <math.h>
double cos( x );
```

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする引数"x"のコサインを計算します。

cosh

数学関数

機　　能: 双曲線コサインを計算します。

書　　式:
```
#include <math.h>
double cosh( x );
```

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の双曲線コサインを計算します。

D

div

剰余算関数

機　　能: int 型整数の除算を行います。

書　　式:
```
#include <stdlib.h>

div_t div( number, denom );
```

実現方法: 関数

引　　数:

int number;	被除数
int denom;	除数

戻 り 値: "number"を"denom"で割った商と剰余を返します。

解　　説:

- "number"を"denom"で割った商と剰余を div_t の構造体で返します。
- div_t は stdlib.h 内で定義しています。この構造体は、int quot、int rem というメンバーから構成されます。

E

exp

数学関数

機　　能: 指数関数を計算します。

書　　式: #include <math.h>

double exp(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の指数関数の計算結果を返します。

F

fabs

数学関数

機　　能: 倍精度浮動小数の絶対値を計算します。

書　　式: #include <math.h>

double fabs(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: 倍精度浮動小数の絶対値を返します。

feof

入出力関数

機　　能: ストリームのファイル状態指示子(EOF)を調べます。

書　　式: #include <stdio.h>

int feof(stream);

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- ストリームが EOF 場合、真(0 以外)を返します。
- それ以外の場合、NULL(0)を返します。

解　　説:
- ストリームが EOF まで読み込んだかどうか判定します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

ferror

入出力関数

機　　能: ストリームのエラー状態を調べます。

書　　式:
```
#include <stdio.h>

int ferror( stream );
```

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- ストリームがエラーの場合、真(0 以外)を返します。
- それ以外の場合、NULL(0)を返します。

解　　説:
- ストリームがエラーかどうか判定します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

fflush

入出力関数

機　　能: 出力バッファをフラッシュします。

書　　式:
```
#include <stdio.h>

int fflush( stream );
```

実現方法: 関数

引　　数: FILE _far *stream; ストリームのポインタ

戻 り 値: 常に 0 を返します。

fgetc

入出力関数

機　　能: ストリームから 1 文字を入力します。

書　　式:
```
#include <stdio.h>

int fgetc( stream );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終りの場合、EOF を返します。

解　　説:
- ストリームから 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

fgets

入出力関数

機　　能: ストリームから文字列を入力します。

書　　式:
```
#include <stdio.h>

char _far * fgets( buffer, n, stream );
```

実現方法: 関数

引　　数:

char _far *buffer;	格納先のポインタ
int n;	最大文字数
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に入力できた場合、格納先のポインタ(引数で与えたポインタと同じ)を返します。
- エラー又はストリームの終わりの場合、NULL ポインタを返します。

解　　説:
- 指定したストリームから文字列を入力し、バッファ(buffer)に格納します。入力を終了するのは、次の３つの場合です。
  (1) 改行文字('¥n')を入力した場合
  (2) n-1 個の文字を入力した場合
  (3) ストリームの終わりまで入力した場合
- 入力した文字列の最後には、ヌル文字('¥0')を付加します。
- 改行文字('¥n')は、そのまま格納します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

floor

数学関数

機　　能: 整数の繰り下げ値を計算します。

書　　式:
```
#include <math.h>

double floor( x );
```

実現方法: 関数

引　　数:

double x;	実数

戻 り 値: 整数の繰り下げ値を double 型で返します

fmod

数学関数

機　　能: 剰余計算を行います。

書　　式:
```
#include <math.h>

double fmod( x ,y );
```

実現方法: 関数

引　　数:

double x;	被除数
double y;	除数

戻 り 値: "x"を"y"で割ったときの剰余を返します。

fprintf

入出力関数

機　　能: ストリームへの書式付き出力を行います。

書　　式:
```
#include <stdio.h>

int fprintf( stream, format, argument... );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列のポインタ

戻 り 値:
- 出力した文字数を返します。
- ハードウェアに起因するエラーの場合、EOF を返します。

解　　説:
- format の指定に従って argument を文字列に変換し、ストリームへ出力します。

- format の指定方法は printf と同様です。

fputc

入出力関数

機　　能: ストリームに 1 文字を出力します。

書　　式:
```
#include <stdio.h>

int fputc( c, stream );
```

実現方法: 関数

引　　数:

int c;	出力する文字
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に出力できた場合、出力した文字を返します。
- エラーの場合、EOF を返します。

解　　説: ストリームに 1 文字を出力します。

fputs

入出力関数

機　　能: ストリームへ文字列を出力します。

書　　式:
```
#include <stdio.h>

int fputs ( str, stream );
```

実現方法: 関数

引　　数:

const char _far *str;	出力する文字列のポインタ
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常に出力できた場合、0 を返します。
- エラーの場合、0 以外(EOF)を返します。

解　　説: ストリームへ文字列を出力します。

fread

入出力関数

機　　能: ストリームから固定長データを入力します。

書　　式:
```
#include <stdio.h>

size_t fread( buffer, size, count, stream );
```

実現方法: 関数

引　　数:

void _far *buffer;	格納先のポインタ
size_t size;	データ 1 項目のバイト数
size_t count;	最大データ項目数
FILE _far *stream;	ストリームのポインタ

戻 り 値: 入力したデータ項目数を返します。

解　　説:
- ストリームから size のデータ長を持つデータを count の項目数だけ入力し、バッファ(buffer)に格納します。
- count 分のデータを入力する前にストリームの終わりになった場合、それまでに入力したデータ項目数を返します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

free

メモリ管理関数

機　　能: メモリの解放を行います。

書　　式:
```
#include <stdlib.h>

void free( cp );
```

実現方法: 関数

引　　数:

void _far *cp;	解放するメモリ領域へのポインタ

戻 り 値: 戻り値はありません。

解　　説:
- 以前に関数 malloc、calloc によって割り当てられたメモリ領域の解放を行います。
- NULL を引数にした場合は処理を行いません。

frexp

数学関数

機　　能: 浮動小数を仮数部と指数部に分割します。

書　　式:
```
#include <math.h>

double frexp( x,prexp );
```

実現方法: 関数

引　　数:

double x;	浮動小数
int _far *prexp;	2 を底とする指数を格納する領域へのポインタ

戻 り 値: "x"の仮数部を返します。

fscanf

入出力関数

機　　能: ストリームからの書式付き入力を行います。

書　　式:
```
#include <stdio.h>

int fscanf( stream, format, argument... );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列のポインタ

戻 り 値:
- 各引数 argument に格納したデータ数を返します。
- ストリームからデータとして EOF を入力した場合、EOF を返します。

解　　説:
- format の指定に従ってストリームからの入力文字を変換し、各引数 argument が示す引数に格納します。
- argument は各引数のポインタでなければいけません。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。
- format の指定方法は、scanf と同様です。

fwrite

入出力関数

機　　能: ストリームへ固定長データを出力します。

書　　式:
```
#include <stdio.h>

size_t fwrite( buffer, size, count, stream );
```

実現方法: 関数

引　　数:

const void _far *buffer;	出力データのポインタ
size_t size;	データ 1 項目のバイト数
size_t count;	最大データ項目数
FILE _far *stream;	ストリームのポインタ

戻 り 値: 出力したデータ項目数を返します。

解　　説:

- ストリームへ size のデータ長を持つデータを count の項目数だけ出力します。
- count 分のデータを出力する前にエラーになった場合は、それまでに出力したデータ項目数を返します。

G

getc

入出力関数

機　　能: ストリームから 1 文字を入力します。

書　　式:
```
#include <stdio.h>

int getc( stream );
```

実現方法: マクロ

引　　数: FILE _far *stream; ストリームへのポインタ

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終わりの場合、EOF を返します。

解　　説:
- ストリームから 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

getchar

入出力関数

機　　能: stdin から 1 文字を入力します。

書　　式:
```
#include <stdio.h>

int getchar( void );
```

実現方法: マクロ

引　　数: 引数はありません。

戻 り 値:
- 入力した 1 文字を返します。
- エラー又はストリームの終わりの場合、EOF を返します。

解　　説:
- ストリーム(stdin)から 1 文字を入力します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

gets

入出力関数

機　　能: stdin から文字列を入力します。

書　　式:
```
#include <stdio.h>

char _far * gets( buffer );
```

実現方法: 関数

引　　数: char _far *buffer; 格納先のポインタ

戻 り 値:
- 正常に入力できた場合、格納先のポインタ(引数で与えたポインタと同じ)を返します。
- エラー又はストリームの終わりの場合、NULL ポインタを返します。

解　　説:
- stdin から文字列を 1 行入力し、バッファ(buffer)に格納します。
- 行末の改行文字('¥n')は、ヌル文字('¥0')に置き換えます。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

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

戻 り 値:
- 英字、又は数字の場合、0 以外を返します。
- 英字、又は数字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isalpha

文字操作関数

機　　能: 英字(A～Z、a～z)を判定します。

書　　式:
```
#include <ctype.h>

int isalpha( c );
```

実現方法: マクロ

引　　数:

int c;		判定する文字

戻 り 値:
- 英字の場合、0 以外を返します。
- 英字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

iscntrl

文字操作関数

機　　能: コントロール文字(0x00～0x1f、0x7f)を判定します。

書　　式:
```
#include <ctype.h>

int iscntrl( c );
```

実現方法: マクロ

引　　数:

int c;		判定する文字

戻 り 値:
- コントロール文字の場合、0 以外を返します。
- コントロール文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isdigit

文字操作関数

機　　能: 数字(0～9)を判定します。

書　　式:
```
#include <ctype.h>

int isdigit( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 数字の場合、0 以外を返します。
- 数字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isgraph

文字操作関数

機　　能: ブランク以外の文字(0x21～0x7e)を印字文字判定します。

書　　式:
```
#include <ctype.h>

int isgraph( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 印字可能な場合、0 以外を返します。
- 印字不可の場合、0 を返します。

解　　説: 引数の文字を判定します。

islower

文字操作関数

機　　能: 英小文字(a～z)を判定します。

書　　式:
```
#include <ctype.h>

int islower( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 英小文字の場合、0 以外を返します。
- 英小文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isprint

文字操作関数

機　　能: ブランク文字を含む文字(0x20～0x7e)の印字文字を判定します。

書　　式:
```
#include <ctype.h>

int isprint( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 印字可能な場合、0 以外を返します。
- 印字不可の場合、0 を返します。

解　　説: 引数の文字を判定します

ispunct

機　　能: 区切り文字を判定します

書　　式:
```
#include <ctype.h>

int ispunct( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 区切り文字の場合、0 以外を返します。
- 区切り文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isspace

文字操作関数

機　　能: ブランク、タブ、改行を判定します。

書　　式:
```
#include <ctype.h>

int isspace( c );
```

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- ブランク、タブ、改行の場合、0 以外を返します。
- ブランク、タブ、改行でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isupper

文字操作関数

機　　能: 文字操作関数

書　　式: #include <ctype.h>

int isupper(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 英大文字の場合、0 以外を返します。
- 英大文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

isxdigit

文字操作関数

機　　能: 16 進文字(0～9、A～F、a～f)を判定します。

書　　式: #include <ctype.h>

int isxdigit(c);

実現方法: マクロ

引　　数: int c; 判定する文字

戻 り 値:
- 16 進文字の場合、0 以外を返します。
- 16 進文字でない場合、0 を返します。

解　　説: 引数の文字を判定します。

L

labs

整数算術関数

機　　能: long 型整数の絶対値を計算します。

書　　式: #include <stdlib.h>

long labs(n);

実現方法: 関数

引　　数: long n; long 型整数

戻 り 値: long 型整数の絶対値(0 からの距離)を返します。

ldexp

数学関数

機　　能: 浮動小数の巾を計算します。

書　　式: #include <math.h>

double ldexp(x,exp);

実現方法: 関数

引　　数: double x; 実数
int exp; 巾乗数

戻 り 値: "x"*(2 の"exp"乗)を返します。

ldiv

整数算術関数

機　　能: long 型整数の除算を行います。

書　　式: #include <stdlib.h>

ldiv_t ldiv(number, denom);

実現方法: 関数

引　　数:

long number;	被除数
long denom;	除数

戻 り 値: "number"を"denom"で割った商と剰余を返します。

解　　説:

- "number"を"denom"で割った商と剰余を ldiv_t の構造体で返します。
- ldiv_t は stdlib.h 内で定義しています。この構造体は、long quot,long rem というメンバーから構成されます。

localeconv

地域化関数

機　　能: 構造体 lconv を初期化します。

書　　式: #include <locale.h>

struct lconv _far *localeconv(void);

実現方法: 関数

引　　数: 引数はありません。

戻 り 値: 初期化された構造体 lconv へのポインタを返します。

log

数学関数

機　　能: 自然対数を計算します。

書　　式: #include <math.h>

double log(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の自然対数を返します。

解　　説: 関数 exp の逆関数です。

log10

数学関数

機　　能: 常用対数を計算します。

書　　式: #include <math.h>

double log10(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: "x"の常用対数を返します。

longjmp

実行制御関数

機　　能: 関数呼び出し時の環境の回復を行います。

書　　式:
```
#include <setjmp.h>

void longjmp( env, val );
```

実現方法: 関数

引　　数:

jmp_buf env;	環境を回復する領域へのポインタ
int val;	setjmp の結果として返す値

戻 り 値: 戻り値はありません。

解　　説:
- "env"で示される領域から環境を回復します。
- プログラムの制御は、以前に setjmp 関数を呼んだ次の文に移ります。
- "val"で指定した値は、setjmp 関数の結果として返します。ただし、"val"が"0"の場合"1"に変換されます。

M

malloc

メモリ管理関数

機　　能: malloc メモリの割り当てを行います。

書　　式: #include <stdlib.h>

void _far * malloc(nbytes);

実現方法: 関数

引　　数: size_t nbytes; 割り当てるメモリの大きさバイト数

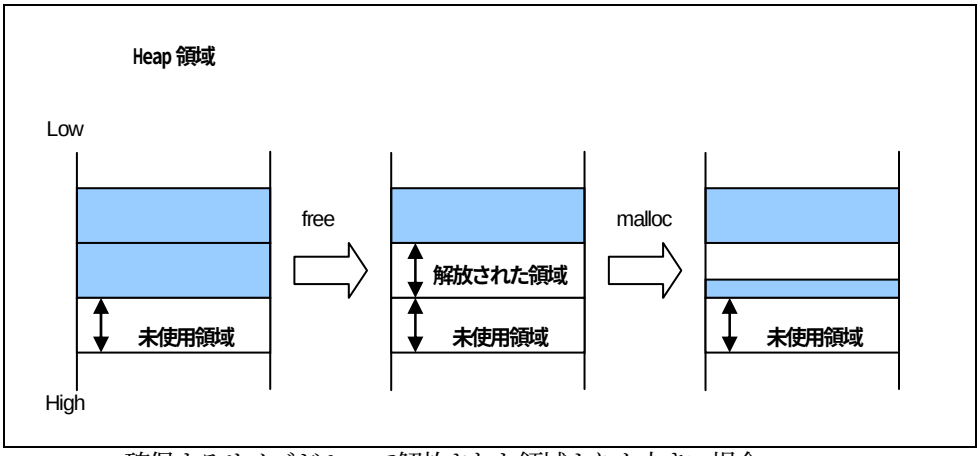
戻 り 値: 指定した大きさの領域が確保できなかった場合は戻り値として NULL を返します。

解　　説: 動的なメモリ領域を割り当てます。

規　　則: malloc を行う場合、以下の(1)～(2)の順にチェックを行い、適合する箇所でメモリを確保します。

(1) free で解放された領域がある場合

- 確保するサイズが free で解放された領域より小さい場合
  free で作成された連続空き領域の上位番地から下位番地方向へ領域が確保されます。



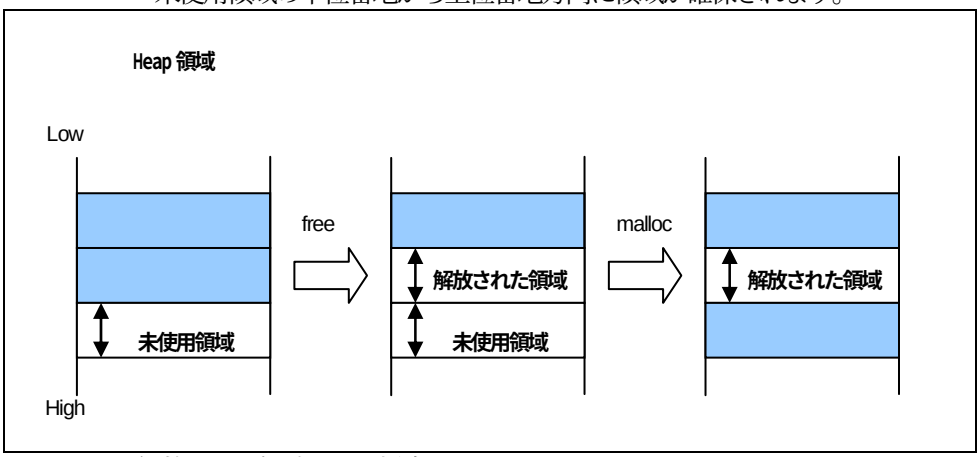
- 確保するサイズが free で解放された領域よりも大きい場合
  未使用領域の下位番地から上位番地方向に領域が確保されます。

malloc

メモリ管理関数

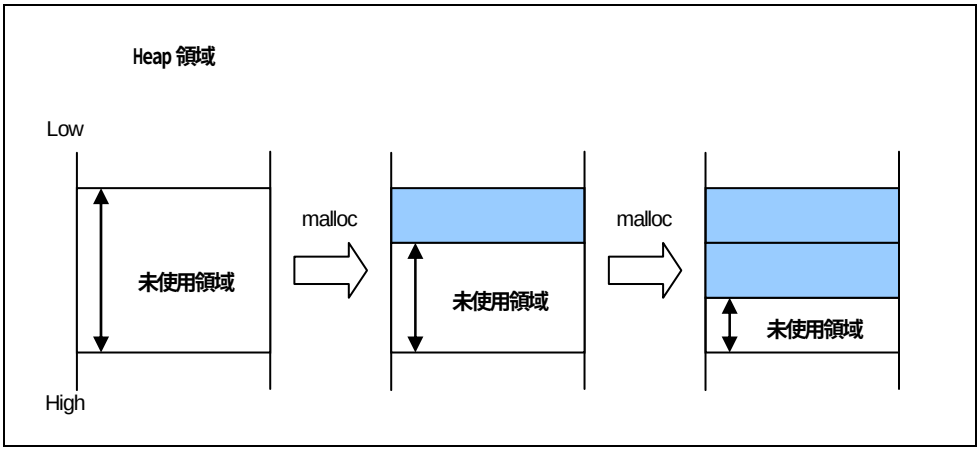
規　　則:

- 確保するサイズが free で解放された領域よりも大きい場合
  未使用領域の下位番地から上位番地方向に領域が確保されます。



(2) free で解放された領域がない場合

- 確保可能な未使用領域が存在する場合
  未使用領域の下位番地から上位番地方向に領域が確保されます。



- 確保可能な未使用領域が存在しない場合
  メモリは確保せずに、NULL を返します。

注意事項: ガーベージコレクションは行っていません。したがって、小さな未使用領域が多く存在していても、指定した領域サイズ以上の未使用領域が無ければメモリを確保できず、NULL を返します。

mblen

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字列の長さを計算します。

書　　式:
```
#include <stdlib.h>

int mblen ( s,n );
```

実現方法: 関数

引　　数:

const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	検索バイト数

戻 り 値:
- s が正しいマルチバイト文字列をなしている場合はそのバイト数を返します。
- s が正しいマルチバイト文字列をなしていない場合は-1 を返します。

解　　説: s が NULL 文字を指している場合は 0 を返します。

mbstowcs

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字列をワイド文字列に変換します。

書　　式:
```
#include <stdlib.h>

size_t mbstowcs( wcs,s,n );
```

実現方法: 関数

引　　数:

wchar_t _far *wcs;	変換ワイド文字列格納領域へのポインタ
const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	格納ワイド文字数

戻 り 値:
- 変換したマルチバイト文字列の文字数を返します。
- 正しいマルチバイト文字列をなしてない場合は-1 を返します。

mbtowc

多バイト文字/多バイト文字列操作関数

機　　能: マルチバイト文字をワイド文字に変換します。

書　　式:
```
#include <stdlib.h>

int mbtowc( wcs,s,n );
```

実現方法: 関数

引　　数:

wchar_t _far *wcs;	変換ワイド文字列格納領域へのポインタ
const char _far *s;	マルチバイト文字列へのポインタ
size_t n;	検索バイト文字数

戻 り 値:
- s が正しいマルチバイト文字をなしている場合は変換したワイド文字数を返します。
- s が正しいマルチバイト文字をなしていない場合は-1 を返します。
- s が NULL 文字の場合は 0 を返します。

memchr

メモリ操作関数

機　　能: メモリ領域より文字の検索を行います。

書　　式:
```
#include <string.h>

void _far * memchr( s, c, n );
```

実現方法: 関数

引　　数:

const void _far *s;	検索先のメモリ領域へのポインタ
int c;	検索する文字
size_t n;	検索するメモリ領域の大きさ

戻 り 値:
- 見つかった場合、指定された文字"c"の位置(ポインタ)を返します。
- メモリ領域中に文字"c"が見つからない場合は NULL を返します。

解　　説:
- "s"で示されるアドレスから始まる"n"で指定された大きさのメモリ領域から、"c"で示される文字を検索します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX 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は、#include <string.h> の記述の後に、 #undef memcpy と記述してください。
- "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, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

memmove

メモリ操作関数

機　　能: メモリ領域を移動します。

書　　式:
```
#include <string.h>

void _far * memmove( s1, s2, n );
```

実現方法: 関数

引　　数:

void _far *s1;	移動先へのポインタ
const void _far *s2;	移動元へのポインタ
size_t n;	移動バイト数

戻 り 値: 移動先へのポインタを返します。

解　　説: オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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は、#include <string.h> の記述の後に、 #undef memcpy と記述してください。
- "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, -OS, -OR_MAX, -OS_MAX 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。

P

perror

入出力関数

機　　能: エラーメッセージを stderr に出力します。

書　　式: #include <stdio.h>

void perror(s);

実現方法: 関数

引　　数: const char _far *s; メッセージの前につく文字列へのポインタ

戻 り 値: 戻り値はありません。

pow

数学関数

機　　能: 巾乗計算を行います。

書　　式: #include <math.h>

double pow(x,y);

実現方法: 関数

引　　数: double x; 被乗数
double y; 巾乗数

戻 り 値: "x"の"y"乗を返します。

printf

入出力関数

機　　能: stdout への書式付き出力を行います。

書　　式:
```
#include <stdio.h>

int printf( format, argument... );
```

実現方法: 関数

引　　数: const char _far *format;　　書式指定文字列のポインタ

format で与えられる文字列の%以降の指定は、次の意味を持ちます。[]内は省略可能です。

書式：
%[フラグ][最小フィールド幅][精度][修飾文字(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。

(1) 変換指定記号

- d、I
  引数の整数を符号付き 10 進数に変換します。
- u
  引数の整数を符号なし 10 進数に変換します。
- o
  引数の整数を符号なし 8 進数に変換します。
- x
  引数の整数を符号なし 16 進数に変換します。0AH～0FH に小文字の"abcdef"を使用します。
- X
  引数の整数を符号なし 16 進数に変換します。0AH～0FH に大文字の"ABCDEF"を使用します。
- c
  引数の文字を 1 文字(ASCII 文字)で出力します。
- s
  引数の文字列 far 型ポインタ(char *)以降(ヌル文字'¥0'又は精度数まで)を文字列に変換します。なお、wchar_t 型の文字列は扱うことができません。
- p
  引数のポインタ(すべての型に対応)を 24 ビットアドレスで出力します。
- n
  n 変換は引数の整数ポインタにそれまでに出力した文字数を格納します。引数は変換されません。
- e
  double の引数を指数形式に変換します。形式は[-]d.dddddde±dd です。
- E
  指数表示の e の代りに E が使用される以外は、e と同じ書式です。

printf

入出力関数

解　説:

- f
  double の引数を[-]d.dddddd 形式に変換します。
- g
  double の引数を e 又は f により指定される形式に変換します。通常は f 型の変換を行います。指数部が-4 以下、又は精度が指数値以下の場合は、e 型に変換されます。
- G
  指数表示の e の代りに E が使用される以外は、g と同じ書式です。
- －
  変換の結果を最小フィールド幅内で左寄せします。デフォルトは右寄せです。
- ＋
  符号付きの変換結果に＋又は－を付けます。デフォルトは、負の数のみ－を付けます。
- ブランク' '
  デフォルトで符号付きの変換結果が符号なしになった場合、頭にブランク' 'を付けます。
- #
  o 変換では、頭に 0 を付けます。
  x、X 変換では 0 以外の時、頭に 0x、0X を付けます。
  e、E、f 変換では、常に小数点を付けます。
  g、G 変換では、常に小数点を付け、小数点以下の 0 も切り捨てずに出力します。

(2) 最小フィールド幅

- 正の 10 進整数で最小のフィールド幅を指定します。
- 変換結果の文字数が指定したフィールド数より少ない場合 左に埋め文字を挿入してフィールド幅を合わせます。
- デフォルトの埋め文字はブランク' 'です。頭に 0 を付けた整数でフィールド幅を指定した場合は、'0'を埋め文字とします。
- －フラグを指定した場合は、変換結果を左寄せし右に埋め文字を挿入します。この場合、埋め文字は常にブランク' 'です。
- 最小フィールド幅にアスタリスク(*)を指定した場合、引数の整数がフィールド幅を指定します。引数の値が負の場合、－フラグの後に正のフィールド幅を指定したことになります。

(3) 精度

'.'の後に正の整数で指定し、'.'のみの場合はゼロを指定したことになります。
機能、及びデフォルト値は変換タイプにより異なります。
精度の指定がない浮動小数点型データの出力は、精度 6 で処理されます。ただし、精度が 0 のときは小数点は出力されません。

- d、i、o、u、x、X 変換の場合
  (1) 変換結果の桁数が指定した桁数より小さい場合、頭に'0'を挿入して桁数を合わせます。
  (2) この指定が最小フィールド幅より大きい場合、こちらの指定が優先されます。
  (3) この指定が最小フィールド幅より小さい場合、最小桁数の処理を行った後、フィールド幅の処理を行います。
  (4) デフォルト値は 1 です。
  (5) ゼロを最小桁数 0 で変換した場合、何も出力しません。

printf

入出力関数

解　　説:

- s 変換の場合
  (1) 最大文字数を表します。
  (2) 変換結果が指定した文字数を越える場合⇒後が切り捨てられます。
  (3) デフォルトには文字数制限がありません。
  (4) 精度の指定にアスタリスク(*)を指定した場合
  (5) → 引数の整数が精度を指定します。
  (6) 引数の値が負の場合⇒精度の指定は無効になります。
- e、E、f 変換の場合
  小数点の後に n 個(n は精度)の数字を出力します。
- g、G 変換の場合
  n 個(n は精度)以上の有効数字を出力しません。

(4) l、L[1]、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。

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。

[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。

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。

Q

qsort

整数算術関数

機　　能: 配列をソートします。

書　　式:
```
#include <stdlib.h>

void qsort( base,nelen,size,cmp( e1,e2 ) );
```

実現方法: 関数

引　　数:

void _far *base;	配列開始アドレス
size_t nelen;	要素数
size_t size;	各要素の大きさ
int cmp( );	要素を比較する関数

戻 り 値: 戻り値はありません

解　　説: 配列をソートします。

R

rand

整数算術関数

機　　能: 疑似乱数を発生します。

書　　式:
```
#include <stdlib.h>

int rand( void );
```

実現方法: 関数

引　　数: 引数はありません。

戻 り 値:
- srand で指定した seed 乱数の系列を返します。
- 発生させる乱数は 0～RAND_MAX 間の値をとります。

realloc

メモリ管理関数

機　　能: 確保済み領域の大きさを変更します。

書　　式:
```
#include <stdlib.h>

void _far * realloc( cp, nbytes );
```

実現方法:

引　　数:

void _far *cp;	変更前のメモリ領域へのポインタ
size_t nbytes;	変更するメモリ領域の大きさ(バイト数)

戻 り 値:
- 大きさが変更された領域のポインタを返します。
- 指定した大きさの領域が確保できなかった場合は NULL を返します。

解　　説:
- 関数 malloc や calloc ですでに確保済みの領域の大きさを変更します。
- 引数"cp"にすでに確保済みのポインタを指定し、引数"nbytes"で変更する大きさを指定します。

S

scanf

入出力関数

機　　能: stdin からの書式付き入力を行います。

書　　式:
```
#include <stdio.h>
#include <ctype.h>

int scanf( format, argument... );
```

実現方法: 関数

引　　数: const char _far *format;　　書式指定文字列のポインタ

戻 り 値: format で与える文字列の%以降の指定は、次の意味を持ちます。[]内は省略可能です。

> **書式：**
> %[＊][最大フィールド幅][修飾子(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。

(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。

scanf

入出力関数

解　説:

- c
  文字を格納します。スペース文字は読み飛ばさずにそのまま格納します。
  最大フィールド幅で 2 以上を指定した場合は、複数の文字を格納します。ただし、この場合はヌル文字'¥0'は付加しません。
  対応する引数は文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。
- p
  引数のポインタを出力します。
- [ ]
  [ ]内の文字(複数が指定可能)を入力している間、入力文字を格納します。 []内の文字以外を入力した場合、格納を中止します。
  また[の次に^(サーカムフレックス)を指定した場合は、^と]の間で指定した文字以外が入力許可文字となります。
  指定した文字を入力した場合、格納を中止します。
  対応する引数は自動的に付加するヌル文字'¥0'を含む文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。
- n
  書式変換で既に読み込まれた文字数を格納します。対応する引数は整数へのポインタでなければいけません。
- e、E、f、g、G
  浮動小数点の形式に変換します。修飾子 l を指定したとき、対応する引数は double へのポインタとなります。デフォルトは float へのポインタです。

(2) *(データ格納の抑止指定)

- *が指定されている場合、変換したデータを引数に格納することを抑止します。

(3) 最大フィールド幅

- 正の 10 進整数で最大入力文字数を指定します。1 つの書式変換において、この文字数よりも多くの文字を読み込むことはありません。
- 指定した文字数分の文字を読み込む以前に、スペース文字(関数 isspace()で真となる文字)、又は要求する書式以外の文字を入力した場合は、その時点で読み込みを中止します。

(4) 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:
```
#include <stdio.h>

int sprintf( pointer, format, argument... );
```

実現方法: 関数

引　　数:

char _far *pointer;	格納先のポインタ
const char _far *format;	書式指定文字列のポインタ

戻 り 値: 出力した文字数を返します。

解　　説:
- format の指定に従って argument を文字列に変換し、pointer 以降に格納します。
- format の指定方法は、printf と同様です。

sqrt

数学関数数

機　　能: 数値の平方根を計算します。

書　　式:
```
#include <math.h>

double sqrt( x );
```

実現方法: 関数

引　　数:

double x;		実数

戻 り 値: 平方根値を返します。

srand

整数算術関数

機　　能: 疑似乱数発生ルーチンにシードを与えます。

書　　式: #include <stdlib.h>

void srand(seed);

実現方法: 関数

引　　数: unsigned int seed; 乱数の系列値

戻 り 値: 戻り値はありません。

解　　説: 引数"seed"を用いて、rand によって与えられる疑似乱数系列を初期化します。

sscanf

入出力関数

機　　能: 文字列からの書式付き入力を行います。

書　　式: #include <stdio.h>

int sscanf(string, format, argument...);

実現方法: 関数

引　　数: const char _far *string; 入力文字列のポインタ
const char _far *format; 書式指定文字列のポインタ

戻 り 値:
- 各引数 argument に格納したデータ数を返します。
- ヌル文字('¥0')をデータとして入力した場合、EOF を返します。

解　　説:
- format の指定に従って入力文字を変換し、各引数 argument が示す変数に格納します。
- argument は各変数の far 型ポインタでなければいけません。
- format の指定方法は、scanf と同様です。

strcat

文字列操作関数

機　　能: 文字列の連結を行います。

書　　式:
```
#include <string.h>

char _far * strcat( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	連結先の文字列へのポインタ
const char _far *s2;	連結する文字列へのポインタ

戻 り 値: 連結された文字列領域(s1)へのポインタを返します。

解　　説:
- "s1"で示される文字列の最後に、"s2"で示される文字列を連結します。
- 連結された文字列は、NULL で終了します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strchr

文字列操作関数

機　　能: 文字列より文字の検索を行います。

書　　式:
```
#include <string.h>

char _far * strchr( s, c );
```

実現方法: 関数

引　　数:

const char _far *s;	検索先の文字列へのポインタ
int c;	検索する文字

戻 り 値:
- 文字列"s"中で最初に現れた文字"c"の位置を返します。
- 文字列"s"が文字"c"を含まない場合は NULL を返します。

解　　説:
- "s"で示される領域の先頭より、"c"で示される文字を検索します。
- ?¥0'も検索対象とすることができます。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strcmp

文字列操作関数

機　　能: 2 つの文字列の比較を行います。

書　　式:
```
#include <string.h>

int strcmp( s1, s2 );
```

実現方法: マクロ、関数

引　　数:

const char _far *s1;	比較する 1 番目の文字列へのポインタ
const char _far *s2;	比較する 2 番目の文字列へのポインタ

戻 り 値:
- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　1 番目の文字列(s1)の方が大きい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説:
- 通常は、マクロが使用されます。ライブラリ関数を使用したい場合は、#include<string.h> の記述の後に、 #undef strcmp と記述してください。
- NULL で終了している 2 つの文字列をバイト単位に比較します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX 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は、#include <string.h> の記述の後に、 #undef strcpy と記述してください。
- 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化により関数のインライン展開を行う可能性があります。

strcspn

文字列操作関数

機　　能: 指定外文字列の長さを求めます。

書　　式:
```
#include <string.h>

size_t strcspn( s1, s2 );
```

実現方法: 関数

引　　数:

const char _far *s1;	検索先の文字列へのポインタ
const char _far *s2;	検索する文字列へのポインタ

戻 り 値: 指定外文字列の長さを返します。

解　　説:
- "s1"で示される領域の先頭より、"s2"で示される文字列に含まる文字以外で構成される最初の部分の文字列の長さを求めます。
- '¥0'は検索対象とすることができません。

stricmp

文字列操作関数

機　　能: 2 つの文字列の比較を行います(英字はすべて大文字として扱います)。

書　　式:
```
#include <string.h>

int stricmp( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	比較する 1 番目の文字列へのポインタ
char _far *s2;	比較する 2 番目の文字列へのポインタ

戻 り 値:
- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　1 番目の文字列(s1)の方が大きい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説: NULL で終了している 2 つの文字列をバイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。

strerror

文字列操作関数

機　　能: エラー番号を文字列に変換します。

書　　式:
```
#include <string.h>

char _far * strerror( errcode );
```

実現方法: 関数

引　　数:

int errcode;	エラーコード

戻 り 値: エラーコードに対するメッセージ文字列へのポインタを返します。

解　　説: stderr は静的な配列に対してポインタを返します。

strlen

文字列操作関数

機　　能: 文字列の長さを求めます。

書　　式:
```
#include <string.h>

size_t strlen( s );
```

実現方法: 関数

引　　数:

const char _far *s;	長さを求める文字列へのポインタ

戻 り 値: 文字列の長さを返します。

解　　説: "s"で示される文字列(NULL まで)の長さを求めます。

strncat

文字列操作関数

機　　能: 文字列の(n 文字)連結を行います。

書　　式:
```
#include <string.h>

char _far * strncat( s1, s2, n );
```

実現方法: 関数

引　　数:

char _far *s1;	連結先の文字列へのポインタ
const char _far *s2;	連結する文字列へのポインタ
size_t n;	連結する文字数

戻 り 値: 連結された文字列領域へのポインタを返します。

解　　説:

- "s1"で示される文字列の最後に、"n"で指定された文字数の文字を"s2"で示される文字列より連結します。
- 連結された文字列は、NULL で終了します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strncpy

文字列操作関数

機　　能: 文字列の複写を行います。

書　　式:
```
#include <string.h>

char _far * strncpy( s1, s2, n );
```

実現方法: 関数

引　　数:

char _far *s1;	複写先の文字列へのポインタ
const char _far *s2;	複写元の文字列へのポインタ
size_t n;	複写する文字数

戻 り 値: 複写先の文字列へのポインタを返します。

解　　説:
- "s1"で示される領域に、"n"で指定された文字数の文字を、"s2"で示される文字列より複写します。ただし、この時"n"で指定された文字数より多い部分は複写されず、その後に'¥0'を付加することも行いません。逆に、"s2"の示す文字列の長さが"n"文字より短い場合は、複写された文字列の後に"n"文字になるまで'¥0'が付加されます。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strnicmp

文字列操作関数

機　　能: 2 つの文字列の(n 文字)比較を行います(英字はすべて大文字として扱います)。

書　　式:
```
#include <string.h>

int strnicmp( s1, s2, n );
```

実現方法: 関数

引　　数:

char _far *s1;	比較する 1 番目の文字列へのポインタ
char _far *s2;	比較する 2 番目の文字列へのポインタ
size_t n;	比較する文字数

戻 り 値:
- 戻り値= =0　2 つの文字列は等しい
- 戻り値>0　2 つの文字列は等しい
- 戻り値<0　2 番目の文字列(s2)の方が大きい

解　　説:
- NULL で終了している 2 つの文字列を"n"文字分バイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strrchr

文字列操作関数

機　　能: 文字列より文字の検索を行います。

書　　式:
```
#include <string.h>

char _far * strrchr( s, c );
```

実現方法: 関数

引　　数:

const char _far *s;	検索先の文字列へのポインタ
int c;	検索する文字

戻 り 値:
- 文字列"s"中で最後に現れた文字"c"の位置を返します。
- 文字列"s"が文字"c"を含まない場合は NULL を返します。

解　　説:
- "s"で示される領域の末尾より、"c"で示される文字を検索します。
- '¥0'も検索対象とすることができます。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtod

文字列操作関数

機　　能: 文字列を倍精度浮動小数に変換します。

書　　式:
```
#include <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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

strtok

文字列操作関数

機　　能: 文字列の切り出しを行います。

書　　式:
```
#include <string.h>

char _far * strtok( s1, s2 );
```

実現方法: 関数

引　　数:

char _far *s1;	切り出し先の文字列へのポインタ
const char _far *s2;	区切り文字へのポインタ

戻 り 値:
- 見つかった場合、切り出したトークンへのポインタを返します。
- 見つからない場合、NULL を返します。

解　　説:
- 最初の呼び出しでは、最初の字句の先頭文字へのポインタを返します。このとき返される文字の最後には、ＮＵＬＬ文字が書き込まれます。その後の呼び出し(" s 1 "が NULL の場合)では、順次、次の字句が返されます。"s1"に字句がなくなると NULL を返します。
- オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX 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, -OS, -OR_MAX, -OS_MAX を指定した場合は、最適化によりコード効率の良い別の関数等を選択する可能性があります。

T

tan

数学関数

機　　能: タンジェントを計算します。

書　　式: #include <math.h>

double tan(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする"x"のタンジェントを返します。

tanh

数学関数

機　　能: 双曲線タンジェントを計算します。

書　　式: #include <math.h>

double tanh(x);

実現方法: 関数

引　　数: double x; 実数

戻 り 値: ラジアンを単位とする"x"の双曲線タンジェントを返します。

tolower

文字操作関数

機　　能: 英大文字を英小文字に変換します。

書　　式:
```
#include <ctype.h>

int tolower( c );
```

実現方法: マクロ

引　　数:

int c;		変換する文字

戻 り 値:
- 引数が英大文字の場合、英小文字を返します。
- それ以外は渡した引数を返します。

解　　説: 引数の英大文字を英小文字に変換します。

toupper

文字操作関数

機　　能: 英小文字を英大文字に変換します。

書　　式:

実現方法: マクロ

引　　数:

int c;		変換する文字

戻 り 値:
- 引数が英小文字の場合、大文字を返します。
- それ以外は渡した引数を返します。

解　　説: 引数の英小文字を英大文字に変換します。

U

ungetc

入出力関数

機　　能: ストリームへ 1 文字戻します。

書　　式:
```
#include <stdio.h>

int ungetc( c, stream );
```

実現方法: マクロ

引　　数:

int c;	戻す文字
FILE _far *stream;	ストリームのポインタ

戻 り 値:
- 正常な場合は、戻した 1 文字を返します。
- ストリームが書き込みモード、エラー、EOF の場合、又は EOF を戻そうとした場合、EOF を返します。

解　　説:
- ストリームに 1 文字を戻します。
- 0x1A コードを終了コードとみなし、以降のデータを受け付けません。

V

vfprintf

入出力関数

機　　能: フォーマットを指定してテキストを指定ストリームに書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( stream, format, ap... );
```

実現方法: 関数

引　　数:

FILE _far *stream;	ストリームのポインタ
const char _far *format;	書式指定文字列へのポインタ
va_list ap;	引数リストの先頭へのポインタ

戻 り 値: 出力した文字数を返します。

解　　説:
- フォーマットを指定してテキストを指定ストリームに書き込みます。
- 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

vprintf

入出力関数

機　　能: フォーマットを指定してテキストを stdout に書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vprintf( format, ap... );
```

実現方法: 関数

引　　数:

const char _far *format;	書式指定文字列へのポインタ

戻 り 値: 出力した文字数を返します

解　　説:
- フォーマットを指定してテキストを stdout に書き込みます。
- 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

vsprintf

入出力関数

機　　能: フォーマットを指定してテキストを指定バッファに書き込みます。

書　　式:
```
#include <stdarg.h>
#include <stdio.h>

int vfprintf( s, format, ap... );
```

実現方法: 関数

引　　数:

char _far *s;	格納先へのポインタ
const char _far *format;	書式指定文字列へのポインタ
va_list ap;	引数リストの先頭へのポインタ

戻 り 値: 出力した文字数を返します。

解　　説: 可変長引数にポインタを記述する場合は、far 型ポインタでなければなりません。

W

wcstombs

多バイト文字/多バイト文字列操作関数

機　　能: ワイド文字列をマルチバイト文字列に変換します。

書　　式:
```
#include <stdlib.h>

size_t _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;	ワイド文字マルチバイト文字に含めたバイト数を返します。

戻 り 値:
- マルチバイト文字に含めたバイト数を返します。
- 対応するマルチバイト文字が存在しない場合は-1 を返します。
- ワイド文字が 0 の場合は 0 を返します。

E.2.4 標準関数ライブラリの使用に関する注意事項

a. 標準ヘッダファイルに関する注意事項

標準関数ライブラリ中の関数を使用する時は、必ず指定された標準ヘッダファイルをインクルードしてください。インクルードしない場合、引数及び戻り値の整合がとれませんのでプログラムが正常に動作しないことがあります。

b. 標準ライブラリ関数の最適化に関する注意事項

最適化オプション-O[3～5], -OR, -OS, -OR_MAX, -OS_MAX のいずれかを指定した場合、標準関数に関する最適化を行ないます。この最適化は-Ono_stdlib を指定することにより、抑止することができます。ユーザー関数として、標準ライブラリ関数と同名の関数を使用する場合は、この最適化を抑止してください。

(1) 関数のインライン埋め込み

関数 strcpy 及び memcpy について、【表 E.13】の条件を満たす時、関数のインライン埋め込みを行ないます。

表E.13 標準ライブラリ関数に対する最適化条件

関数名	最適化条件	記述例
strcpy	第 1 引数：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

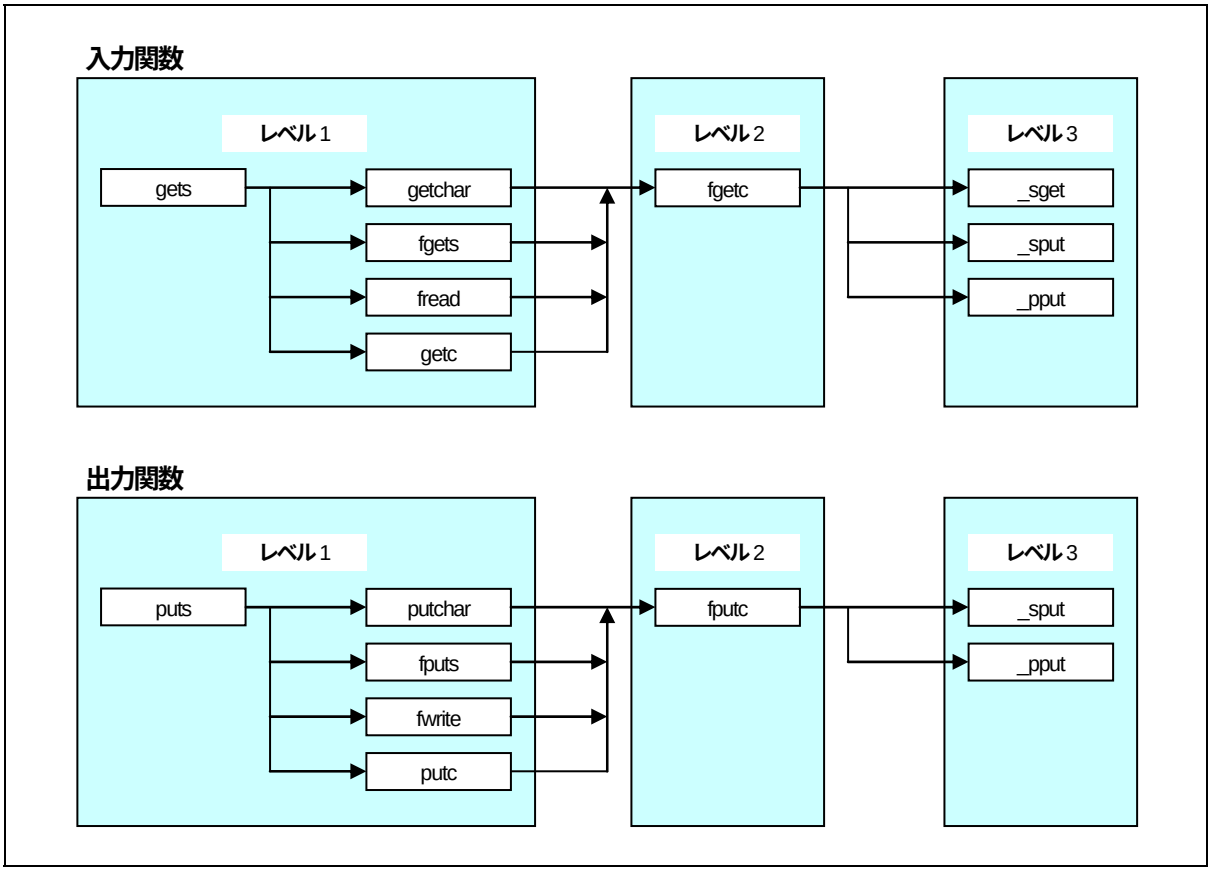
入出力関数として scanf、printf 等の高機能な関数ライブラリを用意しています。これらの関数は一般的に高水準入出力関数と呼ばれます。高水準入出力関数は、ハードウェア環境に依存した低水準入出力関数の組み合わせで実現しています。

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。

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。

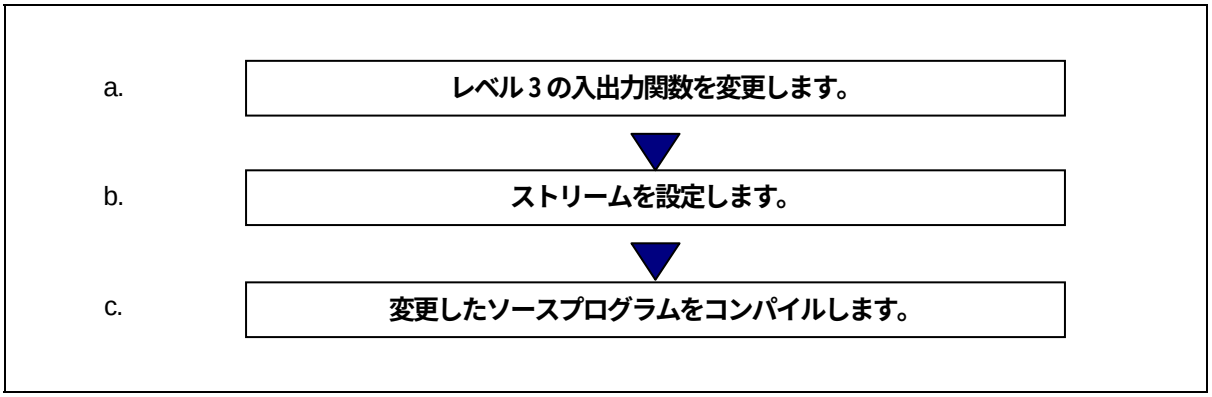
マイコンのハードウェア(入出力ポート)に依存しているのは最下位のレベル 3 の関数のみです。入出力関数をアプリケーションプログラムで使用する場合、必要に応じてレベル 3 の関数のソースファイルを書き換えることで、システムに順応した関数に変更できます。



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。

(1) 回路の諸設定

- プロセッサ動作モード：マイクロプロセッサモード
- クロック発信周波数：20MHz
- 外部バス幅：16 ビット

(2) シリアル通信の初期設定

- UART1 を使用
- ボーレート：9600bps
- データ長：8 ビット
- パリティ：なし
- ストップビット：2 ビット

※これらのシリアル通信の初期設定は init 関数(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.14】に示します。

表E.14 レベル 3 の関数の仕様

入力関数	引数	戻り値(int 型)
_sget _sput _pput	なし	正常に入力できた場合は入力した文字を返します。 エラーの場合は EOF を返します。

出力関数	引数(int 型)	戻り値(int 型)
_sput _pput	出力する文字	正常に出力できた場合は 1 を返します。 エラーの場合は EOF を返します。

シリアル通信は、M16C/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。

(1) device.c ファイルの先頭に記述している条件コンパイルの記述を削除します。

(2) #pragma 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。

(4) speed 関数も UART0 用に複製し、speed0 等の異なった関数名に書き換えます。

以上で device.c ファイルの変更は終了します。

次に、入出力関数の初期設定を行っている init 関数(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

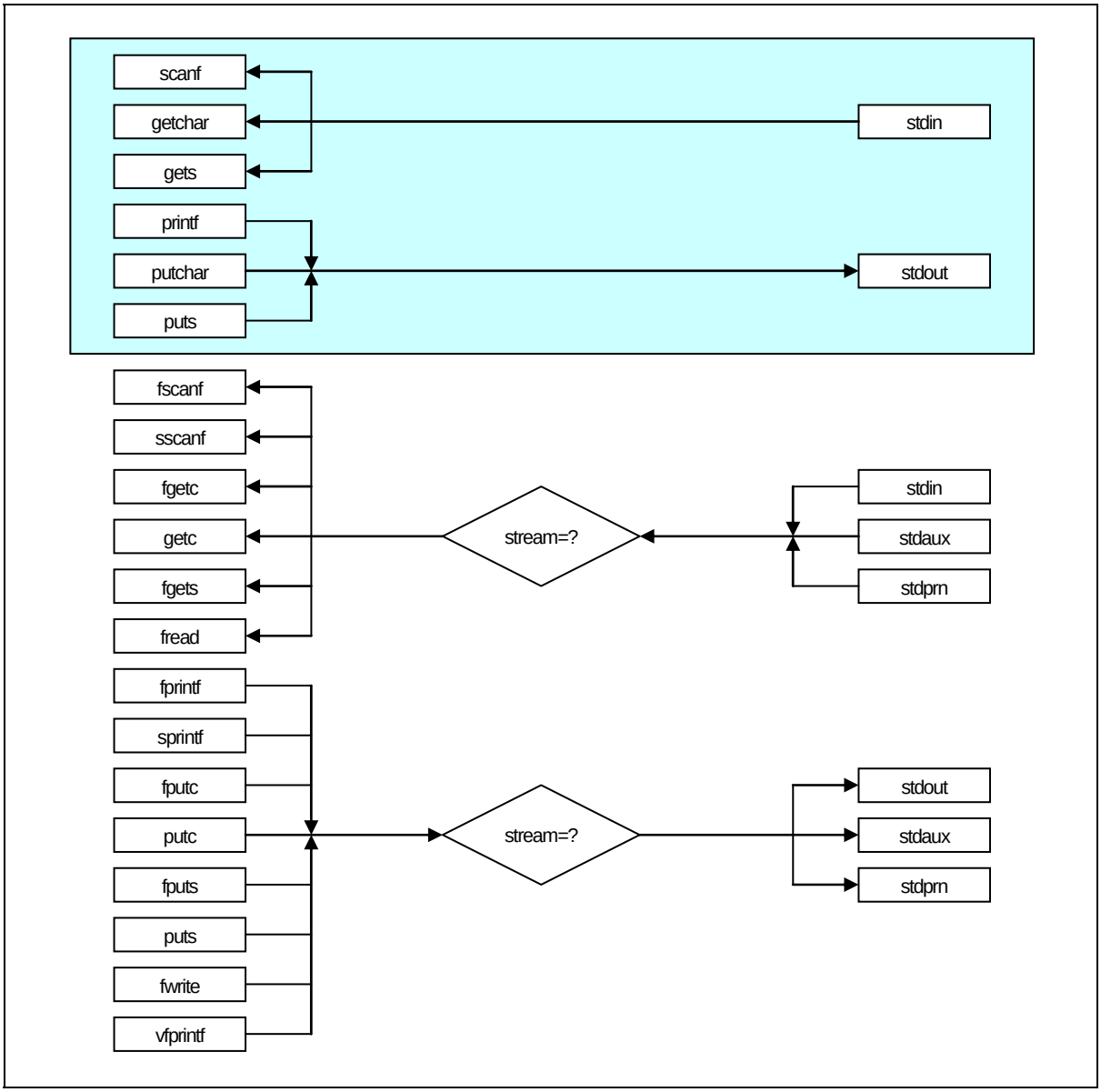
本コンパイラの標準ライブラリは stdin、stdout、stderr、stdaux、stdprn の 5 種類のストリーム情報を外部構造体として持っています。この外部構造体は標準ヘッダファイル stdio.h 内で定義されており、各ストリームのモード情報(入力ストリームか出力ストリームを表すフラグ)やステータス情報(エラー又は EOF を表すフラグ)等を管理しています。

表E.15 ストリーム情報一覧表

ストリーム情報	名称
stdin	標準入力
stdout	標準出力
stderr	標準エラー出力（stderr は stdout に定義されています。）
stdaux	標準補助入出力
stdprn	標準プリンタ出力

本コンパイラの標準ライブラリ関数中、【図 E.3】の網掛部に示す関数の対応するストリームは標準入力(stdin)又は標準出力(stdout)に固定しています。これらの関数に関しては、ストリームを変更することはできません。なお、stderr は stdout に#define で定義されています。

ストリームは、fgetc、fputc 等の引数としてストリームへのポインタを指定できる関数のみ変更することができます。



図E.3 関数とストリームの相互関係図

【図 E.4】に stdio.h 内のストリーム定義部を示します。

```
/***********************************************************************
*
* standard I/O header file
                :
            (省略)
                :
typedef struct _iobuf {
            char    _buff;                          /* Store buffer for ungetc */           ← (1)
            int     _cnt;                           /* Strings number in _buff(1 or 0) */   ← (2)
            int     _flag;                          /* Flag */                              ← (3)
            int     _mod;                           /* Mode */                              ← (4)
            int     (*_func_in)(void);      /* Pointer to one byte input function */        ← (5)
            int     (*_func_out)(int);      /* Pointer to one byte output function */       ← (6)
} FILE;
#define     _IOBUF_DEF
                :
            (省略)
                :
extern FILE _iob[];
#define     stdin       (&_iob[0])  /* Fundamental input */
#define     stdout      (&_iob[1])  /* Fundamental output */
#define     stdaux      (&_iob[2])  /* Fundamental auxialiary input output */
#define     stdprn      (&_iob[3])  /* Fundamental printer output */

#define     stderr      stdout              /* NC no-support */

/***********************************************************
*
***********************************************************/
#define     _IOREAD  1          /* Read only flag */
#define     _IOWRT   2          /* Write only flag */
#define     _IOEOF   4          /* End of file flag */
#define     _IOERR   8          /* Error flag */
#define     _IORW    16         /* Read and write flag */
#define     _NFILE   4          /* Stream number */
#define     _TEXT    1          /* Text mode flag */
#define     _BIN     2          /* Binary mode flag */

            (以降省略)
                :
```

図E.4 stdio.h **内のストリーム定義部** (stdio.h)

【図 E.4】に示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。

(1) char _buff

関数 scanf、fscanf では入力の際に 1 文字分の先読みを行っています。
先読みした文字が不要な場合は関数 ungetc を呼び出して、先読みした文字をこの変数に格納します。
入力関数はこの変数にデータが存在する場合はこのデータを入力データとします。

(2) int _cnt

_buff のデータ数を格納します(0 もしくは 1)。

(3) int _flag

読み込み専用フラグ(_IOREAD)、書き込み専用フラグ(_IOWRT)、読み書き両用フラグ(_IORW)、エンドオブファイルフラグ(_IOEOF)、エラーフラグ(_IOERR)のフラグを格納します。

- _IOREAD、_IOWRT、_IORW
  ストリームの動作モードを指定するフラグです。これらのフラグは、ストリームの初期化部分で設定します。
- _IOEOF、_IOERR
  入出力関数内で EOF、エラーの発生に応じて設定されます。

(4) int _mod

テキストモード(_TEXT)、バイナリモード(_BIN)を表すフラグを格納します。

- テキストモード
  入出力データのエコーバック、文字変換を行います。エコーバック、文字変換の詳細については、関数 fgetc、fputc のソースプログラム(fgetc.c、fputc.c)を参照ください。
- バイナリモード
  入出力データを無変換で扱います。これらのフラグはストリームの初期化部分で設定します。

(5) int (*_func_in)()

ストリームが読み込みモード(_IOREAD)又は読み書き両用モード(_IORW)の場合、レベル 3 入力関数のポインタを格納します。それ以外の場合は NULL ポインタを格納します。

この情報をもとに、レベル 2 入力関数はレベル 3 入力関数を間接呼び出しで呼び出します。

(6) int (*_func_out)()

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 /* UART1 : default */
    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。

(1) UART0 用のストリームには、標準補助入出力(stdaux)を使用します。
(2) 標準補助入出力に対するフラグ(_flag)、モード(_mod)をシステムに合わせて設定します。
(3) 標準補助入出力に対するレベル 3 関数のポインタを設定します。
(4) speed 関数に対する条件コンパイルコマンドを削除し、UART0 用の speed0 関数に書き換えます。

以上の設定で、2 本の UART が使用できます。ただし、標準入出力のストリームを使用する関数は UART0 が使用する標準補助入出力に対して使用することができません。したがって、関数の引数にストリームを記述できる関数で使用してください。【図 E.6】に init 関数の変更例を示します。

```
void init( void )
{
                :
        (省略)
                :
    stdaux->_flag = _IORW;          ← (2) 読み書き両用モードに設定
                :
        (省略)
                :
    stdaux->_mod = _TEXT;           ← (2) テキストモードに設定
                :
        (省略)
                :
    stdaux->_func_in = _sget0;      ← (3) UART0 用のレベル 3 の入力関数を指定
                :
        (省略)
                :
    stdaux->_func_out = _sput0;     ← (3) UART0 用のレベル 3 の出力関数を指定
                :
        (省略)
                :
    speed(_96, _B8, _PN, _S2);      ← (4) UART0 用の speed 関数を指定
    init_prn();
}
```

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.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ウォ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した場合、エラーメッセージを画面に表示後、コンパイルを中止します。【図F.1】～【図F.3】にエラーメッセージとウォ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  ← 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。
```

図F.2 各コマンドのエラー出力形式

```
[Warning(cpp308. ウォーニング番号):ファイル名,行番号] ウォーニングメッセージ
[Warning(ccom):ファイル名,行番号] ウォーニングメッセージ
```

図F.3 各コマンドのウォーニング出力形式

F.2 nc308 エラーメッセージ

【表F.1】および【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 (1/2)

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コンパイルオプションを指定してください。各処理系のコマンドラインは、コンパイルオプション-v で確認することができます。
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 で使用できないコンパイルオプションオプション-?を使用しています。 ⇒ 正しいコンパイルオプション指定してください。
illegal option	• -as308 や-ln308 などの各処理系に指示するオプションが 100 文字以上です。 ⇒ 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 '-?'	• コンパイルオプション-?に必要なパラメータがありません。 ⇒ コンパイルオプション-?の次に必要なパラメータを指定してください。 • コンパイルオプション-?とパラメータの間にスペースがあります。 ⇒ コンパイルオプション-?とパラメータ間のスペースを削除してください。
Invalid option '-o'	• コンパイルオプション-o 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 (2/2)

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	• 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	• 指定されたコンパイルオプションの数が 100 以上です。 ⇒ コンパイルオプションは 99 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.5】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 (1/3)

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。
		• コマンドラインがコンパイルオプション-o で終了しています。 ⇒ コンパイルオプション-o 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コンパイルオプション-l が制限値を越えました。 ⇒ コンパイルオプション-l を 8 個以下にしてください。
		• コマンドラインがコンパイルオプション-l で終了しています。 ⇒ コンパイルオプション-l の後に出力ファイル名を指定してください。
		• コンパイルオプション-D 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。 ⇒ コンパイルオプション-D の後に出力ファイル名を指定してください。
		• コマンドラインがコンパイルオプション-D で終了しています。 ⇒ コンパイルオプション-D の後に出力ファイル名を指定してください。
		• コンパイルオプション-U 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。
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。

表F.4 cpp308 エラーメッセージ一覧表 (2/3)

番号	エラーメッセージ	エラー内容と対策
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、コンパイルオプション-l 指定)を短くしてください。
18	not enough memory for macro include file not found	• 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。 ⇒ 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、コンパイルオプション-l 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 以下にしてください。
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。
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。

表F.5 cpp308 エラーメッセージ一覧表 (3/3)

番号	エラーメッセージ	エラー内容と対策
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 erro	• #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。
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 ウォーニングメッセージ

【表F.6】にプリプロセッサcpp308 が出力するウォ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.6 cpp308 ウォーニングメッセージ一覧表

番号	ウォーニングメッセージ	エラー内容と対策
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 以下にしてください。
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.7】～【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.7 ccom308 エラーメッセージ一覧表 (1/13)

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 redefined	• #pragma プラグマ名 において同じ関数を重複して定義しています。 ⇒ #pragma プラグマ名の宣言を 1 回にしてください。
#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。
'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。

表F.8 ccom308 エラーメッセージ一覧表 (2/13)

エラーメッセージ	エラー内容と対策
'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。
-M82,-M90 duplicated option	• オプション"-M82"と"-M90"が重複して選択されています。 ⇒ オプション"-M82"と"-M90"のいずれかひとつを選択して下さい。
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 回以下にしてください。
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。
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。

表F.9 ccom308 エラーメッセージ一覧表 (3/13)

エラーメッセージ	エラー内容と対策
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アクセス権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。
cannot refer to the range outside of the stack frame	• スタックフレーム領域外を参照しています。 ⇒ 正しく指定してください。
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	• 同じ識別子を持つ auto 変数が 2 回以上記述されています。 ⇒ 正しく記述してください。
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。
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。

表F.10 ccom308 エラーメッセージ一覧表 (4/13)

エラーメッセージ	エラー内容と対策
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。
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。

表F.11 ccom308 エラーメッセージ一覧表 (5/13)

エラーメッセージ	エラー内容と対策
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。
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。

表F.12 ccom308 エラーメッセージ一覧表 (6/13)

エラーメッセージ	エラー内容と対策
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。
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。

表F.13 ccom308 エラーメッセージ一覧表 (7/13)

エラーメッセージ	エラー内容と対策
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 array type	• 不完全型の配列を宣言することはできません。 ⇒ 多次元配列の宣言時、配列の要素数を明示してください。
invalid operation for pointer to incomplete type	• 不完全型へのポインタに対して無効な演算をしています。 ⇒ 構造体のメンバを定義する、または配列の要素数を明示して完全型にしてください。
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 で記述した番号またはフォーマット指定が間違っています。 ⇒ 正しく指定してください。
Invalid #pragma INTERRUPT vector number	• #pragma INTERRUPT で記述した番号またはフォーマット指定が間違っています。 ⇒ 正しく指定してください。
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。 ⇒ witch、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。

表F.14 ccom308 エラーメッセージ一覧表 (813)

エラーメッセージ	エラー内容と対策
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。
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。 ⇒ 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。

表F.15 ccom308 エラーメッセージ一覧表 (9/13)

エラーメッセージ	エラー内容と対策
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, 共用体のメンバを参照しています。 ⇒ 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。
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。

表F.16 ccom308 エラーメッセージ一覧表 (10/13)

エラーメッセージ	エラー内容と対策
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	• メモリ空間が不足しています。 ⇒ メモリを増やすか、Windows 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。
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。
reinitialized of 変数名 '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。

表F.17 ccom308 エラーメッセージ一覧表 (11/13)

エラーメッセージ	エラー内容と対策
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。
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。
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。

表F.18 ccom308 エラーメッセージ一覧表 (12/13)

エラーメッセージ	エラー内容と対策
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。
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。

表F.19 ccom308 エラーメッセージ一覧表 (13/13)

エラーメッセージ	エラー内容と対策
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 ウォーニングメッセージ

【表F.20】～【表F.29】にコンパイラccom308 が出力するウォ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.20 ccom308 ウォーニングメッセージ一覧表 (1/10)

ウォーニングメッセージ	ウォ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 HANDLER の両方を指定しています。 ⇒ #pragma プラグマ名と#pragma HANDLER は、排他的に指定してください。
#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#pragma TASK 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。
#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。

表F.21 ccom308 ウォーニングメッセージ一覧表 (2/10)

ウォーニングメッセージ	ウォーニング内容と対策
#pragma ASM line too long, then cut	• #pragma ASM で記述できる一行の文字数1024バイトを越えています。 ⇒ 1024 バイト以下で記述してください。
#pragma directive conflict	• 1 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 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 to object for 変数名,ignored	• #pragma DMAC 宣言された変数は、オブジェクト型（または不完全型）への 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	• 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	• 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参照しています。 ⇒ 参照しないでください。
#pragma control for function duplicate, ignored	• #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。

表F.22 ccom308 ウォーニングメッセージ一覧表 (3/10)

ウォーニングメッセージ	ウォーニング内容と対策
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。記述が正しい場合は、このウォ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。記述が正しい場合は、このウォ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。
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に関数名を記述しています。
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います。 ⇒ 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。

表F.23 ccom308 ウォーニングメッセージ一覧表 (4/10)

ウォーニングメッセージ	ウォーニング内容と対策
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	• コンパイルオプション-fCE 使用時において、列挙子の値が 255 を越えました。 ⇒ 255 以下で表現できるように記述してください。
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。
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 duplicated, this declare ignored	• メンバ名が重複して定義されています。この宣言を無視します。 ⇒ メンバ名の宣言を 1 つにしてください。
identifier (変数名) is duplicated	• 変数名が重複して定義されています。この宣言を無視します。 ⇒ 変数名の宣言を 1 つにしてください。
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。

表F.24 ccom308 ウォーニングメッセージ一覧表 (5/10)

ウォーニングメッセージ	ウォーニング内容と対策
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います。 ⇒ 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。
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 function 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。

表F.25 ccom308 ウォーニングメッセージ一覧表 (6/10)

ウォーニングメッセージ	ウォ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。
keyword (キーワード) are reserved for future	• 将来のために予約されているキーワードを使用しています。 ⇒ 別の名前に変更してください。
large type was implicitly cast to small type	• 大きい型から小さい型への代入により、値の上位バイト(ワード)が失われる可能性があります。 ⇒ 型を確認してください。記述が正しい場合は、このウォーニングは無視してください。
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 assignment	• レジスタ引数の関数のアドレスを、レジスタ引数でない(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。
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。

表F.26 ccom308 ウォーニングメッセージ一覧表 (7/10)

ウォーニングメッセージ	ウォーニング内容と対策
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。このウォーニングは、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、このウォ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。コンパイルオプション-Wnon_prototype を指定した時のみ出力されます。 ⇒ 関数のプロトタイプ宣言を記述するか、コンパイルオプション-Wnon_prototype を指定しないでください。
non-prototyped 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(コンパイルオプション-Wnon_prototype 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。
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 nonprototyped 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。
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。
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。

表F.27 ccom308 ウォーニングメッセージ一覧表 (8/10)

ウォーニングメッセージ	ウォーニング内容と対策
size of incomplete array type	• 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。 • 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。 ⇒ メンバーは near、far を指定しないでください。
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。

表F.28 ccom308 ウォーニングメッセージ一覧表 (9/10)

ウォーニングメッセージ	ウォーニング内容と対策
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。 ⇒ プ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)	• コンパイルオプション-dx は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown function option (-Wxxx)	• コンパイルオプション-Wxxx は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown function option (-fx)	• コンパイルオプション-fx は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown function option (-gx)	• コンパイルオプション-gx は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown optimize option (-mx)	• コンパイルオプション-mx は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown optimize option (-Ox)	• コンパイルオプション-Ox は、指定できません。 ⇒ コンパイルオプションを正しく指定してください。
Unknown option (-x)	• コンパイルオプション-x は、指定できません。 ⇒ コンパイル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 コンパイルオプション-Wunknown_pragma (-WUP)、および -Wall 指定時のみ表示されます。
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。

表F.29 ccom308 ウォーニングメッセージ一覧表 (10/10)

ウォーニングメッセージ	ウォーニング内容と対策
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。
xxx was declared but never referenced	• 参照されない宣言があります。 ⇒ 宣言を削除してください。

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。

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、

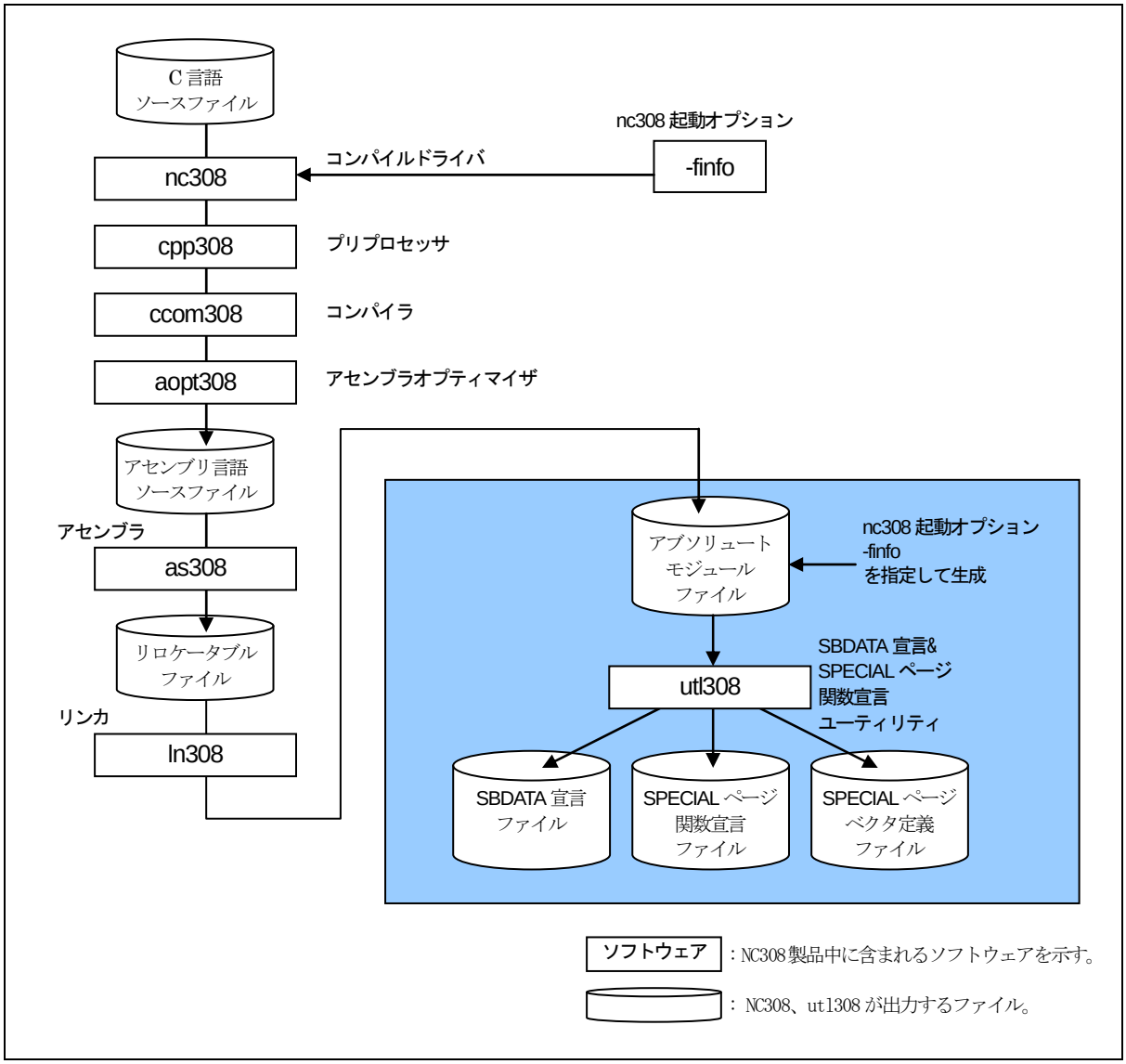
(1) SBDATA 宣言
使用頻度の高い変数から SB 領域に割り当てるための宣言
(#pragma SBDATA)

(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、標準出力へ出力します。)

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 Series Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX-XXXX). Renesas Electronics Corp.
and Renesas Solutions Corp., 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 ELCTRONICS CORPORATION
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 ELCTRONICS CORPORATION
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。どちらのオプションも指定しない場合、utl308 はエラーになります。

(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 の起動は、以下の表に示す書式に従って、必要な情報、パラメータを指定してください。utl308 の起動オプションを【表G.1】に示します。

表G.1 utl308 の起動オプション

オプション	短縮形	機能
-all	なし	● オプション-sb308 と同時に使用する場合 使用頻度が低いため、SB 領域に入らない変数に対しても、コメントの形で SBDATA 宣言を出力します。 ● オプション-sp308 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。
-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。
-fover_write	-fOW	オプション-o 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。
-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、ホストマシンの標準出力に出力します。
-sb308	なし	SBDATA 宣言を出力します。 ● utl308 を使用する場合は、オプション-sb308 またはオプション-sp308 のいずれかを必ず指定してください。いずれも指定しない場合はエラーとなります。
-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。 オプション-sp308 と同時に使用してください。
-sp308	なし	SBDATA 宣言を出力します。 ● utl308 を使用する場合は、オプション-sb308 またはオプション-sp308 のいずれかを必ず指定してください。いずれも指定しない場合はエラーとなります。
-Wstdout	なし	エラー及びウォーニングメッセージを標準出力へ出力します。

-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:
- オプション-sb308 と同時に使用する場合
  使用頻度が低いため、SB 領域に入らない変数に対しても、コメントの形で SBDATA 宣言を出力します。
- オプション-sp308 と同時に使用する場合
  使用頻度が低いため、SPECIAL ページ領域に入らない関数に対しても、コメントの形で SPECIAL 宣言を出力します。

補足説明: 本オプションを使用することにより、一度も呼出される事がない関数を見つけ出す事ができます。
ただし、間接でのみ呼出される関数は、呼出し回数が 0 回と表示されますので、お客様側で確認してください。

-fover_write　　-fOW

SBDATA 宣言または SPECIAL 関数宣言のファイルへの出力

機　　能: オプション-o で指定された出力ファイルが既に存在するか否かをチェックしません。ファイルが存在する場合は上書きします。
オプション-o と同時に指定してください。

-fsection

#pragma SECTION 内の SBDATA 宣言、SPECIAL ページ関数宣言の出力

機　　能: 処理の対象として、#pragma SECTION でセクション変更された領域に配置された変数および関数も含めます。

注　　意: #pragma SECTION を使用して特定の変数および関数を、任意のアドレスへ意図的に配置する事を目的としている場合は、SBDATA 宣言あるいは SPECIAL ページ宣言により、意図したアドレスとは異なるアドレスへ配置されますので、本オプションは指定しないでください。

-o

SBDATA 宣言または SPECIAL 関数宣言のファイルへの出力

機　　能: SBDATA 宣言、または SPECIAL 関数宣言の結果をファイルに出力します。指定が無い場合は表示をホストマシンの標準出力に出力します。
また、指定されたファイルが既に存在する場合は、標準出力に出力します。

-sb308

SBDATA 宣言の出力

機　　能:　SBDATA 宣言を出力します。指定が無い場合はエラーになります。

-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。指定が無い場合はエラーになります。

-sp=*番号*

SPECIAL ページ関数として使用しない番号の指定

機　　能:　SPECIAL ページ関数として使用しない番号を指定します。
オプション-sp308 と同時に使用してください。

-Wstdout

エラーメッセージの標準出力への表示

機　　能:　エラー、およびウォーニングメッセージをホストマシンの標準出力に出力します。

G.3 制限事項

(1) SBDATA 宣言を出力する場合、アセンブラで記述されたファイル中で宣言されているアセンブラ指示命令.sbsym をカウントすることができません。したがって指示命令.sbsym によって定義された外部変数がある場合は、utl308 実行後生成された結果に対して外部変数が SB 領域内に収容できるように調整する必要があります。

(2) SPECIAL ページ関数宣言を出力する場合、アセンブラで記述されたファイル中で宣言されている SPECIAL ページ関数をカウントすることができません。したがってアセンブラで宣言された SPECIAL ページ関数がある場合は、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です。

- _Bool
- unsigned char、signed char
- unsigned short、signed short
- unsigned int、signed int
- unsigned long、signed long
- unsigned long long、signed long lomg

上記の型をもつ変数でも下記の条件を満たす場合は、処理の対象外になります。

- #pragma SECTION で変更されたセクションに配置された変数
- #pragma ADDRESS で定義された変数
- #pragma ROM で定義された変数

既に#pragma SBDATA を用いて宣言された変数がプログラム中に存在する場合は、utl308 はその宣言を優先し、SB 領域の残りから割り当てられる変数を選択します。

G.4.2 SPECIALページ関数宣言が処理対象とする関数

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です。

- static で宣言されていない関数
- 3 回以上呼ばれている関数

上記の関数でも下記の条件を満たす場合は、処理の対象外になります。

- #pragma SECTION で変更されたセクションに配置された関数
- 各#pragma で定義された関数

既に#pragma SPECIAL を用いて宣言された関数がプログラム中に存在する場合は、utl308 はその宣言を優先し、SPECIAL ページ領域の残りから割り当てられる関数を選択します。

G.5 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

utl308 にアブソリュートモジュールファイル (コンパイルオプション-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 のコマンド入力例を、【図G.5】にSBDATA宣言ファイル例を示します。

```
% utl308  -sb308  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)

【図G.4】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"

void        func(void)
{
            (以降省略)
                :
```

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

アセンブラ指示命令.sbsym で外部変数を定義している場合は、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。

アセンブリ言語プログラム:

```
        .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:

```
            :
        (省略)
            :
//#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

utl308 にアブソリュートモジュールファイル(コンパイルオプション-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 のコマンド入力例、【図G.9】にSPECIALページ関数宣言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   SPECIA    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)

【図G.8】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.10】に示します。

```
#include   "special.h"

void          func(void)
{
              (以降省略)
                    :
```

図G.10 SPECIAL ページ関数宣言ファイルの設定例

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】に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	• utl308 オプション-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。utl308 オプション -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 ウォーニングメッセージ

【表G.3】にutl308 が出力するウォ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.3 utl308 ウォーニングメッセージ一覧表

ウォーニングメッセージ	ウォ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。

付録H Call Walker 用.sni ファイル作成ツール gensni 操作方法

High-performance Embedded Workshop のスタック解析ツール Call Walker を使用するためには、入力ファイルとして.sni ファイルが必要です。
.sni ファイルは、アブソリュートモジュールファイルから Call Walker 用.sni ファイル作成ツール gensni を使用して生成します。

H.1 Call Walkerの起動

Call Walker は、High-performance Embedded Workshop に登録されている"Call Walker"を選択、または、High-performance Embedded Workshop の Tools メニューからの選択によって起動します。
Call Walker 起動後は、 [File]メニューの[Import Stack File...]から入力ファイルとして.sni ファイルを指定してください。

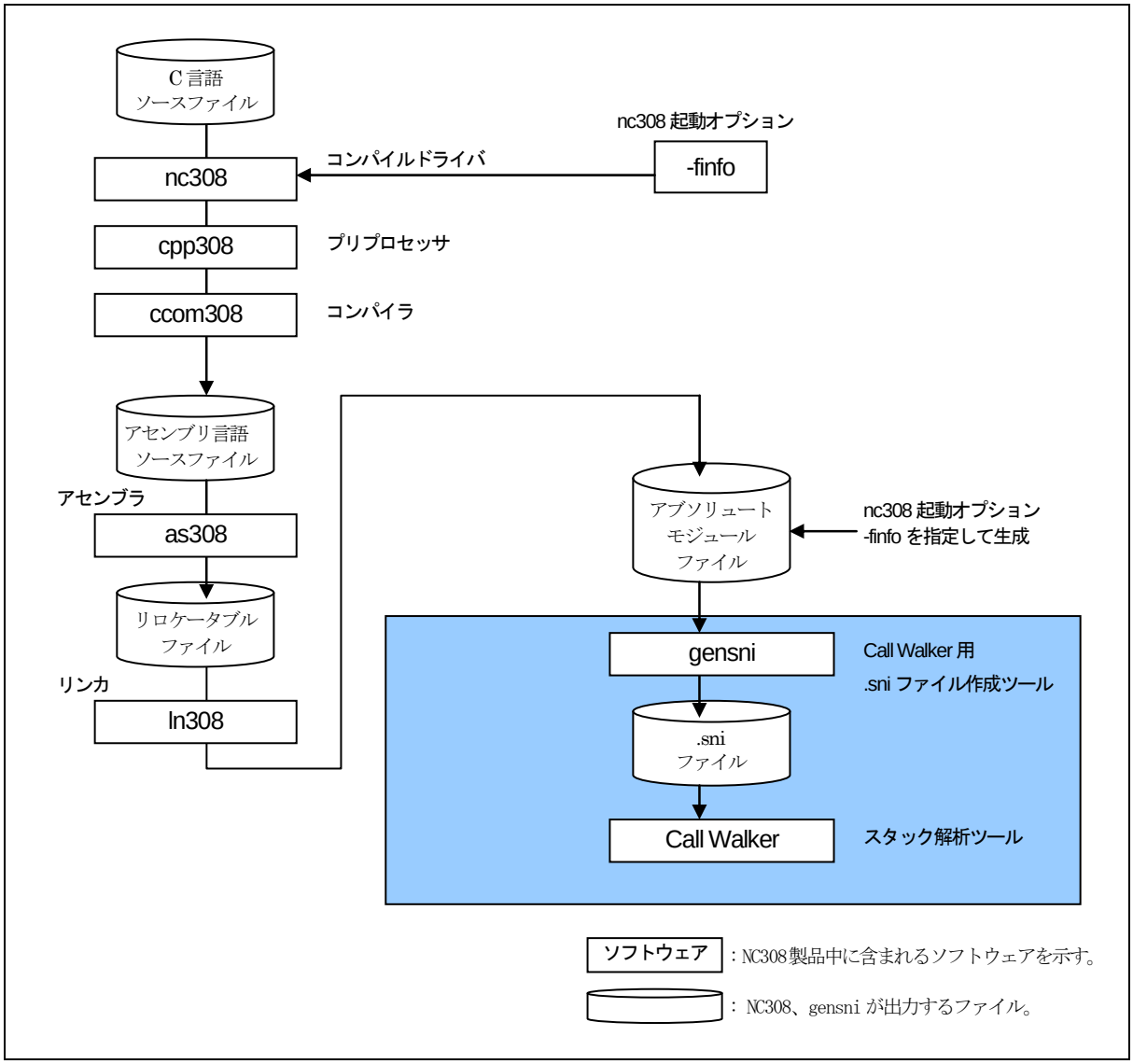
H.1.1 Call Walkerの注意事項

Call Walker が表示するスタックサイズは、厳密な値ではありません。したがいまして、Call Walker が表示するスタックサイズは、スタック領域のサイズを検討する場合の参考値として扱ってください。
なお、Call Walker が表示するスタックサイズを基にスタック領域のサイズを決定した際には、十分に評価を行ってください。

H.2 gensniの概要

H.2.1 gensniの処理概要

gensni は、Call Walker 用.sni ファイルを作成するためのツールです。
gensni 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、.sni ファイルを生成します。gensni を使用するには、コンパイル時にコンパイルオプション“-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H.1】示します。



図H.1　NC308 の処理フロー

H.3 gensniの起動方法

High-performance Embedded Workshop から Call Walker を起動した場合、gensni は自動的に実行します。
しかし、High-performance Embedded Workshop 以外から Call Walker を起動する場合は、gensni は自動的に実行しません。この場合は、Windows のコマンドプロンプト上で gensni を起動してください。

H.3.1 入力書式

gensni を起動するためには、以下の図に示す入力書式にしたがって、入力ファイル名、起動オプションを指定します。

```
% gensni△[起動オプション] △アブソリュートモジュールファイル（拡張子. x30）

% : 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H.2　gensni コマンドの入力書式

gensni 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 gensni に、以下のオプションを指定しています。

- 出力情報の指定ファイルへ出力..... -o オプション

(デフォルトでは、入力ファイルの.x30 を.sni に変更したファイルに出力します。)

アブソリュートモジュールファイルの生成:

```
% nc308 –g –fansi ncrt0.a30 sample.c <RET>
M32C Series Compiler V.X.XX Release XX
Copyright(C) XXXX(XXXX,XXXX,XXXX,XXXX). Renesas Electronics Corp.
and Renesas Solutions Corp., All rights reserved.

ncrt0.a30
sample.c

%
```

. sni ファイルの生成:

```
%gensni -o sample ncrt0.x30<RET>

sample.sni is created.

%
```

図H.3　gensni コマンドの入力例

H.3.2　オプションリファレンス

gensniの起動オプションを【表H.1】に示します。

表H.1　gensni の起動オプション

オプション	短縮形	機能
-o ファイル名	なし	.sni ファイルの名前を指定します。 ● 本オプションが指定されていない場合、.sni ファイル名は入力ファイル名の拡張子を.sni に指定した名前になります。 ● .sni ファイル名に拡張子を指定した場合は.sni を指定した拡張子に変更します。拡張子の指定がない場合は、拡張子を.sni にします。
-V	なし	gensni の起動メッセージを表示して処理を終了します。 .sni ファイルは作成しません。

-o ファイル名

.sni ファイル名を指定する

機　　能:
- 本オプションが指定されていない場合、.sni ファイル名は入力ファイル名の拡張子を.sni に変更した名前となります。
- ファイル名に拡張子がない場合は、拡張子を.sni とします。

補足説明: 本オプションを使用することにより、.sni ファイル名を任意に変更できます。
拡張子の変更も可能です。

-V

gensni の起動メッセージを表示して処理を終了する

機　　能: gensni の起動メッセージを表示して処理を終了します。
- .sni ファイルは作成しません。

M32C シリーズ用　C コンパイラパッケージ　V.5.42
C コンパイラユーザーズマニュアル

発行年月日:　2010 年 4 月 1 日　Rev.2.00

発行:　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

編集:　株式会社ルネサス ソリューションズ

M32C シリーズ用
C コンパイラパッケージ　V.5.42
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

RJJ10J2567-0200