お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010年4月1日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

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

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

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

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

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

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

お客様各位

資料中の「三菱電機」、「三菱XX」等名称の株式会社ルネサス テクノロジへの変更について

2003年4月1日を以って株式会社日立製作所及び三菱電機株式会社のマイコン、ロジック、アナログ、ディスクリート半導体、及びDRAMを除くメモリ(フラッシュメモリ・SRAM等)を含む半導体事業は株式会社ルネサス テクノロジに承継されました。
従いまして、本資料中には「三菱電機」、「三菱電機株式会社」、「三菱半導体」、「三菱XX」といった表記が残っておりますが、これらの表記は全て「株式会社ルネサス　テクノロジ」に変更されておりますのでご理解の程お願い致します。尚、会社商標・ロゴ・コーポレートステートメント以外の内容については一切変更しておりませんので資料としての内容更新ではありません。

注:「高周波・光素子事業、パワーデバイス事業については三菱電機にて引き続き事業運営を行います。」

2003年4月1日
株式会社ルネサス テクノロジ
カスタマサポート部

User's Manual

RENESAS

7900シリーズ用 Cコンパイラ

NC79 V.4.10　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.1.00　2000.12

《安全設計に関するお願い》

三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

《本資料ご利用に際しての留意事項》

- 本資料は、お客様が用途に応じた適切な三菱半導体製品をご購入いただくための参考資料であり、本資料中に記載の技術情報について三菱電機株式会社・三菱電機セミコンダクタシステム株式会社が所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は責任を負いません。
- 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は、予告なしに、本資料に記載した製品または仕様を変更することがあります。三菱半導体製品のご購入に当たりましては、事前に三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店へ最新の情報をご確認頂きますとともに、三菱電機半導体情報ホームページ（http://www.semicon.melco.co.jp/）および三菱開発ツールホームページ（http://www.tool-spt.mesc.co.jp/）などを通じて公開される情報に常にご注意ください。
- 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社はその責任を負いません。
- 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。三菱電機株式会社・三菱電機セミコンダクタシステム株式会社は、適用可否に対する責任は負いません。
- 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店へご照会ください。
- 本資料の転載、複製については、文書による三菱電機株式会社・三菱電機セミコンダクタシステム株式会社の事前の承諾が必要です。
- 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたら三菱電機株式会社・三菱電機セミコンダクタシステム株式会社または特約店までご照会ください。

製品の内容及び本書についてのお問い合わせ先

電子メールの場合：インストーラが生成する以下のテキストファイルに必要事項を記入の上、開発ツールサポート窓口support@tool.mesc.co.jpまで送信ください。

Windows 98/95/Windows NT 4.0版：\SUPPORT\製品名\SUPPORT.TXT
EWS版：/support/製品名/toolinfo.txt

FAXの場合：各ユーザーズマニュアル及びユーザーズマニュアルの最後に添付されている「技術サポート連絡書」に必要事項を記入の上、開発ツールサポート窓口まで送信ください。FAX送信先は「技術サポート連絡書」に記載してあります。

NC79 V.4.10 ユーザーズマニュアル

目　次

目次

はじめに

NC79は、三菱16ビットマイクロコンピュータ 7700ファミリ7900シリーズ用のCコンパイラです。NC79は、C言語で記述したプログラムを 7900シリーズ用のアセンブリ言語ソースファイルに変換します。また、コンパイルオプションを指定することによって、アセンブル / リンクを実行してマイクロコンピュータに書き込み可能な16進数形式ファイルまでを生成することができます。

用語の使い分けの説明

NC79 の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
nc79	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。
NC79	NC79 の製品名、又は製品の総称を意味します。
as79	ﾘﾛｹｰﾀﾌﾞﾙﾏｸﾛｱｾﾝﾌﾞﾗ、又はその実行ファイルを意味します。
AS79	AS79の製品名、又は製品の総称を意味します。
アセンブラ関数	アセンブリ言語で記述したサブルーチンを意味します。
asm関数	NC79の拡張機能で、C言語プログラム中に記述できるasm(アセンブリ言語)を意味します。
インラインアセンブル記述	

使用する記号の説明

NC79 のマニュアルでは、以下に示す記号を使用します。

記号	表示内容
#	ルートユーザのプロンプトを示します。
%	UNIXのプロンプトを示します。
A>	MS-Windows (MS-DOS)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。

NC79 V.4.10

ユーザーズマニュアル

第１章
NC79の処理概要

この章では、NC79が行うコンパイル処理の概要と、NC79を使用したプログラム開発の事例を説明します。

1.1 NC79の構成

NC79は、以下に示す５つの実行ファイルで構成されています。

1.nc79 ..... コンパイルドライバ
2.cpp79 ..... プリプロセッサ
3.ccom79 ..... コンパイラ
4.StkViewer & stk ..... STKビューワ & スタックサイズ計算ユーティリティ
（StkViewer は、GUI(Graphical User Interface)ユーティリティです）
5.utl79 ..... DPnDATA宣言ユーティリティ
6.MapViewer ..... マップビューワ（PC版のみに付属）
（StkViewer は、GUI(Graphical User Interface)ユーティリティです）

1.2 NC79の処理フロー

NC79の処理フローを【図1.1】に示します。

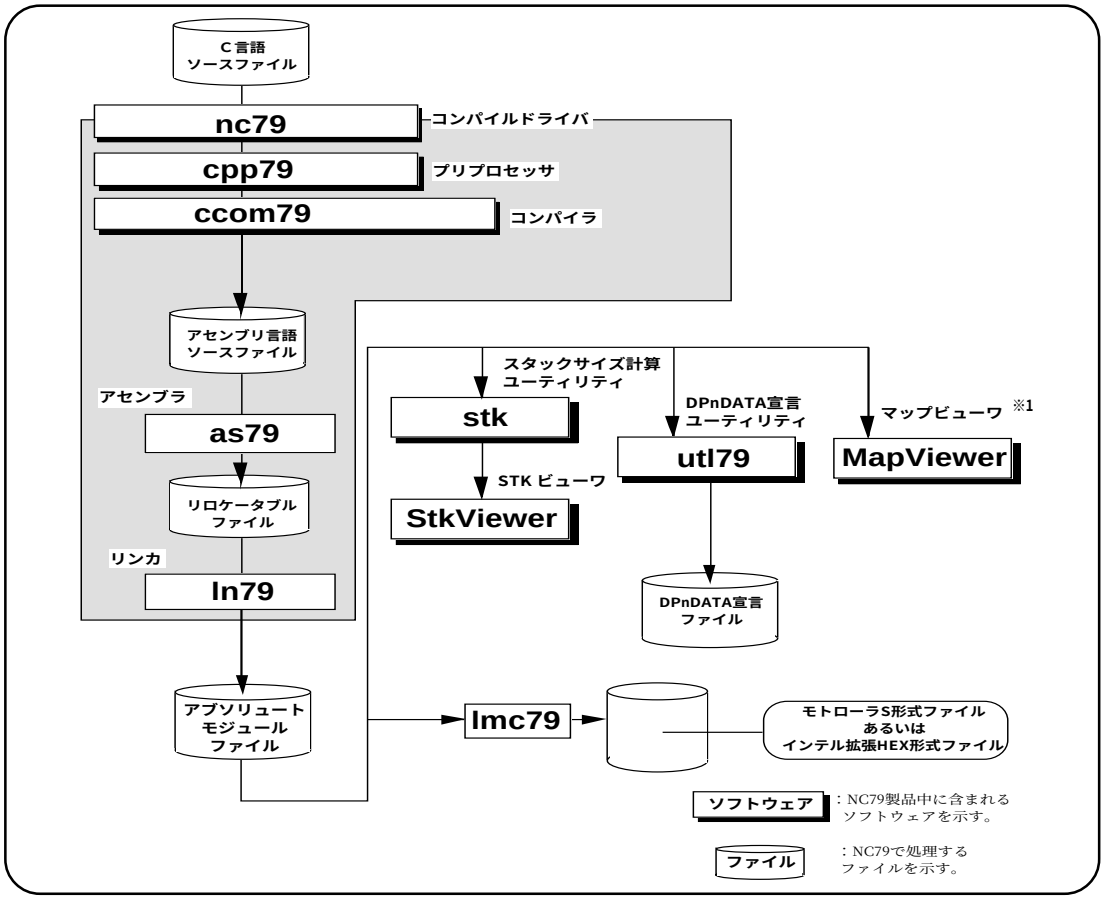


図1.1 NC79の処理フロー

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

1.2.1 nc79

nc79は、コンパイルドライバの実行ファイルです。nc79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、nc79の起動オプション-as79、-ln79に続けてリロケータブルマクロアセンブラas79、リンケージエディタln79のオプションを指定することができます。

1.2.2 cpp79

cpp79は、プリプロセッサの実行ファイルです。cpp79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 ccom79

ccom79は、コンパイラ本体の実行ファイルです。cpp79によって処理されたC言語ソースプログラムをAS79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 StkViewer & stk

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

StkViewer は、stkを呼び出して、アブソリュートモジュールファイル(.x79)に付加されているインスペクタ情報を処理し、プログラムの動作に必要なスタックサイズと関数の呼び出し関係を求め、表示します。

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

StkViewer & stk を使用するには、コンパイル時にコンパイルドライバの起動オプション -finfoを指定して、アブソリュートモジュールファイル(.x79)にインスペクタ情報が付加されるようにしてください。

1.2.5 utl79

utl79は、DPnDATA宣言ユーティリティの実行ファイルです。アブソリュートモジュールファイル(.x79)を処理し、使用頻度の高い外部変数を#pragma DPnDATA(nは１～3)で宣言したファイルを生成します。

utl30 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(.x79)を生成してください。

1.2.6 MapViewer

MapViewer は、マップビューワの実行ファイルです。MapViewer は、アブソリュートモジュールファイル(.x79)を処理し、リンク後のメモリ配置をグラフィカルに表示します。

MapViewer を使用するには、コンパイル時にコンパイルドライバの起動オプション -finfo を指定して、アブソリュートモジュールファイル(.x79)を生成してください。

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

1.3 プログラム開発例

NC79を使用したプログラムの開発例の流れを【図1.2】に示します。このプログラムの概要を以下に示します（項目の[1]～[4]は【図1.2】の[1]～[4]に対応します。

[1]C言語ソースプログラム（AA.c)をnc79でコンパイルし、アセンブラas79デアセンブルし、リロケータブルオブジェクトファイル（AA.r79)を作成します
[2]スタートアッププログラムncrt0.a79とセクション情報を記述したインクルードファイルsect79.incを組み込むシステムにあわせて、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.r79)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.r79とncrt0.r79をnc79から実行されるリンケージエディタln79でリンクし、アブソリュートモジュールファイル(AA.x79)を作成します。

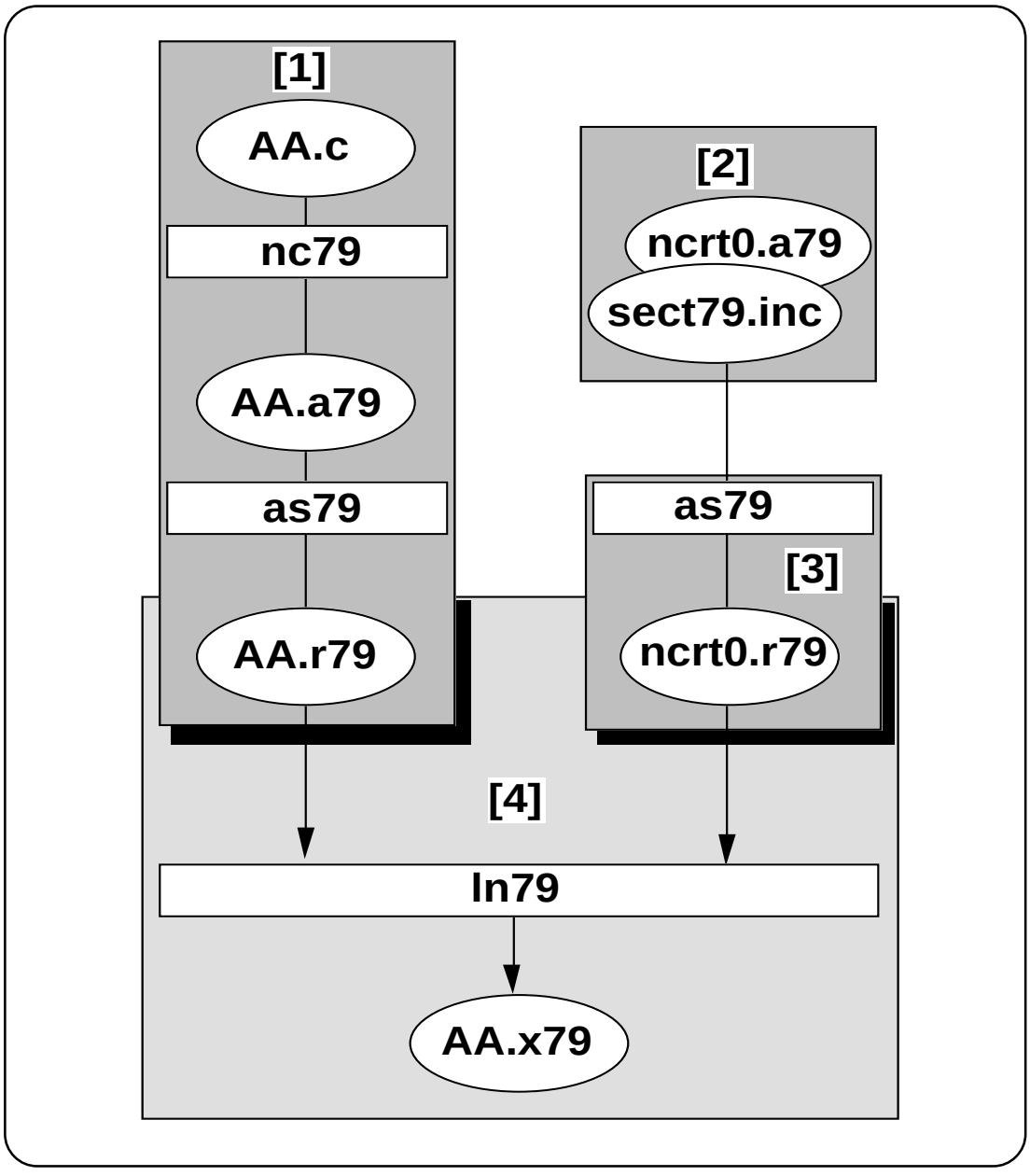


図1.2 プログラム開発フロー

【図1.2】に示した一連の処理を記述した実行手順ファイル(makefile)の例を【図1.3】に示します。

```
AA.x79 : ncrt0.a79 AA.r79
        nc79 -oAA ncrt0.a79 AA.r79

ncrt0.r79 : ncrt0.a79
        as79 ncrt0.a79

AA.r79 : AA.c
        nc79 -c AA.c
```

図1.3 実行手順ファイル(makefile)の記述例

また、コンパイルドライバnc79では、【図1.3】と同様の処理をコマンドラインから【図1.4】に示すように入力できます。

```
%NC79 -oAA ncrt0.a79


%:プロンプトを示します。
<RET>:リターンキーの入力を示します。

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

図1.4 nc79コマンドの入力例

1.4 NC79の出力ファイル

サンプルソースプログラムsmp.cをNC79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、スタック使用量表示ファイルの概要を説明します。

1.4.1 出力ファイルの概要

コンパイルドライバnc79は、起動オプションによって【図1.5】に示すファイルを出力します。次項から【図1.6】に示すC言語ソースファイルsmp.cをコンパイル・アセンブル・リンクした結果出力された個々のファイル例と表示内容を説明します。

なお、as79、およびln79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.r79)リストファイル(拡張子.lst)等に関しては、「AS79ユーザーズマニュアル」を参照してください。

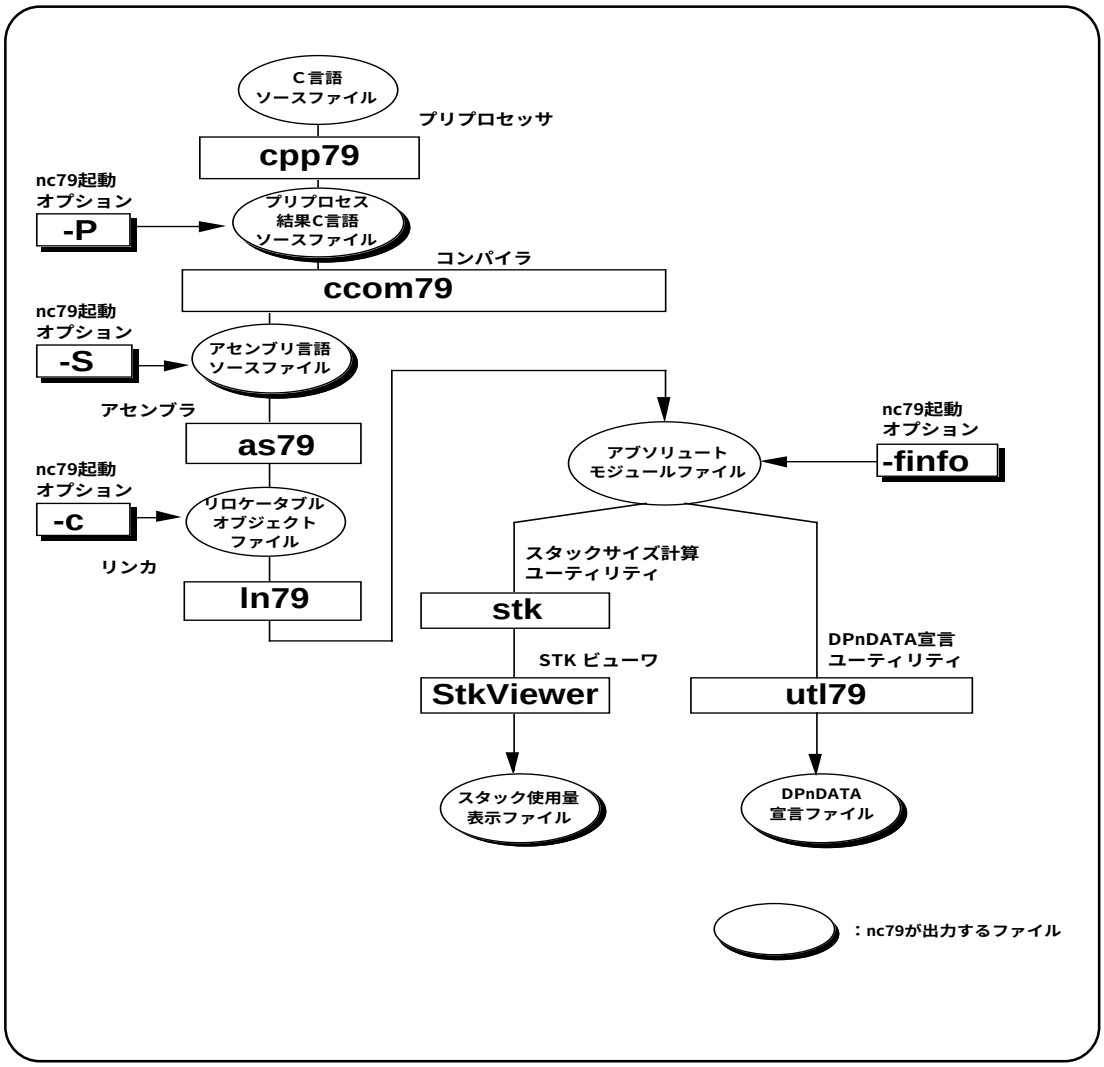


図1.5　nc79起動オプションと出力ファイルの相互関係図

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

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

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

図1.6　C言語ソースファイル例(smp.c)

1.4.2 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cpp79は、#で始まるプリプロセスコマンドに対して、ヘッダーファイルの内容、マクロの展開、およ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、cpp79がC言語ソースファイルを処理した結果を格納しています。したがって、このファイルには#pragma、#line以外のプリプロセス行は出力されません。このファイルの内容を参照することにより、コンパイラが処理を行うプログラムの内容を確認することができます。ファイルの拡張子は.iです。

ファイルの出力例を【図1.7】、【図1.8】に示します。

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

typedef long fpos_t;

typedef unsigned int size_t;

extern FILE _iob[];

int _far       getc(FILE *st);
int _far       getchar(void);
int _far       putc(int c, FILE *st);
int _far       putchar(int c);
int _far       feof(FILE *st);
int _far       ferror(FILE *st);
int _far       fgetc(FILE *st);
char * _far    fgets(char *s, int n, FILE *st);
int _far       fputc(int c, FILE *st);
int _far       fputs(const char *s, FILE *st);
size_t _fa     fread(void *ptr, size_t size, size_t nelem, FILE *st);
size_t _far    fwrite(const void *ptr, size_t size, size_t nelem, FILE *st);
char * _far    gets(char *s);
int _far       puts(const char *s);
int _far       ungetc(int c, FILE *st);
int _far       printf(const char *format, ...);
```

図1.7　プリプロセス結果C言語ソースファイル例(1)

```
int _far       fprintf(FILE *st, const char *format, ...);                    (1)
int _far       sprintf(char *s, const char *format, ...);

int _far       fscanf(FILE *st, const char *format, ...);
int _far       scanf(const char *format, ...);
int _far       sscanf(const char *s, const char *format, ...);
int _far       fflush(FILE *stream);
void _far      clearerr(FILE *stream);
void _far      perror(const char *s);

extern  int _far       init_dev(FILE *, int);
extern  int _far      speed(int, int, int, int);
extern  int _far      init_prn(void);
extern  int _far      _sget(void);
extern  int _far      _sput(int);
extern  int _far       _pput(int);
extern char * _far  _print( int(*)(int), const char *, int **, int * );
```

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

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

}
```

図1.8 プリプロセス結果C言語ソースファイル例(2)

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示します。項目番号(1)～(4)は、【図1.7】、【図1.8】中の(1)～(4)にそれぞれ対応しています。

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

1.4.3 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ccom79がプリプロセス結果C言語ソースファイルをAS79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.a79で示されるアセンブリ言語ソースファイルです。

ファイルの出力例を【図1.9】、【図1.10】に示します。なお、アセンブリ言語ソースファイルの出力にはnc79の起動オプション-dsource(-dS)を指定して、行単位でC言語ソースファイルの内容をコメントとして表示させています。

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

;## NC79 Compiler for 7900 Series OUTPUT
;## ccom79 Version  X.XX.XX
;## Copyright(c) 2000 MITSUBISHI ELECTRIC CORPORATION
;## and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;## All Rights Reserved
;## Compile Start Time Thu May  5 20:18:26 2000

;## COMMAND_LINE: ccom79  D:\MTOOL\NC79WA~1\TMP\smp.i -o .\smp.a79 -dS

;## Normal Optimize              OFF                                          (1)
;## ROM size Optimize            OFF
;## Speed Optimize               OFF
;## DP Register            6 bits offset
;## Default ROM is               near
;## Default RAM is               near

__DT            .EQU  0       ; data bank
__STACK  .EQU  0       ; stack bank
     .GLB  __DP1
     .GLB  __DP2
     .GLB  __DP3
     .DP1  __DP1
     .DP2  __DP2
     .DP3  __DP3

;## #        FUNCTION main
;## #        FRAME       AUTO (    flag)       size   2,       offset 1
;## #        ARG Size(0) Auto Size(2)  Context Size(5)

     .section       program_F
     ._file  'smp.c'
     ._line  6
;## # C_SRC :     {
     .DT    __DT
     .DP0   OFF
     .glb    _main
_main:
     phd    0
     pha
     tsd
     ._line  8
;## # C_SRC :             flag = CLR;
     clrm    DP0:1 ;  flag
     ._line  11
;## # C_SRC :             printf( "flag = %d\n",flag );                        (2)
     pei     DP0:1 ;  flag
     pea     #OFFSET ___T0
     jsrl    _printf
     plx
     plx
     ._line  13
;## # C_SRC :     }
     plx
     rtld    0
     .glb    __print
     .glb    $_pput
     .glb    $_sput
     .glb    __sget
     .glb    _init_prn
     .glb    $speed
     .glb    $init_dev
     .glb    $perror
```

図1.9 アセンブリ言語ソースファイル例(1)(smp.a79)

```
    .glb    $clearerr
    .glb    $fflush
    .glb    _sscanf
    .glb    _scanf
    .glb    _fscanf
    .glb    _sprintf
    .glb    _fprintf
    .glb    _printf
    .glb    $ungetc
    .glb    $puts
    .glb    $gets
    .glb    $fwrite
    .glb    $fread
    .glb    $fputs
    .glb    $fputc
    .glb    $fgets
    .glb    $fgetc
    .glb    $ferror
    .glb    $feof
    .glb    $putchar
    .glb    $putc
    .glb    _getchar
    .glb    $getc
    .glb    __iob

    .SECTION    rom_NO,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 Thu May  8 20:18:26 2000
```

図1.10　アセンブリ言語ソースファイル例(2)(smp.a79)

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(1)～(2)は【図1.9】、【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、およびRAMのnear/far属性の初期設定の情報を示しています。

(2)nc79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nc79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、nc79から起動できるアセンブラas79とリンケージエディタln79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 nc79コマンドの入力書式

コンパイルドライバnc79は、コンパイラの各コマンド(cpp79、ccom79)とアセンブルコマンドas79及びリンクコマンドln79を起動し、機械語データファイルを生成します。このnc79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.a79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.r79とsample.r79をリンク

を行い、アブソリュートモジュールファイルsample.x79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、

●リロケータブルモジュールファイル名sample.a79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 ..... -as79 "-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 ..... -ln79 "-ms"オプション

を行っています。

```
%nc79△[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 nc79コマンドの入力書式

```
%nc79 -osample -as79“ -l ”-ln79“ -ms ”ncrt0.a79 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 nc79コマンドの入力例

2.1.2 コマンドファイル

nc79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）を読み込んでプログラムを実行することができます。

a.コマンドファイルの入力書式

```
%nc79△[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

```
%nc79 -c @test.cmd <RET>

<RET>:リターンキーの入力を示します。
```

図2.4　コマンドファイルの入力例

コマンドファイルの記述は以下のようになります。

```
                                test.cmd<CR>
                                ncrt0.a79<CR>
test.cmdの記述 ==========>      sample1.c sample2.c <CR>
                                -g -as79 -l <CR>
                                -o sample <CR>

<CR>:改行を示します。
```

図2.5　コマンドファイルの記述例

b.コマンドファイルの記述規定

コマンドファイルの記述には以下の規定があります。

(1)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。
(2)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。
(3)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。
(4)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。
(5)コマンドファイルの一行に記述可能な文字数は、2048文字までです。2048文字を越えた場合はエラーとなります。

c.コマンドファイル使用時の注意事項

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

リンク時にファイルを指定するために拡張子".cm$"のln79用コマンドファイルを自動生成します。このため、拡張子が".cm$"のファイルがある場合には、上書きされる可能性があります。拡張子が".cm$"のファイルは、使用しないでください。

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

2.1.3　nc79起動オプションに関する注意事項

a.nc79起動オプションの記述に関する注意事項

nc79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.nc79の制御に関するオプションの優先順位

nc79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.r79)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.a79)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。

2.1.4　nc79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の制御オプション

オプション	機能
-c	リロケータブルファイル(拡張子.r79)を作成し、処理を終了します。※1
-D*識別子名*	識別子を定義します。#defineと同じ機能です。
-I*ディレクトリ名*	#includeで指定するファイルが存在するディレクトリ名を指定します。ディレクトリは最大8個まで指定可能です。
-E	プリプロセスコマンドのみを起動し結果を標準出力に出力します。※1
-P	プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成します。※1
-S	アセンブリ言語ソースファイル(拡張子.a79及び.ext)を作成し、処理を終了します。※1
-U*ﾌﾟﾘﾃﾞﾌｧｲﾝﾄﾞﾏｸﾛ名*	指定したプリデファインドマクロを未定義にします。
-time	各プロセスの処理時間を表示します（UNIX版のみサポートしています）。
-silent	起動時のコピーライトメッセージを出力しません。
-dsource （短縮形 -dS）	出力するアセンブリ言語ソースリスト中にC言語ソースリストをコメントとして出力します。生成された ".a79 "を削除しません。
-dsource_in_list （短縮形 -dSL）	C言語ソースリストをコメントとしてリストファイルに出力します。

b.出力ファイル指定オプション

【表2.2】に出力するアブソリュートモジュールファイルの名称を指定する起動オプションを示します。

表2.2　出力ファイル指定オプション

オプション	機能
-o*ファイル名*	ln79が生成するファイル(機械語データファイル、マップファイル、シンボルファイル等)の名称を指定します。また、ディレクトリ名も指定できます。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*	ln79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。

c.バージョ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、および-Sを指定しない場合、nc79はln79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.x79）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	短縮形	機能
-g	なし	デバッグ時に必要なシンボルファイル（拡張子.a.79）を出力します。
-genter	なし	関数呼び出し時に必ずスタックフレームを生成します。
-gno_reg	なし	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容量ともに最小にする最大限の最適化を行います。
-OR	なし	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。これによりROM容量を削減することができます。ただし、使用するスタック容量が多くなる可能性があります。
-Ostack_frame_align	-OSFA	スタックフレームの偶数アライメントを行います。
-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。

f.生成コード変更オプション

【表2.6】にnc79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。
-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)
-fextend_to_int	-fETI	char型データをint型に拡張し演算を行います(ANSI規格で定められた拡張を行います)。※1
-fchar_enumerator	-fCE	enumerator(列挙子)の型をint型ではなくunsigned char型で扱います。
-fall_far	-fAF	デフォルトをすべてfar型にします。
-fnear_function	-fNF	関数のデフォルトをnearにします。near関数は呼び出すときにjsrで呼出し、rtsで戻ります。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべてoddセクションに配置します。
-ffar_program_section	-fFPS	near関数・far関数をprogram_Fセクションに配置します。
-fnot_use_MVN	-fNUM	MVN命令でのブロック転送を行いません(MVN命令は構造体間の代入で使用されます)。
-bank=n	なし	コンパイル時のデータバンクレジスタ(DT)の値を指定します。指定しない場合、デフォルトは0です。
-ffar_RAM_data	-fFRAM	RAMデータのデフォルト属性をfarにします。
-ffar_ROM_data	-fFROM	ROMデータのデフォルト属性をfarにします。
-fconst_not_ROM	-fCNR	constで指定した型をROMデータとして扱いません。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#pragma EQU)で指定した変数をvolatileで指定したとみなしません。
-fsmall_array	-fSA	far型の配列を参照する場合、その総バイト数が不明な配列を64Kバイト以下の配列として扱います。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fswitch_table	-fST	ROM効率が良くなる場合のみswitch文に対してテーブルジャンプ方式のコードを生成します。(したがって、本オプションを指定してもジャンプテーブルを使用するとは限りません。)
-fDP_offset_8	-fDPO8	DPレジスタをDPR0の1本のみを使用することを前提とした場合のコードを生成します。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。
-finfo	なし	インスペクタ、"StkViewer"、"MapViewer"、"utl79"、に必要な情報をアブソリュートモジュールファイル(.x79)に出力します。
-fauto_128	-fA1	ダイレクトページレジスタ DPR0～DPR3 を使用する場合におけるスタックフレームサイズの上限を63バイトから127バイトに変更します。

※1.ANSI規格では、char型データ、またはsigned char型データを評価するときに必ずint型に拡張します。これは、char型の演算、例えば「c1 = c2 * 2 / c3;」を行う時に、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

ライブラリ	機能
-lﾗｲﾌﾞﾗﾘﾌｧｲﾙ名	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nc79の言語仕様に関する記述の間違いに対して警告(ワーニングメッセージ)を出力する起動オプションを示します。

表2.8　警告オプション

オプション	短縮形	機能
-Wnon_prototype	-WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	-WUP	サポートしていない＃pragmaを使用した場合、警告を出します。
-Wno_stop	-WNS	エラーが発生してもコンパイル作業を停止しません。
-Wstdout	なし	エラーメッセージをホストマシンの標準出力(stdout)に出力します。
-Werror_file<file name>	-WEF	タグファイルを出力します。
-Wmake_tagfile	-WMT	error、およびwarningが発生した場合にタグファイルを出力します。
-Wstop_at_warning	-WSAW	ワーニング発生時にコンパイル処理を停止します。
-Wnesting_comment	-WNC	コメント中に/*を記述した場合に警告を出します。
-Wccom_max_warnings	-WCMW	ccom79の出力するワーニングの回数の上限を指定できます。
-Wall	なし	検出可能な警告をすべて表示します。
-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。
-Wlarge_to_small	-WLTS	大きいサイズから小さいサイズへの暗黙の転送に対してワーニングを出力します。
-Wuninitialize_variable	-WUV	初期化されていないAUTO変数に対してワーニングを出力します。
-Wno_used_argument	-WNUA	使用されていない引数に対してワーニングを出力します。

i. アセンブル / リンクオプション

【表2.9】にas79、およびln79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　警告オプション

オプション	機能
-as79△<*オプション*>	アセンブルコマンドas79のオプションを指定します。最大４個までオプションを指定することができます。2個以上のオプションを渡す場合は、”(ダブルクォーテーション)で囲んでください。
-ln79△<*オプション*>	リンクコマンドln79のオプションを指定します。最大４個までオプションを指定することができます。2個以上のオプションを渡す場合は、”(ダブルクォーテーション)で囲んでください。

j. その他のオプション

【表2.10】にnc79が生成するアセンブリ言語ソースファイルに対して処理を行うする起動オプションを示します。

表2.10 その他のオプション

オプション	短縮形	機能
-dsource	-dS	出力するアセンブリ言語ソースリスト中にC言語ソースリスティングをコメントとして出力します。生成された ”.a79 ”を削除しません。
-dsource_in_list	-dSL	出力するアセンブリ言語ソースリスト中にC言語ソースリスティングをコメントとしてリストファイルに生成します。

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、NC79ではハードウェア（7900シリーズ）の初期設定、セクションの配置、割り込みベクタアドレス、等を設定するアセンブリ言語で記述したサンプルの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あ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

NC79のスタートアッププログラムは、以下の2つのファイルで構成しています。

(1)ncrt0.a79
このプログラムは、プログラムの開始時、またはリセット直後に実行されます。

(2)sect79.inc
このプログラムは、ncrt0.a79からインクルードされます。

ncrt0.a79のソースプログラムリストを【図2.6】、【図2.7】、【図2.8】、【図2.9】に、sect79.incのソースプログラムリストを【図2.10】、【図2.11】にそれぞれ示します。

```
;*****************************************************************
;
;     ncrt0.a79 : NC79 startup program
;
;     NC79 COMPILOR for 7900 Series
;     Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
;     and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;     All Rights Reserved.
;
;     $Id: ncrt0.a79,v 1.71 2000/XX/XX XX:XX:XX usuki Exp $
;
;*****************************************************************
;
      .include       sect79.inc                                    ← (1)
;=======================================
;     initialize MACRO definition
;=======================================
BZERO     .macro        _TOP,_SECT
      ldad    E,#SIZEOF _SECT
      phb                           ; push Eregister(high)
      pha                           ; push Eregister(low)
      ldad    E,#STARTOF _TOP
      phb
      pha
      .glb    _bzero
      jsrl    _bzero
      .endm


(1)sect79.incをインクルードします。
```

図2.6 スタートアッププログラムリスト(1)（ncrt0.a79)

```
BCOPY    .macro      _FROM,_TO,_SECT
     ldad   E,#SIZEOF _SECT
     phb
     pha
     ldad   E,#STARTOF _TO
     phb
     pha
     ldad   E,#STARTOF _FROM
     phb
     pha
     .glb   _bcopy
     jsrl   _bcopy
     .endm

;=======================================
;    start INTERRUPT section
;=======================================
__DT      .equ   00H                                                    ← (2)
     .GLB  __DP1
     .GLB  __DP2
     .GLB  __DP3
     .glb   start
     .section       interrupt
start:                                                                 ← (3)
     ldt    #00H                ; set DTregister
     movmb       DT+:005EH,#0a2H          ; set PM0register             ← (4)
     movmb       DT+:005fH,#02H           ; set PM1register(DPR0,1,2,3)
     ;---------------------------------------------------------------
     ; If tou use a chip-select wait controller, a DRAM control unit and
     ; others, add the settings of these functions here before initializing
     ; bss and data sections.
     ;---------------------------------------------------------------
     clp    m,x,D
     lda.W  A,#OFFSET stack_top - 1  ; set stack pointer                ← (5)
     tas
     ;================
     ;       NEAR
     ;================
     ;---------------------------------------
     ;       Zero clear for data area
     ;---------------------------------------
     BZERO       bss_NE,bss_NE                                         ← (6)
     BZERO       bss_NO,bss_NO
     lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
     tas
     ;---------------------------------------
     ;       Initialize for data area
     ;---------------------------------------
     BCOPY       data_INE,data_NE,data_NE                              ← (7)
     BCOPY       data_INO,data_NO,data_NO
     lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
     tas
```

(2)NC79の起動オプション-bank=nを使用する場合は、__DTの値を変更します。
(3)リセット後はこのラベルからスタートします。
(4)プロセッサ動作モードを設定します。
(5)スタックポインタの初期化を行います。
(6)near領域のbssセクションのゼロクリア処理を行います。
(7)near領域のdataセクションの初期値をRAM領域に転送します。

図2.7 スタートアッププログラムリスト(2)(ncrt0.a79)

```
;---------------------------------------
;       Zero clear for data area
;---------------------------------------
ldd     (1,2,3),#__DP1,#__DP2,#__DP3
BZERO       bss_DP1E,bss_DP1E
BZERO       bss_DP1O,bss_DP1O
BZERO       bss_DP2E,bss_DP2E
BZERO       bss_DP2O,bss_DP2O
BZERO       bss_DP3E,bss_DP3E
BZERO       bss_DP3O,bss_DP3O
lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
tas
;---------------------------------------
;       Initialize for data area
;---------------------------------------
BCOPY       data_IDP1E,data_DP1E,data_DP1E
BCOPY       data_IDP1O,data_DP1O,data_DP1O
BCOPY       data_IDP2E,data_DP2E,data_DP2E
BCOPY       data_IDP2O,data_DP2O,data_DP2O
BCOPY       data_IDP3E,data_DP3E,data_DP3E
BCOPY       data_IDP3O,data_DP3O,data_DP3O
lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
tas
;================
;       FAR
;================
;---------------------------------------
;       Zero clear for data area
;---------------------------------------
BZERO       bss_FE,bss_FE                                   ← (8)
BZERO       bss_FO,bss_FO
lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
tas
;---------------------------------------
;       Initialize for data area
;---------------------------------------
BCOPY       data_IFE,data_FE,data_FE                        ← (9)
BCOPY       data_IFO,data_FO,data_FO
lda.W  A,#OFFSET stack_top - 1  ; reset stack pointer
tas

;---------------------------------------
;       Initialize for HEAP area
;---------------------------------------
.glb    __mbase, __mnext, __msize                           ← (10)
 ldad   E, #heap_top
 stad   E, __mbase
 stad   E, __mnext
 ldad   E, #HEAPSIZE
 stad   E, __msize
;---------------------------------------
;       Call init( initialize standard I/O )
;---------------------------------------
.glb    _init                                               ← (11)
jsrl    _init
;---------------------------------------
;       Call main
;---------------------------------------
ldt     #__DT               ; set DTregister
.glb    _main                                               ← (12)
jsrl    _main
```

(8)far領域のbssセクションのゼロクリア処理を行います。
(9)far領域のdataセクションの初期値をRAM領域に転送します。
(10)heap領域の初期化を行います。メモリ管理関数を使用しない場合にはコメントにします。
(11)標準入出力の初期化を行うinit関数を呼び出します。入出力関数を使用しない場合にはコメントにします。
(12)main関数の呼び出しを行います。M、Xフラグは共にクリアされた状態で呼び出さなければなりません。

図2.8 スタートアッププログラムリスト(3)(ncrt0.a79)

```
        ;---------------------------------------
        ;       Call exit
        ;---------------------------------------
        .glb    _exit                                           ← (14)
        .glb    $exit
_exit:
$exit:
        bra     _exit
        ;---------------------------------------
        ;       Call dummy_int( dummy interrupt function )
        ;---------------------------------------
dummy_int:                                                      ← (15)
        rti
        .end
;************************************************************************
;
;       Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
;       and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;       All Rights Reserved.
;
;************************************************************************
```

(14)exit関数部です。
(15)ダミーの割り込み関数です。

図2.9 スタートアッププログラムリスト(4)(ncrt0.a79)

```
;*****************************************************************
;
;     sect79.inc : section definition
;
;     NC79 COMPILOR for 7900 Series
;     Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
;     and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;     All Rights Reserved.
;
;     $Id: sect79.inc,v 1.8 XX/XX/XX XX:XX:XX xxxxi Exp $
;
;*****************************************************************
;
;=======================================
;     NEAR
;=======================================
STACKSIZE       .equ   300H
HEAPSIZE .equ   300H
;---------------------------------------
;     RAM DATA
;---------------------------------------
      .section        data_NE      ,DATA
      .org            000800H
      .section        data_NO      ,DATA
      .section        bss_NE       ,DATA
      .section        bss_NO       ,DATA
      .section        stack  ,DATA
      .blkb           STACKSIZE
stack_top:
      .section        heap   ,DATA
heap_top:
      .blkb           HEAPSIZE

      .section        data_DP1E,DATA
__DP1:
      .section        data_DP1O,DATA
      .section        bss_DP1E,DATA
      .section        bss_DP1O,DATA

      .section        data_DP2E,DATA
__DP2:
      .section        data_DP2O,DATA
      .section        bss_DP2E,DATA
      .section        bss_DP2O,DATA

      .section        data_DP3E,DATA
__DP3:
      .section        data_DP3O,DATA
      .section        bss_DP3E,DATA
      .section        bss_DP3O,DATA

;---------------------------------------
;     PROGRAM
;---------------------------------------
      .section        interrupt
      .org            004000H
      .section        program_N
;---------------------------------------
;     ROM DATA
;---------------------------------------
      .section        rom_NE       ,ROMDATA
      .section        rom_NO       ,ROMDATA
      .section        data_INE,ROMDATA
      .section        data_INO,ROMDATA
      .section        data_IDP1E,ROMDATA
      .section        data_IDP1O,ROMDATA
      .section        data_IDP2E,ROMDATA
      .section        data_IDP2O,ROMDATA
      .section        data_IDP3E,ROMDATA
      .section        data_IDP3O,ROMDATA
```

図2.10 スタートアッププログラムリスト(5)(sect79.inc)

```
;---------------------------------------
;     VECTOR TABLE
;---------------------------------------
      .section      vector
      .org          0ffc0H
DMA3:             .word         dummy_int
DMA2:             .word         dummy_int
DMA1:             .word         dummy_int
DMA0:             .word         dummy_int
ROMC:             .word         dummy_int
RESERVED3:        .word         dummy_int
RESERVED2:        .word         dummy_int
RESERVED1:        .word         dummy_int
INT4:             .word         dummy_int
INT3:             .word         dummy_int
AD:               .word         dummy_int
UART1S:           .word         dummy_int
UART1A:           .word         dummy_int
UART0S:           .word         dummy_int
UART0A:           .word         dummy_int
TB2I:             .word         dummy_int
TB1I:             .word         dummy_int
TB0I:             .word         dummy_int
TA4I:             .word         dummy_int
TA3I:             .word         dummy_int
TA2I:             .word         dummy_int
TA1I:             .word         dummy_int
TA0I:             .word         dummy_int
INT2:             .word         dummy_int
INT1:             .word         dummy_int
INT0:             .word         dummy_int
NMI:              .word         dummy_int
WDT:              .word         dummy_int
DBC:              .word         dummy_int
BK:               .word         dummy_int
DIV0:             .word         dummy_int
RESET:            .word         OFFSET start
;=======================================
;     FAR
;=======================================
;---------------------------------------
;     RAM DATA
;---------------------------------------
      .section      data_FE       ,DATA
      .org          12000H
      .section      data_FO       ,DATA
      .section      bss_FE        ,DATA
      .section      bss_FO        ,DATA
;---------------------------------------
;     PROGRAM
;---------------------------------------
      .section      program_F
      .org          18000H
;---------------------------------------
;     ROM DATA
;---------------------------------------
      .section      rom_FE        ,ROMDATA
      .section      rom_FO        ,ROMDATA
      .section      data_IFE,ROMDATA
      .section      data_IFO,ROMDATA
;*****************************************************************
;
;
;     Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
;     and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;     All Rights Reserved.
;
;*****************************************************************
;
```

図2.11 スタートアッププログラムリスト(6)(sect79.inc)

2.2.2 スタートアッププログラムのカスタマイズ

a.スタートアッププログラムの処理概要

(1). ncrt0.a79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。

- ●データのnear領域のバンク値(__DT)を設定します。
- ●プロセッサ動作モードを設定します。(デフォルトと、NC79起動オプション"-fDP_offset_8"使用時と異なります。)
- ●スタックポインタの初期化を行います。
- ●データのnear領域の初期化を行います。
  bss_NE、bss_NO、bss_DP1E、bss_DP2E、bss_DP3E、bss_DP1O、bss_DP2O、bss_DP3O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_INE、data_INO、data_IDP1E、data_IDP2E、data_IDP3E、data_IDP1O、data_IDP2O、data_IDP3O)の初期値をRAM領域 (data_NE、data_NO、data_DP1E、data_DP2E、data_DP3E、data_DP1O、data_DP2O、data_DP3O)に転送する処理を行います。※1
- ●データのfar領域の初期化を行います
  bss_FE、bss_FOセクションをゼロクリアします。また、初期値が格納されたROM領域(data_IFE、data_IFO)の初期値をRAM領域(data_FE、data_FO)に転送する処理を行います。※1
- ●標準入出力関数ライブラリの初期化を行います。
- ●heap領域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。

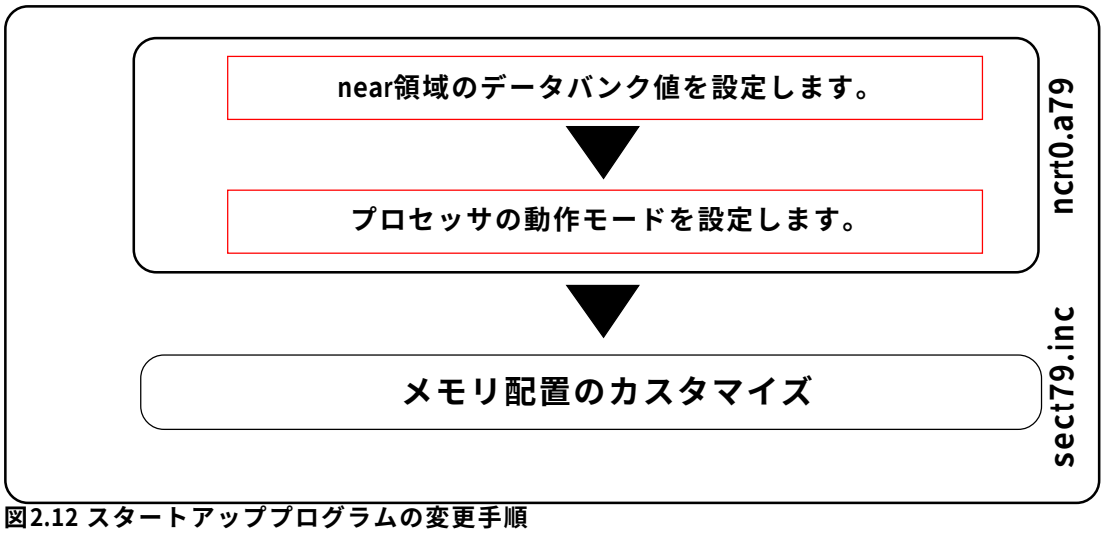


図2.12 スタートアッププログラムの変更手順

※1.NC79は初期値を持つ大域変数に対して、変数としてアクセスされるRAM領域(data_NE、data_NO)とその初期値を格納するROM領域(data_INE、data_INO)の2つの領域に出力します。スタートアッププログラムは、初期値データをRAM領域に転送する処理を行います。

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関数は、7900シリーズの入出力の初期化を行います。init関数は、ncrt0.a79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.a79内のinit関数の呼び出し部をコメントにしてください。

```
;=====================================================================
;Callinit(InitializestandardI/O)
;---------------------------------------------------------------------
        .glb_init
        jsrl_init

;=====================================================================
;Callmain()function
```

図2.13　init関数呼び出し部（ncrt0.a79）

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。

(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.a79中で以下の設定を行っています。

(1)外部変数char＊__mbaseの初期化
(2)外部変数char＊__mnextの初期化
　　heapセクションの先頭アドレスを表すラベルheap_topで初期化します。
(3)外部変数unsigned__msizeの初期化
　　heapセクションのサイズを表す式(sizeofheap)で初期化します。

【図2.14】にncrt0.a79内の初期化部を示します。

```
;InitializeforHEAParea
;-----------------------------------------
        .glb__mbase,__mnext,__msize
        ldad    E,heap_top
        stad    E,__mbase
        stad    E,__mnext
        ldad    E,HEAPSIZE
        stad    E,__msize
```

図2.14　メモリ管理関数を使用するときの初期化部（ncrt0.a79）

メモリ管理関数を使用しないときは、この初期化部をすべてコメントにしてください。コメントにすることで、不要なライブラリがリンクされずROMサイズを節約できます。

(3)独自の初期化プログラムを記述するときの注意事項

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

(1)独自の初期化プログラムにおいて、m、x、Dフラグを変更した場合は、初期化プロ グラムの出口でm、x、Dフラグを元の状態に戻してください。また、データバンクレジスタ(DT)の内容を変更しないでください。
(2)独自の初期化プログラムからC言語で記述されたサブルーチンを呼び出す場合は、以下の2項目に注意してください。
①m、x、Dフラグはクリアした状態で呼び出してください。
②呼び出すサブルーチンのnear・far属性によりJSR・JSRL命令を選択してください。
(3)DPレジスタを4本とも使用するモードでは、必ず、DP1～DP3の値を設定してください。

d.stackセクションのサイズの設定

stackセクションでは、ユーザスタック用に使用する領域のサイズと割り込みスタック用に使用する領域のサイズを設定する必要があります(スタックサイズの設定)。

ユーザスタックは必ず使用しますので、この領域は必ず確保してください。セクション領域の確保は、プログラムでstackセクションのメモリ使用量が最大になるときのサイズをスタックサイズとして設定します。

スタックサイズは、STKビューワ StkViewer & スタックサイズ計算ユーティリティ stk を使用して求めます。

詳しくは「付録G スタックサイズ計算ユーティリティ」を参照してください。

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。

```
;=======================================
;    NEAR
;=======================================
STACKSIZE       .equ   300H
HEAPSIZE .equ   300H
```

図2.15 heapサイズの設定例(ncrt0.a79)

f.データバンクレジスタの設定

スタートアッププログラムncrt0.a79中の【図2.16】で示す__DTに、near型データを配置する領域のデータバンクレジスタの値を設定します。また、near領域を0以外のバンクに設定するnc79の起動オプション-bank=n(n=0～255)を使用するときは、起動オプションで指定した値と同じ値をncrt0.a79中の__DTに設定してください。

```
__DT        .equ        00h
            .glb        start
            .section    interrupt
start:
            ldt         #__DT;setDTregister
```

図2.16　データバンクレジスタの設定例(1)(ncrt0.a79)

【図2.17】に、near型データの領域のバンクを2に設定するときの変更例を示します。このとき、リンクするすべてのC言語プログラムをコンパイルするときにnc79の起動オプション-bank=2を指定してコンパイルしてください。なお、near型データ領域のバンクを0以外に設定した場合、一部の標準関数が使用できなくなります。

```
__DT        .equ        02h
            .section    interrupt
start:
            ldt         #__DT               ;setDTregister
```

図2.17　データバンクレジスタの設定例(2)(ncrt0.a79)

g.プロセッサモードレジスタの設定

ncrt0.a79中の【図2.18】で示す部分において5EH番地、及び5FH番地(プロセッサモードレジスタ0、1)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。

```
            :
          (省略)
            :
    movmb           DT+:5eH,#24H                ;setPM0register
    movmb           DT+:5fH,#02H                ;setPM1register(DPR0,1,2,3)
```

図2.18　プロセッサモードレジスタ設定例(1)(ncrt0.a79)

nc79の起動オプション-fDP_offset_8(-fDPO8)でDPレジスタをDP0のみ使用するモードを選択する場合には、【図2.19】に示す方法で設定してください。

```
            :
          (省略)
            :
    movmb           DT+:5eH,#24H                ;setPM0register
    movmb           DT+:5fH,#00H                ;setPM1register(DPR0Only)
```

図2.19　プロセッサモードレジスタ設定例(2)(ncrt0.a79)

nc79の起動オプション-bank=でnear領域のバンクを0以外に設定したときは、【図2.20】に示す方法で設定してください。

```
                    :
                  (省略)
                    :
    ldab        A,#24H              ;setPM0register
    stab        A,LG:5eH
    ldab        A,#02H              ;setPM1register(DPR0,1,2,3)
    stab        A,LG:5fH
```

図2.20　プロセッサモードレジスタ設定例(3)(ncrt0.a79)

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、NC79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。

NC79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7900シリーズのメモリ上に配置します。セクションを表すセクション名は、セクションベース名とその属性により構成されます。

表2.9 セクション名

表2.10 セクションベース名

セクションベース名	内容
data	初期値を持つデータを格納します。
bss	初期値のないデータを格納します。
rom	文字列、#pragma ROM、const修飾子で指定されたデータを格納します。
section	プログラムを格納します。

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	N ... near属性 F ... far属性	data,bss,rom
E/O	E ... データサイズが偶数 O ... データサイズｶﾞ奇数	data,bss,rom

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5.3】に示します。

表2.12 セクションの名称

セクション名	内容
stack	スタックとして使用する領域です。このセクションは7900シリーズの0バンクに配置してください。
heap	メモリ管理関数(malloc等)により、プログラム実行中に動的に確保されるメモリ領域です。このセクションは7900シリーズの任意のバンクに配置できます。
vector	7900シリーズの割り込みベクトルテーブルの内容を格納します。この割り込みベクトルテーブルを配置するアドレスは機種により異なります。詳しくは各機種のユーザーズマニュアルを参照してください。
interrupt	割り込みプログラム（#pragma INTERRUPT、#pragma INTF、#pragma HANDLARで指定された関数）を格納します。このセクションは7900シリーズの0バンクに配置してください。

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section.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。

nc79のオプション-fDP_offset_8（ダイレクトページレジスタをDPR0の１本のみを使用）することを前提とした場合のインクルードファイルsect79.incのセクション配置図を【図2.21】に示します。

※スタートアッププログラムncrt0.a79では、near領域のバンク値の初期設定は0です。

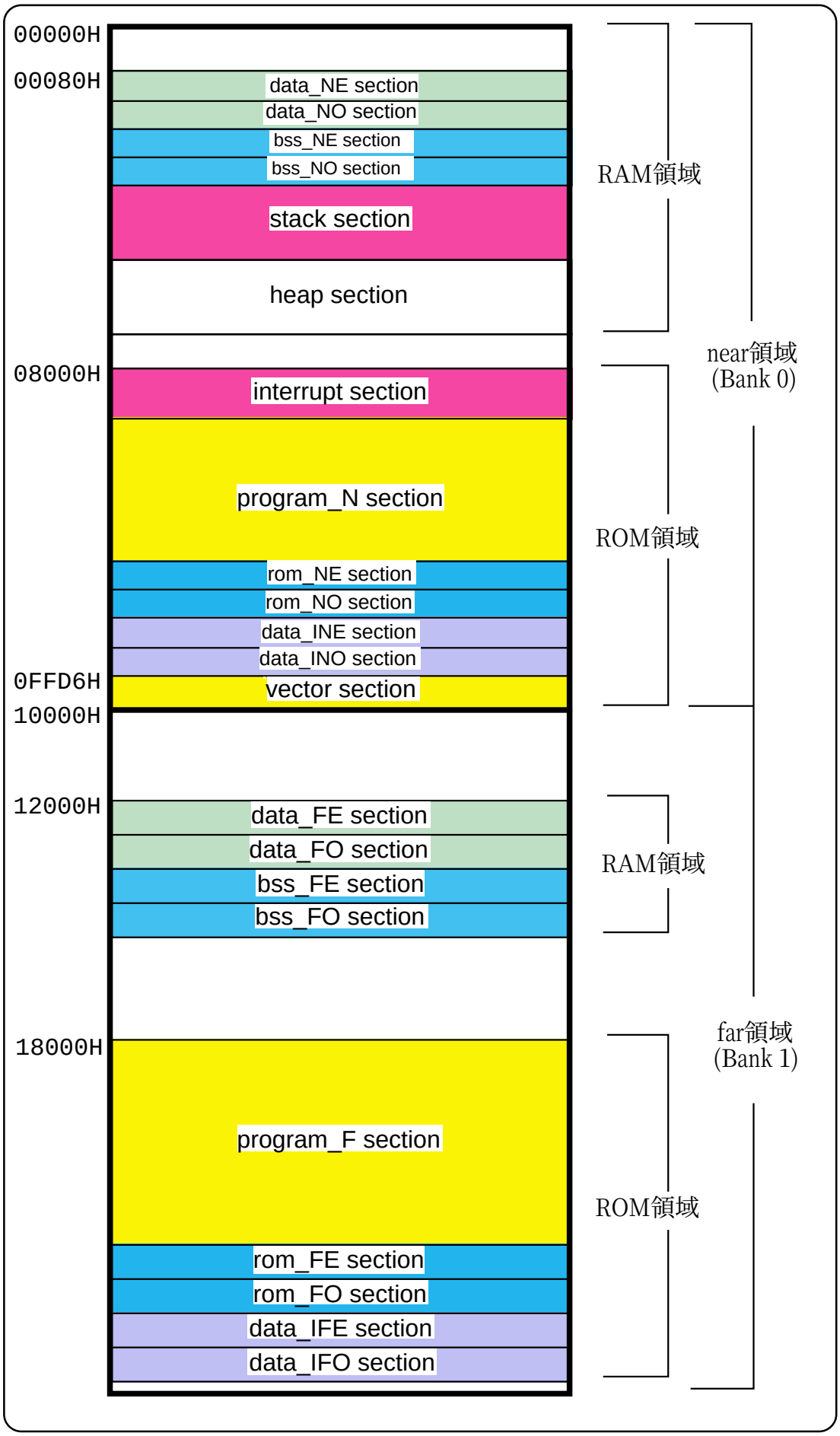


図2.21 セクション配置例

b.メモリ配置設定用ファイルの概要

<sect79.incについて>

このプログラムは、ncrt0.a79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。
- ●割り込みベクタを設定します。

c.sect79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みシステムにあわせて変更する方法の概略を【図2.22】に示します。

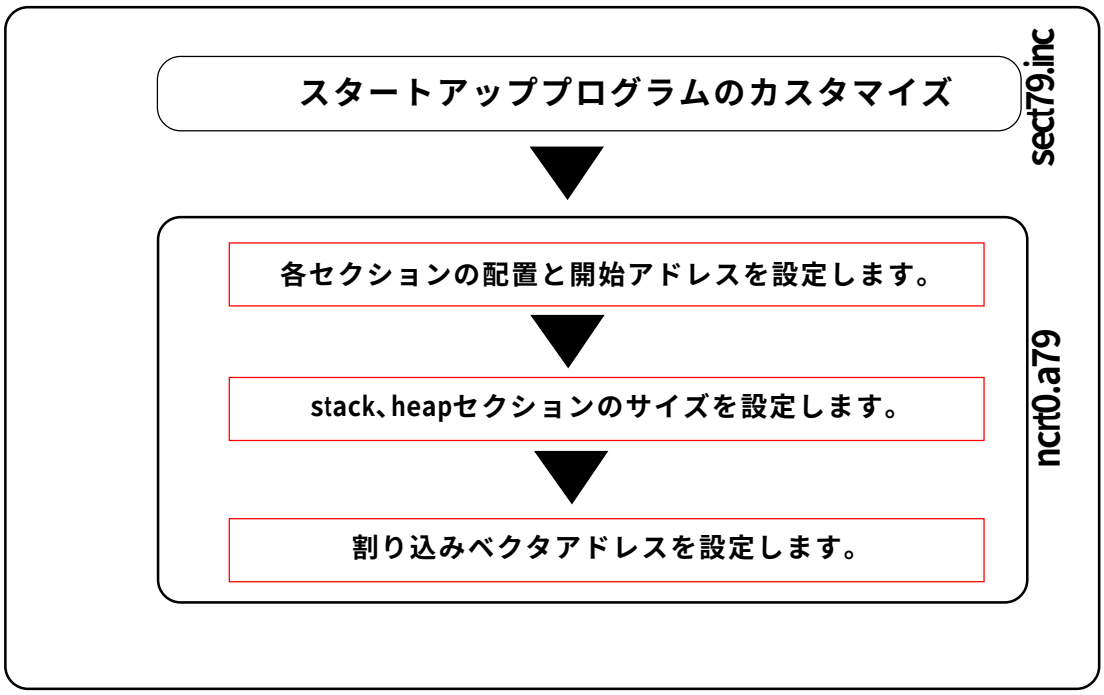


図2.22 メモリ配置の変更手順例

d.セクション配置と開始アドレスの設定

セクションの配置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sect79.incで行います。

セクションの配置は、sect79.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as79の疑似命令.ORGを用いて指定します。【図2.23】に設定例を示します。

```
.section    program_N
            .ORG        0C000H←program_Nセクションの開始アドレスの設定
;
```

図2.23　セクション開始アドレスの設定例(sect79.inc)

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、7900シリーズのメモリ(RAM/ROM)の属性に影響されるため、配置できる領域が限られているセクションがあります。セクションの配置には、以下の規則に従ってください。

(a)RAM領域に配置するセクション

- ●stackセクション
- ●heap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_DP1Eセクション
- ●data_DP2Eセクション
- ●data_DP3Eセクション
- ●data_DP1Oセクション
- ●data_DP2Oセクション
- ●data_DP3Oセクション
- ●bss_NEセクション
- ●bss_NOセクション
- ●bss_FEセクション
- ●bss_FOセクション
- ●bss_DP1Eセクション
- ●bss_DP2Eセクション
- ●bss_DP3Eセクション
- ●bss_DP1Oセクション
- ●bss_DP2Oセクション
- ●bss_DP3Oセクション

(b)ROMに配置するセクション

- ●interruptセクション
- ●vectorセクション
- ●program_Nセクション
- ●program_Fセクション
- ●rom_NEセクション
- ●rom_NOセクション
- ●data_IDP1Eセクション
- ●data_IDP2Eセクション
- ●data_IDP3Eセクション
- ●rom_FEセクション
- ●rom_FOセクション
- ●data_INEセクション
- ●data_INOセクション
- ●data_IFEセクション
- ●data_IFOセクション
- ●data_IDP1Oセクション
- ●data_IDP2Oセクション
- ●data_IDP3Oセクション

なお、偶数サイズの初期値を格納したセクション（data_INE、data_IFE）の先頭アドレスに対してワードアライメントをとっても効果はありません。
また、セクションは7900シリーズのメモリ空間において配置できる領域が限られているセクションがあります。

(a)0バンクにのみ配置できるセクション

- ●stackセクション
- ●interruptセクション
- ●vectorセクション

(b)-bank=オプションで指定されたバンクに配置できるセクション

- ●data_NEセクション
- ●data_NOセクション
- ●bss_NEセクション
- ●bss_NOセクション
- ●rom_NEセクション
- ●rom_NOセクション

※-bank=の指定がない場合、0バンクに配置します。

(c)7900シリーズの全メモリ空間に配置できるセクション

- ●heapセクション
- ●data_INEセクション
- ●data_INOセクション
- ●data_FEセクション
- ●data_FOセクション
- ●bss_FEセクション
- ●bss_FOセクション
- ●program_Nセクション※
- ●program_Fセクション
- ●rom_FEセクション
- ●rom_FOセクション
- ●data_IFEセクション
- ●data_IFOセクション

※program_Nセクションはバンク境界にまたがって配置できません。

(d)「#pragmaDP[n]DATA」を使用時に配置できるセクションすべて ”0バンク"に配置してください。

- ●data_DP1Eセクション
- ●data_DP2Eセクション
- ●data_DP3Eセクション
- ●data_DP1Oセクション
- ●data_DP2Oセクション
- ●data_DP3Oセクション
- ●data_IDP1Eセクション
- ●data_IDP2Eセクション
- ●data_IDP3Eセクション
- ●data_IDP1Oセクション
- ●data_IDP2Oセクション
- ●data_IDP3Oセクション
- ●bss_DP1Eセクション
- ●bss_DP2Eセクション
- ●bss_DP3Eセクション
- ●bss_DP1Oセクション
- ●bss_DP2Oセクション
- ●bss_DP3O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、定義する必要はありません（セクションのサイズはリンク時にマップファイル（拡張子.map）を作成することにより調べることができます）。

●data_NE,data_INEセクション
●data_NO,data_INOセクション
●data_FE,data_IFEセクション
●data_FO,data_IFOセクション
●bss_NEセクション
●bss_NOセクション
●data_DP1Eセクション
●data_DP2Eセクション
●data_DP3Eセクション
●data_DP1Oセクション
●data_DP2Oセクション
●data_DP3Oセクション
●bss_DP1Eセクション
●bss_DP2Eセクション
●bss_DP3Eセクション
●bss_DP1Oセクション
●bss_DP2Oセクション
●bss_DP3Oセクション
●bss_FEセクション
●bss_FOセクション
●rom_NEセクション
●rom_NOセクション
●rom_FEセクション
●rom_FOセクション
●data_IDP1Eセクション
●data_IDP2Eセクション
●data_IDP3Eセクション
●data_IDP1Oセクション
●data_IDP2Oセクション
●data_IDP3Oセクション

なお、program_Fセクションはランタイムライブラリが含まれますので必ず配置してください。

(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sect79.incの記述例を【図2.24】、【図2.25】、【図2.26】に示します。この例においてセクション配置を行うプログラムは以下の3条件を満たしている場合です。

①バンク0以外にデータ及びプログラムを配置しない
②メモリ管理関数ライブラリを使用しない
③-fNF(-fnear_function)オプションですべての関数をnear領域に配置する

```
;************************************************************
;
;    sect79.inc : section definition
;
;    NC79 COMPILOR for 7900 Series
;    Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
;    and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;    All Rights Reserved.
;
;    $Id: sect79.inc,v 1.8 XX/XX/XX XX:XX:XX usuki Exp $
;
;************************************************************
;=======================================
;    NEAR
;=======================================
```

図2.24　シングルチップモードにおけるsect79.incリスト(1)

```
STACKSIZE       .equ   300H
HEAPSIZE .equ   300H
;---------------------------------------
;     RAM DATA
;---------------------------------------
      .section      data_NE     ,DATA
      .org          000800H
      .section      data_NO     ,DATA
      .section      bss_NE      ,DATA
      .section      bss_NO      ,DATA
      .section      stack       ,DATA
      .blkb         STACKSIZE
stack_top:
      .section      heap        ,DATA
heap_top:
      .blkb         0                   ← メモリ管理関数を使用しませんのでheap
                                          セクションのサイズを 0 にします。
      .section      data_DP1E   ,DATA
__DP1:
      .section      data_DP1O   ,DATA
      .section      bss_DP1E    ,DATA
      .section      bss_DP1O    ,DATA

      .section      data_DP2E   ,DATA
__DP2:
      .section      data_DP2O   ,DATA
      .section      bss_DP2E    ,DATA
      .section      bss_DP2O    ,DATA

      .section      data_DP3E   ,DATA
__DP3:
      .section      data_DP3O   ,DATA
      .section      bss_DP3E    ,DATA
      .section      bss_DP3O    ,DATA

;---------------------------------------
;     PROGRAM
;---------------------------------------
      .section      interrupt
      .org          004000H
      .section      program_N
      .section      program_F           ← ランタイムライブラリがprogram_Fセク
;---------------------------------------    ションに出力されるため、シングルチッ
;     ROM DATA                             プにおいても定義が必要になります。
;---------------------------------------
      .section      rom_NE      ,ROMDATA
      .section      rom_NO      ,ROMDATA
      .section      data_INE    ,ROMDATA
      .section      data_INO    ,ROMDATA
      .section      data_IDP1E  ,ROMDATA
      .section      data_IDP1O  ,ROMDATA
      .section      data_IDP2E  ,ROMDATA
      .section      data_IDP2O  ,ROMDATA
      .section      data_IDP3E  ,ROMDATA
      .section      data_IDP3O  ,ROMDATA
;---------------------------------------
;     VECTOR TABLE
;---------------------------------------
      .section      vector
      .org          0ffc0H
DMA3:               .word       dummy_int
DMA2:               .word       dummy_int
DMA1:               .word       dummy_int
DMA0:               .word       dummy_int
ROMC:               .word       dummy_int
RESERVED3:          .word       dummy_int
RESERVED2:          .word       dummy_int
RESERVED1:          .word       dummy_int
```

図2.25　シングルチップモードにおけるsect79.incリスト(2)

```
INT4:           .word       dummy_int
INT3:           .word       dummy_int
AD:             .word       dummy_int
UART1S:         .word       dummy_int
UART1A:         .word       dummy_int
UART0S:         .word       dummy_int
UART0A:         .word       dummy_int
TB2I:           .word       dummy_int
TB1I:           .word       dummy_int
TB0I:           .word       dummy_int
TA4I:           .word       dummy_int
TA3I:           .word       dummy_int
TA2I:           .word       dummy_int
TA1I:           .word       dummy_int
TA0I:           .word       dummy_int
INT2:           .word       dummy_int
INT1:           .word       dummy_int
INT0:           .word       dummy_int
NMI:            .word       dummy_int
WDT:            .word       dummy_int
DBC:            .word       dummy_int
BK:             .word       dummy_int
DIV0:           .word       dummy_int
RESET:          .word       OFFSET start
;=======================================
;   FAR
;=======================================
;---------------------------------------
;   RAM DATA
;---------------------------------------
    .section    data_FE     ,DATA
    .org        12000H
    .section    data_FO     ,DATA
    .section    bss_FE      ,DATA
    .section    bss_FO      ,DATA
;---------------------------------------
;   PROGRAM
;---------------------------------------
    .section    program_F
    .org        18000H
;---------------------------------------
;   ROM DATA
;---------------------------------------
    .section    rom_FE      ,ROMDATA
    .section    rom_FO      ,ROMDATA
    .section    data_IFE,ROMDATA
    .section    data_IFO,ROMDATA
;***************************************************************
;
;
;   Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
;   and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;   All Rights Reserved.
;
;***************************************************************
;
```

← 不要ですので取り除きます。

図2.26　シングルチップモードにおけるsect79.incリスト(3)

e.周辺I/O割り込みベクタアドレスの設定

割り込み処理を使用するプログラムでは、sect79.inc中のvectorセクションの割り込みベクタアドレステーブルを変更します。【図2.27】に割り込みベクタアドレステーブル例を示します。

```
;---------------------------------------
;     VECTOR TABLE
;---------------------------------------
      .section      vector
      .org          0ffc0H
DMA3:           .word       dummy_int       ← DMA3割り込み
DMA2:           .word       dummy_int       ← DMA2割り込み
DMA1:           .word       dummy_int       ← DMA1割り込み
DMA0:           .word       dummy_int       ← DMA0割り込み
ROMC:           .word       dummy_int       ← ROM修正機能割り込み
RESERVED3:      .word       dummy_int
RESERVED2:      .word       dummy_int
RESERVED1:      .word       dummy_int
INT4:           .word       dummy_int       ← 外部割り込みINT4
INT3:           .word       dummy_int       ← 外部割り込みINT3
AD:             .word       dummy_int       ← A-D変換割り込み
UART1S:         .word       dummy_int       ← UART1送信割り込み
UART1A:         .word       dummy_int       ← UART1受信割り込み
UART0S:         .word       dummy_int       ← UART0送信割り込み
UART0A:         .word       dummy_int       ← UART0受信割り込み
TB2I:           .word       dummy_int       ← タイマB2割り込み
TB1I:           .word       dummy_int       ← タイマB1割り込み
TB0I:           .word       dummy_int       ← タイマB0割り込み
TA4I:           .word       dummy_int       ← タイマA4割り込み
TA3I:           .word       dummy_int       ← タイマA3割り込み
TA2I:           .word       dummy_int       ← タイマA2割り込み
TA1I:           .word       dummy_int       ← タイマA1割り込み
TA0I:           .word       dummy_int       ← タイマA0割り込み
INT2:           .word       dummy_int       ← 外部割り込みINT2
INT1:           .word       dummy_int       ← 外部割り込みINT1
INT0:           .word       dummy_int       ← 外部割り込みINT0
NMI:            .word       dummy_int       ← NMI割り込み
WDT:            .word       dummy_int       ← 監視タイマ割り込み
DBC:            .word       dummy_int       ← DBC割り込み
BK:             .word       dummy_int       ← BRK命令
DIV0:           .word       dummy_int       ← 0除算割り込み
RESET:          .word       OFFSET start    ← リセット割り込み

※dummy_intはダミーの割り込み処理関数です。
```

図2.27 割り込みベクタアドレステーブル(sect79.inc)

割り込みベクタの内容は、7900シリーズの機種により異なります。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。

割り込みベクタアドレステーブルの変更は、次の手順で行います。

①NC79で作成した関数への登録ラベル名は、関数名の前に_(アンダースコア)が付加されます。したがって、ここで宣言する割り込み処理関数名にもアンダースコアを付加して記述します。
②割り込み処理関数名をベクタアドレステーブルの該当する割り込み名ラベル(AD:、UART1S:、等)に対応したダミーの割り込み関数名dummy_intを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の登録例を示します。

```
;----------------------------------------
;     VECTOR TABLE
;----------------------------------------
      .section      vector
      .org          0ffc0H
DMA3:               .word         dummy_int
DMA2:               .word         dummy_int
DMA1:               .word         dummy_int
DMA0:               .word         dummy_int
ROMC:               .word         dummy_int
RESERVED3:          .word         dummy_int
RESERVED2:          .word         dummy_int
RESERVED1:          .word         dummy_int
INT4:               .word         dummy_int
INT3:               .word         dummy_int
AD:                 .word         dummy_int
UART1S:             .word         _uarttrn
                      :
              （以下省略）
```

図2.28　割り込みベクタアドレスの設定例(sect79.inc)

第3章
プログラミング

この章では、NC79が生成するコード効率の良いプログラムを記述するための、整数/変数の取り扱い方法、nc79の起動オプションの指定に関して説明します。

3.1 注意事項

3.1.1 バージョンアップについての注意事項

NC79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、またはコンパイラのバージョンアップを行った場合には、再度アプリケーションの動作評価を必ず行ってください。

なお、割り込み処理プログラムと被割り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でも同一のRAM上に確保される場合には、同様に排他制御を行ってください。

3.1.2最適化について

a.最適化の抑止方法

NC79では-Oオプションを指定しなくても【図3.1】に示す記述はデフォルトで最適化されます。

```
extern int    port;

func()
{
      port;
}
```

図3.1 最適化される例

この例はportを読み出すだけの操作を行いたいことを意図して記述したものですが、実際には読み出すコードは最適化されて出力されません。最適化を行わないようにするには【図3.2】に示すようにvolatile修飾子を付加してください。

```
extern int    volatile        port;

func()
{
      port;
}
```

図3.2 最適化を抑止する例

b.コード生成に関する注意事項

NC79では、-Oオプションを指定しなくても【図3.3】に示す記述はデフォルトで最適化されます。

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

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

図3.3 最適化される例

この例の場合、変数cはcharと記述されていますので、NC79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行われない文を記述された場合でも、NC79ではアセンブラコードを生成しませんので注意してください。

c.その他

NC79では、16ビット長データをlong型でキャストすることにより、32ビット長への拡張処理を行わずに32ビットの結果を得るコードを生成します。

```
long func( int i1, int i2 )
{
      long    l1, l2, l3;

      l1 = i1;
      l2 = i2;
      l3 = i1 * i2;
      return l3;
}
```

図3.4 16ビット長データ間の乗算結果を32ビット長で求める例(1)

この【図3.4】の例は、16ビット長変数i1とi2の乗算結果を32ビット長変数l3に格納することを意図して記述したものですが、16ビット長データを32ビット長データに代入している分、コード効率が悪くなります。

このような場合には、【図3.5】のようにキャスト演算子を使用することにより、コード効率を良くすることができます。

```
long func( int i1, int i2 )
{
      long    l

      l = ( long )i1 * ( long )i2;
      return l;
}
```

図3.5 16ビット長データ間の乗算結果を32ビット長で求める例(2)

3.1.3 register変数の使用に関する注意事項

a.register修飾子を有効にするために

関数内でローカルな変数に対してregister修飾子を指定した場合、オプション-fenable_register(-fER)を指定した場合のみ有効になります。オプション-fenable_register(-fER)を指定しない場合には、register修飾子を指定した変数もauto変数として扱います。

b.最適化によるregister変数

引数がregister渡しになる関数では、registerで渡された引数を一度auto領域（スタックフレーム上）に転送を行います。

オプション-O、－OR、－OS、-O1～-O5を指定すると、コード効率の向上のために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。

3.2 生成コードの向上

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3.6】に示しますようにfar型配列のextern宣言においてサイズを明記することによりコード効率を良くすることができます。※2

```
extern int far array[];       ← サイズが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3.6 far型配列のextern宣言例

c.配列の添え字に関して

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

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

②サイズが64K以上のfar型配列の場合
　添え字は必ずlong型に拡張されて演算されます。

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

※1.char型、ビットフィールド構造体のメンバで符合指定がない場合、符号なしとして扱われます。
※2.-fsmall_array(-fSA)を指定することにより、サイズが不明の配列を64Kバイト以内と仮定したコードを生成することができます。

d.プロトタイプ宣言の活用

NC79では関数のプロトタイプ宣言を行わない場合、その関数を呼び出す時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つみます。このため、関数のプロトタイプ宣言を行わない場合、冗長な型拡張を行うことがあります。

表3.1 引数に関するスタックの使用規則

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

NC79では、関数のプロトタイプ宣言を行うことにより関数の引数をレジスタに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うことができます。

e.関数のnear・far制御テクニック

関数に対するnear・far指定により、その関数を呼び出す場合にJSR命令で呼び出すかJSRL命令で呼び出すかを制御することができます。

プログラムの容量が64Kバイトを越える場合、一般にすべての関数をfar関数にしますが、ROM容量とスタック容量をできるだけ少なくするために局所的にnear関数を使用することがあります。

関数のnear・far指定はデータの場合と異なり、現在のプログラムバンクレジスタ（PG）の示すバンク内にその関数が有るか無いかで、near・farの指定を行います。

したがって、near関数用の静的なバンクが存在するのではなく、あくまで現在のPGの値に依存したバンクになります。そのため、局所的にnear関数を定義した場合、その呼びだし側と定義側のプログラムバンクが同一でなければなりません。

そこで同一ファイル内に記述された関数はセクションが同じであれば、同じプログラムバンク内に配置される確率が高い特性を利用してstatic関数のみをnear関数にする方法があります。

このためには、near関数とfar関数を同一セクションに配置するための起動オプション-ffar_ program_section（-fFPS）を指定してコンパイルを行います。この方法を用いれば、局所的にnear関数を定義してもリンク時にエラーになる確率が少なくなります。

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、構造体等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)関数呼び出しをまたいで使用する変数をレジスタ変数にしない。

3.2.2 スタートアップ処理を高速化する方法

スタートアッププログラムncrt0.a79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3.7】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領域のクリア処理）が必要になります。

```
static int i;
```

図3.7 初期値を持たない変数の宣言例

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

```
;================
;       NEAR
;================
;----------------------------------------
;       Zero clear for data area
;----------------------------------------
;BZERO      bss_NE,bss_NE
;BZERO      bss_NO,bss_NO
;lda.W A,#OFFSET stack_top - 1  ; reset stack pointer
;tas
                :
          （省略）
                :
;----------------------------------------
;       Zero clear for data area
;----------------------------------------
;ldd    (1,2,3),#__DP1,#__DP2,#__DP3
;BZERO      bss_DP1E,bss_DP1E
;BZERO      bss_DP1O,bss_DP1O
;BZERO      bss_DP2E,bss_DP2E
;BZERO      bss_DP2O,bss_DP2O
;BZERO      bss_DP3E,bss_DP3E
;BZERO      bss_DP3O,bss_DP3O
;lda.W A,#OFFSET stack_top - 1  ; reset stack pointer
;tas
```

図3.8 bss領域のクリア処理のコメントアウト例

3.3 アセンブリ言語プログラムとの結合方法

3.3.1 C言語プログラムからアセンブラ関数の呼び出し方法

a.引数のないアセンブラ関数の呼び出し方法

C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、このアンダースコアをとった名前を使用します。

アセンブラ関数をJSR命令で呼び出すかJSRL命令で呼び出すかを示すために、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。※1

【図3.9】にアセンブラ関数asm_funcを呼び出すときの記述例を、【図3.10】にコンパイル結果の例を示します。

```
extern void near    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3.9 引数がない場合のアセンブラ関数の呼び出し例(1)

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

図3.10【図3.9】のコンパイル結果(抜粋)

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、A、B、X、Yレジスタを介してアセンブラ関数に引数を渡します。ただし、A、B、X、Yレジスタはすべて16ビットで使用されます。引数は最大4個まで記述できます。

#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※1。このときには、引数の型宣言が必要となります。

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。

※1.プロトライプ宣言に記述されたnear、またはfarによって、アセンブラ関数を呼び出す命令が変わります。near属性の関数はJSR命令、far属性の関数はJSRL命令で呼び出されます。また、呼び出し元のC言語プログラムにリターンする時は、near属性のアセンブラ関数ではRTS命令、far属性のアセンブラ関数ではRTL命令を記述する必要があります。

【図3.11】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( X,Y )              ← 引数をX、Yレジスタを介して
                                                  アセンブラに渡します。
void main()
{
      int      i = 0x02;
      int      j = 0x05;

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

図3.11 引数がある場合のアセンブラ関数の呼び出し(2)

```
      .glb    _main
_main:
      phd     0
      pha
      pha
      tsd
      ._line  6
;## # C_SRC :            int      i = 0x02;
      movm.W      DP0:3,#0002H        ; i
      ._line  7
;## # C_SRC :            int      j = 0x05;
      movm.W      DP0:1,#0005H        ; j
      ._line  9
;## # C_SRC :            asm_func( i,j );
      ldy     DP0:1 ; j                         ← 引数をX、Yレジスタを介して
      ldx     DP0:3 ; i                           アセンブラ関数に渡しています。
      jsrl    _asm_func
      ._line  10
;## # C_SRC :    }
      plx
      plx
      rtld    0
```

図3.12【図3.11】のコンパイル結果（抜粋）

3.3.2 アセンブラ関数の記述方法

a.呼び出されるアセンブラ関数の記述方法

アセンブラ関数の入り口処理の記述手順を以下に示します。

①アセンブラの疑似命令.SECTIONでセクション名を指定します。セクション名には任意の名称を付けることができます。

②関数名ラベルをアセンブラの疑似命令.GLBでグローバル指定します。

③関数名に_(アンダースコア)を付加して、ラベルとして記述します。

データ長選択フラグ(m)、インデックスレジスタ長選択フラグ(x)、10進演算モードフラグ(D)は、すべてクリアされた状態でC言語プログラムから呼び出されます。アセンブラ関数からリターンするときはすべてクリアの状態に戻してください。

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。

④m、x、Dフラグをすべてクリアします。

⑤RTS又はRTL命令を記述します。※2

【図8.3】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NC79が出力するセクション名と同じprogram_Nを用いています。

```
      .SECTION    program_N                ← (1)
      .GLB        _asm_func
_asm_func:                                 ← (2)
      sta     A,MEMO1                      ← (3)
      sta     B,MEMO2
      ldab    A,#7H
              :
        （省略）
              :
      clp     M,X,D                        ← (4)
      rts                                  ← (5)
      .END
```

図3.13　アセンブラ関数の呼び出し例（引数がない場合）

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、Aレジスタ、又はEレジスタに戻り値を格納して返します。【表3.2】に戻り値に関する呼び出し規則を、【図3.14】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
char型	Aレジスタの下位8ビットに格納して返します。この時Aレジスタの上位8ビットは不定となります。
int型 nearポインタ型	Aレジスタに格納して返します。
float型 long型 farポインタ型	Eレジスタに格納して返します。
集合体 double型 long double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_N
      .GLB        _asm_func
_asm_func:
              :
        （省略）
              :
      ldad    E,#00001A00H              ;32ビットデータ
      clp     M,X,D
      rts
      .END
```

図3.14　long型の戻り値を返すアセンブラ関数の記述例

※2.アセンブラ関数内でDT、DPRレジスタとm、x、Dフラグの内容を書き換える操作は行わないでください。DT、DPR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。この場合には、RTSD、またはRTLD命令を使用することができます。また、m、x、Dフラグの内容を書き換える場合は、アセンブラ関数の出口ですべてクリアの状態に戻してください。

c.C言語変数の参照方法

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

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

【図3.15】にC言語プログラムの大域変数counterをアセンブラ関数asm_func内で参照する例を示します。

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

main()
{
                :
          (省略)
                :
}


[アセンブラ関数]
      .GLB   _counter       ← C言語プログラムの大域変数を外部参照宣言
_asm_func:
                :
          (省略)
                :
      lda.W  A,_counter     ← 参照
```

図3.15 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場合の注意事項

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(A、B、X、Y、DPR、DT)を一括に退避します。
2.関数の出口でレジスタ(A、B、X、Y、DPR、DT)を一括に復帰します。
3.関数からのリターンにRTI命令を使用します。

- レジスタの退避と復帰は、必ずデータ長選択フラグ(m)とインデックスレジスタ長選択フラグ(x)を同じ状態にしたあとで行ってください。
- 割り込み処理プログラム中でDTレジスタに依存するアドレッシングモードの命令を使用する場合は、割り込み処理プログラムの入り口でDTレジスタの値を必ずロードしてください。※1

【図3.16】に割り込み処理のアセンブラ関数の記述例を示します。

※1.Cコンパイラの生成するコードにおいて、DTレジスタを一時変更するコードを出力することがあります。したがって、アセンブリ言語で記述した割り込み処理プログラム中で「DTを用いたアドレッシングモードを使用する」場合、必ず割り込み処理プログラムの先頭でDT値を退避した上で必要なDT値をロードしてください。

```
      .SECTION    interrupt
      .DT     __DT
      .GLB  _int_func
_int_func:
      clp     m,x                 ← M、Xフラグをクリア
      psh     A,B,X,Y,DP,DT       ← レジスタの一括待避
      ldt     __DT                ← 割り込み処理プログラム中で使用する
      ldab    A,#7H               ← DT値のセット
              :
        （省略）
              :
      clp     m,x                 ← M、Xフラグをクリア
      pul     A,B,X,Y,DP,DT       ← レジスタの一括復帰
      rti                         ← C言語プログラムへリターン
      .ENDFUNC
      .END
```

図3.16 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場合の注意事項

アセンブラ関数内からC言語関数を呼び出すときは以下の点に注意してください。

1.m、x、Dフラグはすべてクリアされた状態で呼び出してください。
2.near属性の関数はJSR命令、far属性の関数はJSRL命令で呼び出してください。

3.3.3 アセンブラ関数の記述に関する注意事項

C言語プログラムから呼び出すアセンブリ言語の関数(サブルーチン)を記述する場合、以下の点に注意してください。

a.m、x、Dフラグの取り扱いに関する注意事項

プロセッサステータスレジスタ中のm、x、Dフラグは、すべてクリアされた状態でC言語プログラムから呼び出されます。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m、x、Dフラグをすべてクリアの状態にしてください。

b.DT、DPR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DPR(ダイレクトページレジスタ)、DT(データバンクレジスタ)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DT及びDPR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.A、B、X、Y、E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A、B、X、Y、Eレジスタの内容を変更しても問題はありません。

d.アセンブラ関数への引数に関する注意事項

アセンブリ言語で記述した関数に対して引数を渡す場合、#pragma PARAMETER機能を使用してその引数をA、B、X、Yレジスタを介して渡すことができます。ただし、NC79のEレジスタを介して引数を渡すことはできません（NC79においてもA、B、X、Yレジスタを使用してください）。アセンブラ関数へレジスタを介して引数を渡す場合の書式を【図14.7】に示します（図中のasm_funcはアセンブラ関数名です）。

```
unsigned int near  asm_func( unsigned int, unsigned int );
                                        ↑アセンブラ関数のプロトタイプ宣言

#pragma PARAMETER   asm_func( X, Y )
```

図3.17 アセンブラ関数の記述例

#pragma PARAMETER機能で使用するA、B、X、Yレジスタは、すべて16ビット長で使用されます。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ってください。

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

- ●構造体型、又は共用体型
- ●浮動小数点型
- ●long型、char型、void型などのサイズが16ビット以外の型
- ●far属性のポインタ型

また、関数の戻り値として

- ●doubleを返す関数
- ●構造体、又は共用体を返す関数

を宣言することはできません。

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

NC79は、弊社製Cコンパイラ「NCxx」とは拡張機能を含む言語仕様レベルで基本的に互換性を有しています。ただし、以下の点については異なりますのでご注意ください。

a.near/farデフォルトの違い

NCシリーズのnear/farデフォルトは以下の【表3.3】の通りとなっています。このため、移植時には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
NC79	near	near	far
NC77	near	near	far
NC30	near	far	far固定
NC308	near (ただし、ポインタ型はfar)	far	far固定

3.4.2 NC79とNC77間の移植に関する注意事項

NC79とNC77ではアセンブラレベルでの関数名の扱いが異なっています。

表3.4 関数名とアセンブラシンボル名の変換規則

関数	NC79	NC77
レジスタ引数を持つ	$関数名	?関数名
レジスタ引数を持たない	__関数名	__関数名

なお、NC77で作成したプログラムをNC79でリコンパイルする場合には、NC79起動オプション-fDP_offset_8(-fDPO8)を指定してください。

付録A
コマンドオプションリファレンス

付録Aでは、コンパイルドライバnc79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、nc79から起動できるアセンブラas79とリンケージエディタln79の起動オプションをあわせて記載しています。

A.1　nc79コンパイルドライバの入力書式

```
%nc79△[起動オプション]△<[アセンブリ言語ソースファイル名]△
      [リロケータブルオブジェクトファイル名]△[C言語ソースファイル名]>

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

図A.1 nc79コマンドの入力書式

```
%nc79 -osample -as79“ -l ”-ln79“ -ms ”ncrt0.a79 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 nc79コマンドの入力例

A.2　nc79の起動オプション

A.2.1　コンパイルドライバの制御に関するオプション

【表A.1】にコンパイルドライバの制御に関する起動オプションを示します。

表A.1 コンパイルドライバの制御オプション

オプション	機能
-c	リロケータブルファイル(拡張子.r79)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。
-l*ディレクトリ名*	#includeで指定するファイルが存在するディレクトリ名を指定します。ディレクトリは最大8個まで指定可能です。
-E	プリプロセスコマンドのみを起動し結果を標準出力に出力します。※1
-P	プリプロセスコマンドのみを起動しファイル(拡張子.i)を作成します。※1
-S	アセンブリ言語ソースファイル(拡張子.a79及び.ext)を作成し、処理を終了します。※1
-Uﾌﾟﾘﾃﾞﾌｧｲﾝﾄﾞﾏｸﾛ名	指定したプリデファインドマクロを未定義にします。
-silent	起動時のコピーライトメッセージを出力しません。

※1.起動オプション-c、-E、-P、および-Sを指定しない場合、nc79はln79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.x79）まで作成します。

-c

コンパイルドライバの制御

[機能]　リロケータブルオブジェクトファイル（拡張子.r79）を作成し、処理を終了します。

[実行例]

```
D:\work>nc79 -c sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c


D:\work>dir sample.*
 ドライブ D のボリューム ラベルは WinNT です
 ボリューム シリアル番号は D812-16E7 です

 D:\work のディレクトリ

98/07/31  03:42p             2,939 sample.c
99/02/09  05:05p               774 SAMPLE.R79
          2 個のファイル           3,713 バイト
                693,170,688 バイトの空き領域

D:\work>
```

[注意事項]　このオプションを指定したときは、アブソリュートモジュールファイル（拡張子.x79）等、ln79で処理した結果出力されるファイルは生成されません。

-D*識別子名*

コンパイルドライバの制御

[機能]　プリプロセスコマンドの#defineと同じ機能です。複数の識別子を定義するときは、スペースで区切ってください。

[書式]　nc79△-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。

-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で指定するファイルが存在するディレクトリ名を指定します。
最大8個のディレクトリを指定できます。

[書式]　nc79△-Iディレクトリ名△<C言語ソースファイル名>

[実行例]

```
D:\work>% nc79 -c -Id:\test\include -Id:\test\inc sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c


D:\work>
```

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

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

-E

コンパイルドライバの制御

[機能]　プリプロセスコマンドのみを起動し結果を標準出力に出力します。

[実行例]

```
D:\work>nc79 -E sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

#line 1 "sample.c"
        :
      (省略)
        :
#line 1 "D:\mtool\inc79\stdio.h"
        :
      (省略)
        :

D:\work>
```

[注意事項]　このオプションを指定したときは、アセンブリ言語ソースファイル（拡張子.a79）、リロケータブルオブジェクトファイル（拡張子.r79）、機械語データファイル（拡張子.x79）等、ccom79、as79、及びln79で処理した結果出力されるファイルは生成されません。

-P

コンパイルドライバの制御

[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）を作成します。

[実行例]

```
D:\work>nc79 -P sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c


D:\work>
```

[注意事項]
1. このオプションを指定したときは、アセンブリ言語ソースファイル（拡張子.a79)、リロケータブルオブジェクトファイル（拡張子.r79）、アブソリュートモジュールファイル（拡張子.x79)等、ccom79、as79、およびln79で処理した結果出力されるファイルは生成されません。
2. このオプションにより生成されるファイル（拡張子.i）には、プリプロセッサが生成する#lineは含まれません。#lineを含む結果を得る場合には、-Eオプションを指定し、リダイレクトしてください。

-S

コンパイルドライバの制御

[機能]　アセンブリ言語ソースファイル（拡張子.a79）を作成し、処理を終了します。

[実行例]

```
D:\work>nc79 -S sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c

D:\work>dir sample.*
 ドライブ D のボリューム ラベルは WinNT です
 ボリューム シリアル番号は D812-16E7 です

 D:\work のディレクトリ

99/02/09  05:13p             3,394 sample.a79
98/07/31  03:42p             2,939 sample.c
               2 個のファイル           7,107 バイト
                     693,271,552 バイトの空き領域

D:\work>
```

[注意事項]　このオプションを指定したときは、リロケータブルオブジェクトファイル（拡張子.r79）、アブソリュートモジュールファイル（拡張子.x79)等、as79、およびln79で処理した結果出力されるファイルは生成されません。

-Uプリデファインドマクロ名

コンパイルドライバの制御

[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]　nc79△-Uﾌﾟﾘﾃﾞﾌｧｲﾝﾄﾞﾏｸﾛ名△<C言語ソースファイル名>

[実行例]

```
D:\work>nc79 -c -UNC79 sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c


D:\work>
```

※この例では、マクロ定義NC79を未定義にしています。

[注意事項]　未定義にできるマクロの数は、使用しているホストマシンのOSのコマンドラインの最大文字数により制限されることがあります。
__STDC__、__LINE__、__FILE__、__DATE__、__TIME__は未定義にすることはできません。

-silent

コンパイルドライバの制御

[機能]　起動時のコピーライトメッセージを出力しません。

[実行例]

```
D:\work>nc79 -c -silent sample.c

sample.c


D:\work>
```

A.2.2 出力ファイル指定オプション

【表A.2】に出力するアブソリュートモジュールファイルの名称を指定する起動オプションを示します。

表A.2 出力ファイル指定オプション

オプション	機能
-o*ファイル名*	ln79が生成するファイル(機械語データファイル、マップファイル、シンボルファイル等)の名称を指定します。また、ディレクトリ名も指定できます。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*	ln79が生成するファイル(アブソリュートモジュールファイル、マップファイル、等)の出力先ディレクトリを指定できます。

-oファイル名

出力ファイル指定

[機能]　ln79が生成するファイル(機械語データファイル、マップファイル、シンボルファイル、等)の名称を指定します。また、出力先のディレクトリ名も指定できます。ファイルの拡張子は必ず省略してください。

[書式]　nc79△-oファイル名△<C言語ソースファイル名>

[実行例]

```
D:\work>nc79 -od:\test\sample ncrt0.a79 sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a79
sample.c


D:\work>
```

※この例では、ディレクトリd:\testにアブソリュートモジュールファイルsample.x79のファイルを出力する設定を行っています。

-dirディレクトリ名

出力ファイル指定

[機能]　出力ファイルの出力先を指定できます

[書式]　nc79△-dirディレクトリ名

[実行例]

```
D:\work>nc79 -dird:\test ncrt0.a79 sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a79
sample.c


D:\work>
```

※この例では、ディレクトリd:\testにアブソリュートモジュールファイルncrt.a79のファイルを出力する設定を行っています。

A.2.3 バージョン情報表示オプション

【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 バージョン情報表示オプション

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

-V

コマンドプログラム名の表示

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

[実行例]

```
D:\work>nc79 -c -v sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
D:\MTOOL\LIB79\cpp79 -DNC79 sample.c -o D:\MTOOL\NC79WA~1\TMP\sample.i

D:\MTOOL\LIB79\ccom79 D:\MTOOL\NC79WA~1\TMP\sample.i -o .\sample.a79
D:\MTOOL\BIN\as79 -. -N sample.a79 -o.


D:\work>
```

[注意次項]　このオプションは、小文字のvを記述します。

-V

バージョン情報の表示

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

[実行例]

```
C:\>nc79 -V
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

NC79 C Compile Driver   for 7900 Series Version X.XX.XX
NC Preprocessor              Version X.XX.XX
NC79 Compiler for 7900 Series   Version  X.XX.XX (NC_CORE Version X.XX.XX)
7900 Series Assembler system Version X.XX ReleaseX
  Assembler Driver (as79) for 7900 Series Version X.XX.XX
  Macro Processor (mac79) for 7900 Series Version X.XX.XX  (core X.XX.XX)
  Structured Processor (pre79) for 7900 Series Version X.XX.XX
  Assembler Processor (asp79) for 7900 Series Version X.XX.XX
  Linkage Editor (ln79) for 7900 Series Version X.XX.XX
  Librarian (lb79) for 7900 Series Version X.XX.XX
  Load Module Converter (lmc79) for M37900 Series Version X.XX.XX (core X.XX.XX)
  Cross Referencer (xrf79) for 7900 Series Version X.XX.XX
  Absolute Lister (abs79) for 7900 Series Version X.XX.XX

C:\>
```

[補足説明]　本オプションはコンパイラが正常にインストールされたか否かを確認するために使用します。コンパイラ内部で実行される各コマンドの正しいバージョン番号はリリースノートに記載しています。
リリースノートに記載されているバージョン番号と、本オプションの表示内容が異なる場合、インストールが正常に行われていない可能性があります。インストール方法の詳細は「NC79WA/AS79ガイドブック」を参照してください。

[注意事項]　1.このオプションは、大文字のVを記述します。
2.このオプションを指定したときは、他のオプションはすべて無効になります。

A.2.4 デバッグ用オプション

【表A.4】にC言語レベルデバッグ情報を出力するデバッグの起動オプションを示します。

表A.4 デバッグ用オプション

オプション	短縮形	機能
-g	なし	デバッグ情報をアセンブリ言語ソースファイルファイル（拡張子.a.79）に出力します。
-genter	なし	関数呼び出し時に必ずスタックフレームを生成します。
-gno_reg	なし	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ファイル（拡張子.a.79）に出力します。

[実行例]

```
D:\work>nc79 -g -v sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
D:\MTOOL\LIB79\cpp79 -DNC79 sample.c -o D:\MTOOL\TMP79

D:\MTOOL\LIB79\ccom79 D:\MTOOL\TMP\sample.i -o .\sample.i

D:\MTOOL\BIN\as79 -. -N --N sample.a79 -o.

D:\MTOOL\BIN\ln79 sample.r79 -. -G -MS -l nc79lib -o sample.r79
                :
             (省略)
                :
D:\work>dir /w sample.*
                :
             (省略)
                :
sample.c    sample.map   sample.x79
```

[注意事項]

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

-genter

スタックフレームの構築

[機能]　関数呼び出し時に必ずスタックフレームを形成します。

[注意事項]　デバッガのスタックトレース機能を使用するときには必ずこのオプションを指定してください。指定しない場合は、正しい結果が得られません
このオプションを指定した場合、必要性の有無にかかわらず必ずスタックフレームを関数の入り口で生成します。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。

[補足説明]　レジスタ変数に対するデバッグ情報が必要でない場合には本オプションを指定して、レジスタ変数に対するデバッグ情報の出力を抑止して下さい。デバッガへのダウンロードの高速化が期待できます。

A.2.5 最適化オプション

【表A.5】にプログラムの実行速度、および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	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。これによりROM容量を削減することができます。ただし、使用するスタック容量が多くなる可能性があります。
-Ostack_frame_align	-OSFA	スタックフレームの偶数アライメントを行います。
-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。

[主な最適化オプションの効果を以下に示します。]

最適化オプション効果一覧表

効果	-O	-OR	-OS	-OSA	-OSFA
速度	良	悪	良	良	良
ROMサイズ	良	良	悪	良	同

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、-Oは-O3と同じです。

[補足説明]　速度には効果があるがコードサイズが増える、もしくは、コードサイズは減るが速度は低下するような最適化は行いません。

[レベル概要]　●-O1
-O3、-Ono_bit、-Ono_break_source_debug、-Ono_float_const_fold、-Ono_stdlibを有効にします。

●-O2
-O1と同じ動作をします。

●-O3
速度、およびROM容量ともに効果のある最大限の最適化を行います。

●-O4
-O3と-Oconstを有効にします。

●-O5
共通部分式の最適化（-OR同時指定時）、文字列転送、比較（－OS同時指定時）等の最適化をより強化します。ただし、以下のような場合、正常なコードを出力しません。
　○ポインタ等により、異なる変数が同時に同じメモリ位置を指す場合で、かつ、それらの変数を同一関数内で使用する場合

（例）

```
void          main( void )
{
      test1();
}

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

void          test1( void )
{
      *p = a * 3;              /* a = *p = 9 */
      a = 10;                  /* a = *p = 10 */
      b = *p;          /* a = *p = b = 10 */

      printf(“ b = %d( expect b = 10 )\n ”,b );
}
```

(実行結果)

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

-OR

最適化

[機能]　速度は低下しますが、ROM容量を最小にする最大限の最適化を行います。このオプションは、-gオプション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が変更される最適化が行われる可能性があります。ソース行情報を変更されたくない場合、-Ono_break_source_debug(-ONBSD)オプションを使用して最適化を抑止してください。

-OS

最適化

[機能]　ROM容量は増大する場合がありますが、速度重視の最大限の最適化を行います。このオプションは、-gオプションと同時に指定することができます。

-Oconst　　-OC

最適化

[機能]　const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。

[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. 構造体、共用体、及び配列を除く外部変数
2. const修飾子を指定した外部変数
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

[記述例]

```
int const i = 10;

func()
{
      int k = i;  /* iを10に置き換える。*/
                :
                :

}
```

-Ono_bit　　-ONB

最適化の抑止

[機能]　ビット操作をまとめる最適化を抑止します。

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

[注意事項]　-O(もしくは-OR、-OS)オプションを指定したときのみ効果があります。

-Ono_break_source_debug　-ONBSD

最適化の抑止

[機能]　ソース行情報に影響する最適化を抑止します。

[補足説明]　-OR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、その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オプションを指定したときのみ効果があります。

-Ono_float_const_fold　-ONFCF

最適化の抑止

[機能]　浮動小数点の定数畳み込み処理を抑止します。

[補足説明]　NC79では、デフォルトで定数の畳み込み処理を行います。定数の畳み込み処理の例を以下に示します。

```
[最適化前]
(val/1000e250)*50.0

[最適化後]
val/20e250
```

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

-Ono_stdlib　-ONS

最適化の抑止

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

[注意事項]　標準関数ライブラリ関数と同名の関数をユーザー側で作成する時に、本オプションを指定する必要がある場合があります。

-Osp_adjust　-OSA

スタック補正コードの削除

[機能]　関数呼び出し後のスタック補正コードを取り除く最適化を行います。

[注意事項]　ROM容量を削減することができますが、使用するスタック量が多くなる場合があります。

-Ostack_frame_align　　-OSFA

スタックフレームのアライメント

[機能]　スタックフレームの、偶数アライメントを行います。

[補足説明]　偶数サイズのauto変数が奇数アドレスに配置された場合は、偶数アドレスに配置された場合よりもメモリアクセスが１サイクル多く必要になります。

[注意事項]　本オプションを指定すると、偶数サイズのauto変数を偶数アドレスに配置するようにアライメントを行う為、メモリアクセスを高速に行うことができます。

(1)以下の#pragmaで指定した関数はアライメントを行いません。

- #pragma INTHANDLER
- #pragma HANDLER
- #pragma ALMHANDLER
- #pragma CYCHANDLER
- #pragma INTERRUPT※１

(2)スタートアッププログラムでは、必ずスタックポインタの初期値を偶数アドレスに設定してください。又、全てのプログラムを本オプションを用いてコンパイルしてください。

-Oloop_unroll[=[ループ回数]　　-OLU

ループの展開

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

[補足説明]　実行回数が明確である“ for“ 文 に対して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。
“ -OS“ 指定時にも有効です。

[注意事項]　for文を展開するため、ROMサイズは増加します。

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

A.2.6 生成コード変更オプション

【表A.6】にnc79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しめします。

表A.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。
-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)
-fextend_to_int	-fETI	char型データをint型に拡張し演算を行います(ANSI規格で定められた拡張を行います)。※１
-fchar_enumerator	-fCE	enumerator(列挙子)の型をint型ではなくunsigned char型で扱います。
-fall_far	-fAF	デフォルトをすべてfar型にします。
-fnear_function	-fNF	関数のデフォルトをnearにします。near関数は呼び出すときにjsrで呼出し、rtsで戻ります。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべてoddセクションに配置します。
-ffar_program_section	-fFPS	near関数・far関数をprogram_Fセクションに配置します。
-fnot_use_MVN	-fNUM	MVN命令でのブロック転送を行いません(MVN命令は構造体間の代入で使用されます)。
-bank=n	なし	コンパイル時のデータバンクレジスタ(DT)の値を指定します。指定しない場合、デフォルトは０です。
-ffar_RAM_data	-fFRAM	RAMデータのデフォルト属性をfarにします。
-ffar_ROM_data	-fFROM	ROMデータのデフォルト属性をfarにします。
-fconst_not_ROM	-fCNR	constで指定した型をROMデータとして扱いません。
-fnot_address_volatile	-fNAV	#pragma ADDRESS(#pragma EQU)で指定した変数をvolatileで指定したとみなしません。
-fsmall_array	-fSA	far型の配列を参照する場合、その総バイト数が不明な配列を64Kバイト以下の配列として扱います。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fswitch_table	-fST	ROM効率が良くなる場合のみswitch文に対してテーブルジャンプ方式のコードを生成します。(したがって、本オプションを指定してもジャンプテーブルを使用するとは限りません。)
-fDP_offset_8	-fDPO8	DPレジスタをDPR0の1本のみを使用することを前提とした場合のコードを生成します。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。
-finfo	なし	インスペクタ、”StkViewer ”、”MapViewer ”、”utl79 ”、に必要な情報をアブソリュートモジュールファイル(.x79)に出力します。
-fauto_128	-fA1	ダイレクトページレジスタDPR0～DPR3 をしようする場合におけるスタックフレームサイズの上限を63バイトから127バイトに変更します。

-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]　このオプションを指定することにより、ANSI規格に添ったコードを生成します。

-fnot_reserve_asm　-fNRA

生成コードの変更

[機能]　asmを予約語として扱いません。ただし、同じ機能の_asmは予約語として扱われます。

-fnot_reserve_far_and_near　-fNRFAN

生成コードの変更

[機能]　far、nearを予約語として扱いません。ただし、同じ機能の_far、_nearは予約語として扱われます。

-fnot_reserve_inline　-fNRI

生成コードの変更

[機能]　inlineを予約語として扱いません。ただし、同じ機能の_inlineは予約語として扱われます。

-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。
以下に例を示します。

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

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

この場合、「c2 ＊ 2」の演算でchar型をオーバーフローするため正しい結果を求めることができません。このような場合には、本オプションを指定することにより正しい結果を求めることができます。

※NC79は、ROM効率をよくするため、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。

-fall_far　-fAF

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、データ、自動ポインタアドレス変数、自動変数のアドレス、文字列データの取り扱い及び関数のデフォルトをすべてfarにします。

- -ffar_RAM_data(-fFRAM)
- -ffar_ROM_data(-fFROM)

-fnear_function　-fNF

生成コードの変更

[機能]　関数のデフォルトをnearにします。near関数は呼び出すときにJSR命令で呼出し、RTS命令で戻ります。

[注意事項]　ランタイムライブラリなどは常にprogram_Fセクションに出力するように設定しています。したがって、本オプションを指定した場合でも、ライブラリをリンクした結果、program_Fセクションが含まれます。
本オプションを使用してバンク0内のみで動作するプログラムを作成した場合は、program_Fセクションをライブラリ用のセクションとしてバンク0に配置してください。

-fno_even　　-fNE

生成コードの変更

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

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

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

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

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

-ffar_program_section　　-fFPS

生成コードの変更

[機　能]　near関数、far関数をすべてprogram_Fセクションに配置します。

[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。

-fnot_use_MVN　　-fNUM

生成コードの変更

[機能]　MVN命令でのブロック転送を行いません(MVN命令は構造体間の代入で使用されます)。

[注意事項]　MVN命令実行中、7900シリーズは割り込みを受け付けません。このため、大きいサイズの構造体間の代入を行った場合、割り込みの応答性能が悪くなります。このような場合に本オプションを使用してください。なお、NC79ではMVP命令は生成しません。

-bank=*バンク番号*

生成コードの変更

[機能]　near領域のバンクの値(DT)を指定します。指定しない場合、デフォルトのバンク番号は0です。バンク番号は10進数もしくは16進数で指定できます。16進数で指定する場合、数字の前に0Xもしくは0xを付加してください。

[書式]　nc79△-bank=*バンク番号*△<C言語ソースファイル名>

[注意事項]　1.このオプションを使用してバンク値を0以外に変更するときは、スタートアッププログラムncrt0.a79中の__DTも同じ値に定義してください。
2.-bank及びバンク番号と、=の間にスペースを入れないで下さい。

-ffar_RAM_data　-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。

-ffar_ROM_data　-fFROM

生成コードの変更

[機能]　ROMデータのデフォルト属性をfar属性にします。

-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const指定したデータはROM領域に配置されます。
例えば

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

の場合、配列「array」はROMとして配置されます。本オプションを指定することにより、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指定する必要はありません。

-fnot_address_volatile　-fNAV

生成コードの変更

[機能]　#pragma ADDRESS又は#pragma EQUで指定した大域変数又は関数外に宣言したstatic変数を、volatileで指定された変数として扱いません。

[補足説明]　I/O変数をRA上にあ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回避することができます（#pragma ADDRESS、または#pragma EQUは、通常I/O変数に対して使用するため、volatile指定がなくてもvolatile指定がされているものとして処理されます。）

本オプションは、この処理を抑止します。

通常の用途では、本オプションを使用する必要はありません。

-fsmall_array　-fSA

生成コードの変更

[機能]　総サイズがコンパイル時不明の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]　ROM上のfar型配列等において、配列要素を参照する場合にfar型配列の総サイズが不定であれば、64Kバイト以上の配列にも対応できるように添え字の計算を32ビットで行います。
例えば、

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

の場合、配列「array」の総サイズがコンパイラには分からないため、添え字「j」の計算を32ビットで求めます。

本オプションを指定することにより、配列「array」の総サイズを64Kバイト以下と仮定するため添え字「j」の計算を16ビットで行います。この結果、処理速度の向上、およびコードサイズの削減が可能になります。

※１つの配列のサイズが64Kバイト未満の場合には、常に本オプションを使用することを推奨します。

[注意事項]　配列が、バンクをまたがっている場合には、正常にアクセスすることができません。
この場合には、本オプションを使用しないでください。

-fswitch_table　-fST

生成コードの変更

[機能]　switch文のcase文において、コードサイズが良くなる場合のみジャンプテーブルを用います。

[注意事項]　本オプションを指定してもジャンプテーブルを使用するとは限りません。また、生成されたジャンプテーブルがバンクにまたがって配置される場合には正しく動作しません。

-fDP_offset_8　-fDPO8

生成コードの変更

[機能]　ダイレクトページレジスタ（DPR）をDPR0の1本のみを使用することを前提とした場合のコードを生成します。

[補足説明]　本オプションを指定した場合、ダイレクトページレジスタはDPR0のみを使用し、256バイトオフセットでアクセスします。また、スタックフレームサイズの上限は、255バイトになります。

[注意事項]　1. 本オプションを指定しない場合、ダイレクトページレジスタをDPR0～DPR3の4本を使用することを前提としたコードを生成します。

2. -fauto_128 (-fA1) オプションとの同時指定はできません。

3. 本オプションを指定した場合は、標準ライブラリファイルとして、nc78m8.lib を使用してください。

4. 拡張機能 #pragma DPnDATA (n は 1～3) を指定することはできません。

-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において、必ずしもすべての状況下で最適解を得られない場合があります。本オプションは、上記状況下で、プログラム上でレジスタ割り当てを指示する事により効率を高める手段として用意しています。
本オプションを指定することによりregister指定された
　1.整数型変数
　2.ポインタ変数
を強制的にレジスタに割り当てます。

注意事項]　register指定を行うと逆に効率を低下させる場合があります。本オプションを使用する場合には、必ず生成されたアセンブラ言語ファイルの内容を確認してください。

-fuse_DIV　　-fUD

生成コードの変更

[機能]　除算に対する生成コードを変更します。

[補足説明]　以下のような除算を行う場合に、マイクロコンピュータのdiv、およびdivs命令を生成します。

●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
●被除数が2バイト値、除数が1バイト値で、かつ、演算結果が1バイト値

[注意事項]　本オプションを指定した場合に、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。※1

-finfo

生成コードの変更

[機能]　インスペクタ、”Stk Viewer”、”Map Viewer”、”utl79”、に必要な情報を出力します。

[補足説明]　“ Stk Viewer”、”Map Viewer ”、”utl79“ を使用する時は、このオプションで出力されたアブソリュートファイル、”.x79”ファイルが必要です。

[注意事項]　asm関数内でのグローバル変数の使用はチェックされません。このため、utl79でも、asm関数の使用は無視されます。

※1.7900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にはその値は不定になります。このため、オーバーフローする場合には、本オプションを指定しないでください。

-fauto_128　　-fA1

生成コードの変更

[機能]　ダイレクトページレジスタ DPR0～DPR3 を使用する場合におけるスタックフレームサイズの上限を 63バイトから 127バイトに変更します。

[補足説明]　フレームレジスタとして DPR0 と DPR1 を使用します。

[注意事項]
1. -fDP_offset_8 (-fDPO8) オプション との同時指定はできません。
2. 本オプションを指定した場合は、標準ライブラリとして nc79lib.lib を使用してください。
3. 拡張機能 #pragma DP1DATA を指定することはできません。

A.2.7 ライブラリ指定オプション

【表A.7】にライブラリファイルを指定する起動オプションを示します。

表A.7 ライブラリ指定オプション

ライブラリ	機能
-lﾗｲﾌﾞﾗﾘﾌｧｲﾙ名	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]　　リンク時にln79が使用するライブラリファイル名を指定します。

[実行例]

```
D:\work>nc79 -v -lusrlib ncrt0.a79 sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a79
D:\MTOOL\BIN\as79 -. -N ncrt0.a79 -o.

sample.c
D:\MTOOL\LIB79\cpp79 -DNC79 sample.c -o D:\MTOOL\TMP\sample.i

D:\MTOOL\LIB79\ccom79 D:\MTOOL\TMP\sample.i -o .\sample.a79
D:\MTOOL\BIN\as79 -. -N sample.a79 -o.

D:\MTOOL\BIN\ln79 ncrt0.r79 sample.r79 -. -l usrlib -l nc79lib -o ncrt0


DATA    0002576(00A10H) Byte(s)
ROMDATA  0000298(0012AH) Byte(s)
CODE    0010718(029DEH) Byte(s)

D:\work>
```

※この例では、ライブラリusrlib.libを指定しています。

[注意事項]

1. ファイルの拡張子は ”.lib ”にしてください。
2. ファイルの拡張子を省略することができます。
3. NC79WAは環境変数LIB79で指定されたディレクトリ内にあるライブラリファイルnc79m8.lib（-fDP_offset_8[-fDPO8]オプション指定時）、または、nc79lib.lib（-fDP_offset_8[-fDPO8]オプション指定なし）をデフォルトでリンクします。（複数のライブラリファイルを指定した場合、nc79m8.libやnc79lib.libを参照する優先順位は最も低くなります。）

A.2.8 警告オプション

【表A.8】にnc79の言語仕様に関する記述の間違いに対して警告（ワーニングメッセージ）を出力する起動オプションを示します。

表A.8 警告オプション

オプション	短縮形	機能
-Wnon_prototype	-WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	-WUP	サポートしていない＃pragmaを使用した場合、警告を出します。
-Wno_stop	-WNS	エラーが発生してもコンパイル作業を停止しません。
-Wstdout	なし	エラーメッセージをホストマシンの標準出力（stdout）に出力します。
-Werror_file<file name>	-WEF	タグファイルを出力します。
-Wmake_tagfile	-WMT	error、およびwarningが発生した場合にタグファイルを出力します。
-Wstop_at_warning	-WSAW	ワーニング発生時にコンパイル処理を停止します。
-Wnesting_comment	-WNC	コメント中に/*を記述した場合に警告を出します。
-Wccom_max_warnings	-WCMW	ccom79の出力するワーニングの回数の上限を指定できます。
-Wall	なし	検出可能な警告をすべて表示します。
-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。
-Wuninitialize_variable	-WUV	初期化されていないAUTO変数に対してワーニングを出力します。
-Wlarge_to_small	-WLTS	大きいサイズから小さいサイズへの暗黙の転送に対してワーニングを出力します。
-Wno_used_argument	-WNUA	使用されていない引数に対してワーニングを出力します。

-Wnon_prototype　-WNP

警告オプション

[機能]　前もってプロトタイプ宣言がされていない関数を使用した場合、または関数のプロトタイプ宣言を行っていない場合に警告を出します。

[補足説明]　プロトタイプ宣言を行うことにより、関数引数をレジスタ渡しにすることができます。
レジスタ渡しにすることにより、速度向上、コードサイズ削減が期待できます。
また、プロトタイプ宣言を行うことにより、コンパイラが関数の引数を検査するようになります。このため、プログラムの信頼性向上を期待できます。
したがって、本オプションは、常に使用することを推奨します。

-Wunknown_pragma　-WUP

警告オプション

[機能]　サポートしていない＃pragma を使用した場合、警告を出します。

[補足説明]　デフォルトでは、サポートされていない未知の ”#pragma ”が使用されていても警告を出しません。
NCシリーズコンパイラのみを使用する場合、本オプションを使用することにより、”#pragma ”のスペルミスなどを発見することができます。
NCシリーズコンパイラのみを使用する場合は、本オプションを常に用いてコンパイルすることを推奨します。

-Wno_stop　　-WNS

警告オプション

[機　能]　エラーが発生してもコンパイル作業を停止しません。

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

[注意事項]　記述によるエラーが原因で System Error が発生する場合があります。その場合には、コンパイル作業が停止します。

-Wstdout

警告オプション

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

[実行例]

```
D:\work>nc79 -c -Wstdout sample.c > err.doc
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.


D:\work>type err.doc
[Error(ccom):sample.c,line 41] unknown variable port00
  ===>         port00 = 0x00;
Sorry, compilation terminated because of these errors in main().
sample.c

D:\work>
```

[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as79、ln79のエラー出力は、本オプションに関係なくWindows版（パソコン版）であれば標準出力に出力されます。

-Werror_file <*エラー出力ファイル名*>　　-WEF

警告オプション

[機能]　　エラーメッセージを指定したファイルに出力します。

[書式]　　nc79△-Werror_file△<*エラー出力ファイル名*>

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

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

-Wstop_at_warning　　-WSAW

警告オプション

[機能]　　ワーニング発生時にコンパイラの終了コードを ”10 ”で戻します。

[補足説明]　デフォルトでは、コンパイル時にワーニングが発生した場合、コンパイルの終了コードは、” 1 （正常終了）”で終了します。
本オプションは、makeユーティリティ等を用いている場合に、ワーニングが発生下場合にコンパイル処理を停止したいときに使用します。

-Wnesting_comment　-WNC

警告オプション

[機能]　コメント内に"/*"を記述している場合に警告を発生します。

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

-Wccom_max_warnings　-WCMW

警告オプション

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

[補足説明]　ワーニングの出力の上限回数は、0回以上で指定してください。また、指定回数の省略はできません。0 回を指定するとワーニング出力を完全に抑止します。

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

-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転送に対してワーニングを出力します。

-Wuninitialize_variable　-WUV

警告オプション

[機　能]　初期化されていないAUTO変数に対してワーニングを出力します。

-Wno_used_argument　-WNUA

警告オプション

[機　能]　使用されていない引数に対してワーニングを出力します。

-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。
オプション ”-Wnon_prototype(-WNP) ”、”-Wunknown_pragma(-WUP) ”と同等の警告を表示します。またこれらに加えて、以下の場合にも警告を表示します。

(1)if文、for文や、&&、|| 演算子の比較文に代入演算子 ”= ”を使用した場合。
例） if (i = 0)
func();

(2)代入演算子 ”= ”を間違って ”== ”と記述した場合。
例） i == 0;

(3)古い形式の関数定義を行った場合。
例） func(i)
int i ;
{
...
}

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

-Wmake_tagfile　-WMT

警告オプション

[機能]　error および warning が発生した場合に、file 単位に、タグファイル に生成メッセージの内容を出力します。

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

-Wno_warning_stdlib　-WNWS

警告オプション

[機能]　”-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体ス警告」を抑止します。

A.2.9 アセンブル / リンクオプション

【表A.9】にas79、およびln79のオプションを指定する起動オプションを示します。

表A.9 アセンブル / リンクオプション

オプション	機能
-as79△<*オプション*>	アセンブルコマンドas79のオプションを指定します。最大４個までオプションを指定することができます。2個以上のオプションを渡す場合は、”(ダブルクォーテーション)で囲んでください。
-ln79△<*オプション*>	リンクコマンドln79のオプションを指定します。最大４個までオプションを指定することができます。2個以上のオプションを渡す場合は、”(ダブルクォーテーション)で囲んでください。

-as79"*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as79のオプションを指定します。
2個以上のオプションを指定する場合は、" (ダブルクオーテーション)で囲んでください。

[書式]　　nc79△-as79△"*オプション1*△*オプション2*"△<C言語ソースファイル名>

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

```
D:\work>nc79 -v -as79 " -l -s " sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
D:\MTOOL\LIB79\cpp79 -DNC79 sample.c -o D:\MTOOL\TMP\sample.i

D:\MTOOL\LIB79\ccom79 D:\MTOOL\TMP\sample.i -o .\sample.a79
D:\MTOOL\BIN\as79 -. -N -l -s sample.a79 -o.

D:\MTOOL\BIN\ln79 sample.r79 -. -l nc79lib -o sample

DATA    0001012(003F4H) Byte(s)
ROMDATA  0000282(0011AH) Byte(s)
CODE    0008364(020ACH) Byte(s)

D:\work>dir /w sample.*
 ドライブ D のボリューム ラベルは WinNT です
 ボリューム シリアル番号は D812-16E7 です

 D:\work のディレクトリ

sample.c    SAMPLE.LST   sample.map   sample.x79
          4 個のファイル          41,854 バイト
                691,895,296 バイトの空き領域

D:\work>
```

[注意事項]　　as79の-.、-C、-M、-O、-P、-T、-Vおよび-Xオプションは指定しないでください。

参考としてAS79 のオプションを以下に示します。

ｵﾌﾟｼｮﾝ	機能
-.	画面へのすべてのメッセージ出力を抑止します。バッチファイル等でAS79を実行する場合において、画面に何も表示したくない時にお使いください。 ※オプション-as79で、このオプションを指定しないでください。
-A	ニーモニックオペランドを評価します。
-C	as79がmac79とasp79を起動した時のコマンドラインの内容を表示します。 ※オプション-as79で、このオプションを指定しないでください。
-D	シンボルに定数を設定します。
-F	..FILE展開のファイル名をソースファイル名に固定します。
-H	アセンブラリストファイルのヘッダ情報を出力しません。
-I	インクルードファイルの検索ディレクトリを指定します。
-L	アセンブラリストファイル(拡張子.lst)を生成します。
-M	構造化記述命令をバイト型でニーモニックに変換する(AS79のみ) ※オプション-as79で、このオプションを指定しないでください。
-N	高級言語のソース行情報を行いません。
-O	生成ファイルの出力先パスを指定します。パスにはディレクトリ又はドライブ名が指定できます。この指定がない場合、ソースファイルと同じパスに出力します。 ※オプション-as79、このオプションを指定しないでください。
-P	構造化記述命令を変換する(AS79のみ) ※オプション-as79このオプションを指定しないでください。
-S	ローカルシンボル情報を出力します。
-T	アセンブラエラータグファイルを生成します。 ※オプション-as79で、このオプションを指定しないでください。
-V	アセンブラシステムプログラムのバージョンを表示します。 ※オプション-as79で、このオプションを指定しないでください。
-X	タグファイルを引数として外部プログラムを起動します。 ※オプション-as79で、このオプションを指定しないでください。

※ NC79は、オプション-as79を用いてアセンブラの動作を指定することができますが、その場合はas79のオプション-.、-C、-M、-O、-P、-T、-Vおよび-Xを指定しないで下さい。

-ln79"*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ln79 のオプションを指定します。
リンクコマンドのオプションは、最大4個までオプションを指定することができます。2個以上のオプションを指定する場合は、" (ダブルクオーテーション) で囲んでください。

[書式]　nc79△-ln79△"*オプション1*△*オプション2*"△<C言語ソースファイル名>

[実行例]　以下の例では、マップファイルをコンパイル時に生成しています。

```
D:\work>nc79 -g -v -osample -ln79 -ms ncrt0.a79 sample.c
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 1998,1999,2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a79
D:\MTOOL\BIN\as79 -. -N --N ncrt0.a79 -o.

sample.c
D:\MTOOL\LIB79\cpp79 -DNC79 sample.c -o D:\MTOOL\NC79WA~1\TMP\sample.i

D:\MTOOL\LIB79\ccom79 D:\MTOOL\NC79WA~1\TMP\sample.i -o .\sample.a79 -g

D:\MTOOL\BIN\as79 -. -N --N sample.a79 -o.

D:\MTOOL\BIN\ln79 ncrt0.r79 sample.r79 -. -G -MS -ms -l nc79lib -o samp
le

DATA    0002576(00A10H) Byte(s)
ROMDATA  0000298(0012AH) Byte(s)
CODE    0010718(029DEH) Byte(s)

D:\work>dir /w sample.*
 ドライブ D のボリューム ラベルは WinNT です
 ボリューム シリアル番号は D812-16E7 です

 D:\work のディレクトリ

sample.c    sample.map   sample.x79
          3 個のファイル          53,163 バイト
                691,805,184 バイトの空き領域

D:\work>
```

[注意事項]　※ln79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。

参考としてAS79の製品パッケージに含まれているln79のオプションを以下に示します。

ｵﾌﾟｼｮﾝ	機　能
-.	画面へのすべてのメッセージ出力を抑止します。バッチファイル等でAS79を実行する場合において、画面に何も表示したくない時にお使いください。 ※オプション-ln79で、このオプションを指定しないでください。
-E	アブソリュートオブジェクトモジュールの開始アドレスを指定します。
-G	ソースデバッグ情報をアブソリュートファイルに出力します。 ※このオプションは指定しないでください。
-L	参照するライブラリファイル名を指定します。 ※オプション-ln79で、このオプションを指定しないでください。
-LD	参照するライブラリのディレクトリを指定します。
-LOC	セクションデータを指定アドレスから配置します。
-M	マップファイルを生成します。
-MS	シンボル情報を含むマップファイルを生成します。
-MSL	16文字を越えるシンボルをそのままマップファイルに出力します。
-NOSTOP	発生したエラー全てを画面に出力します。
-O	アブソリュートファイル名を指定します。 ※オプション-ln79で、このオプションを指定しないでください。
-ORDER	セクションのアドレス及び、配置順序を指定します。 ※オプション-ln79、このオプションを指定しないでください。
-T	リンクエラータグファイルを出力します。 ※オプション-ln79で、このオプションを指定しないでください。
-V	リンケージエディタのバージョンを表示します。 ※オプション-ln79で、このオプションを指定しないでください。
@file	コマンドファイルの記述内容に従ってリンケージエディタを実行します。 ※オプション-ln79で、このオプションを指定しないでください。

※ NC79は、オプション-ln79を用いてリンカの動作を指定することができますが、その場合はln79のオプション-.、-G、-O、-ORDER、-L、-T、-Vおよび@fileを指定しないで下さい。

A.2.10 その他のオプション

【表A.10】にnc79が生成するアセンブリ言語ソースファイルに対して処理を行うする起動オプションを示します。

表A.10 その他のオプション

オプション	短縮形	機能
-dsource	-dS	出力するアセンブリ言語ソースリスト中にC言語ソースリスティングをコメントとして出力します。 生成された ".a79 "を削除しません。
-dsource_in_list	-dSL	出力するリストファイル中にC言語ソースリスティングをコメントとして出力します。 ".lst "を生成します。

-dsource　-dS

コメントオプション

[機能]　出力するアセンブリ言語ソースリスト中に対応するC言語ソースリストをコメントとして出力します。

[補足説明]　-S オプションを使用した場合、自動的に、-dsource オプションが有効になります。また、生成された ".a79 "を削除しません。
本オプションは、アセンブルリストファイルに、C言語ソースリストを出力したいときに使用します。

-dsource_in_list　-dSL

コメントオプション

[機能]　出力するリストファイル中に対応するC言語ソースリストをコメントとして出力します。　".lst "ファイルが生成されます。

[補足説明]　-S オプションを使用した場合、自動的に、-dsource オプションが有効になります。

A.3　nc79起動オプションに関する注意事項

A.3.1　nc79起動オプションの記述に関する注意事項

nc79の起動オプションは、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　nc79の制御に関するオプションの優先順位

nc79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.r79）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.a79）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。このため、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

NC79は、7900シリーズを用いたシステムへの組み込みを容易にするために独自の拡張機能を追加しています。

付録Bでは、言語仕様に関する機能以外の拡張機能の使用方法を説明します。

表B.1 拡張機能(1)

拡張機能	機能の内容
near/far修飾子	1.データのアクセスに使用するアドレッシングモードに、アブソリュート系、またはアブソリュートロング系を使用するかを指定します。 near.....64Kバイト以内の領域(0H～0FFFFH)のアクセス far.....64Kバイトを超える領域（全メモリ）のアクセス 2.関数の呼び出しにJSR命令、またはJSRL命令を使用するかを指定します。 near.....JSR命令で呼び出します。 far.....JSRL命令で呼び出します。
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(“　LDA　A,DP:1 ”); 2.プロセッサステータスレジスタ中のM、Xフラグの切り替えをコンパイラに指示する事ができます（関数内のみ記述可能）。 記述例）asm(0,0);　　/* CLP M、X */ 3.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(“　LDA　A,DP:$$ ”,i); 記述例2) asm(“　LDA　A,DP:$$ ”,s.i); 記述例3) asm(“　LDA　A,DP:$$ ”,a[3]); 記述例4) asm(“　LDA　A,DP:$$ ”,$b); 記述例5) asm(“　LDA　A,DP:$@ ”,f); 4.最適化を部分的に抑止するときに使用するダミーの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.文字列中に日本語文字を使用することができます。 記述例）L ’漢字 ” 2.日本語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.コメント中に日本語文字を記述することができます。 記述例）/* 漢字 */ ※シフトJIS、およびEUCコードをサポートしています。
関数のﾃﾞﾌｫﾙﾄ引数宣言	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 i=1,char c=0); 記述例2)　extern int func(int i=a,char c=0); ※デフォルト値として変数を記述するときは、関数を宣言するよりも前にデフォルト値として使用する変数の宣言を行ってください。 ※デフォルト値は引数の後ろから順に埋めてください。
inline記憶クラスのサポート	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態定義を行ってください。

表B.2 拡張機能(2)

拡張機能	機能の内容
C++風コメント	C++ライクでのコメント ”// ”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7900シリーズハードウェア仕様を効率よく活かすための拡張機能を使用できます。
マクロアセンブラ関数	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 asl_b(char val); 記述例）int asl_w(int val);

B.1　near/far修飾子

7900シリーズは、バンク（64Kバイト）を境界としてデータの参照・配置、関数の呼び出し等により使用されるアドレッシングモードが変わります。NC79は、near/far修飾子によりアドレッシングモードの切り替えを制御できます。

B.1.1　near/far修飾子の概要

7700ファミリのアドレッシングモードは、以下の3種類に大別できます。

1.ダイレクト系アドレッシングモード
2.アブソリュート系アドレッシングモード
3.アブソリュートロング系アドレッシングモード

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 ..... アブソリュート系アドレッシングモード
  (アドレスを16ビット長で扱います)
- far修飾子 ..... アブソリュートロング系アドレッシングモード
  (アドレスを32ビット長で扱います)

NC79では、ダイレクトページレジスタ(DPR)をフレームポインタとして使用しています。このため、ダイレクト系のアドレッシングモードはnear・far修飾子で制御することができません。

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、NC79は属性を以下のように解釈します。

- 変数 ..... near属性
- 関数 ..... far属性

また、NC79はコンパイルドライバnc79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。

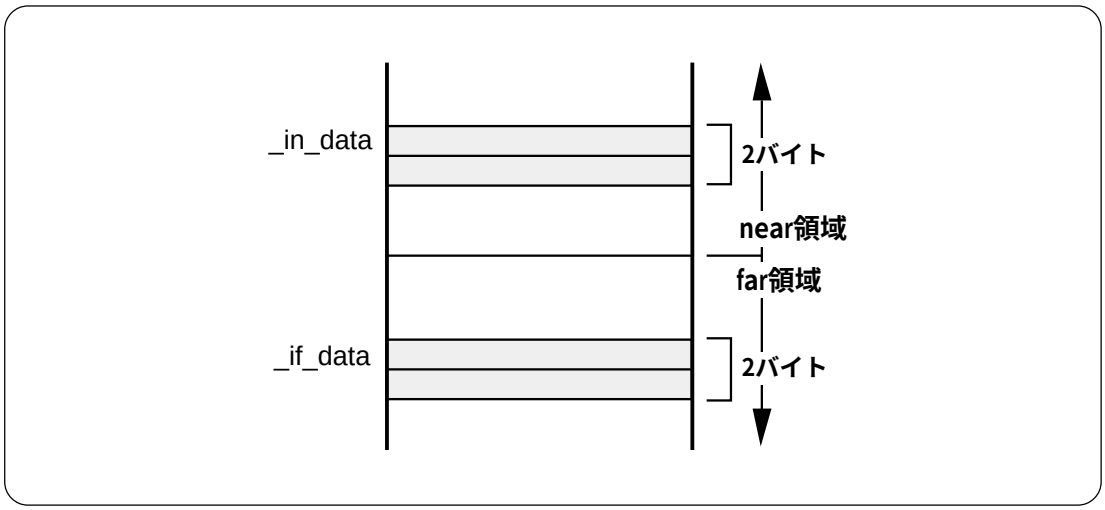
```
型指定子△near又はfar△変数;
```

図B.1 near/farr修飾子を付加した変数の宣言書式

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

```
int near in_data;
int far  if_data;

func()
{
      (以下省略)
       :
```

図B.2 変数の宣言例



図B.3 変数のメモリ配置

B.1.3 ポインタ変数の宣言書式

near・far修飾子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示します。アドレスを扱うポインタ型の変数の宣言例を【図B.4】に示します。

```
●例1
   int far *ptr1

●例2
   int * far ptr2
```

図B.4 ポインタ型変数の宣言例(1)

例1では、変数ptr1はfar領域にあるint型変数を指し示す32ビットサイズの変数です

が、その変数自身はnear領域に配置されます。

例2では、変数ptr2はnear領域にあるint型変数を指し示す16ビットサイズの変数ですが、その変数自身はfar領域に配置されます。

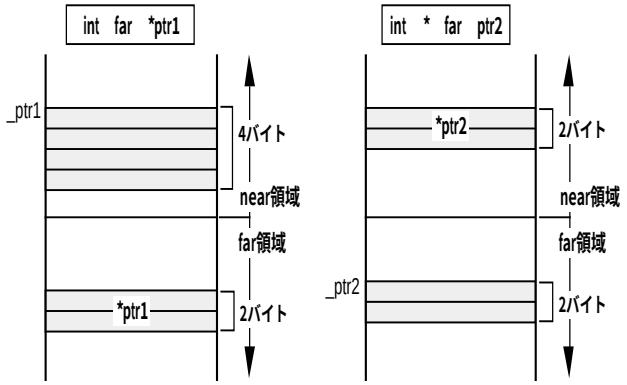
先にも説明したように、変数はnear・farの指定がない場合、near属性として扱われます。したがって、例1、例2はそれぞれ【図B.5】のように解釈されます。

●例1
```
int far * near ptr1
```

●例2
```
int near * far ptr2
```

図B.5 ポインタ型変数の宣言例(2)

例1、例2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

B.1.4 関数の宣言

【図B.7】に宣言時の書式を、【図B.8】に宣言例を示します。

```
型指定子△near又はfar△関数;
```

図B.7 near・far修飾子を付加した関数の書式

```
void near func1( void );
int far func2( int );

void near func1()
{
      :
    (省略)
      :
   func2( idata );
}

int far func2( x )
int x;
{
      :
    (省略)
      :
   return x;
}
```

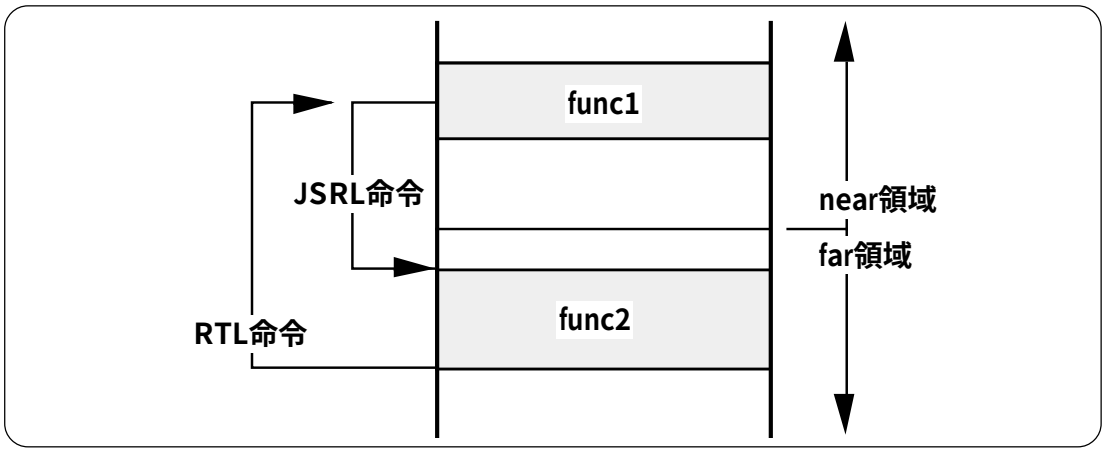
図B.8 変数・関数の宣言例

【図B.8】の例では、関数func2はnear領域以外のバンクに配置することを宣言しています。

near・far修飾子を付加して宣言された関数は、呼び出し・リターンのときに使用する命令が異なります。

●near属性の関数[呼び出し時] ..... JSR命令
　　[リターン時] ..... RTS命令
●far属性の関数 [呼び出し時] ..... JSRL命令
　　[リターン時] ..... RTL命令

【図B.8】に示しました例のメモリ配置と、関数呼び出し・リターンの関係を【図B.9】に示します。



図B.9 関数メモリ配置と呼び出し・リターン関係図

B.1.5 nc79の起動オプションによるnear/farの制御

NC79では、near/far属性を指定しない場合、関数はfar属性、変数(データ)はnear属性として扱われます。NC79の起動オプションには、変数のデフォルトを変更するオプションを用意しています(【表B.1】)。

表B.1 near/far属性を制御するnc79起動オプション

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。
-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)。
-fall_far	-fAF	デフォルトをすべてfar型にします。
-fnear_function	-fNF	関数のデフォルトをnearにします。near関数は呼び出すときにJSR命令で呼出し、RTS命令で戻ります。
-ffar_program_section	-fFPS	near関数・far関数をprogram_Fセクションに配置します。
-ffar_ROM_data	-fFROM	ROMデータのデフォルト属性をfarにします。
-ffar_RAM_data	-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。

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

図B10 nearからfarへの型変換

farに型変換される際、以下の規則でアドレスが拡張されます。

- ●記憶クラスautoの変数のアドレスは、0(ゼロ)を上位アドレスとして拡張
- ●上記以外のnear型アドレス又はポインタは、nc79の起動オプション-bank=で指定されたバンク値を上位アドレスとして拡張

B.1.7 farからnearへの型変換機能

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

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

main()
{
      n_ptr = f_ptr; /* farポインタをnearポインタに代入 */
        :
      (省略)
        :
      func ( f_pyr );/* 引数にnearポインタを持つ関数としてプロトタイプ宣言した */
                          /* farポインタを暗黙的にnearにキャスト */

      n_ptr = (near *)f_ptr;          /* farポインタを明示的にnearにキャスト */
}
```

図B.11 farからnearへの型変換

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

B.1.8 複数の宣言でnear/farの確定を行う機能

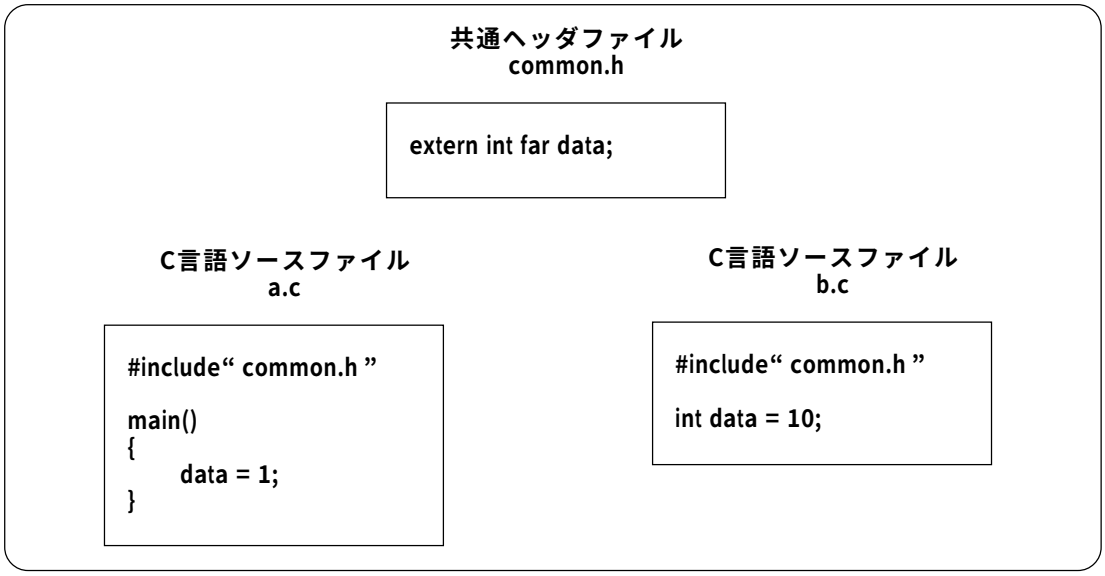
【図B.12】に示すように同一の変数に対して複数の宣言を行った場合、変数の型の情報が結合された型として解釈されます。

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

func()
{
        :
      (以下省略)
        :


この宣言は、以下の宣言として解釈されます。

extern int far idata = 10;

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

図B.12 関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。

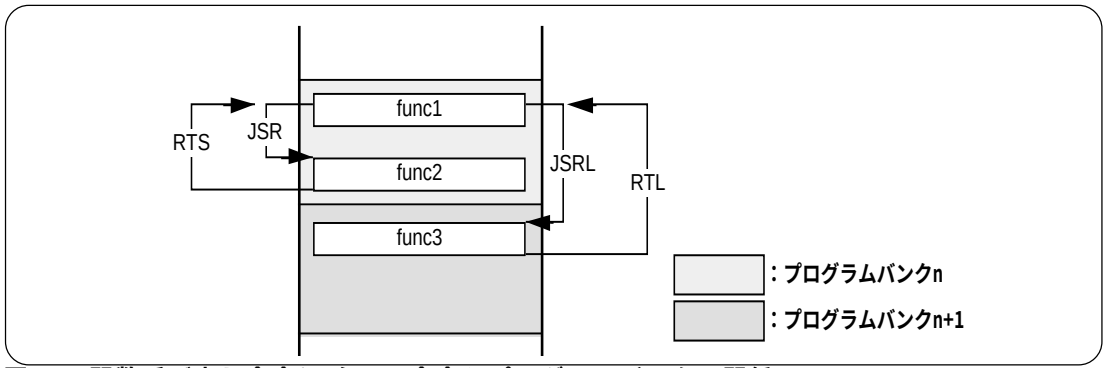


図B.13 共通ヘッダファイルの宣言例

B.1.9 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

関数に対する呼び出し命令、およびリターン命令は、プロトタイプ宣言で指定された関数のnear/far属性により決定されます（デフォルトでは、プロトタイプ宣言が無い場合、far属性として扱われます）。このため、関数が配置されている同一プログラムバンク内に配置されるnear関数を呼び出す時はJSR命令を、関数から戻るときはRTS命令を使用するコードを生成します。また、関数が配置されているプログラムバンクの内外を問わずに関数を呼び出す時はJSRL命令を使用し、関数から戻る時はRTL命令を使用します。【図B.14】に関数呼び出しに関わる命令とプログラムバンクの関係を示します。



図B.14 関数呼び出し命令/リターン命令とプログラムバンクの関係

※変数に対するnear/farの不整合の大半はリンク時に発見できますが、関数に対する整合性はコンパイル時及びリンク時には発見できません。呼び出し元の関数と呼び出される関数との間に不整合がある場合、プログラムは正常に動作しませんので注意してください。

このように、NC79では呼び出す関数がプログラムバンク内外により使用する命令が異なります。したがって、呼び出し元の関数と呼び出される関数との間に不整合がある場合、プログラムは正常に動作しません。また、この不整合はコンパイル、およびリンク時に判定されませんので、不整合が発生しないように十分注意してください。

b.関数のアドレス値の取り扱いに関する注意事項

関数、および変数の宣言にnear/far属性を指定しない場合、関数はfar属性、変数はnear属性として扱われます。また、near/far属性を指定しない関数のアドレスと通常の変数のアドレスを表すビット長は異なります。

●関数のアドレス……32ビット長
●変数のアドレス……16ビット長

このため、char＊型、またはvoid＊型の変数に対して関数のアドレスを代入することはできません（エラーとなります）。このような場合は、char far ＊型、またはvoid far ＊型と宣言することにより、変数を32ビット長で扱うようにしてください。

c.near/far修飾子の文法上の注意事項

near/far修飾子は、文法上const修飾子と全く同じに扱われます。このため、以下に示す記述はエラーになります。

```
int i, far j;   ← この記述はできません
```

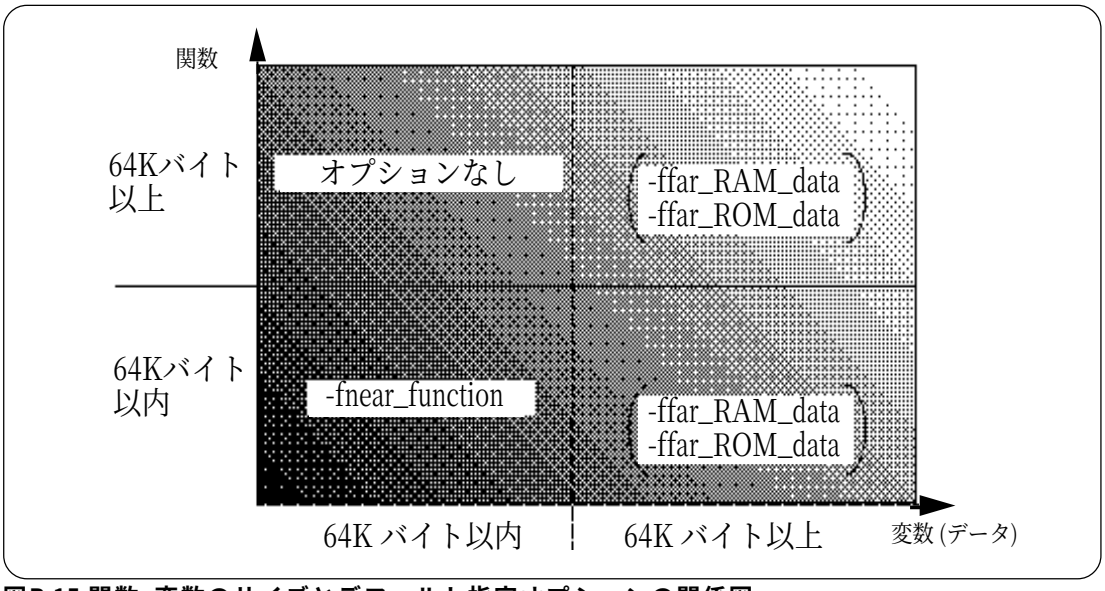
▼

```
int i;
int far j;
```

図B.15 変数の宣言例

d.near/far属性のデフォルト指定オプションに関する注意事項

NC79は起動オプションとして、関数、および変数のデフォルトを変更するオプションを用意しています。【図B.15】に関数・変数のサイズとデフォルト指定オプションの関係図を示します。



図B.15 関数・変数のサイズとデフォルト指定オプションの関係図

e.near領域のバンク値の変更に関する注意事項

nc79の起動オプションに-bank=を指定してnear領域のバンク値を変更した場合、以下の点に注意してください。

1. 記憶クラスがautoの変数のアドレスをnearポインタ変数に代入する、または関数の引数として指定しないでください。
2. fgetc、fputc等の入出力関数を使用できません。
3. 全てのソースファイルは-bank=で指定した同じバンク値でコンパイルしてください。
4. スタートアッププログラムncrt0.a79中野データバンクレジスタの初期化は、__DTに-bank=オプションで指定した同じバンク値を設定してください。

f.farのビットフィールド構造体に関する注意事項

以下の条件をすべて満たすビットフィールド構造体の記述に対して、リンク時にエラーとなることがあります。

- ●far属性のビットフィールド構造体
- ●複数のバンクにまたがって配置される
- ●.(ピリオド)演算子で参照するメンバに定数値を代入する

この条件をすべて満たすビットフィールド構造体の記述に対して、構造体の先頭が配置されたバンクと異なったバンクに配置されたメンバに対して定数値を代入すると、リンク時にエラーが発生します。この場合は、ビットフィールド構造体自身をバンクにまたがらない領域に配置するか、代入するメンバを構造体の先頭が配置されているバンク内に入るように並べ替えてください。

B.2　asm関数

NC79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)を記述することができます。asm関数では拡張機能として、m、xフラグの操作、C言語で記述したauto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.17】に示すようにasm("　　");のダブルクォーテーションの中にAS79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。なお、アセンブラの機能である構造化記述の言語仕様に準じた記述はできません。

```
#pragma ADDRESS ta0_int   75H
char          ta0_int;

void func
{
       :
      (省略)
       :
      ta0_int = 0x07;
      asm("          CLI");
}
```

図B.17 asm関数の記述例(1)

また、【図B.18】に示す記述文字列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最適化処理を部分的に抑止することができます。

```
asm();
```

図B.18 asm関数の記述例(2)

NC79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の変数のDPオフセット値をC言語の変数名で指定できます。
- ●データ長選択フラグ(m)とインデックスレジスタ長選択フラグ(x)をasm(MFLAG, XFLAG)の書式で記述できます。
- ●C言語プログラムの記憶クラスregisterの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の変数のシンボル名をC言語の変数名で指定できます。

※asm("　　sem");等の書式を用いてm、xフラグを操作しないで、asm(MFLAG, XFLAG)の書式を用いてください。また、asm関数内でbra等の分岐命令を記述した場合は、分岐以降の生成コードを確認してください。

B.2.2　m、x フラグの切り替え機能

プロセッサステータスレジスタ内のデータ長選択フラグ(m)とインデックスレジスタ長選択フラグ(x)の切り替えを、【図B.19】に示す形式で記述することができます。

```
asm(MFLAG, XFLAG);

MFLAG:データ長選択フラグの状態
XFLAG:インデックスレジスタ長選択フラグの状態
    状態:
        0 :    フラグのクリアを意味します
        1 :    フラグのセットを意味します
        2 :    フラグの切り替えを一切行わないことを意味します
```

図B.19 m、xフラグの切り換え記述形式

m及びxフラグの状態を切り換えるときは、必ず【図B.19】に示した書式で記述してください。また、asm関数はC言語ソースプログラムの関数外に記述することができますが、この記述は必ず関数内に記述してください。m、xフラグの切り換え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 near func()
    {
        asm( 0, 0 );
            :
          (省略)
            :
        asm( 1, 1 );
            :
          (省略)
            :
        asm( 2, 0 );
            :
          (省略)
            :
        asm( 0, 2 );
            :
          (省略)
            :

●アセンブリ言語ソースファイル(コンパイル結果)
    ;### C_SRC :      asm( 0, 0 );
        clp m,x
        :
      (省略)
        :

    ;### C_SRC :      asm( 1, 1 );
        sep m,x
        :
      (省略)
        :

    ;### C_SRC :      asm( 2, 0 );
        clp x
        :
      (省略)
        :

    ;### C_SRC :      asm( 0, 2 );
        clm
```

図B.20 m、xフラグの切り換え例

B.2.3　auto変数のDPオフセット値の指定

C言語で記述した記憶クラスautoの変数（引数を含む）は、ダイレクトページレジスタ(DP)に対するオフセット値で参照・配置されます。
【図B.21】に示す書式で記述することにより、asm関数中でauto変数を使用することができます。

```
asm("      オペコード   A, DP0:$$", auto変数名);          ←デフォルト
asm("      オペコード   A, DP:$$", auto変数名);           ←-fDPO8オプション指定時
```

図B.21 DP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1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

```
void near func()
{
      int idata;
      int a[3];
      struct TAG{
               int  i;
              int  k;
      } s;
              :
      asm("      LDA.W   A,DP:$$", idata);
              :
      asm("      LDA.W   A,DP:$$", a[2]);
              :
      asm("      LDA.W   A,DP:$$", s.i);
              :
        (以下省略)
              :
}
```

図B.22 DP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.23】に示します。

●C言語ソースファイル

```
void near func()
{
        int idata;
        asm("  LDA.W   A,DP:$$", idata);
        asm("  CMP.W   A, #1000H ");
                :
          (以下省略)
                :
}
```

●アセンブリ言語ファイル（コンパイル結果）

```
;###    FUNCTION func
;###       FRAME AUTO  (   idata)  size   2,   offset 1
                :
           (省略)
                :
;### C_SRC :       asm("      LDA.W   A,DP:$$", idata);

;#### ASM START
        LDA.W   A,DP:1
        .cline  6
;### C_SRC :       asm("      CMP.W   A, #1000H ");
        CMP.W   A, #1000H

;#### ASM END
                :
          (以下省略)
                :
```

図B.23 auto変数の参照例

なお、auto変数をasm関数内で参照する場合は、必ずこの機能を使用してください。この機能を使用しなかった場合、auto変数の領域確保が行われないことがあります。また、将来のバージョンアップ時に正常に動作しなくなる等の不良動作が発生する可能性があります。

また、【図B.24】に示す書式で記述することにより、asm関数中でauto変数のビットフィールドを使用することができます。

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

図B.24 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.25】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("oramb  $b",s.bit1);
}
```

図B.25 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.26】に示します。auto領域のビットフィールドを参照する場合には、ビット操作命令で参照可能な範囲に配置していることを確認してください。

●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(" oramb $b",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    (      s)     size   1,      offset 1
      ;## #   ARG Size(0)     Auto Size(1)    Context Size(5)

            .section        program_F
            ._file  'bit.c'
            ._line  3
            .DT     __DT
            .DP0    OFF
            .glb    _func
      _func:
            phd     0
            pht
            tsd
            ._line  10
      ;## ASM START
            oramb #02H,DP0:1        ;  s
      ;## ASM END
            ._line  11
            adds    #0001H
            rtld    0
```

図B.26 auto領域のビットフィールドの参照例

B.2.4　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registerの変数名（引数を含む）は、レジスタ上で管理されます。

【図B.27】に示す書式で記述することにより、asm関数中でレジスタ変数を使用することができます。

```
asm("　　オペコード　$$,#00H",レジスタ名);
```

図B.27 レジスタ変数の記述書式

NC79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。したがって、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。

この記述形式で指定できる変数名は1つです。レジスタ変数の参照例とコンパイル結果を【図B.28】に示します。

●C言語ソースファイル

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

        asm("lda.W   $$,#0000H", 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(3)

        .section        program_F
        ._file  'reg.c'
        ._line  3
        .DT     __DT
        .DP0    OFF
        .glb    _func
_func:
        ._line  4
        lda.W   A,#0001H ;  i
        ._line  6
;## ASM START
        lda.w   A,#0000H        ;  i
        ;## ASM END
```

図B.28 register変数の参照例

B.2.5　global、extern変数、および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.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の変数を使用することができます。

```
asm("       オペコード   A,$$",extern変数名);
asm("       オペコード   B,$$",static変数名);
```

図B.29 シンボル名指定の記述形式

この記述形式で指定できる変数名は1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

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

void func()
{
              asm(" lda     A,$$",idata);
              asm(" lda     A,$$",a[2]);
              asm(" lda     A,$$",s.i);
                    :
               （以下省略）
                    :
}
```

図B.30 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.31】に示します。

```
●C言語ソースファイル
    extern int      ext_val;              ←extern変数

    func()
    {
            static  int s_val;            ←static変数

            asm("lda      A,$$", ext_val);
            asm("lda      B,$$, s_val;
    }

●アセンブリ言語ソースファイル（コンパイル結果）
            .glb    _func
    _func:
            ._line  8
    ;## ASM START
            lda     A,_ext_val
            ._line  9
            lda     B,___S0_s_val
    ;## ASM END
            ._line  10
            rtl
            .glb    _ext_val

          .SECTION      bss_NE,DATA
    ___S0_s_val:         ;### C's name is s_val
            .blkb   2
            .END
```

図B.31 extern変数及びstatic変数の参照例

また、【図B.32】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変数のビットフィールドを使用することができます。

extern変数及びstatic変数のビットフィールドを参照する場合には、ビット操作命令で参照可能な範囲に配置していることをユーザ側で確認してください。

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

図B.32 ビットフィールド名指定の記述書式

この記述形式で指定できる変数名は1つです。【図B.33】に参照例とコンパイル結果を示します。

●C言語ソースファイル

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

void
func(void)
{
        asm(" oramb $b", 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
;## #   ARG Size(0)     Auto Size(0)    Context Size(3)

        .section        program_F
        ._file  'kk.c'
        ._line  10
        .DT     __DT
        .DP0    OFF
        .glb    _func
_func:
        ._line  11
;## ASM START
        oramb #02H,DT:_s
;## ASM END
        ._line  12
        rtl

        .SECTION        bss_NO,DATA
        .glb    _s
_s:
        .blkb   1
```

図B.33 シンボルに対するビットフィールドの参照例

B.2.6　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、register変数、extern変数、static変数）に依存することなく、asm関数中で使用することができます。
【図B.34】に示す書式で記述することにより、asm関数中でC言語で記述した変数を使用することができます。※1

```
asm("　　オペコード　　A,$@", 変数名);
```

図B.34　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.35】に示します。

●C言語ソースファイル

```
extern int     e_val;        ←extern変数

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

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

●アセンブリ言語ソースファイル(コンパイル結果)

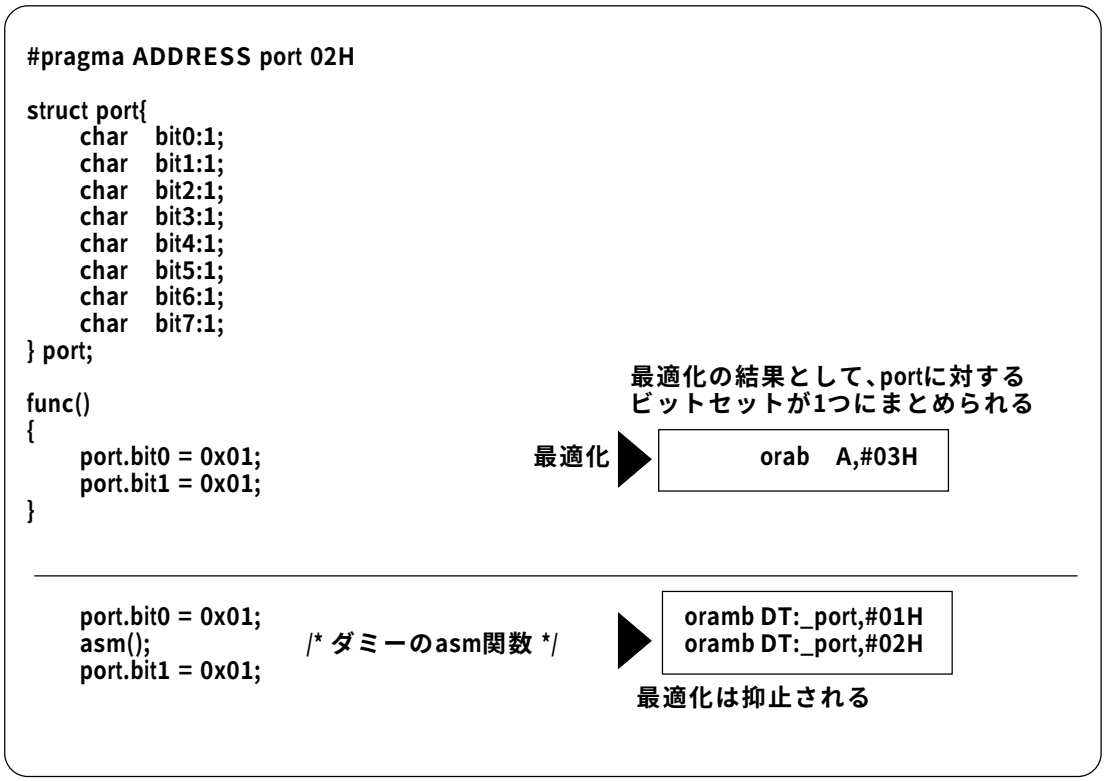
```
     .glb    _func
    _func:
     phd
     pha
     pha
     tsd
     ._line 9
    ;## # C_SRC :            asm("   movm.w   $@,#1", e_val);
    ;## ASM START
       movm.w   DT:_e_val,#1          ←-------------------------   extern変数の参照
     ._line 10
    ;## # C_SRC :            asm("   movm.w   $@,#2", f_val);
       movm.w   DP0:3,#2              ←-------------------------   auto変数の参照
     ._line 11
    ;## # C_SRC :            asm("   movm.w   $@,#3", r_val);
       movm.w   DP0:1,#3              ←-------------------------   register変数の参照
     ._line 12
    ;## # C_SRC :            asm("   movm.w   $@,#4", s_val);
       movm.w   DT:___S0_s_val,#4     ←-------------------------   static変数の参照
    ;## ASM END
```

図B.35　各記憶クラスの変数の参照例

※1 register変数を使用している場合、register修飾子を有効にするため、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.2.7　最適化の部分的な抑止方法

【図B.36】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、部分的に最適化を抑止することができます。



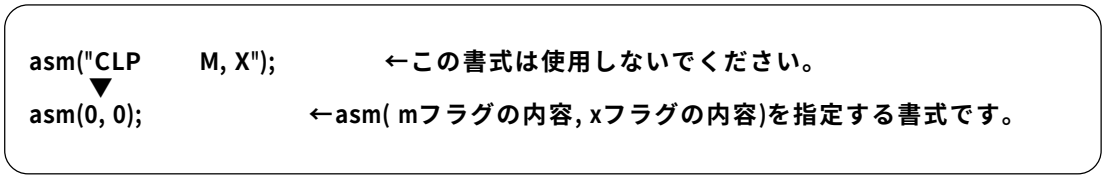
図B.36 ダミーのasm関数による最適化の抑止例

B.2.8　asm関数に関する注意事項

a.　asm関数の拡張機能

asm関数※1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)asm関数では、CLM、CLP、SEM、SEP等の命令を用いてプロセッサステータスレジスタ中のm、xフラグを変更しないでください。m、xフラグの変更は、【図B.37】に示す書式で記述してください。



図B.37 asm関数の記述例(1)

(2)記憶クラスがautoの変数、引き数もしくは1ビットのビットフィールドをダイレクトページレジスタ(DPR)からのオフセット値を用いて指定しないでください。この場合は、【図B.38】に示す書式で記述してください。

※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。

```
asm("LDA      A,DP:$$", i);     ←記憶クラスautoの変数を参照する書式です。
asm("SEB      $b",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.38 asm関数の記述例(2)

(3)NC79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変数を記述するときは、【図B.39】に示す書式で記述してください。

```
asm("LDAB   $$,#00H",i);   ←レジスタ変数を参照する書式です。
```

図B.39 asm関数の記述例(3)

b.　asm関数の使用について

(1)データバンクレジスタ(DT)の内容をasm関数で変更した場合、連続するasm関数の最後で【図B.40】に示すDTを元に戻す記述を行ってください。

```
asm("   LDT   #1");                    ←DT変更
asm("   .DT   1");
asm("   LDA   A.DT:TABLE");
     :
    (省略)
     :
asm("   LDT   #__DT");          ←DTを元に戻す
```

図B.40 変更したデータバンクレジスタの復帰方法

(2)ダイレクトページレジスタ(DPR)は、スタックフレームポインタとして使用しますので、asm関数で変更しないでください。

c. ラベルの記述に関する注意事項

NC79が生成するアセンブルソースファイルでは、【図B.41】に示す形式の内部ラベルが出力されます。したがって、asm関数を用いてこの規定と重複する可能性のあるラベルを記述しないでください。

```
●アルファベットの大文字１文字と数字で構成されるラベル名
例）A1:
    C9877:
●２文字以上の_(アンダースコア)で始まるラベル名
例）__LABEL:
    ___START:
```

図B.41 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

NC79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。NC77/NC79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。

- ●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.42】に示します。

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

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

図B.42 環境変数NCKINとNCKOUTの設定例

NC79は、入力漢字コードをプリプロセッサcpp79で処理しますcpp79でEUCコードに変換し、その後コンパイラccom79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するときの書式を【図B.43】に示します。

```
L"漢字文字列"
```

図B.43 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。

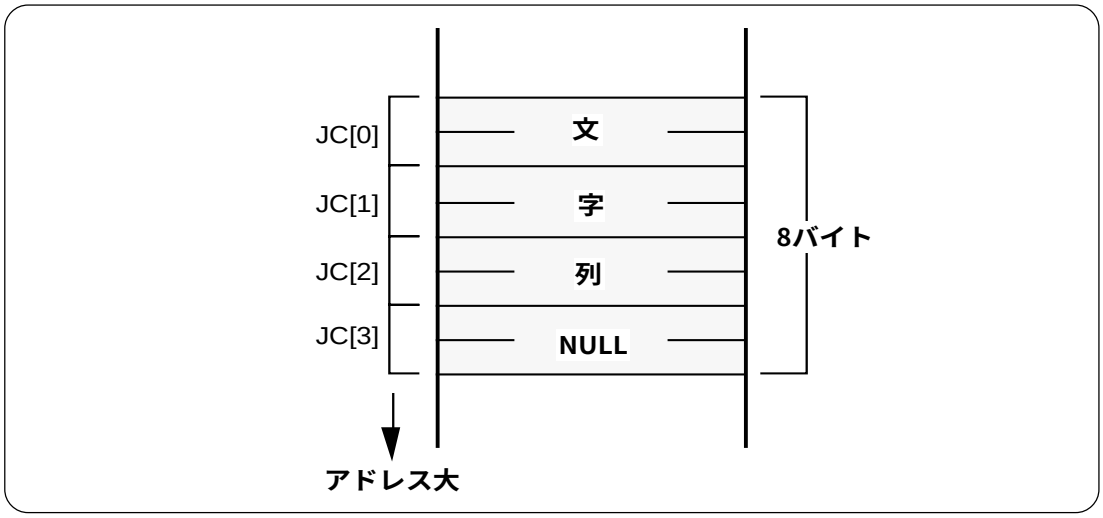
2バイト文字として扱う場合は、文字列の先頭にLを付加してwchar_t型へのポインタ型として使用します。wchar_t型は、標準ヘッダファイルstdlib.hの中でunsigned short型にtypedefしています。

【図B.44】に日本語の文字列の記述例を示します。

```
#include <stdlib.h>

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

      (以下省略)
        :
```

図B.44 日本語文字列の記述例

【図B.44】中の①の初期化された文字列のメモリ配置を【図B.45】に示します。



図B.45 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するときの書式を【図B.46】に示します。

```
L'漢'
```

図B.46 文字列中の漢字コード記述書式

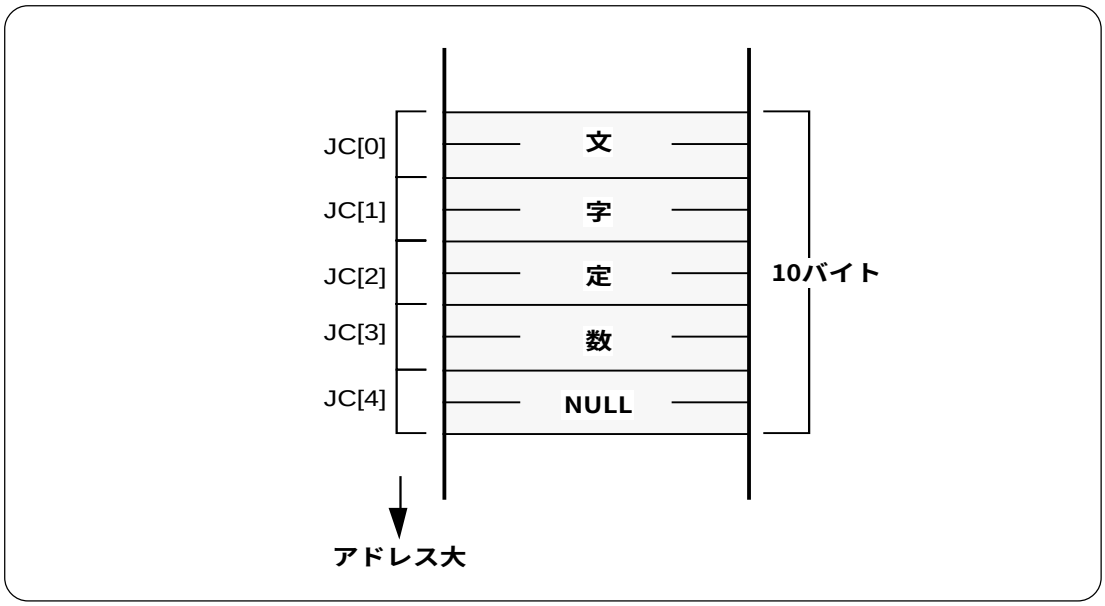
文字列と同様に、文字定数の前にLを付加した場合、wchar_t型として扱われます。文字定数として'文字'のように複数の文字を記述した場合は、最初の文字「文」のみが文字定数として扱われます。

【図B.47】に日本語の文字定数の記述例を示します。

```
#include <stdlib.h>

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

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

      (以下省略)
         :
```

図B.47 日本語文字定数の記述例

【図B.47】中の文字定数を代入した配列のメモリ配置を【B.48】に示します。



図B.48 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

NC79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。

B.4.1　関数のデフォルト引数宣言の概要

NC79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.49】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.49 関数のデフォルト引数宣言書式

関数のデフォルト引数の宣言例を【図B.50】に、コンパイル結果を【図B.51】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一引数：1、
                                        第二引数：2に宣言しています
void main( void )
{
      func();                         ← 実引数は第一引数：1、第二引数：2になります。
      func( 3 );                      ← 実引数は第一引数：3、第二引数：2になります。
      func( 3,5 );                    ← 実引数は第一引数：3、第二引数：5になります。
}
```

図B.50 関数のデフォルト引数宣言の例

```
_main:
      ._line  5
;## # C_SRC :           func();
      pea     #0002H                  ← 第二引数：2
      lda.W   A,#0001H                ← 第一引数：1
      jsrl    $func
      plx
      ._line  6
;## # C_SRC :           func( 3 );
      pea     #0002H                  ← 第二引数：2
      lda.W   A,#0003H                ← 第一引数：3
      jsrl    $func
      plx
      ._line  7
;## # C_SRC :           func( 3,5 );
      pea     #0005H                  ← 第二引数：5
      lda.W   A,#0003H                ← 第一引数：3
      jsrl    $func
      plx
      ._line  8
;## # C_SRC :     }
      rtl

※NC79では、関数の引数は、宣言した引数の最後から積んでいきます。
※本例では、第一引数がレジスタ渡しで処理しています。
```

図B.51 関数のデフォルト引数宣言の例（コンパイル結果）

関数の引数には変数を記述することができます。関数のデフォルト引数の変数指定例を【図B.52】に、コンパイル結果を【図B.53】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います
}
```

図B.52 関数のデフォルト引数の変数指定例

```
_main:
      ._line  6
;## # C_SRC :            func();
      lda     A,DT:_sym          ← 変数(sym)を引数として関数呼び出しを行います
      jsrl    $func
```

図B.53 関数のデフォルト引数の変数指定例（コンパイル結果）

関数のデフォルト引数の宣言を行う時には、以下の点に注意してください。

(1)複数の引数にデフォルト値を指定する時

関数の引数が複数ある場合にデフォルト値を指定する時には、必ず引数の後ろから埋めてください。【図B.54】に不適切な記述例を示します。

```
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.54 プロトタイプ宣言の記述例

(2)変数をデフォルト値として指定する時

デフォルト値として変数を指定する場合には、指定する変数の宣言を行った後に関数のプロトタイプ宣言を行ってください。もし、関数のプロトタイプ宣言を行った時点で宣言していない変数を関数のデフォルト値に指定した場合、エラーとして処理します。

B.4.3 関数のデフォルト引数宣言の規定事項

関数のデフォルト引数宣言を行なう場合には、以下の規定を遵守してください。

- ●関数の引数が複数ある場合にデフォルト引数を指定する時には、必ず引数の後ろから順に埋めてください。
- ●デフォルト引数に変数を指定することができます。ただし、変数を指定する場合には、指定する変数の宣言を行なった後に、関数のデフォルト引数宣言を行なってください。もし、関数の関数のデフォルト引数宣言を行なった時点で宣言の行なわれていない変数を引数のデフォルト値に指定した場合には、コンパイル時にエラーとなります。

B.5 inline関数宣言

NC79では、C++ライクに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マクロとして定義されます。

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.55】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.56】に、コンパイル結果を【図B.57】に示します。

```
extern int s;

inline int func( int i )
{
	return i+1;
}

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

図B.56 inline記憶クラスの宣言例

```
      .section        program_F
;## # C_SRC :     {
      .DT     __DT
      .DP0    OFF
_func         .MACRO
;## # C_SRC :             return i+1;
      inc     A
      .ENDM

;## #         FUNCTION main
;## #         ARG Size(0)  Auto Size(0)  Context Size(3)

      ._file   'smp.c'
      ._line  10
;## # C_SRC :     {
      .DT     __DT
      .DP0    OFF
      .glb    _main
_main:
      ._line  11
;## # C_SRC :             func( s );
      lda     A,DT:_s
      _func
      ._line  12
;## # C_SRC :     }
      rtl
      .glb    _s
      .END
```

図B.57 inline記憶クラスの宣言例(コンパイル結果)

B.5.3 inline記憶クラスの規定事項

inline記憶クラス指定時には、以下の点に注意してください。

(1)インライン関数のネストについて

　インライン関数からインライン関数を呼び出すことは可能です。ただし、インライン関数はマクロを使用しているため、アセンブラのマクロのネスト可能段数によって制限を受けます。マクロのネスト可能段数についての詳細はAS79ユーザーズマニュアルを参照してください。また、インライン関数の再帰呼び出しはできません。

(2)インライン関数の定義について

　関数に対してinline記憶クラスを指定する時には、宣言だけでなく実体定義を行ってください。実体定義は必ず同一ファイル内に記述してください。【図B.58】の記述はNC79ではエラーとして処理します。

```
inline void func( int i );

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

【エラーメッセージ】

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

図B.58 インライン関数の不適切な記述例(1)

また、ある関数を通常の関数として使用した後にその関数をインライン関数として定義した時には、inlineの指定は無効になり、すべてstaticな関数として扱います。この時NC79は警告を発します（【B.59】）。

```
int func( int i );

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

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

【ワーニングメッセージ】

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

図B.59 インライン関数の不適切な記述例(2)

(3)インライン関数のアドレスについて

インライン関数はマクロ定義ですので、関数自身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にはエラーになります（【B.60】）。

```
int func( int i );

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):smp.c,line 12] can't get inline function's address by '&' operator
   ===>        f = &func;
Sorry, compilation terminated because of these errors in main().
```

図B.60 インライン関数の不適切な記述例(3)

(4)staticデータの宣言

インライン関数内でstaticデータを宣言した場合、宣言したstaticデータの実体はファイル単位で確保されます。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アクセスする領域が異なります。インライン関数内で使用するstaticデータは関数外で宣言してください。NC79では、インライン関数内でstatic宣言をした場合警告を発します。また、インライン関数内のstatic宣言は推奨しません(【B.61】)。

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

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

【ワーニングメッセージ】

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

図B.61 インライン関数の不適切な記述例(4)

(5)デバッグ情報について

NC79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うことになります。

B.6 コメント ’// ’の概要

NC79では、"/*"と"*/"で記述するコメント以外にC++言語ライクのコメント"//"を記述することができます。

B.6.1 コメント ’// ’の概要

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

B.6.2 コメント ’// ’の書式

行中で"//"を記述した場合、同一行以降の記述はコメントとして扱われます。【図B.62】に書式を示します。

```
//コメント
```

図B.62 コメントの記述書式

コメントの記述例を【図B.63】に示します。

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

図B.63 コメントの記述例

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.メモリに関する拡張機能の使用方法

表B.4 メモリ配置に関する拡張機能

拡張機能	機能の内容
#pragma SECTION	NC79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 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 i
#pragma STRUCT	1.指定したタグを持つ構造体のパックを禁止します。 記述形式)#pragma STRUCT 構造体のタグ名 unpack 記述例)#pragma STRUCT TAG1 unpack 2.指定したタグを持つ構造体のメンバの並べ替えを行い、偶数サイズのメンバを先に配置します。 記述形式)#pragma STRUCT 構造体のタグ名 arrange 記述例)#pragma STRUCT TAG1 arrange

b.組み込み機器に関する拡張機能の使用方法

表B.5 組み込み機器に関する拡張機能の使用方法

拡張機能	機能の内容
#pragma INTERRUPT (#pragma INTF)	C言語で記述した割り込み処理関数を宣言します。この宣言により、関数の出入り口で割り込み処理関数の手続きを行うコードを生成します。 記述形式)#pragma INTERRUPT 割り込み処理関数名 記述例)#pragma INTERRUPT int_func
#pragma ADDRESS (#pragma EQU)	変数の絶対アドレスを指定します。near指定された変数は、バンク内アドレスを指定します。 記述形式)#pragma ADDRESS 変数名 絶対アドレス 記述例)#pragma ADDRESS port0 2H
#pragma PARAMETER	アセンブラ関数を呼び出す際に、その引数をレジスタを介して渡すことを宣言します。 記述形式)#pragma PARAMETER 関数名(レジスタ名) 記述例)#pragma PARAMETER asm_func(X, Y)

c.MR79に関する拡張機能の使用方法

表B.6 MR79に関する拡張機能の使用方法

拡張機能	機能の内容
#pragma INTHANDLER #pragma HANDLER	MR79の割り込みハンドラ関数名を宣言します。 記述形式1)#pragma INTHANDLER 関数名 記述形式2)#pragma HANDLER 関数名 記述例)#pragma INTHANDLER int_func
#pragma ALMHANDLER	MR79のアラームハンドラ関数名を宣言します。 記述形式)#pragma ALMHANDLER 関数名 記述例)#pragma ALMHANDLER alm_func
#pragma CYCHANDLER	MR79の周期起動ハンドラ関数名を宣言します。 記述形式)#pragma CYCHANDLER 関数名 記述例)#pragma CYCHANDLER cyc_func
#pragma TASK	MR79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.DTレジスタの操作に関する拡張機能の使用方法

表B.7 DTレジスタの操作に関する拡張機能の使用方法

拡張機能	機能の内容
#pragma LOADDT	コンパイル時のデータバンクレジスタ(DT)の値を関数の先頭でロードする関数を指定します。 記述形式)#pragma LOADDT 関数名 記述例)#pragma LOADDT func

e.DPレジスタの操作に関する拡張機能の使用方法

表B.8 DPレジスタの操作に関する拡張機能の使用方法

拡張機能	機能の内容
#pragma DP[n]DATA (nは1～3)	外部変数を指定したDPレジスタに割り当てます。 記述形式)#pragma DP[n]DATA 外部変数名 記述例)#pragma DP1DATA GLDATA1 記述例)#pragma DP2DATA GLDATA2 記述例)#pragma DP3DATA GLDATA3

f.関数呼び出しに関する拡張機能の使用方法

表B.9 関数呼び出しに関する拡張機能の使用方法

拡張機能	機能の内容
#pragma M1FUNCTION	Mフラグを1の状態のまま呼び出す関数を指定します。 記述形式)#pragma M1FUNCTION 関数名 記述例)#pragma M1FUNCTION func
#pragma MX1FUNCTION	関数の出入口におけるMXフラグの状態がセットされていることを前提としたコードを生成します。 記述形式)#pragma MX1FUNCTION 関数名 記述例)#pragma MX1FUNCTION func

g.その他の拡張機能の使用方法

表B.10 その他の拡張機能

拡張機能	機能の内容
#pragma ASM #pragma ENDASM	アセンブリ言語で記述を行う領域を指定します。 記述形式)#pragma ASM #pragma ENDASM 記述例) #pragma ASM lda.W A,#0000H tad #pragma ENDASM
#pragma PAGE	アセンブラリスティングファイルの改ページの指定を行います。 記述形式) #pragma PAGE 記述例)#pragma PAGE
#pragma TBLJMPOFF	指定された関数も対しては-fswtich_tableオプション指定時でもテーブルジャンプのコードを生成しません。

B.7.2 メモリ配置に関する拡張機能の使用方法

NC79は、次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 SECTION

セクション名変更機能

[機能]　NC79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。
この宣言をデータ(data及びbss)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すべ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。

[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 near 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_N             ← pro1セクションに配置
          .DT __DT
          .DP OFF
          .func   _func
          .pub    _func
     _func:
```

#pragma ROM

romセクションへの配置機能

[機能]　指定データ(変数)をromセクションに配置します。

[書式]　#pragma ROM△変数名

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

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

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

[使用例]

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

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


[アセンブリ言語ソースプログラム]
    .section    rom_NE,ROMDATA
    ;### C's name is i
    ___S0_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
          :
        (以下省略)
```

#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]　NC79では、構造体はパックされます。例えば、【図B.64】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。

```
strucr s
{
        int     i;
        char    c;
        int     j;
};
```

メンバ名	データ型	サイズ	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.64 構造体メンバの配置例(1)

①パックの禁止
　NC79では、拡張機能として構造体メンバの配置を制御することができます【図B.65】に示した構造体を#pragma STRUCTでパックを禁止したときのメンバの配置例を【図B.66】に示します。

```
strucr s
{
        int     i;
        char    c;
        int     j;
};
```

メンバ名	データ型	サイズ	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
ﾊﾟﾃﾞｨﾝｸﾞ	(char)	8ビット	--

図B.65 構造体メンバの配置例(2)

【図B.65】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。

②メンバ配置の並び換え
NC79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.64】に示した構造体を#pragma STRUCTで配置を並び替えたときの配置例を【図B.66】に示します。

```
strucr s
{
        int     i;
        char    c;
        int     j;
};
```

メンバ名	データ型	サイズ	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.66 構造体メンバの配置例(3)

[解説]　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。

```
#pragma STRUCT TAG unpack

struct TAG{
      int      i;
      char    c;
} s1;
```

図B.67 #pragma STRUCT宣言の使用例

B.7.3 組み込み機器に関する拡張機能の使用方法

NC79は、次に示す組み込み機器に関する拡張機能を持っています。

#pragma INTERRUPT(#pragma INTF)

割り込み関数の記述機能

[機能]　割り込み処理関数の宣言を行います。

[書式]　#pragma INTERRUPT△[/E]△割り込み処理関数名

[解説]　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、7900シリーズのすべてのレジスタ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、RTI命令でリターンします。

また、宣言時に以下のスイッチを指定できます。

[/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。

#pragma INTERRUPTにより宣言された関数は、interruptセクションに配置されます。

[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により宣言された関数に対してnear/far宣言を行わないでください。
4.#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。
5.関数名以外を指定した場合、何の効果も及ぼしません。
6.#pragma INTERRUPT宣言の二重定義はエラーとなりません。

[使用例]

```
#pragma INTERRUPT i_func


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

[備考]　C77 V.2.10以前のバージョンとの互換性のために#pragma INTFの記述もできます。

#pragma ADDRESS(#pragma EQU)

入出力変数の絶対アドレス指定機能

[機能]　変数の絶対アドレスを指定します。near指定された変数は、バンク内アドレスを指定します。

[書式]　#pragma ADDRESS△変数名△絶対アドレス

[解説]　この宣言により指定した絶対アドレスは文字列としてアセンブラファイルに展開され、疑似命令.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、エラーとなります。
7.#pragma ADDRESS宣言で指定された変数のアドレス値は絶対番地として扱われます。

[使用例]

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

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

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

[注意点]　NC79WA V.3.20 Release1より#pragma ADDRESSで宣言された変数のアドレス値は絶対番地として扱うようになりました。
このため。変数のアドレス値をバンク値からの相対番地で記述している場合には絶対番地に修正してください。

[規定]　DT値に依存しない絶対アドレスを指定するときは、変数の宣言時にfarを指定してください。

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

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

```
#pragma ADDRESS port0       2H
#pragma ADDRESS port1       3H
#pragma ADDRESS p0d 4H
#pragma ADDRESS p1d 5H
char    port0, port1, p0d, p1d;

void main()
{
      p0d = p1d = 0xFF;
      port0 = 0xAA;
      port1 ^= port0;
}
```

#pragma PARAMETER

レジスタ渡しのアセンブラ関数宣言機能

[機能]　引数をレジスタに格納して渡すアセンブラ関数を宣言します。

[書式]　#pragma PARAMETER△アセンブラ関数名(レジスタ名, レジスタ名, ..)

[解説]　アセンブラ関数を呼び出すときに、その引数を以下のレジスタに格納して渡すことを宣言します。

- Aレジスタ
- Bレジスタ
- Xレジスタ
- Yレジスタ

これらのレジスタは、すべて16ビットモードで使用されます。

[規定]
1. #pragma PARAMETER宣言を行う前には、必ずアセンブラ関数のプロトタイプ宣言を行ってください。プロトタイプ宣言を行わない場合は警告を出力し、この宣言を無効にします。
2. プロトタイプ宣言では、以下の項目を守ってください。
   a. 引数の数の最大は4個です。また、プロトタイプ宣言の引数の数と#pragma PARAMETER宣言の引数の数が一致している必要があります。
   b. 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
      - 浮動小数点型(float型、double型)
      - long型、char型、void型等のサイズが16ビット以外の型
      - farポインタ型
   c. アセンブラ関数のリターン値の型として以下の関数は宣言できません。
      - doubleを返す関数
      - 構造体、共用体を返す関数

[使用例]

```
int asm_func(unsigned int, unsigned int);
#pragma PARAMETER asm_func(X, Y);

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

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

}
```

B.7.4 MR79に関する拡張機能の使用方法

NC79は、次に示すリアルタイムOS MR79サポートに関する拡張機能を持っています。

#pragma INTHANDLER(#pragma HANDLER)

割り込みハンドラ関数の記述機能

[機能]　MR79の割り込みハンドラを宣言します。

[書式]　①#pragma INTHANDLER△割り込みハンドラ関数名
②#pragma HANDLER△割り込みハンドラ関数名

[解説]　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。

1.入口処理
- ●レジスタを現在のスタックへ退避します。
- ●スタックポインタ(SP)をタスクコントロールブロック(TCB)へ退避します。
- ●システムスタックへの切り替えを行います。

2.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。

#pragma INTHANDLERにより宣言された関数は、interruptセクションに配置されます。

[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、コンパイルエラーとなります。
- ●#pragma INTERRUPT
- ●#pragma HANDLER
- ●#pragma ALMHANDLER
- ●#pragma CYCHANDLER
- ●#pragma TASK

[使用例]

```
#include    <mr79.h>
#include    "id.h"

#pragma INTHANDLER  hand

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

#pragma ALMHANDLER

アラームハンドラの記述機能

[機能]　MR79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命令でリターンします。
- ●入口処理でデータバンクレジスタ(DT)の値をロードし、出口処理では旧DT値を復帰します。

#pragma ALMHANDLERにより宣言された関数は、interruptセクションに配置されます。

[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、コンパイルエラーとなります。
- ●#pragma INTERRUPT
- ●#pragma INTHANDLER
- ●#pragma HANDLER
- ●#pragma CYCHANDLER
- ●#pragma TASK

[使用例]

```
#include    <mr79.h>
#include    "id.h"

#pragma ALMHANDLER alm

void alm()
{
        :
      (省略)
        :
}
```

#pragma CYCHANDLER

周期起動ハンドラの記述機能

[機能]　MR79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命令でリターンします。
- ●入口処理でデータバンクレジスタ(DT)の値をロードし、出口処理では旧DT値を復帰します。

#pragma CYCHANDLERにより宣言された関数は、interruptセクションに配置されます。

[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、コンパイルエラーとなります。

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

[使用例]

```
#include    <mr79.h>
#include    "id.h"

#pragma CYCHANDLER cyc

void cyc()
{
        :
     (省略)
        :
}
```

#pragma TASK

タスク開始関数の記述機能

[機能]　MR79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、関数の出入口でタスク専用の処理を行うコードを生成します。

1.入口処理
- ●旧フレームポインタ(DPR)の待避を行いません。

2.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.引数を持つタスクの開始関数を記述することはできません。
2.タスクからのリターンにext_tskシステムコールを記述しないでください。
3.タスクの戻り値がvoid型であるように宣言してください。
4.#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。
5.関数名以外を指定した場合、無効となります。
6.#pragma TASK宣言の二重定義はエラーとなりません。
7.#pragma TASK宣言で以下に示す宣言と同じ関数を指定した場合、コンパイルエラーとなります。

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

[使用例]

```
#include    <mr79.h>
#include    "id.h"

#pragma TASK    main
#pragma TASK    tsk1

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

void tsk1()
        :
    (以下省略)
```

B.7.5 DTレジスタの操作に関する拡張機能の使用方法

NC79は、次に示すデータバンクレジスタ（DT)の操作に関する拡張機能を持っています。

#pragma LOADDT

DTレジスタのロード関数指定機能

[機能]　コンパイル時のDTの値を、関数の先頭でロードする関数を指定します。

[書式]　#pragma LOADDT△関数名

[解説]　#pragma LOADDTにより宣言された関数は、コンパイル時に-bank=オプションで指定されたデータバンクレジスタ(DT)の値を、関数の先頭でロードするコードを生成します。

1.#pragma LOADDT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。
2.関数名以外を指定した場合、無効となります。
3.#pragma LOADDT宣言の二重定義はエラーとなりません。

[規定]　この機能は64Kバイト以上のデータを持つようなアプリケーションプログラムで、DT値(near領域となるバンク値)を切り替えながら効率の良い処理を行わせるときに使用します。この機能を使用するときは、near/far属性と7900シリーズのアドレッシングモードの関係を熟知された上で使用してください。

[使用例]

```
[C言語ソースプログラム]
#pragma LOADDT          func

void func()
{
      int       i, j;
          :
      (以下省略)
          :

[アセンブリ言語ソースプログラム]
      .section        program_F
      ._file   'smp.c'
      ._line  4
;## # C_SRC: {
      .DT               __DT
      .DP0              OFF
      .glb              _func
_func:
      phd
      pht
      ldt               #__DT

          :
      (以下省略)
          :
```

B.7.6 DPレジスタの操作に関する拡張機能の使用方法

NC79は、次に示すダイレクトページレジスタ（DPR)の操作に関する拡張機能を持っています。

#pragma DP[n]DATA

DPレジスタのロード関数指定機能

[機能]　外部変数を指定したDPレジスタに割り当てます。

[書式]　#pragma DP[n]DATA△外部変数名
（nは、1～3です）

[解説]　#pragma DP[n]DATAにより宣言された外部変数は、疑似命令 .DPSYM を用いて定義されます。

[規定]　1.指定した変数が関数内で宣言された static変数の場合は無効になります。
2.#pragma DP[n]DATAで宣言された変数の領域は、下記のセクションに割り当てられます。
- data_DP[n]E
- data_DP[n]O
- bss_DP[n]E
- bss_DP[n]O
- data_IDP[n]E
- data_IDP[n]O

3.コンパイルオプション-fDP_offset_8(-fDPO8)指定時には無効になります。この場合は、オプション-fDPO8が指定されたものとしてコード生成を行います。
4.本宣言は外部変数の定義よりも前に行う必要があります。

[使用例]

```
#pragma  DP1DATA     data1
unsigned  int       data1;

void  func(void)
{
     unsigned   intdata_buff ;

     data1  =  100;
     data_buff   =  data1;

        :
     (以下省略)
        :

}
```

[注意]　1. デフォルトでは、DPレジスタの値はスタートアッププログラムncrt0.a79の中の __DP[n]の値です。

2. -fauto_128 (-fA1) オプション指定時は、#pragma DP1DATA を使用することはできません。

B.7.7 関数呼び出しに関する拡張機能の使用方法

NC79は、次に示す関数呼び出しに関する拡張機能を持っています。

#pragma M1FUNCTION

Mフラグ1状態指定関数呼び出し機能

[機能]　Mフラグを1の状態のまま関数を呼び出します。

[書式]　#pragma M1FUNCTION△関数名

[解説]　#pragma M1FUNCTIONにより宣言された関数は、Mフラグ、Xフラグが以下の状態に設定されます。

1. 呼び出す側の関数
   - ●Mフラグを1、Xフラグを0の状態にして呼び出します。
2. 呼び出される側の関数
   - ●Mフラグが1、Xフラグが0の状態であるものとして処理します。

[規定]　呼び出し側と呼ばれる側の関数に対して、同一の宣言を行ってください。

[使用例]

```
[C言語ソースプログラム]
#pragma M1FUNCTION func
char  func(char);

void main()
{
   func(1,2);
}


char func(char c,char d)
{
      return (c+d);
}

[アセンブリ言語ソースプログラム]
_main:
      ._line  6
;## #C_SRC:              func( 1,2 )
      ldab    A,#02H
      sem
      pha
      ldab    A,#01H
      jsrl    $func
        :
      (以下省略)
        :

$func:
      phd     0
      pht
      tsd
      stab    A,DP0:1       ;c
      ._line 11
        :
      (以下省略)
        :
```

#pragma MX1FUNCTION

MXフラグ1状態指定関数呼び出し機能

[機能]　関数の出入り口におけるMXフラグの状態がセットされていることを前提としたコードを生成します。

[書式]　#pragma MX1FUNCTION△関数名

[解説]　#pragma MX1FUNCTIONにより宣言された関数は、Mフラグ、Xフラグが以下の状態に設定されます。

1. 呼び出す側の関数
   - ●M、Xフラグをセットして呼び出します。
2. 呼び出される側の関数
   - ●M、Xフラグがセットされているものとして処理します。

[規定]　呼び出し側と呼ばれる側の関数に対して、同一の宣言を行ってください。

[使用例]

```
[C言語ソースプログラム]
#pragma MX1FUNCTION    func
          char    func(char,char);

          void    main(void)
          {
                func(1,2);
          }

          char    func(char c,char d)
          {
                return (c+d);
          }

[アセンブリ言語ソースプログラム]
 _main:
                ._line  6
          ;## # C_SRC :         func(1,2);
                ldab    A,#02H
                sem             <--Mフラグをセット
                pha
                ldab    A,#01H
                     sep    x       <--Xフラグをセット
                     jsrl   $func
                     adds   #0001H
                     ._line  7
                      ;## # C_SRC :  }
                clp     m,x
                        rtl
               :
               (略)
               :
             $func:
                        phd
               pht
               tsd
               stab    A,DP0:1
               ._line  11
          ;## # C_SRC :         return (c+d);
                     ldab    A,DP0:7
               add     A,DP0:1
               adds    #0001H
               rtld    0
```

B.7.8 その他の拡張機能の使用方法

NC79は、前述以外に次の拡張機能を持っています。

#pragma ASM(ENDASM)

インラインアセンブラ指定機能

[機能]　アセンブリ言語で記述を行なう領域を指定します。

[書式]
```
#pragma ASM
        アセンブリ言語記述
#pragma ENDASM
```

[解説]　#pragma ASMと#pragma ENDASMの間の領域を直接、アセンブリ言語ファイルにそのまま出力します。

[規定]　#pragma ASM を記述する際は、必ず#pragma ENDASMを組み合わせて記述してください。#pragma ASMと対になる#pragma ENDASMがない場合にはコンパイルを中断します。

[使用例]
```
void func()
{
      int      i, j;

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

#pragma ASM                    この領域をアセンブリ言語ファイルにそのまま
      lda      A,#0003H        出力します
LOOP:
      dec      A
        :
      (省略)
        :
#pragma ENDASM

}
```

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

#pragma PAGE

.PAGE疑似命令出力機能

[機能]　アセンブラで出力するリストファイルでの改行位置を宣言します。

[書式]　#pragma PAGE

[解説]　Cソースファイル中に#pragma PAGEを記述した場合、コンパイラが出力するアセンブリ言語ファイルに、.PAGE疑似命令を出力します。
アセンブラによりアセンブラリスティングファイルを出力する場合に改ページ指定を行うことができます。

[規定]　1.アセンブラ疑似命令.PAGEのヘッダに指定する文字列の指定はできません。
2.auto変数の宣言中に#pragma PAGEを記述することできません。

[使用例]

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

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

#pragma PAGE

        i++;

}
```

#pragma TBLJMPOFF

テーブルジャンプコード非出力機能

[機能]　本機能で指定された関数に対しては-fswitch_table(-fST)オプション指定時でもテーブルジャンプコードを生成しません。

[書式]　#pragma TBLJMPOFF 関数名

[規定]　1.#pragma TBLJMPOFF宣言以降に実体を定義した関数のみ有効です。
2.関数名以外を指定した場合は無効になります。
3.#pragma TBLJMPOFFの二重宣言はエラーとなりません。

[使用例]

```
#pragma TBLJMPOFF func

void  func(int i)
{

      switch(int i){
      case 1:
            .
            .
            .
      }

}
```

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

NC79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ため、プログラムのチューンアップが行いやすくなります。

B.8.2 アセンブラマクロ関数の記述例

アセンブラマクロ関数は、下記のようにＣ言語プログラム中にＣ言語の関数と同じ書式で記述することができます。アセンブラマクロ関数を使用する場合、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;
int    i, j;

func()
{
       i = div_w( l,j );
}
```

図B.68 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。

ASL

[機能]　valを1ビット算術左シフトした値を返します。

[書式]　#include <asmmacro.h>

```
char  asl_b( char val );  /* 8ビットでの演算の場合 */
int   asl_w( int val );   /* 16ビットでの演算の場合 */
```

ASR

[機能]　valを1ビット論理左シフトした値を返します。

[書式]　#include <asmmacro.h>

```
char  asr_b( char val );  /* 8ビットでの演算の場合 */
int   asr_w( int val );   /* 16ビットでの演算の場合 */
```

DIV

[機能]　val1をval2で除算した結果（商）を返します。除算時にオーバーフローが発生した場合、演算結果は不定値になります。

[書式]　#include <asmmacro.h>

```
int div_w( long val1, int val2 );       /* 16ビットでの演算のみ */
```

DIVS

[機能]　val1をval2で符号付き除算した結果（商）を返します。除算時にオーバーフローが発生した場合、演算結果は不定値になります。

[書式]　#include <asmmacro.h>

```
int divs_w( long val1, int val2 );      /* 16ビットでの演算のみ */
```

LSR

[機能]　valを1ビット論理左シフトした値を返します。

[書式]
```
#include <asmmacro.h>

char  lsr_b( char val );        /* 8ビットでの演算の場合 */
int   lsr_w( int val);          /* 16ビットでの演算の場合 */
```

MVN

[機能]　転送元アドレス(src)から転送先アドレス(dest)に指定バイト数(num)のデータを転送します。転送にはmvn命令を使用します。

※バンク値には__DTを使用します。

[書式]
```
#include <asmmacro.h>

void  mvn( char _near * _near dest, char _near * _near src, int num );
```

MVP

[機能]　転送元アドレス(src)から転送先アドレス(dest)に指定バイト数(num)のデータを転送します。転送にはmvp命令を使用します。

※バンク値には__DTを使用します。

[書式]
```
#include <asmmacro.h>

void  mvp( char _near * _near dest, char _near * _near src, int num );
```

ROL

[機能]　valをキャリーフラグを含めて1ビット左へ回転した値を返します。

[書式]
```
#include <asmmacro.h>

char  rol_b( char val );   /* 8ビットでの演算の場合 */
int   rol_w( int val );    /* 16ビットでの演算の場合 */
```

ROR

[機能]　valをキャリーフラグを含めて1ビット右へ回転した値を返します。

[書式]

```
#include <asmmacro.h>

char  ror_b( char val );   /* 8ビットでの演算の場合 */
int   ror_w( int val );    /* 16ビットでの演算の場合 */
```

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

NC79は、7900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7900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 NC79性能概要

以下にNC79の性能の概要を示します。

a.測定環境

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

表C.1 EWS標準環境

項目	EWSの種類	UNIXのバージョン
EWSの環境	SPARCstation	SunOS V.4.1.3 JLE1.1.3
		日本語Solaris 2.5
	HP9000 700シリーズ	HP-UX V.10.20
スワップ領域の空き容量	50Mバイト以上	

表C.2 PC標準環境

項目	PCの種類	OSのバージョン
ﾊﾟｰｿﾅﾙｺﾝﾋﾟｭｰﾀの環境	IBM PC/AT互換機	Windows95 WindowsNT 4.0
搭載CPUの種類	Pentium II	
メモリ容量	32Mバイト以上	

b.C言語ソースファイル記述仕様

【表C.3】にNC79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.3 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.NC79の仕様

【表C.4】にNC79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.4 NC79の仕様

項　目	仕 様
NC79で指定可能なファイル数	メモリの容量に依存します
ファイル名の長さ	OSに依存します
nc79の起動オプション-Dで指定可能なマクロ名の総数	メモリの容量に依存します
nc79の起動オプション-Iで指定可能なディレクトリ数	最大8
nc79の起動オプション-as79で引き渡し可能なパラメータ数	メモリの容量に依存します
nc79の起動オプション-ln79で引き渡し可能なパラメータ数	メモリの容量に依存します
複文、繰り返し制御構造、及び選択制御構造に対するネスト数	メモリの容量に依存します
条件コンパイルにおけるネスト数	メモリの容量に依存します
宣言中の基本型を修飾するポインタ、配列、及び関数宣言子の数	メモリの容量に依存します
関数定義の数	メモリの容量に依存します
１つのブロック中におけるブロック有効範囲を持つ識別子の数	メモリの容量に依存します
１つのソースファイル中で同時に定義され得るマクロ識別子の数	メモリの容量に依存します
マクロ名の置き換えの数	メモリの容量に依存します
入力プログラムにおける論理ソース行の数	メモリの容量に依存します
#includeファイルに対するネスト数	最大8
１つのswitch文中におけるcase名札の数 (switch文のネストがない場合)	メモリの容量に依存します
#if、#elif文で指定できる演算子、被演算子の合計数	メモリの容量に依存します
関数1個あたりで確保可能なスタックフレーム容量(バイト数)	255
#pragma ADDRESSで定義可能な変数の数	メモリの容量に依存します
括弧のネスト数	メモリの容量に依存します
初期化式付きの変数定義を行う場合の定義可能な初期値の数	メモリの容量に依存します
修飾宣言子のネスト数	YACCスタックに依存
宣言子の括弧によるネスト数	YACCスタックに依存
演算子の括弧によるネスト数	YACCスタックに依存
１つの内部識別子又はマクロ名で意味をもつ文字数	メモリの容量に依存します
１つの外部識別子で意味をもつ文字数	メモリの容量に依存します
１ソースファイル中の外部識別子の数	メモリの容量に依存します
１ブロックでブロックスコープを持つ識別子	メモリの容量に依存します
１ソースファイル中のマクロ数	メモリの容量に依存します
１つの関数、関数呼び出しのパラメータ数	メモリの容量に依存します
１つのマクロ定義、マクロ呼び出しのパラメータ数	最大31
結合後の文字列リテラル内の文字数	メモリの容量に依存します
１つのオブジェクトサイズ(バイト数)	メモリの容量に依存します
１つの構造体／共用体のメンバ数	メモリの容量に依存します
１つの列挙中の列挙定数の数	メモリの容量に依存します
１つのstruct宣言リスト中の構造体／共用体のネスト数	メモリの容量に依存します
１つの文字列の文字数	OSに依存します
１ファイルの行数	メモリの容量に依存します

C.2 基本言語仕様

NC79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。NC77/NC79は以下のものを字句(トークン)として処理します。

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

a.キーワード

NC79は以下のものをキーワードとして解釈します。

表C.5 キーワード一覧表

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

b.識別子

識別子は、以下の要素で構成されます。

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

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

c.定数

定数は以下に示す3種類のタイプがあります。

●整数定数
●浮動小数点定数
●文字定数

(1)整数定数

整数定数は、10進数のほか、8進数、16進数を指定することができます。各進数の書式を【表C.6】に示します。

表C.6 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

整数定数の型は、その値の大小により以下の順で決定されます。

●8進数、16進数 ..... signed int型→unsigned int型→signed long型→unsigned long型
●10進数　　signed int型→signed long型→unsigned long型

また、接尾詞U又はu、L又はl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

②long型定数

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

●signed　long型→unsigned 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.7 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.7】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

NC79は、【表C.8】に示すものを演算子として解釈します。

表C.8 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	?:
	ミ	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	sizeof
	!=		

f.区切り子

NC79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。

C.2.2 型

a.データ型

NC79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

NC79は、以下に示すものを型修飾子として解釈します。

- const
- volatile
- 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.5】に示します。

表C.9 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
char	なし	8	0～255
unsigned char			
signed char	有り	8	-128～127
int	有り	16	-32768～32767
short			
signed int			
signed short			
unsigned int	なし	16	0～65535
unsigned short			
long	有り	32	-2147483648～2147483647
signed long			
unsigned long	なし	32	0～4294967295
float	有り	32	1.17549435e-38F～3.40282347e+38F
double	有り	64	2.2250738585072014e-308
long double			～1.7976931348623157e+308
nearポインタ	なし	16	0～0xFFFF
farポインタ	なし	32	0～0xFFFFFFFF

●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。
●構造体のビットフィールドメンバで符号指定がない型は符号なしとして解釈します。

C.2.3 式

【表C.10】、【表C.11】に式の種類と式の構成要素の関係を示します。

表C.10 式の種類と式の構成要素(1)

式の種類	式の構成要素
一次式	識別子
	定数
	文字リテラル
	(式)
	一次式
後置式	後置式[式]
	後置式(引数の並び,...)
	後置式.識別子
	後置式－>識別子
	後置式＋＋
	後置式－－
	後置式

式の種類	式の構成要素
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j	式 ? 式 : 式
代入式	単項式 += 式
	単項式 -= 式
	単項式 ＊= 式
	単項式 /= 式
	単項式 %= 式
	単項式 <<= 式
	単項式 >>= 式
	単項式 &= 式
	単項式 ¦= 式
	単項式 ^= 式
	代入式
コンマ演算子	式 ,単項式

C.2.4 宣言

宣言には以下に示す2種類のタイプがあります。

●変数宣言
●関数宣言

a.変数宣言

変数の宣言は、【図C.1】に示す書式で記述します。

```
記憶クラス指定子△型宣言子△宣言指定子△初期化式;
```

図C.1 変数の宣言書式

(1)記憶クラス指定子

NC79は、以下の記憶クラス指定子をサポートしています。

●extern　●auto　●typedef
●static　●register

(2)型宣言子

NC79は、以下の型宣言子をサポートしています。

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

(3)宣言指定子

NC79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

NC79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

NC79は、以下の記憶クラス指定子をサポートしています。

- ●extern
- ●inline
- ●static

(2)型宣言子

NC77/NC79は、以下の型宣言子をサポートしています。

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

(3)宣言指定子

NC79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

NC79は、以下の文をサポートしています。

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

a.名札付き文

名札付き文は、【図C.7】に示す書式で記述します。

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

図C.7 名札付き文の書式

b.複文

複文は、【図C.8】に示す書式で記述します。

```
{ 宣言の並びopt 文の並びopt }

※optはオプション部であることを示します。
```

図C.8 複文の書式

c.式・空文

式及び空文は、【図C.9】に示す書式で記述します。

```
式:
式;
空文:
;
```

図C.9 式・空文の書式

d.選択文

選択文は、【図C.10】に示す書式で記述します。

```
if( 式 )文
if( 式 )文 else 文
switch( 式 )文
```

図C.10 選択文の書式

e.繰り返し文

繰り返し文は、【図C.11】に示す書式で記述します。

```
while( 式 )文
do 文 while ( 式 );
for( 式opt;式opt;式opt )文;

※optはオプション部であることを示します。
```

図C.11 繰り返し文の書式

f.分岐文

分岐文は、【図C.12】に示す書式で記述します。

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

※optはオプション部であることを示します。
```

図C.12 分岐文の書式

g.アセンブリ言語記述文

アセンブリ言語記述文は、【図C.13】に示す書式で記述します。

```
asm( "文字列" );
asm( mフラグ,xフラグ);

文字列　　：　　アセンブリ言語ステートメント
mフラグ　：　　データ長選択フラグの内容
xフラグ　：　　インデックス長選択フラグの内容
```

図C.13 アセンブリ言語記述文の書式

C.3 プリプロセスコマンド

プリプロセスコマンドは、#で始まるコマンドでプリプロセッサcpp77/cpp79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

NC79は、【表C.12】に示すプリプロセスコマンドを用意しています。

表C.12 プリプロセスコマンド一覧表

コマンド名	機能
#define	マクロの定義を行います。
#undef	マクロを未定義にします。
#include	指定したファイルの取り込みを行います。
#error	メッセージを標準出力に出力し処理を中断します。
#line	ファイルの行番号を指定します。
#assert	定数式が偽のときに警告を出力します。
#pragma	NC77/NC79の拡張機能の処理を指示します。
#if	条件コンパイルを行います。
#ifdef	条件コンパイルを行います。
#ifndef	条件コンパイルを行います。
#elif	条件コンパイルを行います。
#else	条件コンパイルを行います。
#endif	条件コンパイルを行います。

C.3.2 プリプロセスコマンドリファレンス

以降にNC79のプリプロセスコマンドの詳細仕様を説明します。プリプロセスコマンドは、【表C.12】の順序で掲載しています。

#define

[機能]　マクロの定義を行います。

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

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

●下記の記述をした場合、識別子は空白文字に置換されます。

```
#define SYMBOL
```

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

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

NC79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。

```
NC79
```

●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
```

#undef

[機能]　マクロを未定義にします。

[書式]　#undef△識別子

[解説]　●マクロとして定義された識別子を無効にします。

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

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

#include

[機能]　指定したファイルの取り込みを行います。

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

[解説]　①nc79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INC79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INC79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8です。
●該当するファイルが存在しないときはインクルードエラーとなります。

#error

[機能]　メッセージを標準出力に出力し処理を中断します。

[書式]　#error△文字列

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

#line

[機能]　ファイル中の行番号を付け替えます。

[書式]　#line△整数△"ファイル名"

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

#assert

[機能]　定数式が偽のときに警告を出力します。

[書式]　#assert△定数式

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

```
[Warning(cpp79.82):x.c, line xx]assertion warning
```

#pragma

[機能]　NC79の拡張機能の処理を指示します。

[書式]
①#pragma SECTION△既定セクションベース名△変更セクションベース名
②#pragma ROM△変数名
③#pragma STRUCT△構造体のタグ名△unpack
③#pragma STRUCT△構造体のタグ名△arrange
④#pragma INTERRUPT△割り込み処理関数名
④#pragma INTF△割り込み処理関数名
⑤#pragma ADDRESS△変数名△絶対アドレス
⑤#pragma EQU△変数名 = 絶対アドレス
⑥#pragma PARAMETER△アセンブラ関数名(レジスタ名, レジスタ名, ..)
⑦#pragma INTHANDLER△割り込みハンドラ関数名
⑦#pragma HANDLER△割り込みハンドラ関数名
⑧#pragma ALMHANDLER△アラームハンドラ関数名
⑨#pragma CYCHANDLER△周期起動ハンドラ関数名
⑩#pragma TASK△タスクの開始関数名
⑪#pragma LOADDT△関数名
⑫#pragma DPnDATA△外部変数名
⑬#pragma M1FUNCTION△関数名
⑭#pragma MX1FUNCTION△関数名
⑮#pragma ASM
⑮#pragma ENDASM
⑯#pragma PAGE

[解説]
①セクションベース名変更機能
②romセクションへの配置機能
③構造体配列制御機能
④割り込み関数の記述機能
⑤入出力変数の絶対アドレス指定機能
⑥レジスタ渡しのアセンブラ関数宣言機能
⑦割り込みハンドラ関数の記述機能
⑧アラームハンドラ関数の記述機能
⑨周期起動ハンドラ関数の記述機能
⑩タスク開始関数の記述機能
⑪DTレジスタのロード関数指定機能
⑫DPレジスタのロード関数指定機能
⑬Mフラグ1指定機能
⑭MXフラグ1指定機能
⑮インラインアセンブラ記述機能
⑯アセンブラリスティングファイル改ページ機能

- #pragmaでは上記14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。
- 処理の指定(SECTION、INTERRUPT等)の記述は、必ず大文字で記述してください。
- 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。nc79の起動オプション-Wunknown_pragma(-WUP)を指定したときのみ警告を出力します。

#if～#elif～#else～#endif

[機能]　条件コンパイルを行います(式が真であるか否かを調べます)。

[書式]
```
#if△定数式
      :
#elif△定数式
      :
#else
      :
#endif
```

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

#ifdef～#elif～#else～#endif

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

[書式]
```
#ifdef△識別子
        :
#elif△定数式
        :
#else
        :
#endif
```

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

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

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

#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]

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

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

●#elseは#ifndefと対で使用します。#elseと改行の間に字句列を記述することはできません。ただし、コメントを記述することはできます。

●#elifは#if、#ifdef、#ifndefと対で使用します。

●#endifは#ifndefで制御する範囲の終了を示します。#ifndefコマンドを使用する場合には必ずこのコマンドを記述してください 。

●#ifndef～#else～#endifの組み合せはネストすることができます。ネストレベルは特に制限はありません(ただし、メモリ容量に依存します)。

●識別子中にsizeof演算子、cast演算子、変数を使用することはできません。

C.3.3 プリデファインドマクロ

NC79では、予め以下のマクロが定義されています。

●NC79

C.3.4 プリデファインドマクロの使用方法

プリデファインドマクロを使用して、NC79以外のC言語プログラム中でマシン依存部をプリプロセスコマンドを使用して切り替える時等に使用します。

```
#ifdef NC79
#pragma ADDRESS      port0   2H
#pragma ADDRESS      port1   3H

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

NC79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・関数からの戻り値に関する規則を説明します。

D.1 データの内部表現

D.1.1 整数型

【表D.1】に整数型のデータが使用するバイト数を示します。

表D.1 整数型のデータサイズ

型名	符号の有無	ビットサイズ	表現できる数値
char	なし	8	0～255
unsigned char			
signed char	有り	8	-128～127
int	有り	16	-32768～32767
short			
signed int			
signed short			
unsigned int	なし	16	0～65535
unsigned short			
long	有り	32	-2147483648～2147483647
signed long			
unsigned long	なし	32	0～4294967295

(1)char型は符号指定がない場合、unsigned char型と解釈します。
(2)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。
(3)long型は符号指定がない場合、signed long型と解釈します。

D.1.2 浮動小数点型

【表D.2】に浮動小数点型のデータが使用するバイト数を示します。

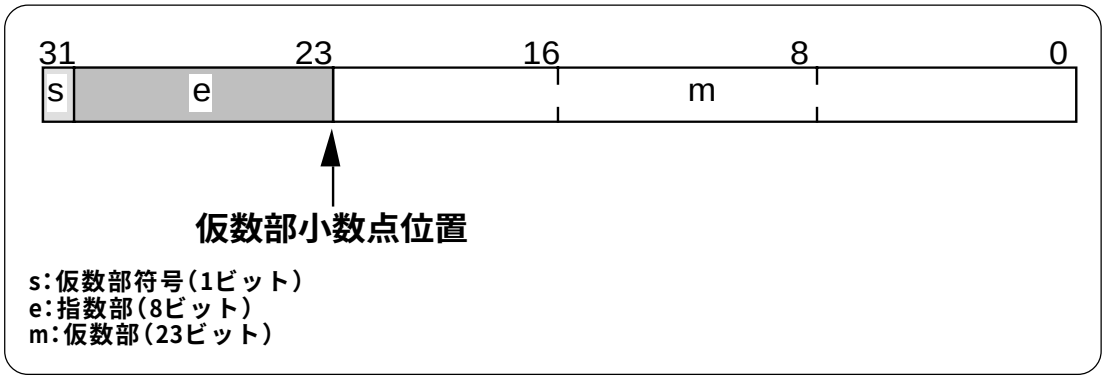
表D.2 浮動小数点型のデータサイズ

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

NC79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

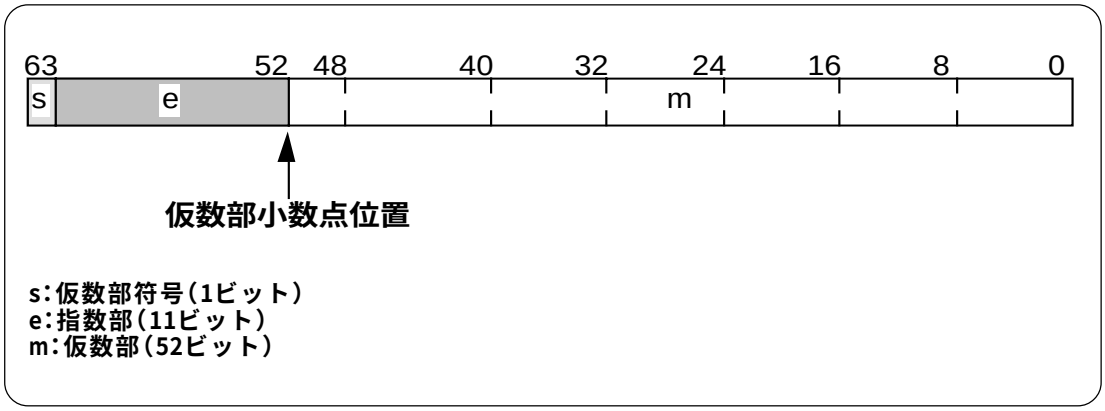
【図D.1】に示すデータ形式で2進数の浮動小数点(float)データを表現します。



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、nc79の起動オプション-fchar_enumerator(-fCE)を使用することにより列挙型をunsigned char型と同じ内部表現にできます。

D.1.4 ポインタ型

【表D.3】にポインタ型のデータが使用するバイト数を示します。

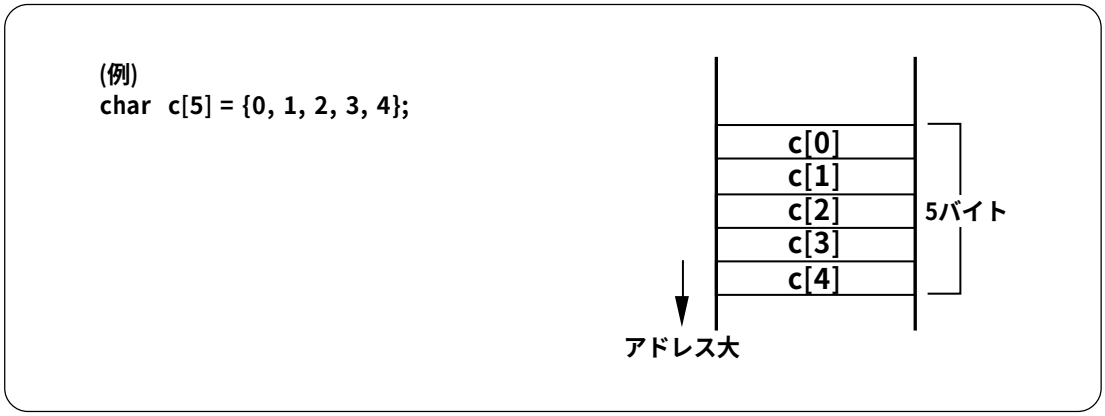
表D.3 ポインタ型のデータサイズ

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

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

D.1.5 配列型

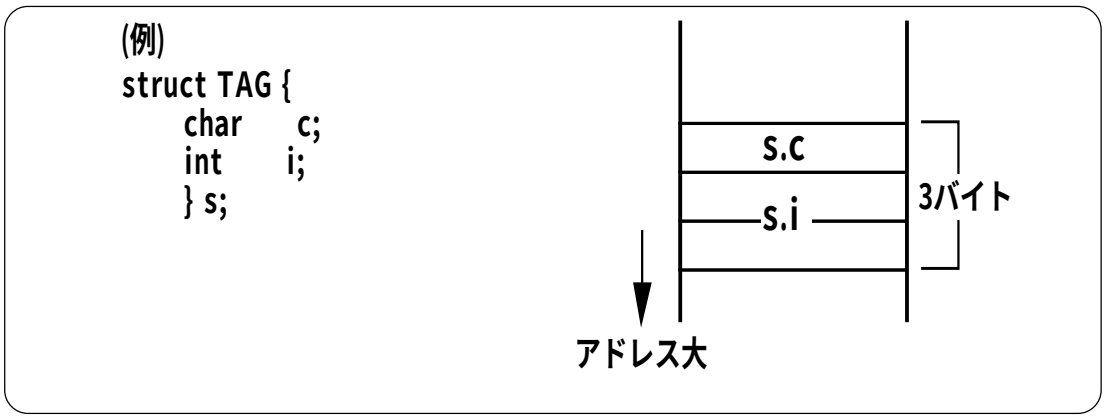
配列型は、要素のサイズ（バイト数）と要素数との積で表す領域に連続して配置されます。要素の出現順にメモリに配置されます。【図D.3】に配置例を示します。



図D.3 配列の配置例

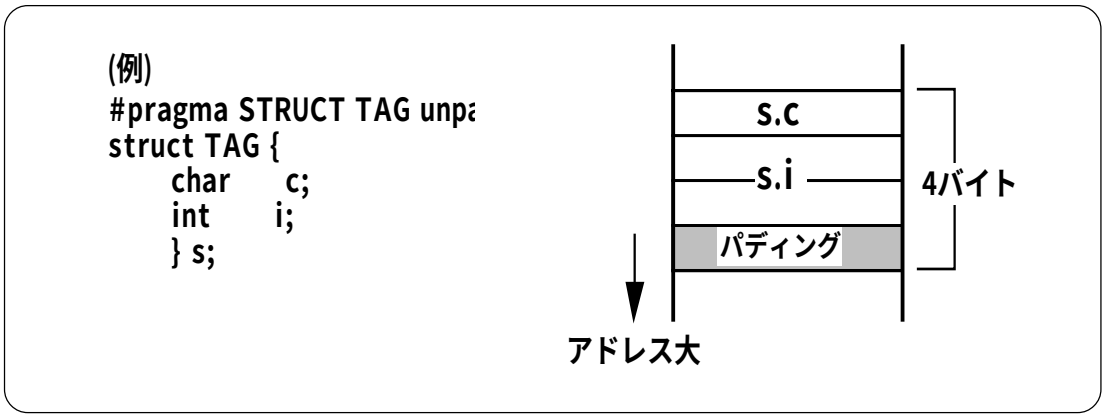
D.1.6 構造体型

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。



図D.4 構造体の配置例(1)

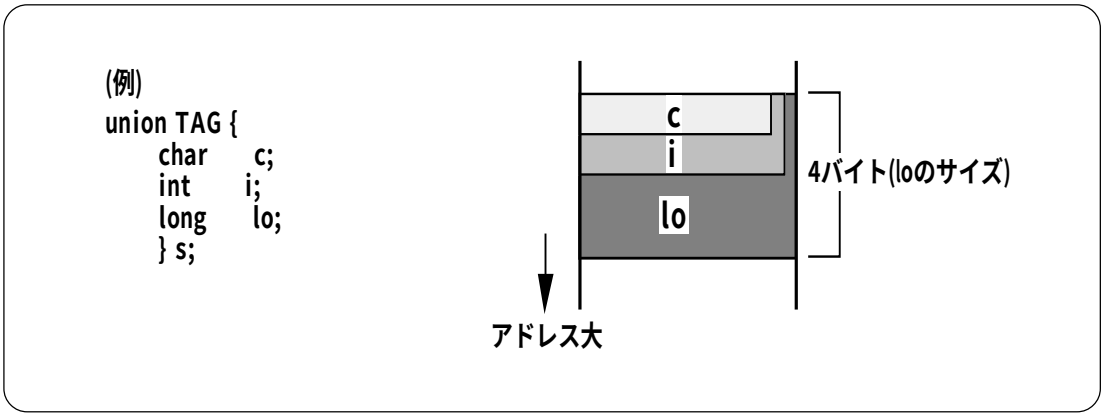
構造体は通常の場合、ワードアライメントを行いません。複数の構造体のメンバは連続して配置されます。ワードアライメントを行う場合は、拡張機能の#pragma STRUCTを使用します。#pragma STRUCTで宣言することにより、メンバのサイズの合計が奇数バイトであるときに1バイトのパディングを付加します。【図D.5】に配置例を示します。



図D.5 構造体の配置例(2)

D.1.7 共用体型

共用体型は、メンバの中で最大のデータサイズの領域をとります。【図D.6】に配置例を示します。

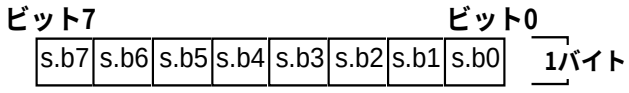


図D.6 共用体の配置例

D.1.8 ビットフィールド型

ビットフィールド型は、最下位のビットから配置されます。【図D.7】に配置例を示します。

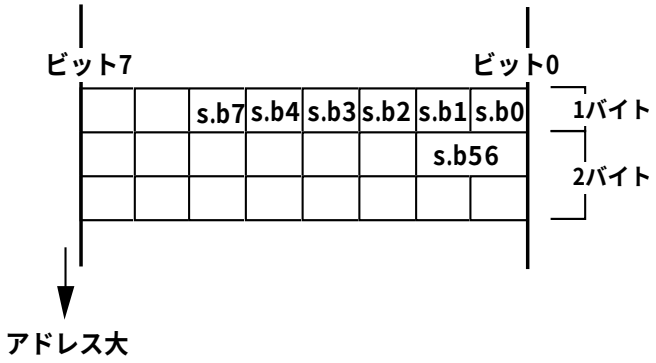
```
(例)
struct BTAG {
     char  b0 : 1;
     char  b1 : 1;
     char  b2 : 1;
     char  b3 : 1;
     char  b4 : 1;
     char  b5 : 1;
     char  b6 : 1;
     char  b7 : 1;
     } s;
```



図D.7 ビットフィールドの配置例(1)

ビットフィールドのメンバ中で、データ型が異なるものは次のアドレスに配置されます。このような場合、同じデータ型のメンバは同じデータ型が配置されるアドレス上に最下位アドレスから連続して配置されます。

```
(例)
struct BTAG {
     char  b0 : 1;
     char  b1 : 1;
     char  b2 : 1;
     char  b3 : 1;
     char  b4 : 1;
     int   b56: 2;
     char  b7 : 1;
     } s;
```



図D.8 ビットフィールドの配置例(2)

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。

D.2 符号拡張規則

ANSI規格等で定められた標準のC言語仕様では、char型のデータは演算時等においてint型に符号拡張して処理を行う規則を記しています。この仕様は、【図D.9】に示すようなchar型の演算を行うときに、演算の途中でchar型で表現できる最大値をオーバフローして結果が予期しない値になることを防ぐためです。

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

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

図D.9 C言語のサンプルプログラム例

NC79では、デフォルトでコード効率と実行速度を重視したコードを生成するために、char型をint型に符号拡張を行いません。この仕様は、コンパイルドライバnc77/nc79の起動オプション-fansi又は-fextend_to_int(-fETI)を使用することにより無効となり、標準のC言語と同様の符号拡張を行います。

-fansi又は-fextend_to_int(-fETI)オプションを使用しないで【図D.9】のように演算結果をchar型に代入するような演算を記述するときは、char型で表現できる最小値及び最大値※1が演算途中でオーバフローしないように注意してください。

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、Aレジスタ又はEレジスタに戻り値を格納して返します。Aレジスタ又はEレジスタで表すことのできる8～32ビット幅の戻り値を返すことができます。【表D.4】に戻り値に関する呼び出し規則を示します。

表D.4 戻り値に関する呼び出し規則

戻り値の型	規則
char型	Aレジスタの下位8ビットに格納して返します。 このときAレジスタの上位8ビットは不定となります。
int型 nearポインタ型	Aレジスタに格納して返します。
float型 long型 farポインタ型	Eレジスタに格納して返します。
集合体 double型 long double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。

※1.NC79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.3.2 引き数渡しに関する規則

NC79は、関数への引数渡しの方法として、レジスタ渡しとスタック渡しの2通りの方法を使用します。

(1)引き数のレジスタ渡し

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

●関数のプロトタイプ宣言を行ない、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

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

(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

引数の型	第1引数	第2引数
char型	Aレジスタ	スタック
int型	Aレジスタ	スタック
nearポインタ型		
その他の型	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センブ(アンダーバー)あるいは$(ダラー)を付加したもの、もしくは、関数名それ自身になります。付加文字列と文字列が付加される条件を【表D.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)	第一引数がレジスタ渡しとなる関数
_(アンダーバー)	上記条件以外の関数※1

【図 D.10】に示すプログラムは、関数の引数がレジスタ引数を持つものと、関数の引数をスタック渡しのみで扱う例です。

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

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

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

void main( void )                                   ④
{
        int sum;
        sun = func_proto( 1,2,3 );             ⑤
        sun = 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っています[新しい形式])
③関数func_no_protoの実体です(K&R形式の記述です)。
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)

※1.NC79では、プロトタイプ宣言を行なった時(新しい形式の記述の時)のみ、レジスタ渡しを適応します。つまり、K&R形式の記述(旧形式の記述)を行なった場合には、すべての引数をスタック渡しで行ないます。また、C言語の言語仕様上、関数に対してプロトタイプ宣言を行なう記述形式(新しい形式)とK&R形式(旧形式)の記述を混在すると、引数が関数に正しく渡されない場合があることに注意してください。NC79では、上記の理由によって、プロトタイプ宣言を行なう記述形式に統一してC言語ソースファイルを記述することを推奨しています。

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の呼び出しを(④の部分)を【図D.15】及び【図D.D】に示します。

```
;## #       FUNCTION func_proto
;## #       FRAME      AUTO (     i)size  2,      offset 1
;## #       FRAME      ARG  (     j)size  2,      offset 8          ←⑧
;## #       FRAME      ARG  (     k)      size  2,      offset 10   ←⑦
;## #       REGISTER ARG    (     i)size  2,      REGISTER A        ←⑨
;## #       ARG Size(4)  Auto Size(2)  Context Size(5)

     .section      program_F
     ._file  'smp.c'
     ._line  4
;## # C_SRC :    {
     .DT   __DT
     .DP0  OFF
     .glb  $func_proto
$func_proto:                                                        ←⑩
     phd   0
     pha
     tsd
     sta   A,DP0:1      ; i
     ._line  5
;## # C_SRC :           return i+j+k;
     lda   A,DP0:8      ; j
     add   A,DP0:1      ; i
     add   A,DP0:10     ; k
     plx
     rtld  0


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をレジスタ渡ししています。
⑩関数func_protoの先頭アドレスです。
```

図D.11 サンプルプログラム(sample.c)のコンパイル結果(1)

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

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

```
;## #        FUNCTION func_no_proto
;## #        FRAME       ARG  (      i)size   2,       offset 6
;## #        FRAME       ARG  (      j)size   2,       offset 8
;## #        FRAME       ARG  (      k)       size   2,       offset 10
;## #        ARG Size(6) Auto Size(0)  Context Size(5)

     ._line  13
;## # C_SRC :     {
     .DT     __DT
     .DP0    OFF
     .glb    _func_no_proto
_func_no_proto:
     phd     0
     tsd
     ._line  14
;## # C_SRC :             return i+j+k;
     lda     A,DP0:8       ; j
     add     A,DP0:6       ; i
     add     A,DP0:10      ; k
     rtld    0
```

図D.12 サンプルプログラム(sample.c)のコンパイル結果(2)

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

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

```
;## #       FUNCTION main
;## #       FRAME      AUTO (    sum)     size  2,      offset 1
;## #       ARG Size(0) Auto Size(2)  Context Size(5)

      ._line  18
;## # C_SRC :    {
      .DT     __DT
      .DP0    OFF
      .glb    _main
_main:
      phd     0
      pha
      tsd
      ._line  20
;## # C_SRC :          sum = func_proto( 1,2,3 );
      pea     #0003H                                          ⑪
      pea     #0002H
      lda.W   A,#0001H
      jsrl    $func_proto
      plx
      plx
      sta     A,DP0:1     ;  sum
      ._line  21
;## # C_SRC :          sum = func_no_proto( 1,2,3 );
      pea     #0003H                                          ⑫
      pea     #0002H
      pea     #0001H
      jsrl    _func_no_proto
      tds
      sta     A,DP0:1     ;  sum
      ._line  22
;## # C_SRC :    }
      plx
      rtld    0
      .END


⑪すべての引数をスタック渡しにしています。
⑫関数func_no_protoの先頭アドレスです。
```

図D.13 サンプルプログラム(sample.c)のコンパイル結果(3)

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

D.3.4 関数間のインターフェース

【図D.17】に示すプログラムにおいて、引数及び戻り値の処理を【図D.20】～【図D.22】に示します。なお、【図D.18】～【図D.19】は、【図D.17】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。【図D.23】～【図D.24】に《NC79》で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を示します。

```
int   func( int, int, int );

void main( void )
{
      int     i=0x1234;           ←関数funcへの引数
      int     j=0x5678;           ←関数funcへの引数
      int     k=0x9abc;           ←関数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;                 ←関数mainへの引数
}
```

図D.17 C言語のサンプルプログラム例(sample2.c)

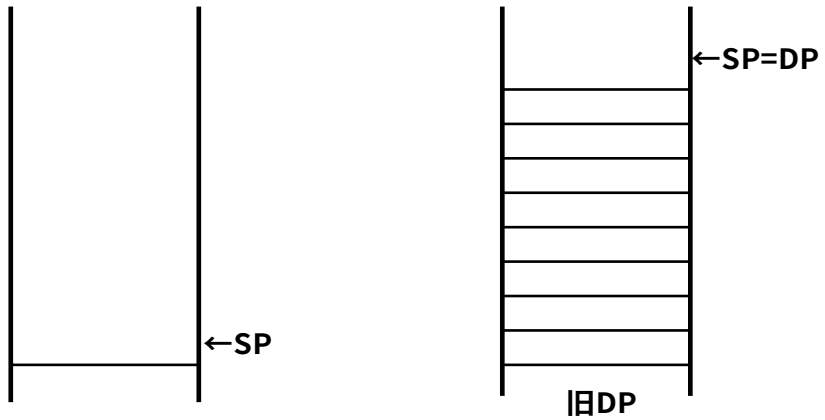
```
      .section      program_F
      ._file  'smp.c'
      ._line  4
;## # C_SRC :    {
      .DT     __DT
      .DP0    OFF
      .glb    _main
_main:                                          ←①
      phd     0
      subs    #0006H
      tsd                                       ←②
      ._line  5
;## # C_SRC :          int     i=0x1234;
      movm.W      DP0:5,#1234H      ; i
      ._line  6
;## # C_SRC :          int     j=0x5678;
      movm.W      DP0:3,#5678H      ; j
      ._line  7
;## # C_SRC :          int     k=0x9abc;
      movm.W      DP0:1,#9abcH      ; k
      ._line  9
;## # C_SRC :          k=func( i,j,k );
      pei     DP0:1 ; k                         ←③
      pei     DP0:3 ; j                         ←④
      lda     A,DP0:5     ; i                   ←⑤
      jsrl    $func                             ←⑥
      plx
      plx                                       ←⑩
      sta     A,DP0:1     ; k                   ←⑪
      ._line  10
;## # C_SRC :    }
      adds    #0006H
      rtld    0
```

図D.18 アセンブリ言語サンプルプログラム(1)

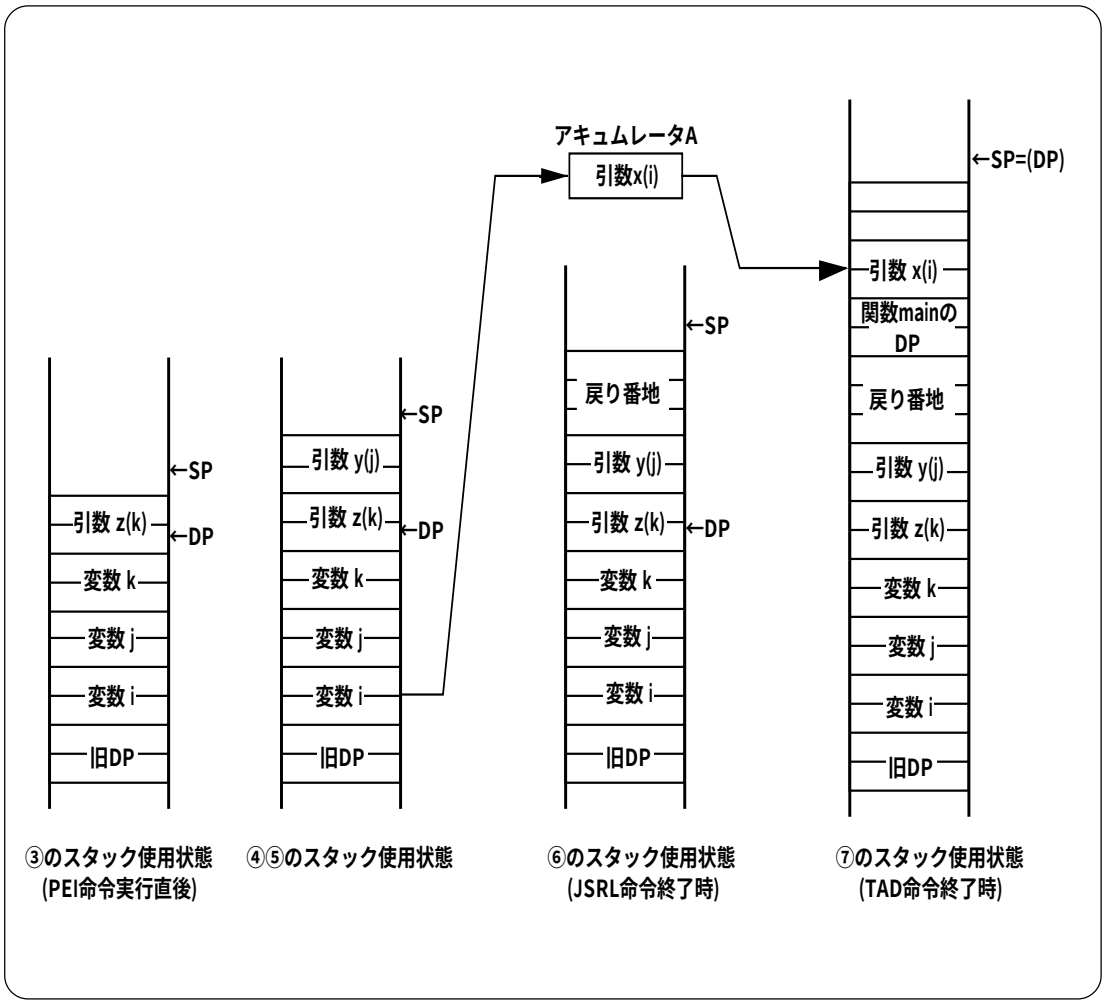
```
      ._line  13
;## # C_SRC :    {
      .DT     __DT
      .DP0    OFF
      .glb    $func
$func:
      phd     0
      pha
      tsd
      sta     A,DP0:1       ; x                        ←⑦
      ._line  16
;## # C_SRC :             sum=x+y+z;
      lda     A,DP0:8       ; y
      add     A,DP0:1       ; x
      add     A,DP0:10      ; z
      sta     A,DP0:1       ; sum
      ._line  17
;## # C_SRC :             return sum;
      lda     A,DP0:1       ; sum                      ←⑧
      plx
      rtld    0                                        ←⑨
      .END
```

図D.19 アセンブリ言語サンプルプログラム(2)

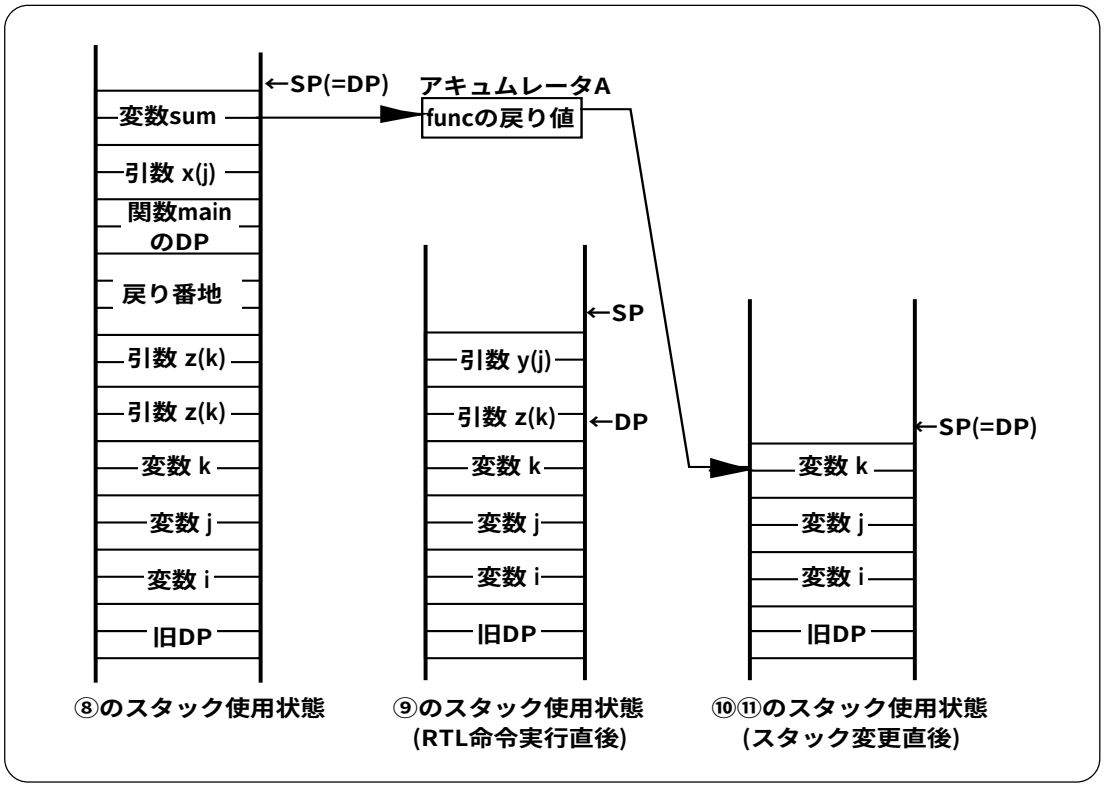
【図D.18】～【図D.19】中の①→②の処理(関数mainの入り口処理)を【図D.20】に、③→④→⑤→⑥→⑦の処理(関数funcの呼び出し及び関数funcで使用するスタックフレームの構築処理)を【図D.21】に、⑧→⑨→⑩→⑪の処理(関数funcから関数mainへの戻り処理)を【図D.22】に、各々のスタック及びレジスタの遷移を示します。



図D.20 関数 mainの入り口処理



図D.21 関数funcの呼び出し及び、入り口処理



図D.22 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、7900シリーズのスタック上に配置されます。【図D.25】に示すようなC言語ソースプログラムでは、記憶クラスautoの変数が有効となる領域が互いに重ならない場合、1つの領域のみ確保を行い複数の変数でその領域を共有します。

```
func()
{
    int     i, j, k;
  ┌──────────────────────────────────────┐
  │ for( i=0 ; i<=0 ; i++ ){             │
  │       処理                  iの有効範囲 │
  │ }                                    │
  └──────────────────────────────────────┘
      :
    (省略)
      :
  ┌──────────────────────────────────────┐
  │ for( j=0xFF ; j<=0 ; j-- ){          │
  │       処理                  jの有効範囲 │
  │ }                                    │
  └──────────────────────────────────────┘
      :
    (省略)
      :
  ┌──────────────────────────────────────┐
  │ for( k=0 ; k<=0 ; k++ ){             │
  │       処理                  kの有効範囲 │
  │ }                                    │
  └──────────────────────────────────────┘
}
```

図D.25 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(DPからのオフセット1)を共有します。【図D.25】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.26】に示します。

```
;###   FUNCTION func
;###         FRAME AUTO    (     k)    size  2,     offset 1     ←①
;###         FRAME AUTO    (     j)    size  2,     offset 1     ←②
;###         FRAME AUTO    (     i)    size  2,     offset 1     ←③

    .source tst1.c
    .section      program_F
    .DT    __DT
    .DP    OFF
    .func  _func
    .pub   _func
_func:
      :
    (以下省略)



※3つのauto変数は①②③に示すように、DPオフセット1の領域を共有しています。
```

図D.26 アセンブリ言語ソースプログラム例

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

　NC79は、【表E.1】に示すように15個の標準ヘッダファイルを用意しています。

表E.1 標準ヘッダファイル一覧表

ヘッダファイル名	内容
assert.h	プログラムの診断情報の出力
ctype.h	文字判定関数のマクロ宣言
errno.h	エラー番号の定義
float.h	浮動小数点数の内部表現に関する各種制限値の定義
limits.h	コンパイラの内部処理に関する各種制限値の定義
locale.h	関数・マクロの地域化
math.h	数値計算
setjmp.h	分岐関数、分岐関数で使用する構造体の定義
signal.h	非同期割り込みを処理するための定義・宣言
stdarg.h	可変個の実引数を持つ関数の宣言と定義
stddef.h	各標準インクルードファイルで共通に使用するマクロ名の定義
stdio.h	FILE構造体の定義
	ストリーム名の定義
	入出力関数のプロトタイプ宣言
stdlib.h	メモリ管理関数、終了関数のプロトタイプ宣言
string.h	文字列操作関数、メモリ操作関数のプロトタイプ宣言
time.h	現在の暦時間を得る

E.1.2 標準ヘッダファイルリファレンス

　以降にNC79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NC79の標準関数と、データ型の数値表現における制限値を定義しているマクロを、対応するヘッダファイルと共に説明します。

assert.h

[機能]　関数assertを定義しています。

ctype.h

[機能]　文字操作関数を宣言、及びマクロを定義しています。文字操作関数を以下に示します。

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

errno.h

[機能]　エラー番号を定義しています。

float.h

[機能]　浮動小数点数の内部表現に関する各種制限値を定義しています。以下に、浮動小数点数の制限値を定義したマクロを示します。

NC79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
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
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
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
MB_LEN_MAX	マルチバイト文字型のバイト数の最大値	1
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
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
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
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
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

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。

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	7700ファミリの入出力の初期化
	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	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型浮動小数に変換
atol	文字列をlong型浮動小数に変換
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

NC79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

NC79は119個の標準関数ライブラリを用意しています。各関数は機能的に以下の11種類に分類されます。

1.文字列操作関数
文字列のコピー、比較等を行う関数です。

2.文字判定関数
アルファベット、10進文字等の判定、及び大文字から小文字へ、小文字から大文字へ変換する関数です。

3.入出力関数
文字、文字列の入出力を行う関数です。中には、書式変換付きの入出力、及び文字列操作を行う関数も含まれています。

4.メモリ管理関数
動的メモリ領域の確保、及び解放を行う関数です。

5.メモリ操作関数
メモリ領域の複写、設定、及び比較を行う関数です。

6.実行制御関数
プログラムの実行を終了する関数と、現在実行中の関数から別の関数へジャンプするための関数です。

7.数学関数
sin、cos等の演算を行う関数です。
※これらの関数は時間を要します。このため、監視タイマには十分注意してください。

8.整数算術関数
整数値に対しての演算を行う関数です。

9.文字列数値変換関数
文字列を数値に変換する関数です。

10.多バイト文字・多バイト文字列操作関数
多バイト文字・多バイト文字列を扱う関数です。

11.地域化関数
ロケールに関する関数です。

E.2.2 標準関数ライブラリ機能別一覧

a.文字列操作関数

文字列操作関数の一覧を【表E.2】に示します。

表E.2 文字列操作関数

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

※ 標準関数の幾つかは､その関数専用の大域変数を使用しています｡関数が呼び出されて実行している最中に割り込みが入り､割り込み処理プログラム中で同じ関数が呼び出された場合､最初に呼び出された関数で使用していた大域変数の内容を書き換えることがあります｡リエントラント性がある関数(表中では○)は､このような大域変数の書き換えは発生しません｡リエントラント性がない関数(表中では×)を割り込み処理プログラムで使用するときは注意してください｡

b.文字操作関数

文字操作関数の一覧を【表E.3】に示します。

表E.3 文字操作関数

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

c.入出力関数

入出力関数の一覧を【表E.4】に示します。

表E.4 入出力関数

種別	関数名	機能	ﾘｴﾝﾄﾗﾝﾄ性
初期化	init	7900シリーズの入出力の初期化	○
	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の設定	○

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	疑似乱数にシードを与える	○

i.文字列数値変換関数

文字列数値変換関数の一覧表を【表E.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	○
strtoul	文字列をunsigned long型に変換	○

j.多バイト文字・多バイト文字列操作関数

多バイト文字・多バイト文字列操作関数の一覧表を【表E.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NC79が提供する標準関数の詳細仕様を説明します。関数は、アルファベット順に掲載しています。

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

abort

実行制御関数

[機能]　　プログラムを異常終了します。

[書式]　　#include <stdlib.h>
void _far 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]　　整数の絶対値を計算します。

[書式]　　#include <stdlib.h>
int _far abs(n);

[実現方法]　関数

[引数]　　int n; 整数

[戻り値]　●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 _far 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 _far 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 _far atan(x);

[実現方法]　関数

[引数]　double x; ..... 実数

[戻り値]　●-π/2からπ/2ラジアンの範囲の値を返します。

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 _far 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 _far atof( s );
```

[実現方法]　関数

[引数]　const char ＊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 _far atoi( s );
```

[実現方法]　関数

[引数]　const char ＊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 _far atol(s);

[実現方法]　関数

[引数]　const char ＊s;　変換文字列へのポインタ

[戻り値]　●文字列をlong型整数に変換した値を返します。

bcopy

メモリ操作関数

[機能]　メモリ領域の複写を行います。

[書式]　#include <string.h>
void _far 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]　●戻り値はありません。

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

bsearch

整数算術変換関数

[機能]　配列内のバイナリサーチを行います。

[書式]
```
#include <stdlib.h>
void _far bsearch( key, base, nelem, cmp );
```

[実現方法]　関数

[引数]
```
const void ＊s; ..... 検索キー
const void＊s; ..... 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]
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。
- 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 _far bzero( top, size );
```

[実現方法]　関数

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

[戻り値]　●戻り値はありません。

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

calloc

メモリ管理関数

[機能]　メモリの割り当てとゼロクリアを行います。

[書式]　#include <stdlib.h>
void _far ＊ _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]　●確保した領域の先頭アドレスを返します。
●指定した大きさの領域が確保できなかった場合、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 _far 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 _far clearerr( stream );
```

[実現方法]　関数

[引数]　FILE ＊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 _far 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 _far cosh(x);

[実現方法]　関数

[引数]　double x; ..... 実数

[戻り値]　●"x"の双曲線コサインを計算します。

div

剰余算関数

[機能]　int型整数の除算を行います。

[書式]　#include <stdlib.h>
div_t _far div(number, denom);

[実現方法]　関数

[引数]　int number; ..... 被除数
int denom; ..... 除数

[戻り値]　●"number"を"denom"で割った商と剰余を返します。

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

exp

数学関数

[機能]　　指数関数を計算します。

[書式]　　#include <math.h>
double _far exp(x);

[実現方法]　関数

[引数]　　double x; .....実数

[戻り値]　●"x"の指数関数の計算結果を返します。

fabs

数学関数

[機能]　　倍精度浮動小数の絶対値を計算します。

[書式]　　#include <math.h>
double _far 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 _far feof( stream );
```

[実現方法]　マクロ

[引数]　FILE ＊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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far ferror( stream );
```

[実現方法]　マクロ

[引数]　FILE ＊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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far fflush( stream );
```

[実現方法]　関数

[引数]　FILE ＊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 _far fgetc( stream );
```

[実現方法]　関数

[引数]　FILE ＊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。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 ＊buffer; ..... 格納先のポインタ
int n; 最大文字数
FILE ＊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。
　・改行文字('\n')を入力した場合
　・n-1個の文字を入力した場合
　・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。
●buffer及びstreamをfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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

[実現方法]　関数

[引数]
FILE ＊stream; ..... ストリームのポインタ
const char ＊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。
●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、printfと同様です。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far fputc( c, stream );
```

[実現方法]　関数

[引数]
int c; 出力する文字
FILE ＊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。
●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。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far fputs ( str, stream );
```

[実現方法]　関数

[引数]
const char ＊str;　出力する文字列のポインタ
FILE ＊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。
●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。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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

[実現方法]　関数

[引数]
void ＊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 ＊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。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far 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 _far frexp(x,prexp);

[実現方法]　関数

[引数]　double x; ..... 浮動小数
int ＊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 _far fscanf( stream, format, argument... );
```

[実現方法]　関数

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

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

[解説]
- formatの指定に従ってストリームからの入力文字を変換し、各引数argumentが示す変数に格納します。
- argumentは各変数のポインタでなければいけません。
- 0x1Aコードを終了コードとみなし、以降のデータを受け付けません。
- formatの指定方法は、scanfと同様です。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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

[実現方法]　関数

[引数]
const void ＊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 ＊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。
- 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。
- count分のデータを入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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far getc( stream );
```

[実現方法]　マクロ

[引数]　FILE ＊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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far 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 ＊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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far(Windows版はmakefar.dos)を使用してライブラリファイルを作り直してください。

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 _far init( void );
```

[実現方法]　関数

[引数]　引数はありません。

[戻り値]
- 戻り値はありません。

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

isalnum

文字操作関数

[機能]　英字(A～Z、a～z)、数字(0～9)を判定します。

[書式]
```
#include <ctype.h>
int isalnum( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●英字、又は数字の場合、0以外を返します。
●英字、又は数字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isalpha

文字操作関数

[機能]　英字(A～Z、a～z)を判定します。

[書式]
```
#include <ctype.h>
int isalpha( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●英字の場合、0以外を返します。
●英字でない場合、0を返します。

[解説]　●引数の文字を判定します。

iscntrl

文字操作関数

[機能]　コントロール文字(0x00～0x1f、0x7f)を判定します。

[書式]
```
#include <ctype.h>
int iscntrl( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●コントロール文字の場合、0以外を返します。
●コントロール文字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isdigit

文字操作関数

[機能]　数字(0～9)を判定します。

[書式]
```
#include <ctype.h>
int isdigit( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●数字の場合、0以外を返します。
●数字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isgraph

文字操作関数

[機能]　ブランク以外の文字(0x21～0x7e)を印字文字判定します。

[書式]
```
#include <ctype.h>
int isgraph( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●印字可能な場合、0以外を返します。
●印字不可の場合、0を返します。

[解説]　●引数の文字を判定します。

islower

文字操作関数

[機能]　英小文字(a～z)を判定します。

[書式]
```
#include <ctype.h>
int islower( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●英小文字の場合、0以外を返します。
●英小文字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isprint

文字操作関数

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

[書式]
```
#include <ctype.h>
int isprint( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●印字可能な場合、0以外を返します。
●印字不可の場合、0を返します。

[解説]　●引数の文字を判定します。

ispunct

文字操作関数

[機能]　区切り文字を判定します。

[書式]
```
#include <ctype.h>
int ispunct( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●区切り文字の場合、0以外を返します。
●区切り文字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isspace

文字操作関数

[機能]　ブランク、タブ、改行を判定します。

[書式]
```
#include <ctype.h>
int isspace( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●ブランク、タブ、改行の場合、0以外を返します。
●ブランク、タブ、改行でない場合、0を返します。

[解説]　●引数の文字を判定します。

isupper

文字操作関数

[機能]　英大文字(A～Z)を判定します。

[書式]
```
#include <ctype.h>
int isupper( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
●英大文字の場合、0以外を返します。
●英大文字でない場合、0を返します。

[解説]　●引数の文字を判定します。

isxdigit

文字操作関数

[機能]　16進文字(0～9、A～Z、a～z)を判定します。

[書式]
```
#include <ctype.h>
int isxdigit( c );
```

[実現方法]　マクロ

[引数]　int c; 判定する文字

[戻り値]
- 16進文字の場合、0以外を返します。
- 16進文字でない場合、0を返します。

[解説]
- 引数の文字を判定します。

labs

整数算術関数

[機能]　long型整数の絶対値を計算します。

[書式]
```
#include <stdlib.h>
long _far 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

地域化関数

[機能]　浮動小数の巾を計算します。

[書式]
```
#include <math.h>
double _far 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

地域化関数

[機能]　int型整数の除算を行います。

[書式]
```
#include <stdlib.h>
ldiv_t  _far 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。
●div_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 _far 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 _far 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 _far longjmp(env, val);

[実現方法]　関数

[引数]　jmp_buf _far ＊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。
●"value"で指定した値は、setjmp関数の結果として返します。ただし、"val"が"0"の場合"1"に変換されます。

malloc

メモリ管理関数

[機能]　mallocメモリの割り当てを行います。

[書式]　#include <stdlib.h>
void _far ＊ _far malloc(nbytes);

[実現方法]　関数

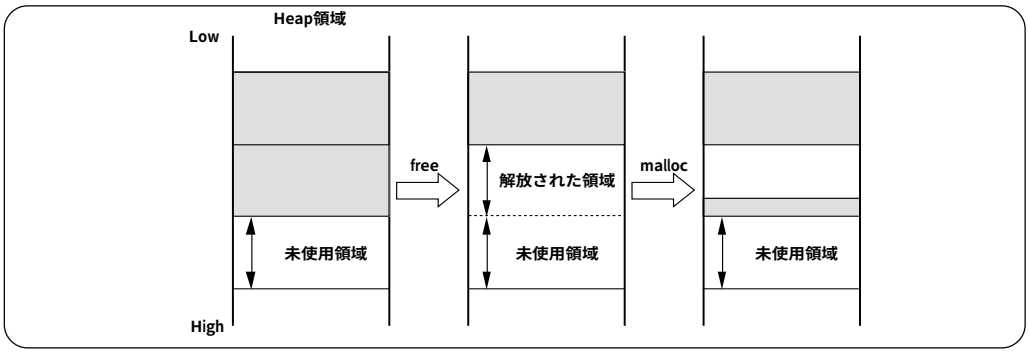
[引数]　size_t nbytes; ..... 割り当てるメモリの大きさバイト数

[戻り値]
●確保した領域の先頭アドレスを返します。
●指定した大きさの領域が確保できなかった場合には戻り値としてNULLを返します。

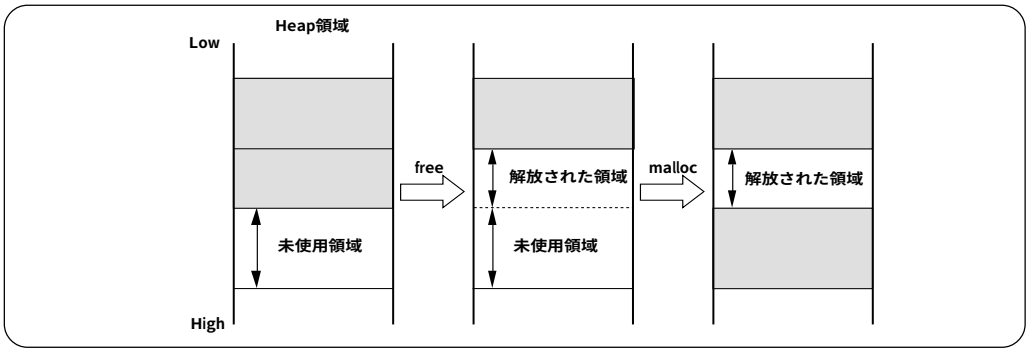
[解説]
●動的にメモリ領域を割り当てます。

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

(1)freeで解放された領域がある場合
(1-1)確保するサイズがfreeで解放された領域より小さい場合
freeで作成された連続空き領域の上位番地から下位番地方向へ領域が確保されます。



(1-2)確保するサイズがfreeで解放された領域よりも大きい場合
未使用領域の下位番地から上位番地方向に領域が確保されます。

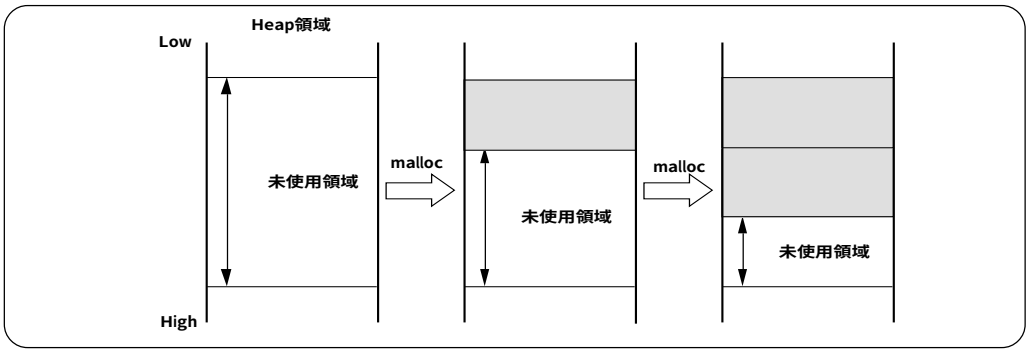


malloc

[注意]　　(2)freeで解放された領域がない場合

(2-1)確保可能な未使用領域が存在する場合

未使用領域の下位番地から上位番地方向に領域が確保されます。



(2-2)確保可能な未使用領域が存在しない場合

メモリは確保せずに、NULLを返します。

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 _far mblen ( s,n );
```

[実現方法]　関数

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

[戻り値]
●sが正しいマルチバイト文字列をなしている場合はそのバイト数を返します。
●sが正しいマルチバイト文字列をなしていない場合は-1を返します。
●sがNULL文字を指している場合は0を返します。

mbstowcs

多バイト文字･多バイト文字列操作関数

[機能]　マルチバイト文字列をワイド文字列に変換します。

[書式]
```
#include <stdlib.h>
size_t  _far mbstowcs( wcs,s,n );
```

[実現方法]　関数

[引数]
wchar_t ＊wcs; ..... 変換ワイド文字列格納領域へのポインタ
const char ＊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  _far mbtowc( wcs,s,n );
```

[実現方法]　関数

[引数]
wchar_t ＊wcs; ..... 変換ワイド文字列格納領域へのポインタ
const char ＊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 ＊ _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。

[解説]
- メモリ領域中に文字"c"が見つからない場合にはNULLを返します
- "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,-OR及び-OSを指定した場合は､最適化によりコード効率の良い関数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 _far 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

- ●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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _far memcpy( s1, s2, n );
```

[実現方法]　関数

[引数]
```
void _far ＊s1; ..... 複写先のメモリ領域へのポインタ
const void _far ＊2; ..... 複写元のメモリ領域へのポインタ
size_t n; ..... 複写するバイト数
```

[戻り値]
- ●複写先のメモリ領域へのポインタを返します。

[解説]
- ●"s1"で示される領域に、"n"で指定されたバイト数分"s2"で示されるメモリ領域より複写します。
- ●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far 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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _far memmove( s1, s2, n );
```

[実現方法]　関数

[引数]
void ＊s1; ..... 移動先へのポインタ
const void ＊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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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>
char _far＊ _far memset(s, c, n);

[実現方法]　関数

[引数]　void _far ＊s; ..... 設定先のメモリ領域への先頭ポインタ
int c; 設定するデータ
size_t n; ..... 設定するバイト数

[戻り値]　●設定先のメモリ領域への先頭ポインタを返します。

[解説]　●"s"で示される領域に、"n"で指定されたバイト数分"c"で指定されたデータを設定します。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far modf (val,pd);

[実現方法]　関数

[引数]　double val; .....実数
double ＊pd; .....整数部格納領域へのポインタ

[戻り値]　●実数の小数部を返します。

perrror

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 _far perror(s);

[実現方法]　関数

[引数]　const char ＊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 _far pow(x,y);

[実現方法]　関数

[引数]　double x; ..... 被乗数
double y; ..... 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 _far printf( format, argument... );
```

[実現方法]　関数

[引数]　const char ＊format; .....書式指定文字列のポインタ

formatで与えられる文字列の%以降の指定は、次の意味を持ちます。[]内は省略可能です。書式の各指定については次のページで説明します。

書式：%[フラグ][最小フィールド幅][精度][修飾文字(l、L、又はh)]変換指定記号
書式例：%-05.8ld

[戻り値]
- 出力した文字数を返します。
- ハード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。
- 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。
- 引数にfarポインタを扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

printf - formatの書式指定

1.変換指定記号

●d、i

引数の整数を符号付き10進数に変換します。

●u

引数の整数を符号なし10進数に変換します。

●o

引数の整数を符号なし8進数に変換します。

●x

引数の整数を符号なし16進数に変換します。0AH～0FHに小文字の"abcdef"を使用します。

●X

引数の整数を符号なし16進数に変換します。0AH～0FHに大文字の"ABCDEF"を使用します。

●c

引数の文字を1文字(ASCII文字)で出力します。

●s

引数の文字列ポインタ(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)をデータバンクレジスタとオフセットという形式で出力します。(出力例：00:1205)

●n

n変換は引数の整数ポインタにそれまでに出力した文字数を格納します。引数は変換されません。

●e

doubleの引数を指数形式に変換します。形式は[-]d.dddddde±ddです。

●E

指数表示のeの代りにEが使用される以外は、eと同じ書式です。

●f

doubleの引数を[-]d.dddddd形式に変換します。

●g

doubleの引数をe又はfにより指定される形式に変換します。通常はf型の変換を行います。指数部が-4以下、又は精度が指数値以下の場合は、e型に変換されます。

●G

指数表示のeの代りにEが使用される以外は、gと同じ書式です。

printf - formatの書式指定

2.フラグ

●－

変換の結果を最小フィールド幅内で左寄せします。デフォルトは右寄せです。

●＋

符号付きの変換結果に＋又は－を付けます。デフォルトは、負の数のみ－を付けます。

●ブランク' '

デフォルトで符号付きの変換結果が符号なしになった場合、頭にブランク' 'を付けます。

●#

o変換では、頭に0を付けます。

x、X変換では0以外の時、頭に0x、0Xを付けます。

e、E、f変換では、常に小数点を付けます。

g、G変換では、常に小数点を付け、小数点以下の0も切り捨てずに出力します。

3.最小フィールド幅

●正の10進整数で最小のフィールド幅を指定します。

●変換結果の文字数が指定したフィールド数より少ない場合 左に埋め文字を挿入してフィールド幅を合わせます。

●デフォルトの埋め文字はブランク' 'です。頭に0を付けた整数でフィールド幅を指定した場合は、'0'を埋め文字とします。

●－フラグを指定した場合は、変換結果を右寄せし右に埋め文字を挿入します。この場合、埋め文字は常にブランク' 'です。

●最小フィールド幅にアスタリスク(*)を指定した場合、引数の整数がフィールド幅を指定します。引数の値が負の場合、－フラグの後に正のフィールド幅を指定したことになります。

4.精度

'.'の後に正の整数で指定し、'.'のみの場合はゼロを指定したことになります。機能、及びデフォルト値は変換タイプにより異なります。

精度の指定がない浮動小数点型データの出力は、精度6で処理されます。ただし、精度が0のときは小数点は出力されません。

●d、i、o、u、x、X変換の場合

・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。

・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。

・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。

・デフォルト値は1です。

・ゼロを最小桁数0で変換した場合、何も出力しません。

printf - formatの書式指定

●s変換の場合
・最大文字数を表します。
・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。
・デフォルトには文字数制限がありません。
・精度の指定にアスタリスク(＊)を指定した場合⇒引数の整数が精度を指定します。
・引数の値が負の場合⇒精度の指定は無効になります。
●e、E、f変換の場合
・小数点の後にn個(nは精度)の数字を出力します。
●g、G変換の場合
・n個(nは精度)以上の有効数字を出力しません。

5.l、L、又はh
●lの場合、d、i、o、u、x、X、n変換をlong int又はunsigned long intの引数に対して行います。
●hの場合、d、i、o、u、x、X変換をshort int又はunsigned short intの引数に対して行います。
●l、hをd、i、o、u、x、X、n変換以外で指定した場合、指定が無視されます。
●Lの場合、e、E、f、g、G変換をdoubleの引数に対して行います。※1

※1.標準のC言語の仕様では、Lの場合、e、E、f、g変換をlong doubleの引数に対して行います。NC79では、long doubleはdoubleとして扱いますので、Lを指定した場合doubleの引数として扱います。

putc

入出力関数

[機能]　ストリームに1文字を出力します。

[書式]
```
#include <stdio.h>
int _far putc( c, stream );
```

[実現方法]　マクロ

[引数]
```
int c; 出力する文字
FILE＊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。
- 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。
- streamをfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far 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。
- 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。

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 _far puts( str );
```

[実現方法]　マクロ

[引数]　char ＊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。
- 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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

qsort

整数算術関数

[機能]　配列をソートします。

[書式]
```
#include <stdlib.h>
void _far qsort( base,nelen,size,cmp( e1,e2) );
```

[実現方法]　関数

[引数]
```
void ＊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]
- 引数をfarで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 _far 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

メモリ操作関数

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

[実現方法]　関数

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

[戻り値]
●大きさが変更された領域へのポインタを返します。
●指定した大きさの領域が確保できなかった場合にはNULLを返します。

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

scanf

入出力関数

[機能]　stdinからの書式付き入力を行います。

[書式]
```
#include <stdio.h>
#include <ctype.h>
int _far scanf( format, argument... );
```

[実現方法]　関数

[引数]　char ＊format; .....書式指定文字列のポインタ

formatで与える文字列の%以降の指定は、次の意味を持ちます。[]内は省略可能です。書式の各指定については次のページで説明します。

書式：%[＊][最大フィールド幅][修飾子(l、L、又はh)]変換指定記号
書式例：%*5ld

[戻り値]
- 各引数argumentに格納したデータ数を返します。
- stdinからデータとしてEOFを入力した場合、EOFを返します。

[解説]
- formatの指定に従ってstdinからの入力文字を変換し、各引数argumentが示す変数に格納します。
- argumentは各変数の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。
- 1A16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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラファイルを作り直してください。

scanf - formatの書式設定

1.変換指定記号

●d

符号付きの10進数を変換します。対応する引数は整数へのポインタでなければいけません。

●i

符号付きの10進数、8進数、16進数の入力を変換します。8進数は先頭が0、16進数は先頭が0x、0Xで始まります。対応する引数は整数へのポインタでなければいけません。

●u

符号なし10進数を変換します。対応する引数は、符号なし整数へのポインタでなければいけません。

●o

符号付き8進数を変換します。対応する引数は整数へのポインタでなければいけまん。

●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。また頭の0xは付けられません。対応する引数は整数へのポインタでなければいけません。

●s

ヌル文字'\0'までの文字列を格納します。対応する引数はヌル文字'\0'を含む文字列を格納するのに、充分な大きさを持つ文字配列を指すポインタでなければいけません。

最大フィールド幅に達して入力を中止した場合は、それまでの文字にヌル文字を付加した文字列を格納します。

●c

文字を格納します。スペース文字は読み飛ばさずにそのまま格納します。最大フィールド幅で2以上を指定した場合は、複数の文字を格納します。ただし、この場合はヌル文字'\0'は付加しません。対応する引数は文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。

●p

データバンクレジスタとオフセットという形式 (例：00:1205)の入力を変換します。対応する引数はすべての型のポインタへのポインタです。

●[]

[]内の文字(複数が指定可能)を入力している間、入力文字を格納します。[]内の文字以外を入力した場合、格納を中止します。また[の次に ^ (サーカムフレックス)を指定した場合は、^ と]の間で指定した文字以外が入力許可文字となります。指定した文字を入力した場合、格納を中止します。

対応する引数は自動的に付加するヌル文字'\0'を含む文字列を格納するのに充分な大きさを持つ文字配列を指すポインタでなければいけません。

●n

書式変換で既に読み込まれた文字数を格納します。対応する引数は整数へのポインタでなければいけません。

●e、E、f、g、G

浮動小数点の形式に変換します。修飾子lを指定したとき、対応する引数はdoubleへのポインタとなります。デフォルトはfloatへのポインタです。

scanf - formatの書式設定

2.＊(データ格納の抑止指定)

＊が指定されている場合、変換したデータを引数に格納することを抑止します。

3.最大フィールド幅

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

4.l、L、又はh

- lの場合、d、i、o、u、xの変換結果をlong int又はunsigned long intとして格納します。また、e、E、f、g、Gの変換結果をdoubleとして格納します。
- hの場合、d、i、o、u、xの変換結果をshort int又はunsigned short intとして格納します。
- l、hをd、i、o、u、x変換以外で指定した場合、指定が無視されます。
- Lの場合、e、E、f、g、Gの変換結果をfloatとして格納します。

setjmp

実行制御関数

[機能]　関数呼び出し時の環境の退避を行います。

[書式]　#include <setjmp.h>
int _far setjmp(env);

[実現方法]　関数

[引数]　jmp_buf _far ＊env; ..... 環境を退避する領域へのポインタ

[戻り値]　●longjmpの引数で与えられた数値を返します。

[解説]　●"env"で示される領域に環境の退避を行います。

setlocale

地域化関数

[機能]　プログラムのロケール情報の設定と検索を行います。

[書式]　#include <locale.h>

[実現方法]　char _far ＊setlocale(category,locale);

[引数]　関数

[戻り値]　int category; ..... ロケール情報、検索部情報
const char ＊locale; ..... ロケール情報文字列へのポインタ

[解説]　●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 _far 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 _far sinh( x );
```

[実現方法]　関数

[引数]　double x; .....実数

[戻り値]　●"x"の双曲線サインを返します。

sprintf

入出力関数

[機能]　文字列への書式付き出力を行います。

[書式]　int _far sprintf(pointer, format, argument...);

[実現方法]　関数

[引数]　char ＊pointer; ..... 格納先のポインタ
const char ＊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。
●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、printfと同様です。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリを作り直してください。

sqrt

数学関数

[機能]　数値の平方根を計算します。

[書式]　#include <math.h>
double _far 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 _far 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 _far sscanf( string, format, argument... );
```

[実現方法]　関数

[引数]　const char ＊string; ..... 入力文字列のポインタ
const char ＊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。
●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。
●引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

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 ＊ _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
char _far ＊s2; .....連結する文字列へのポインタ

[戻り値]　●連結された文字列領域(s1)へのポインタを返します。

[解説]　●"s1"で示される文字列の最後に、"s2"で示される文字列を連結します。※1
●連結された文字列は、NULLで終了します。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _far strchr(s, c);

[実現方法]　関数

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

[戻り値]　●文字列"s"中で最初に現れた文字"c"の位置を返します。
●文字列"s"が文字"c"を含まない場合にはNULLを返します。

[解説]　●"s"で示される領域の先頭より、"c"で示される文字を検索します。
●'\0'も検索対象とすることができます。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

※1.s1はs1とs2が充分に入る大きさが必要です。

strcmp

文字列操作関数

[機能]　2つの文字列の比較を行います。

[書式]
```
#include <string.h>
int _far strcmp( 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]
●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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far 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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 * _far strcpy( s1, s2 );
```

[実現方法]　関数

[引数]
char _far *s1; .....複写先の文字列へのポインタ
const char _far *s2; .....複写元の文字列へのポインタ

[戻り値]
●複写先の文字列へのポインタを返します。

[解説]
●"s1"で示される領域に"s2"で示される(NULLで終了している)文字列を複写します。
●複写先の文字列は、NULLで終了します。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

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 _far 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。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

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 _far 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。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

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 _far 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 ＊ _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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far 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
[戻り値] size_t n; ..... 比較する文字数

[解説]
●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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _far strncpy( s1, s2, n );
```

[実現方法]
関数

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

[戻り値]　●複写先の文字列へのポインタを返します。

[解説]
●"s1"で示される領域に、"n"で指定された文字数の文字を、"s2"で示される文字列より複写します。ただし、この時"n"で指定された文字数より多い部分は複写されず、その後に'\0'を付加することも行いません。逆に、"s2"の示す文字列の長さが"n"文字より短い場合には、複写された文字列の後に"n"文字になるまで'\0'が付加されます。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far strnicmp( s1, s2, n );
```

[実現方法]　関数

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

[戻り値]
●戻り値==0 ..... 2つの文字列は等しい
●戻り値>0 ..... 1番目の文字列(s1)の方が大きい
●戻り値<0 ..... 2番目の文字列(s2)の方が大きい

[解説]
●NULLで終了している2つの文字列を"n"文字分バイト単位に比較します。ただし、この時英字の大文字と小文字は区別せず小文字は大文字に変換して比較します。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _far strrchr( s, c );
```

[実現方法]　関数

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

[戻り値]
●文字列"s"中で最後に現れた文字"c"の位置を返します。
●文字列"s"が文字"c"を含まない場合にはNULLを返します。

[解説]
●"s"で示される領域の末尾より、"c"で示される文字を検索します。
●'\0'も検索対象とすることができます。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

strspn

文字列操作関数

[機能]　指定外文字列の長さを求めます。

[書式]
```
#include <string.h>
size_t  _far 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。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

strtod

文字列数値変換関数

[機能]　文字列を倍精度浮動小数に変換します。

[書式]
```
#include <string.h>
double _far strtod( s,endptr );
```

[実現方法]　関数

[引数]
const char＊s; ..... 変換文字列
char ＊＊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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 ＊ _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]
●"s1"で示される領域の先頭より、"s2"で示される文字列中の文字が最初に現れた位置(ポインタ)を返します。
●最初の呼び出しでは、最初の字句の先頭文字へのポインタを返します。このとき返される文字の最後には、NULL文字が書き込まれます。その後の呼び出し("s1"がNULLの場合)では、順次、次の字句が返されます。"s1"に字句がなくなるとNULLを返します。
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far strtol( s,endptr,base );
```

[実現方法]　関数

[引数]
const char＊s; ..... 変換文字列
char＊＊endptr; ..... 変換されなかった残りの文字列へのポインタ
int base; ..... 読み込む数値の基数(0～36)
※ 0 の場合には整数定数の形式を読み込みます。

[戻り値]
●戻り値 == 0L ..... 数を構成しません。
●戻り値 != 0L ..... 構成した数をlong型で返します。

[解説]
●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数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 _far strtoul( s,endptr,base );
```

[実現方法]　関数

[引数]
```
const char＊s ..... 変換文字列
char＊＊endptr; ..... 変換されなかった残りの文字列へのポインタ
int base; ..... 読み込む数値の基数( 0～36 )
※ 0 の場合には整数定数の形式を読み込みます。
```

[戻り値]
```
●戻り値 == 0L ..... 数を構成しません。
●戻り値 != 0L ..... 構成した数をunsigned long型で返します。
```

[解説]　●オプション-O,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

strxfrm

文字列数値変換関数

[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 _far strxfrm( s1,s2,n );
```

[実現方法]　関数

[引数]
```
char＊s1; ..... 変換結果文字列格納領域へのポインタ
const char＊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,-OR及び-OSを指定した場合は、最適化によりコード効率の良い関数を選択する可能性があります。

tan

数学関数

[機能]　タンジェントを計算します。

[書式]
```
#include <math.h>
double _far 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 _far tanh( x );
```

[実現方法]　関数

[引数]　double x; ..... 実数

[戻り値]　●ラジアンを単位とする"x"の双曲線タンジェントを返します。

tolower

文字操作関数

[機能]　英大文字を英小文字に変換します。

[書式]
```
#include <ctype.h>
int tolower( c );
```

[実現方法]　マクロ

[引数]　int c; 変換する文字

[戻り値]
●引数が英大文字の場合、英小文字を返します。
●それ以外は渡した引数を返します。

[解説]　●引数の英大文字を英小文字に変換します。

toupper

文字操作関数

[機能]　英小文字を英大文字に変換します。

[書式]
```
#include <ctype.h>
int toupper( c );
```

[実現方法]　マクロ

[引数]　int c; 変換する文字

[戻り値]
●引数が英小文字の場合、大文字を返します。
●それ以外は渡した引数を返します。

[解説]　●引数の英小文字を英大文字に変換します。

ungetc

入出力関数

[機能]　ストリームへ1文字戻します。

[書式]
```
#include <stdio.h>
int _far ungetc( c, stream );
```

[実現方法]　関数

[引数]
int c; 戻す文字
FILE ＊stream; ..... ストリームのポインタ

[戻り値]
- 正常ならば戻した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。
- 引数をfarポインタで扱う場合は、メイクファイルmake.far（Windows版はmakefar.dos）を使用してライブラリファイルを作り直してください。

vfprintf

入出力関数数

[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 _far vfprintf( stream,format,ap );
```

[実現方法]　関数

[引数]
FILE ＊stream; ..... ストリームのポインタ
const char ＊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]
- 出力した文字列を返します。

[解説]
- 引数をfarポインタで扱う場合は、メイクファイルmake.far(Windows版はmakefar.dos)を使用してライブラリファイルを作りなおしてください。

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 _far vprintf( format,ap );
```

[実現方法]　関数

[引数]
const char ＊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]　●出力した文字列を返します。

[解説]　●引数をfarポインタで扱う場合は、メイクファイルmake.far(Windows版はmakefar.dos)を使用してライブラリファイルを作りなおしてください。

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 _far vfprintf( s,format,ap );
```

[実現方法]　関数

[引数]
char ＊s; ..... 格納先へのポインタ
const char ＊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]　●出力した文字列を返します。

[解説]　●引数をfarポインタで扱う場合は、メイクファイルmake.far(Windows版はmakefar.dos)を使用してライブラリファイルを作りなおしてください。

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 ＊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 ＊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 _far wctomb( s,wchar );
```

[実現方法]　関数

[引数]
```
char ＊s; ..... 変換マルチバイト文字格納領域へのポインタ
wchar_t wchar; ..... ワイド文字
```

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

E.2.4 標準関数ライブラリの使用に関する注意事項

a.標準ヘッダファイルに関する注意事項

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

b.標準関数ライブラリの使用に関する注意事項

標準関数ライブラリ中で、ポインタを戻り値にもつもの、もしくはポインタを引数にもつものは、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、ポインタのnear・far属性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。

また、入出力関数ライブラリ（printf, sprintf等）のポインタ型の引数はデフォルトでnear属性になっています。可変長の引数にfar属性のポインタを積むプログラムでは、src79/libディレクトリ内のmake.farを使用してライブラリファイルをつくり直してください。【図E.1】にmakeコマンドの実行例を示します。

```
% make -f make.far <RET>

[記号説明]
%:プロンプト
<RET>:リターンキーの入力を意味します。
```

図E.1 makeコマンドの実行方法例

c.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、-OS、-ORのいずれかを指定した場合、標準関数に関する最適化を行ないます。この最適化は、-Ono_stdlibを指定することにより抑止することができます。ユーザー関数として、標準ライブラリ関数と同名の関数を使用する時には、この最適化を抑止してください。

(1)関数のインライン埋め込み

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

表E.13 標準ライブラリ関数に対する最適化条件

関数名	最適化条件	記述例
strcpy	第1引数：nearポインタ 第2引数：文字列定数	strcpy(str, "sample");
memcpy	第1引数：nearポインタ 第2引数：文字列定数 第3引数：定数	memcpy(str ,"sample", 6);

E.3 標準入出力関数ライブラリのカスタマイズ

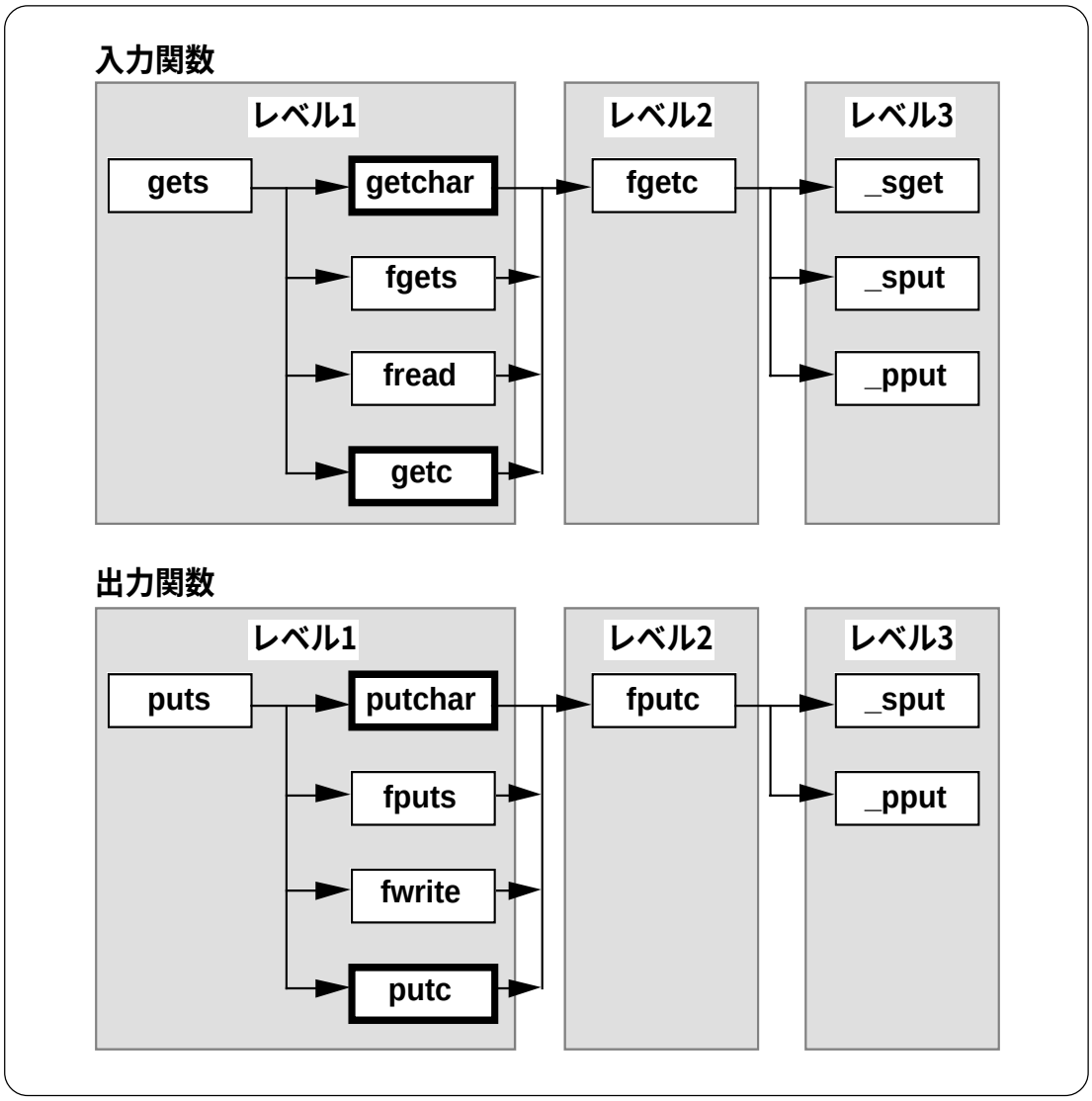
入出力関数としてscanf、printf等の高機能な関数ライブラリを用意しています。これらの関数は一般的に高水準入出力関数と呼ばれます。高水準入出力関数は、ハードウェア環境に依存した低水準入出力関数の組み合わせで実現しています。

7900シリーズの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.2】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。

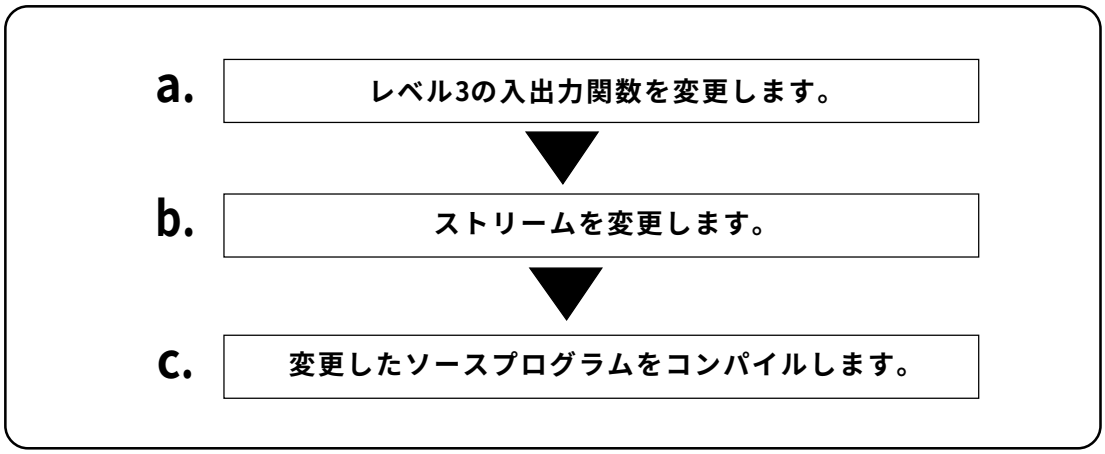
7900シリーズのハードウェア(入出力ポート)に依存しているのは最下位のレベル3の関数のみです。入出力関数をアプリケーションプログラムで使用する場合、必要に応じてレベル3の関数のソースファイルを書き換えることで、システムに順応した関数に変更できます。



図E.2 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.3】に示します。



図E.3 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、7900シリーズの入出力ポートに対して1バイトの入出力を行う関数です。レベル3の入出力関数は、シリアル通信回路(UART)に対して入出力を行う_sget、_sputと、セントロニクス仕様の通信回路に対して入出力を行う_pputがあります。

[回路の諸設定]
●プロセッサ動作モード：マイクロプロセッサモード
●クロック発信周波数：8MHz
●外部バス幅：16ビット

[シリアル通信の初期設定]
●UART1を使用
●ボーレート：9600bps
●データ長：8ビット
●パリティ：なし
●ストップビット：2ビット
※これらのシリアル通信の初期設定はinit関数(init.c)中で設定されています。

レベル3の入出力関数は、C言語で記述されたライブラリのソースファイルdevice.cに記述しています。レベル3の関数の仕様を【表E.14】に示します。

表7.1 レベル3の関数の仕様

入力関数	引数	戻り値(int型)
_sget	なし	正常に入力できた場合は入力した文字を返します エラーの場合はEOFを返します
_sput		
_pput		
出力関数	**引数(int型)**	**戻り値(int型)**
_sput	出力する文字	正常に出力できた場合は1を返します。 エラーの場合はEOFを返します。
_pput		

シリアル通信は、7900シリーズが持つ2本のUARTの内UART1に設定しています。また、システムのクロック発信周波数は、8MHzに設定しています。device.cでは、条件コンパイルコマンドでUARTとクロック発信周波数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
●16MHzのクロックを使用する場合 ..... #define CLOCK_16 1

をdevice.cファイルの先頭で記述します。

2本のUARTを使用する場合は、以下の手順で変更します。

①device.cファイルの先頭に記述している条件コンパイルの記述を削除します。
②#pragma EQUで定義されているUART0の特殊レジスタ名をUART1と異なった変数に書き換えます。
③レベル3の関数_sget、_sputをUART0用にそれぞれ複製し、_sget0、_sput0等の異なった関数名に書き換えます。
④speed関数もUART0用に複製し、speed0等の異なった関数名に書き換えます。

以上でdevice.cファイルの変更は終了します。

次に、入出力関数の初期設定を行っているinit関数(init.c)を変更し、ストリームの設定を変更します。このストリームの設定方法は、次節で説明します。

b.ストリームの設定

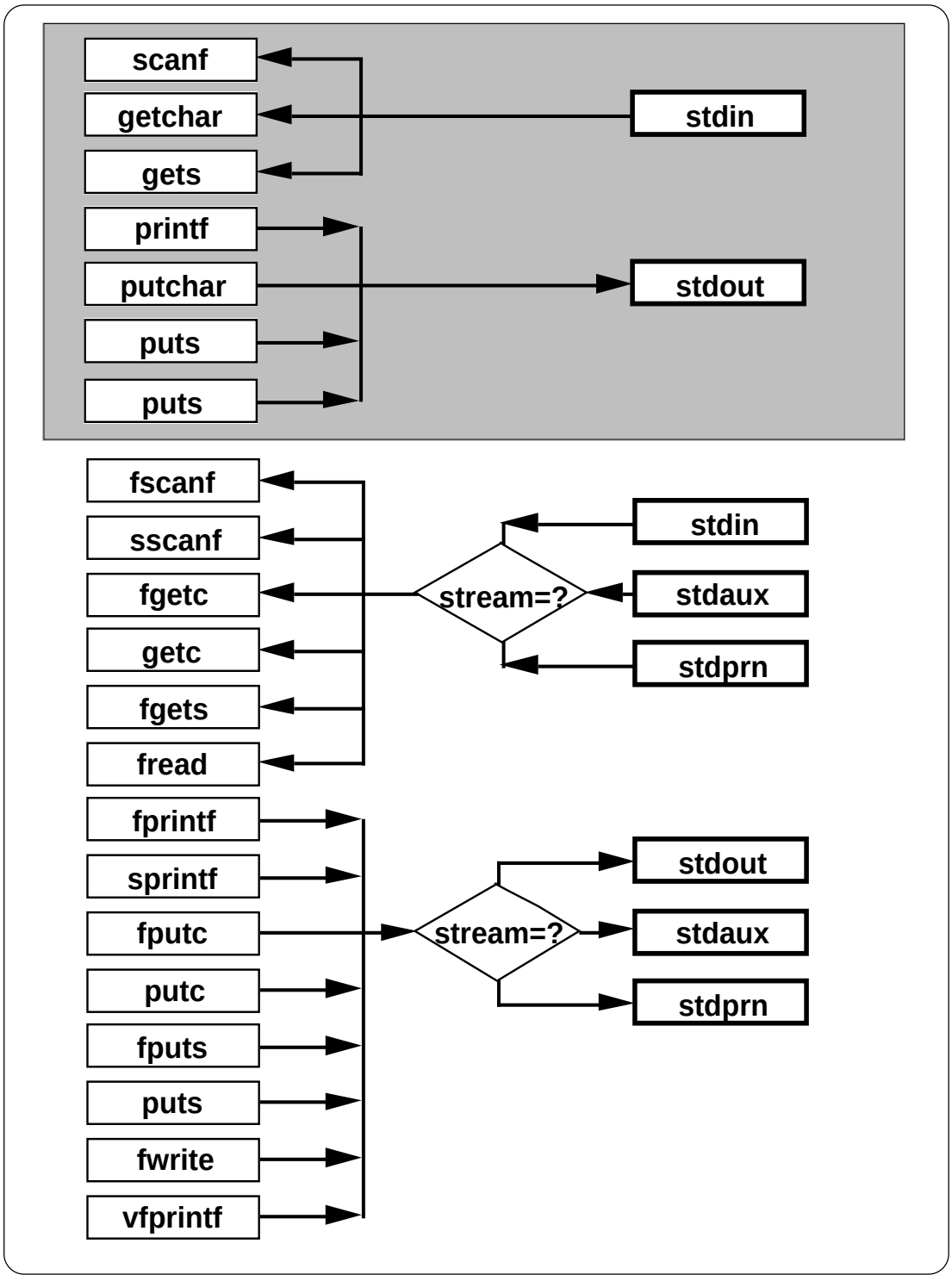
NC79の標準ライブラリはstdin、stdout、stderr、stdaux、stdprnの5種類のストリーム情報を外部構造体として持っています。この外部構造体は標準ヘッダファイルstdio.h内で定義されており、各ストリームのモード情報(入力ストリームか出力ストリームを表すフラグ)やステータス情報(エラー又はEOFを表すフラグ)等を管理しています。

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

NC79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.4 関数とストリームの相互関係図

【図E.5】にstdio.h内のストリーム定義部を示します。

```
/*****************************************************************************
*
* standard I/O header file
*
* stdio.h -- Version 1.00.00
* $Id: stdio.h,v 1.7 99/08/17 18:13:22 usuki Exp $
*
* Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
* AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
* All Rights Reserved.    *
*****************************************************************************/
#ifndef __STDIO_H
#define __STDIO_H

/*********************************************************
*   ANSI conform definition
*********************************************************/
#define _IOFBF    0x0
#define _IOLBF    0x40
#define _IONBF    0x04

#define BUFSIZ    256    /* Stream buffer size for setbuf */
#define EOF       (-1)   /* End of file */

#ifndef _IOBUF_DEF
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
#endif

#define FILENAME_MAX 63
#define FOPEN_MAX 8
#define  L_tmpnam sizeof("\\")+8
#ifndef NULL
#define NULL      0
#endif
#define SEEK_CUR 1
#define SEEK_END 2
#define SEEK_SET 0
#define TMP_MAX 32767
#ifndef _FPOS_T_DEF
typedef long fpos_t;
#define _FPOS_T_DEF
#endif

#ifndef _SIZE_T_DEF
typedef unsigned int size_t;
#define _SIZE_T_DEF
#endif
extern FILE _iob[];
#define  stdin   (&_iob[0])        /* Fundamental input */
#define  stdout  (&_iob[1])        /* Fundamental output */
#define  stdaux  (&_iob[2])        /* Fundamental auxiliary input output */
#define  stdprn  (&_iob[3])        /* Fundamental printer output */

#define  stderr  stdout           /* NC79 no-support */

        :
    （以降省略）
        :
```

図E.5 stdio.h内のストリーム定義部(stdio.h)

【図E.4】に示しましたファイル構造体の要素を以下に説明します。説中の①～⑥は、【図E.4】中の①～⑥に対応しています。

①char _buff

関数scanf、fscanfでは入力の際に1文字分の先読みを行っています。先読みした文字が不用な場合は関数ungetcを呼び出して、先読みした文字をこの変数に格納します。

入力関数はこの変数にデータが存在する場合はこのデータを入力データとします。

②int _cnt

_buffのデータ数を格納します(0もしくは1)。

③int _flag

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

●_IOREAD、_IOWRT、_IORW

ストリームの動作モードを指定するフラグです。これらのフラグは、ストリームの初期化部分で設定します。

●_IOEOF、_IOERR

入出力関数内でEOF、エラーの発生に応じて設定されます。

④int _mod

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

●テキストモード

入出力データのエコーバック、文字変換を行います。エコーバック、文字変換の詳細については、関数fgetc、fputcのソースプログラム(fgetc.c、fputc.c)を参照ください。

●バイナリモード

入出力データを無変換で扱います。これらのフラグはストリームの初期化部分で設定します。

⑤int (＊_func_in)()

ストリームが読み込みモード(_IOREAD)又は読み書き両用モード(_IORW)の場合、レベル3入力関数のポインタを格納します。それ以外の場合はNULLポインタを格納します。この情報をもとに、レベル2入力関数はレベル3入力関数を間接呼び出しで呼び出します。

⑥int (＊_func_out)()

ストリームが書き込みモード(_IOWRT)の場合、レベル3出力関数のポインタを格納します。また、ストリームが入力可能な場合(_IOREAD又は_IORW)で、かつテキストモードの場合、エコーバックするためのレベル3出力関数のポインタを格納します。それ以外の場合は、NULLポインタを格納します。

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

ストリームの初期化ではchar _buff以外のすべての要素に値を設定してください。NC79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.a79から呼び出されています。

【図E.6】にinit関数のソースプログラムを示します。

```
#pragma LOADDT init
#include <stdio.h>

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.6 init関数のソースファイル(init.c)

7900シリーズの2本のUARTを使用するシステムでは、init関数を以下の手順で変更します。前節では、device.cソースファイル中でUART0用の関数を仮に_sget0、_sput0、speed0と設定しました。

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

以上の設定で、2本のUARTが使用できます。ただし、標準入出力のストリームを使用する関数はUART0が使用する標準補助入出力に対して使用することができません。したがって、関数の引数にストリームを記述できる関数で使用してください。【図E.7】にinit関数の変更例を示します。

```
void init( void )
{
        :
    (省略)
        :
    stdaux->_flag = _IORW;          ←②(読み書き両用モードに設定)
        :
    (省略)
        :
    stdaux->_mod = _TEXT;           ←②(テキストモードに設定)
        :
    (省略)
        :
    stdaux->_func_in = _sget0;      ←③(UART0用のレベル3の入力関数を指定)
        :
    (省略)
        :
    stdaux->_func_out = _sput0;     ←③(UART0用のレベル3の出力関数を指定)
        :
    (省略)
        :
    speed0(_96, _B8, _PN, _S2);     ←④(UART0用のspeed関数を指定)
    speed(_96, _B8, _PN, _S2);
    init_prn();
}
```

※図中の②～④は、上記の設定手順の項番に対応しています。

図E.7 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又はmake.far(Windows版(PC版)はmakefile.dos又はmakefar.dos))を使用してライブラリファイルを更新します。

手順の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.8】に、②の方法を【図E.9】にそれぞれ示します。

```
% nc79 -c -g -osample ncrt0.a79 device.r79 init.r79 sample.c <RET>

※この例は、device.cとinit.cを変更した時の記述方法です。
```

図E.8 変更したソースプログラムを直接リンクする方法

```
C:\> make <RET>
```

図E.9 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

NC79が出力するすべてのエラーメッセージとワーニングメッセージ、そのメッセージに対する対処方法を説明します。

F.1 メッセージの出力形式

NC79は、処理中にエラーを検出すると、エラーメッセージを画面に表示した後コンパイルを中止します。
以下にエラーメッセージとワーニングメッセージの出力形式を示します。

```
nc79:[エラーメッセージ]
```

図F.1 コンパイルドライバnc79のエラー出力形式

```
[Error(cpp79.エラー番号):ファイル名,行番号]エラーメッセージ
[Error(ccom):ファイル名,行番号]エラーメッセージ
[Fatal(ccom):ファイル名,行番号]エラーメッセージ          ←※1
```

図F.2 各コマンドのエラー出力形式

```
[Warning(cpp79.ワーニング番号):ファイル名,行番号]ワーニングメッセージ
[Warning(ccom):ファイル名,行番号]ワーニングメッセージ
```

図F.3 各コマンドのワーニング出力形式

次項から各コマンドのエラーメッセージとワーニングメッセージの内容と対処策を示します。cpp79のメッセージは番号順で、その他のコマンドのメッセージはアルファベット順(記号、英字)で掲載しております。

※1. 致命的エラーの場合のメッセージです。

F.2 nc79エラーメッセージ

【表F.1】と【表F.2】にコンパイルドライバnc79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 nc79エラーメッセージ一覧表(1)

エラーメッセージ	エラー内容と対策
Arg list too long	●各処理系を起動するときのコマンドラインがシステムで定義された文字数を超えています。 ⇒システムで定義された文字数を超えないようにNC79のオプションを指定してください。各処理系のコマンドラインは、-vオプションで確認することができます。
Cannot analyze error	●通常は発生しません(内部エラーです)。 ⇒弊社までご連絡ください。
command-file line characters exceed 2048.	●コマンドファイルの１行の文字数が2048文字を越えています。 ⇒コマンドファイルの１行の文字数を2048文字以下にしてください。
Core dump(command_name)	●処理系がCore Dumpを起こしました。カッコ内はCore dumpを起こした処理系です。 ⇒各処理系が正しく実行されていません。環境変数又は各処理系が存在するディレクトリを確認してください。その上で正しく起動しない場合は、弊社にご連絡ください。
Exec format error	●各処理系の実行ファイルが壊れています。 ⇒再度、インストールしなおしてください。
Ignore option '-?'	●NC79で使用できないオプション-?を使用しています。 ⇒正しいオプション指定してください。
illegal option	●-as79や-ln79などの各処理系に指示するオプションが100文字を越えました。 ⇒各処理系に指示するオプションは99文字までにしてください。
Invalid argument	●通常は発生しません(内部エラーです)。 ⇒弊社までご連絡ください。
Invalid option '-?'	●-?オプションに必要なパラメータがありません。 ⇒-?オプションの次に必要なパラメータを指定してください。 ●-?オプションと必要なパラメータの間にスペースがあります。 ⇒-?オプションとパラメータ間のスペースを削除してください。
Invalid option '-o'	●-oオプションの次に出力ファイル名がありません。 ⇒出力ファイル名を指定してください。ファイル名は拡張子を指定しないでください。

表F.2 nc79エラーメッセージ一覧表(2)

エラーメッセージ	エラー内容と対策
Invalid suffix '.xxx'	●NC79が認識できないファイル拡張子(.c,.i,.a79,.r79,.x79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。
No such file or directory	●各処理系が実行できません。 ⇒各処理系が格納されているディレクトリを環境変数で正しく設定しているかを確認してください。
Not enough core	[UNIX版] ●スワップ領域が不足しています。 ⇒セカンダリスワップを追加するなど、スワップ領域を増やしてください。 [MS-Windows 95/NT版] ●メモリ空間が不足しています。 ⇒メモリを増やすか、Windows95/NTのスワップ空間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。
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 cpp79エラーメッセージ

【表F.3】～【表F.6】にプリプロセッサcpp79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 cpp79エラーメッセージ一覧表(1)

番号	エラーメッセージ	エラー内容と対策
1	illegal command option	●入力ファイル名を2回指定しています。 ⇒入力ファイル名の指定を1つにしてください。
11	cannot open input file	●入力ファイル名と出力ファイル名が同じ名前です。 ⇒出力ファイル名は入力ファイル名と違う名前を指定してください。
		●出力ファイル名を2回指定しています。 ⇒出力ファイル名の指定を1つにしてください。
		●コマンドラインが-oオプションで終了しています。 ⇒-oオプションの後に出力ファイル名を指定してください。
		●インクルードファイルのパスを指定する-Iオプションが制限値を越えました。 ⇒-Iオプションを8個以下にしてください。
		●コマンドラインが-Iオプションで終了しています。 ⇒-Iオプションの後に出力ファイル名を指定してください。
		●-Dオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は_)でありません。不当なマクロ名定義を行っています。 ⇒正しいマクロ名、正しいマクロ定義を指定してください。
		●コマンドラインが-Dオプションで終了しています。 ⇒-Dオプションの後に出力ファイル名を指定してください。
		●-Uオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は_)でありません。 ⇒正しいマクロ名定義を指定してください。
		●CPP79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。
		●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。

表F.4 cpp79エラーメッセージ一覧表(2)

番号	エラーメッセージ	エラー内容と対策
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。
17	input file name buffer overflow	●入力ファイル名のバッファがオーバーフローしました。ファイル名は、ディレクトリパス名を含みますので注意してください。 ⇒ファイル名、パス名（標準ディレクトリ、-Iオプション指定）を短くしてください。
18	not enough memory for macro identifier	●マクロ名、綴り文字を登録するためのメモリが足りません。 [UNIX版] ⇒スワップ領域を増やしてください。 [MS-Windows版　MS-DOS版] ⇒拡張メモリを増やしてください。
21	include file not found	●インクルードファイルがオープンできません。 ⇒インクルードファイルは、カレントディレクトリ、-Iオプションや環境変数で指定したディレクトリに存在します。これらのディレクトリを確認してください。
22	illegal file name error	●ファイル名の指定が誤っています。 ⇒ファイル名の指定を正しく行ってください。
23	include file nesting over	●インクルードファイルのネスティングの上限（8）を越えました。 ⇒インクルードファイルのネスティングを8までにしてください。
25	illegal identifier	●#defineの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
26	illegal operation	●プリプロセスコマンド#if～#elseif～#assertの演算式中に誤りがあります。 ⇒演算式を正しく記述してください。
27	macro argument error	●マクロ展開時のマクロ引数の数に誤りがあります。 ⇒マクロ定義及び参照を確認して正しく記述してください。
28	input buffer over flow	●ソースファイルリード中に入力行バッファがオーバーフローしました。又は、マクロ変換中にバッファがオーバーフローしました。 ⇒ソースファイルの1行を1023文字以下にしてください。マクロ変換を予想される場合は、変換結果が1023文字以下になるように変更してください。

表F.5 cpp79エラーメッセージ一覧表(3)

番号	エラーメッセージ	エラー内容と対策
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。 ⇒CPP79で使用できるプリプロセスコマンドは、次のコマンドのみです。 #include、#define、#undef、#if、#ifdef、#ifndef、#else、#endif、#elseif、#line、#assert、#pragma、#error
33	new_line in string	●文字定数又は、文字列定数中に改行が含まれています。 ⇒プログラムを正しく記述しなおしてください。
34	string literal out of range 509 characters	●文字列が509文字をこえました。 ⇒文字列は509文字以下にてください。
35	macro replace nesting over	●マクロのネスティングが制限値(20)を越えました。 ⇒制限値を越えないようにしてください。
41	include file error	●#include命令の記述に誤りがあります。 ⇒正しく記述してください。
43	illegal id name	●#defineコマンドの以下のマクロ名又は引数の記述に誤りがあります。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒ソースファイルを正しく記述してください。
44	token buffer over flow	●#defineの綴り文字のバッファがオーバーフローしました。 ⇒綴り文字を短くしてください。
45	illegal undef command usage	●#undefの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
46	undef id not found	●#undefで未定義にしようとしている以下のマクロ名が定義されていません。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒マクロ名を確認してください。
52	illegal ifdef / ifndef command usage	●#ifdefの記述に誤りがあります。 ⇒ソースファイルを正しく記述してください。
53	elseif / else sequence error	●#if～#ifdef～#ifndefがないのに#elseif又は#elseを使用しています。 ⇒#elseif又は#elseは#if～#ifdef～#ifndefの後に使してください。
54	endif not exist	●#if～#ifdef～#ifndefに対応した#endifがありません。 ⇒#endifをソースファイル中に記述してください。

表F.6 cpp79エラーメッセージ一覧表(4)

番号	エラーメッセージ	エラー内容と対策
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 cpp79ワーニングメッセージ

【表F.7】にプリプロセッサcpp79が出力するワーニングメッセージとその内容及び対処方法を示します。

表F.7 cpp79ワーニングメッセージ一覧表

番号	ワーニングメッセージ	ワーニング内容と対策
81	reserved id used	●CPP79が予約済みの以下のマクロ名を定義又は未定義にしようとしています。 __FILE__、__LINE__、__DATE__、__TIME__ ⇒他のマクロ名を使用してください。
82	assertion warning	●#assertの演算式の結果が0になりました。 ⇒演算式を確認してください。
83	garbage argument	●プリプロセスコマンドの後にコメント以外の文字があります。 ⇒プリプロセスコマンドの後の文字はコメント(/＊.....＊/)で記述してください。
84	escape sequence out of range for character	●文字定数、文字列定数に含まれるエスケープ文字が255越えました。 ⇒エスケープ文字は255までにしてください。
85	redefined	●一度定義したマクロを以前定義したときと異なる内容で再度定義しています。 ⇒以前定義した内容と比較して確認してください。
87	/* within comment	●コメント内に/*を記述しています。 ⇒ネストしないようにコメントを記述してください。

F.5 ccom79エラーメッセージ

【表F.8】～【表F.16】にコンパイラccom79が出力するエラーメッセージとその内容及び対処方法を示します。

表F.8 ccom79エラーメッセージ一覧表(1)

エラーメッセージ	エラー内容と対策
#pragma PARAMETER 関数名 re-defined	●#pragma PARAMETERにおいて同じ関数を重複して 定義しています。 ⇒#pragma PARAMETERの宣言を1回にしてください。
#pragma PARAMETER & function prototype mismatched	● #pragma PARAMETER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 PARAMETER's function argument is struct or union	●#pragma PARAMETER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 PARAMETER must be declared before use	●#pragma PARAMETER宣言で指定した関数の定義が、その関数の呼び出しの後に記述されています。 ⇒関数の呼び出しを行う前に宣言してください。
'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。 ⇒far/near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。
'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。

表F.9 ccom79エラーメッセージ一覧表(2)

エラーメッセージ	エラー内容と対策
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 1 $$ or $@	●asmステートメントで$$または$@を2回以上記述しています。 ⇒$$($@)の記述を1回にしてください。
auto variable's size is zero	●auto領域に要素数が0の配列、あるいは要素数のない配列を宣言しています。 ⇒正しく宣言してください。
bitfield width exceeded	●ビットフィールドの幅が、データ型のビット幅を超えています。 ⇒ビットフィールドで宣言したデータ型のビット幅以内で記述してください。
bitfield width is not constant integer	●ビットフィールドのビット幅が定数ではありません。 ⇒ビット幅を定数で記述してください。
can't get bitfield address by '&' operator	●ビットフィールドタイプに&演算子を記述しています。 ⇒ビットフィールドタイプに&演算子を記述しないでください。
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 void value	●代入式の右辺がvoid型等のように、void型のデータを取り出そうとしています。 ⇒データの型を確認してください。
can't output to ファイル名	●ファイルに書き込みができません。 ⇒ディスクの残り容量又はファイルのパーミッションを確認してください。
can't open ファイル名	●ファイルがオープンできません。 ⇒ファイルのパーミッションを確認してください。
can't set argument	●プロトタイプ宣言と実引数との型の不一致により、レジスタ(引数)に実引数をセットできません。 ⇒型の不一致を修正してください。
can't 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。

表F.10 ccom79エラーメッセージ一覧表(3)

エラーメッセージ	エラー内容と対策
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 'long'	●long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。
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。
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。

表F.11 ccom79エラーメッセージ一覧表(4)

エラーメッセージ	エラー内容と対策
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。
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。
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 strage 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 strage 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 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 initialized	●メンバが確定していない(不完全な)構造体,共用体を初期化しています。 ⇒完全な構造体,共用体を先に定義してください。
incomplete struct return function call	●メンバが確定していない(不完全な)構造体,共用体の型をリターン値にもつ、関数を呼び出しています。 ⇒完全な構造体,共用体を先に定義してください。
incomplete struct / union's member access	●メンバが確定していない(不完全な)構造体,共用体のメンバを参照しています。 ⇒完全な構造体,共用体を先に定義してください。
incomplete struct / union(タグ名)'s member access	●メンバが確定していない(不完全な)構造体,共用体のメンバを参照しています。 ⇒完全な構造体,共用体を先に定義してください。
inline function'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。
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採れていない場合等に発生します。整合が採れるように記述してください。

表F.12 ccom79エラーメッセージ一覧表(5)

エラーメッセージ	エラー内容と対策
invalid push	●関数引数等で、void型をpushしています。 ⇒voidをoushすることはできません。
invalid ' ? : ' operand	●?:演算子の記述に誤りがあります。 ⇒演算子の各式を確認してください。また、:の左右の式の型は、互換型である必要があります。
invalid '!=' operands	●!=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '&&' operands	●&&演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '&' operands	●&演算子の記述に誤りがあります。 ⇒演算子の右辺式を確認してください。
invalid '&=' operands	●&=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '()' operand	●( )の左辺式が関数ではありません。 ⇒( )の左辺式は関数又は関数へのポインタを記述してください。
invalid '＊' operands	●乗算の場合＊演算子の記述に誤りがあります。ポインタ演算子の場合、右辺式がポインタ型でありません。 ⇒乗算の場合、演算子の左辺式,右辺式を確認してください。ポインタの場合、右辺の型を確認してください。
invalid '＊=' operands	●＊=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '+' operands	●+演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '+=' operands	●+=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '-' operands	●-演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '-=' operands	●-=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '/=' operands	●/=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<<' operands	●<<演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<<=' operands	●<<=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<=' operands	●<=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '=' operands	●=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '==' operands	●==演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>=' operands	●>=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>>' operands	●>>演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>>=' operands	●>>=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。

表F.13 ccom79エラーメッセージ一覧表(6)

エラーメッセージ	エラー内容と対策
invalid '[ ]' operands	●[ ]の左辺式が配列,ポインタ型ではありません。 ⇒[ ]の左辺式は配列,又はポインタ型を記述してください。
invalid '^=' operands	●^=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '|=' operands	●|=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '||' operands	●||演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '%=' operands	●%=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid ++ operands	●++単項演算子又は後置演算子の記述に誤りがあります。 ⇒単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。
invalid -- operands	●--単項演算子又は後置演算子の記述に誤りがあります。 ⇒単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。
invalid -> used	●->の左辺式が、構造体,共用体型ではありません。 ⇒->の左辺式を、構造体,共用体型で記述してください。
invalid (? ;)'s condition	●三項演算子の記述に誤りがあります。 ⇒三項演算式を確認してください。
invalid CAST operand	●cast演算子に誤りがあります。void型を他の型にキャスト及び、構造体,共用体からもしくは他の構造体、共用体へのキャストはできません。 ⇒正しく記述してください。
invalid asm()'s argument	●asmステートメントに使用できる変数は、auto変数と引数です。 ⇒auto変数か引数で記述してください。
invalid bitfield declare	●ビットフィールドの宣言で誤りがあります。 ⇒正しく記述してください。
invalid break statements	●break文を記述できないところで使用しています。 ⇒switch,while,do-while,forのなかで記述してください。
invalid case statements	●switch文に誤りがあります。 ⇒switch文を正しく記述してください。
invalid case value	●caseの値に誤りがあります。 ⇒整数型,列挙型の定数を記述してください。
invalid cast operator	●cast演算子の記述に誤りがあります。 ⇒正しく記述してください。
invalid continue statements	●continue文を記述できないところで使用しています。 ⇒while,do-while,forのなかで記述してください。
invalid default statements	●switch文に誤りがあります。 ⇒switch文を正しく記述してください。

表F.14 ccom79エラーメッセージ一覧表(7)

エラーメッセージ	エラー内容と対策
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。 ⇒左辺式に代入可能な式を記述してください。
invalid lvalue at '=' operator	●代入文の左辺が、lvalueではありません。 ⇒左辺式に代入可能な式を記述してください。
invalid member	●メンバ参照の記述に誤りがあります。 ⇒正しく記述してください。
invalid member used	●メンバ参照の記述に誤りがあります。 ⇒正しく記述してください。
invalid redefined type name　of (識別子)	●typedefで同じ識別子名を2回以上定義しています。 ⇒識別子名を正しく記述してください。
invalid return type	●関数の戻り値の型が正しくありません。 ⇒正しく記述してください。

表F.15 ccom79エラーメッセージ一覧表(8)

エラーメッセージ	エラー内容と対策
invalid sign specifier	●signed/unsignedを2回以上記述しています。 ⇒型指定子を正しく記述してください。
invalid strage class for data	●記憶クラスの指定に誤りがあります。 ⇒正しく記述してください。
invalid struct or union type	●列挙型のデータに対して構造体,共用体のメンバ参照しています。 ⇒正しく記述してください。
invalid truth expression	●条件式(?:)の1つめの式でvoid,struct,union型を使用してします。 ⇒スカラ型で記述してください。
invalid type specifier	●int int i; 等のように同じ型指定子を2回以上記述しているか、float int i; 等のように矛盾した型指定子を記述しています。 ⇒型指定子を正しき記述してください。
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 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 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。
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。
No initialized of xxx	●register変数 xxx の初期化を行っていません。 ⇒register変数の初期化を行ってください。
Not enough memory	[UNIX版] ●スワップ領域が不足しています。 ⇒スワップ領域を増やしてください。 [MS-Windows95/NT版 ●メモリ空間が不足しています。 ⇒メモリを増やすか、Windows95/NTのスワップ空間を増やしてください。 [MS-DOS版] ●拡張メモリが不足しています。 ⇒拡張メモリを増やしてください。

表F.16 ccom79エラーメッセージ一覧表(9)

エラーメッセージ	エラー内容と対策
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 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。
Sorry stack fram memory exhaust, max. 64(or 255) bytes but now nnn bytes	●スタックフレームは最大64バイト、または255バイト(起動オプション-fDPO8指定時)までです。現在 nnnバイト使用しています。 ⇒auto変数などのスタックフレーム領域に確保される変数を減らしてください。
Sorry, compilation terminated because of these errors in関数名	●関数名で示される関数でエラーが発生しました。コンパイルを中止します。 ⇒このメッセージが出力される以前のエラーを修正してください。
Sorry, compilation terminated because of too many errors.	●ソースファイル中のエラーがエラーの上限(50個)を超えました。 ⇒このメッセージが出力される以前のエラーを修正してください。
struct or enum's tag used for union	●構造体,列挙型のタグ名を共用体のタグ名として使用しています。 ⇒タグ名を変更してください。
struct or union's tag used for enum	●構造体,共用体のタグ名を列挙型のタグ名として使用しています。 ⇒タグ名を変更してください。
struct or union,enum does not have long or sign	●struct/union/enum型指定子にlongやsignedの型指定子を記述しています。 ⇒型指定子を正しく記述してください。
switch's condition is floating	●switch文の式に浮動小数点型を使用しています。 ⇒整数型,列挙型を使用してください。

表F.17 ccom79エラーメッセージ一覧表(10)

エラーメッセージ	エラー内容と対策
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)。 ⇒メッセージの内容を弊社までご連絡ください。
too many storage class of typedef	●宣言中にextern/typedef/static/auto/registerなどの記憶域クラス指定子を2以上記述しています ⇒記憶域クラス指定子を2回以上指定しないでください。
type redeclaration of 変数名	●1度目と2度目で型の異なる重複定義を行っています。 ⇒変数を2度宣言する場合は同じ型で行ってください。
typedef initialized	●typedefで宣言した変数に初期化式を記述しています。 ⇒初期化式を削除してください。
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 valuable "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 valuable 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 valuable 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。

表F.18 ccom79エラーメッセージ一覧表(11)

エラーメッセージ	エラー内容と対策
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。
zero size array member	●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。。

F.6 ccom79ワーニングメッセージ

【表F.19】～【表F.28】にコンパイラccom79が出力するワーニングメッセージとその内容及び対処方法を示します。

表F.19 ccom79ワーニングメッセージ一覧表(1)

ワーニングメッセージ	ワーニング内容と対策
#pragma プラグマ名 & INTERRUPT both specified	●１つの関数に#pragmaプラグマ名と#pragmaINTERRUPTの両方を指定しています。 ⇒#pragmaプラグマ名と#pragmaINTERRUPTは、排他的に指定してください。
#pragma プラグマ名 & TASK both specified	●１つの関数に#pragmaプラグマ名と#pragmaTASKの両方を指定しています。 ⇒#pragmaプラグマ名と#pragmaTASK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。処理を続行します。 ⇒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 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 ASM format error, ignored	●#pragmaASM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 ASM line too long, then cut	●#pragmaASMで記述できる一行の文字数1024バイトを越えています。 ⇒1024バイト以内で記述してください。
#pragma directive conflict	●一つの関数に対して、異なる機能の#pragmaを指定しています。 ⇒正しく記述してください。

表F.20 ccom79ワーニングメッセージ一覧表(2)

ワーニングメッセージ	ワーニング内容と対策
#pragma ADDRESS format error, ignored	●#pragmaADDRESS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 ADDRESS variable initialized, ADDRESS ignored	●#pragmaADDRESSで指定した変数を初期化しています。#pragmaADDRESSの指定を無視します。 ⇒#pragmaADDRESSか、初期化式のどちらかを削除してください。
#pragma CYCHANDLER format error, ignored	●#pragmaCYCHANDLER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 EQU format error,ignored	●#pragmaEQU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 HANDLER format error, ignored	●#pragmaHANDLER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 INTERRUPT format error, ignored	●#pragmaINTERRUPT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 LOADDT format error, ignored	●#pragmaLOADDT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 M1FUNCTION format error, ignored	●#pragmaM1FUNCTION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 PARAMETER & HANDLER both specified	●１つの関数に#pragmaPARAMETERと#pragmaHANDLERの両方を指定しています。 ⇒#pragmaPARAMETERと#pragmaHANDLERは、排他的に指定してください。
#pragma PARAMETER & INTERRUPT both specified	●１つの関数に#pragmaPARAMETERと#pragmaINTERRUPTの両方を指定しています。 ⇒#pragmaPARAMETERと#pragmaINTERRUPTは、排他的に指定してください。
#pragma PARAMETER & TASK both specified	●１つの関数に#pragmaPARAMETERと#pragmaTASKの両方を指定しています。 ⇒#pragmaPARAMETERと#pragmaTASKは、排他的に指定してください。
#pragma PARAMETER format error	●#pragmaPARAMETERの記述に誤りがあります。処理を続行します。 ⇒正しく記述してください。
#pragma PARAMETER format error, ignored	●#pragmaPARAMETER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。

表F.21 ccom79ワーニングメッセージ一覧表(3)

ワーニングメッセージ	ワーニング内容と対策
#pragma PARAMETER not function,ignored	●#pragmaPARAMETER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 PARAMETER function's address used	●#pragmaPARAMETERで指定された関数のアドレスをポインタ変数に代入しています。 ⇒代入しないで、正しく記述してください。
#pragma PARAMETER's function must be predeclared,ignored	●#pragmaPARAMETERで指定した関数が、宣言されていません。 ⇒#pragmaPARAMETER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 PARAMETER's function must be prototyped,ignored	●#pragmaPARAMETERで指定した関数が、宣言されていません。 ⇒#pragmaPARAMETER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 PARAMETER's function return type invalid,ignored	●#pragmaPARAMETER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 ROM format error,ignored	●#pragmaROM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 SECTION format error, ignored	●#pragmaSECTION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 STRUCT format error, ignored	●#pragmaSTRUCT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 TASK format error,ignored	●#pragmaTASK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。
'auto' is illegal storage class	●不当な記憶域クラスを使用しています。 ⇒正しい記憶域クラスを指定してください。
'register' is illegal storage class	●不当な記憶域クラスを使用しています。 ⇒正しい記憶域クラスを指定してください。
argument is define by 'typedef', 'typedef' ignored	●引数の宣言においてtypedefを使用しています。typedefを無視します。 ⇒typedefを削除してください。
assign far pointer to near pointer, bank value ignored	●farポインタをnearポインタに代入しています。バンクアドレスを無効にします。 ⇒データの型、near/farを確認してください。
assignment from const pointer to non-const pointer	●constポインタから非constポインタへの代入により、const性が失われます。 ⇒記述を確認してください。記述が正しい場合には、このワーニングは無視してください
assignment from volatile pointer to non-volatile pointer	●volatileポインタから非volatileポインタへの代入により、volatile性が失われます。 ⇒記述を確認してください。記述が正しい場合には、このワーニングは無視してください

表F.22 ccom79ワーニングメッセージ一覧表(4)

ワーニングメッセージ	ワーニング内容と対策
block level extern valuable 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 registerstorage class valuable	●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を正しく記述してください。
can't get size of function,unit size 1 assumed	●関数へのポインタを++,--しています。増分、減分の値を1として処理を継続します。 ⇒関数へのポインタを++,--しないでください。
char array initialized by wchar_t string	●char型の初期化式をwchar_t型の文字列で初期化しています。 ⇒初期化式の型を合わせてください。
case value is out of range	●caseの値がswitchの引数の範囲を越えてます。 ⇒正しく記述してください。
character buffer overflow	●文字列のサイズが512文字を超えました。 ⇒511文字以下で記述してください。
character constant too long	●文字定数（シングルクォートに囲まれた文字）の文字数が多すぎます。 ⇒正しく記述してください。
constant valuable assignment	●const型修飾子で指定した変数に対して代入しています。 ⇒代入先の宣言部を確認してください。
cyclic or alarm handler always Bank 0	●#pragmaCYCHANDLER又はALMHANDLERで指定した関数は、常にバンク0（アドレスが10000H未満）の領域にコンパイルされます。 ⇒ありません。
cyclic or alarm handler always load DT	●#pragmaCYCHANDLER又はALMHANDLERで指定した関数は、#pragmaLOADDTをする必要はありません。 ⇒#pragmaLOADDTを削除してください。
cyclic or alarm handler function has argument	●#pragmaCYCHANDLER又はALMHANDLERで指定した関数が、引数を使用しています。 ⇒#pragmaCYCHANDLER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。

表F.23 ccom79ワーニングメッセージ一覧表(5)

ワーニングメッセージ	ワーニング内容と対策
enumerator value overflow size of unsigned char	●-fCEオプション使用時において、列挙子の値が255を越えました。 ⇒255以下で表現できるように記述してください。
enumerator value overflow size of unsigned int	●列挙子の値が65535を越えました。 ⇒65535以下で表現できるように記述してください。
enum's bitfield	●ビットフィールドのメンバに列挙型を用いて定義しています。 ⇒違う型のメンバを用いてください。
external valuable 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。
handler function called	●#pragmaHANDLER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	●#pragmaHANDLERで指定した関数が、戻り値を使用しています。 ⇒#pragmaHANDLER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	●#pragmaHANDLERで指定した関数が、引数を使用しています。 ⇒#pragmaHANDLERで指定した関数は、引数を使用できません。引数を削除してください。
hex character is out of range	●文字定数においてHEX文字が長すぎます。また、\の後に16進数以外の文字が入っています。 ⇒HEX文字を短くしてください。
identifier (メンバ名) is dupilicated, this declare ignored	●メンバ名が重複して定義されています。この宣言を無視します。 ⇒メンバ名の宣言を1つにしてください。
identifier (変数名) is dupilicated	●変数名が重複して定義されています。この宣言を無視します。 ⇒変数名の宣言を1つにしてください。
identifier (変数名) is shadowed	●引数で宣言した変数名と同じ変数名のauto変数を使用しています。auto変数を無視します。 ⇒引数で使用した変数名以外を使用してください。
illegal storage class for argument, 'extern' ignore	●関数定義の引数リストにおいて、不当な記憶域クラスを使用しています。 ⇒正しい記憶域クラスを指定してください。
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。

表F.24 ccom79ワーニングメッセージ一覧表(6)

ワーニングメッセージ	ワーニング内容と対策
init elements overflow,ignored	●初期化式が初期化しようとする変数のサイズを超えました。 ⇒初期化式の数が、初期化する変数のサイズを超えないようにしてください。
inline function is called as normal funciton before, change to static function	●記憶クラスinlineで宣言された関数が通常の関数として呼び出されています。 ⇒inline関数は使用する前に必ず定義を行ってください。
integer constant is out of range	●整数定数の値がunsignedlongで表現できる値を超えました。 ⇒定数の値をunsignedlongで表現できる値で記述してください。
interrupt function called	●#pragmaINTERRUPT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	●#pragmaINTERRUPTで指定した割り込み処理関数が、戻り値を使用しています。 ⇒割り込み関数では戻り値を使用できません。戻り値を削除してください。
interrupt function has argument	●#pragmaINTERRUPT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 bank value (バンクアドレス)	●オプション-bankの値（バンクアドレス）の記述に誤りがあります。 ⇒バンクアドレスは、0から255の間で指定してください。
invalid #pragma EQU	●#pragmaEQU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	●#pragmaSECTION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 asm's M flag	●asmステートメントにおいて、Mフラグに設定する値に誤りがあります。 ⇒整数型の定数（0,1,2）で記述してください。
invalid asm's MX flag,ignored	●asmステートメントにおいて、MXフラグに設定する値が整数型の定数ではありません。 ⇒整数型の定数（0,1,2）で記述してください。
invalid 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 ccom79ワーニングメッセージ一覧表(7)

ワーニングメッセージ	ワーニング内容と対策
invalid asm's X flag	●asmステートメントにおいて、Xフラグに設定する値に誤りがあります。 ⇒整数型の定数(0,1,2)で記述してください。
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	●#pragmaADDRESS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 implicity cast to small type	●大きい型から小さい型への代入により、値の上位バイト(ワード)が失われる可能性があります。 ⇒型を確認してください。記述が正しい場合は、このワーニングを無視してください。
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。
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ってください。
No storage class & data type in declare, global storage class & int type assumed	●記憶域クラス指定子、型指定子なしに、変数を宣言しています。intとして処理します。 ⇒記憶域クラス指定子、型指定子を記述してください。
non-initialized variable ‘ 変数名 ’ is used	●初期化していない変数を参照している可能性があります。 ⇒記述を確認してください。このワーニングは、関数の最後の行で発生することもあります。この場合、関数内のauto変数の記述を確認してください。記述が正しい場合は、このワーニングは無視してください。

表F.26 ccom79ワーニングメッセージ一覧表(8)

ワーニングメッセージ	ワーニング内容と対策
non-prototyped function used	●プロトタイプ宣言していない関数を呼び出しています。-Wnon_prototypeオプションを指定した時のみ出力されます。 ⇒プロトタイプ宣言を行うか、-Wnon_prototypeオプションを削除してください。
non-propototyped function declared	●定義されている関数のプロトタイプ宣言が存在しません(-WNPオプション指定時のみ表示)。 ⇒プロトタイプ宣言を行ってください。
octal constant is out of range	●8進数の定数に8進数で使用できない文字が含まれています。 ⇒8進数の定数は0から7で記述してください。
octal_character is out of range	●8進数の定数に8進数で使用できない文字が含まれています。 ⇒8進数の定数は0から7で記述してください。
overflow in floating value converting to integer	●整数型には格納できない巨大な浮動小数点値を、整数型に代入しています。 ⇒代入式を再確認してください。
old style function declaration	●ANSI(ISO)C以前の形式で関数定義を記述しています。 ⇒ANSI(ISO)形式で関数定義を記述してください。
prototype function is defined as non-prototyped function before	●プロトタイプ宣言していない関数を再度プロトタイプ宣言しています。 ⇒関数の宣言方法を統一してください。
redefined type	●typedefで、定義済みの型名を再定義しています。 ⇒別の型名を使用するか、記述ミスがないか確認してください。
register parameter function used before as stack parameter function	●レジスタ引数の関数がいぜん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。
section name is renamed twice	●データのセクションにおいて、#pragmaSECTIONを用いて2回以上セクション名を変更しています。 ⇒データのセクションについては、セクション名の変更は１回のみ行ってください。
sorry, get stack's address , but DT not 0	●オプション-bankを指定した時に発生します。auto変数のアドレスをポインタに代入し、そのポインタでオブジェクトを参照した場合、DTがバンク0以外を示しているためバンク0を参照できません。 ⇒ポインタをfarで宣言してください。
size of incomplete type	●sizeof演算子のオペランドに定義されていない構造体、共用体を記述しています。 ⇒構造体、共用体を先に定義してください。
	●sizeof演算子のオペランドに定義されている配列の要素数が決定していません。 ⇒構造体、共用体を先に定義してください。
size of incomplete array type	●大きさが不明な配列のsizeofを求めています。無効なサイズです。 ⇒配列の大きさをしてください。

表F.27 ccom79ワーニングメッセージ一覧表(9)

ワーニングメッセージ	ワーニング内容と対策
size of void	●voidのサイズを求めています。無効なサイズです。 ⇒voidのサイズは求められません。
standard library“ 関数名() ” need “ インクルードファイル名 ”	●標準ライブラリ関数をヘッダファイルをインクルードしないで使用しています。 ⇒ヘッダファイルをインクルードしてください。
static variable in inline function	●記憶クラスinlineで宣言した関数内でstaticデータを宣言しています。 ⇒インライン関数内で、staticデータを宣言しないでください
string size bigger than array size	●初期化する変数のサイズより、初期化式のサイズが大きい。 ⇒初期化式のサイズは、変数と同じか、変数より小さくしてください。
string terminator not added	●配列の要素数と初期化式のサイズが同じであるため、文字列の最後に付加する'\0'は、付加しません。 ⇒配列の要素数を増やしてください。
struct (or union) member's address can't has no near far information	●構造体(もしくは共用体)のメンバー(変数)の配置位置情報としてnear,farを指定しています。 ⇒メンバーについてはnear,farは指定しないでください。
task function called	●#pragmaTASK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	●#pragmaTASKで指定した関数が、戻り値を使用しています。 ⇒#pragmaTASK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	●#pragmaTASKで指定した関数が、引数を使用しています。 ⇒#pragmaTASKで指定した関数は、引数を使用できません。引数を削除してください。
this comparison is always false	●常に偽になる比較を行っています。 ⇒条件式を確認してください。
this comparison is always true	●常に真になる比較を行っています。 ⇒条件式を確認してください。
this feature not supported now, ignored	●文法エラーです。この記述は、将来拡張用の文法ですので使用しないでください。 ⇒正しく記述してください。
this function used before with non-default argument	●使用されたことのある関数をデフォルト引数を持つ関数として宣言しています。 ⇒関数を使用する前にデフォルト引数を宣言してください。
this interrupt function is called as normal function before	●使用したことのある関数を#pragmaINTERRUPTで宣言した。 ⇒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。

表F.28 ccom79ワーニングメッセージ一覧表(10)

ワーニングメッセージ	ワーニング内容と対策
too few parameters	●プロトタイプ宣言で宣言した時より引数が足りません。 ⇒引数の数を確認してください。
too many parameters	●プロトタイプ宣言で宣言した時より引数が多すぎます。 ⇒引数の数を確認してください。
uncomplete 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。
unknown #pragma STRUCT xxx	●#pragmaSTRUCTxxxは、処理できません。この行を無視します。 ⇒正しく記述してください。
unknown debug option (-dx)	●オプション-dxは、指定できません。 ⇒オプションを正しく指定してください。
unknown function option (-Wxxx)	●オプション-Wxxxは、指定できません。 ⇒オプションを正しく指定してください。
unknown function option (-fx)	●オプション-fxは、指定できません。 ⇒オプションを正しく指定してください。
unknown function option (-gx)	●オプション-gxは、指定できません。 ⇒オプションを正しく指定してください。
unknown optimize option (-Ox)	●オプション-Oxは、指定できません。 ⇒オプションを正しく指定してください。
unknown option (-x)	●オプション-xは、指定できません ●。⇒オプションを正しく指定してください。
unknown pragma pragma-指示 used	●サポートされていない#pragmaを記述しています。 ⇒#pragmaの内容を確認してください。※この警告は-Wunknown_pragma(-WUP)、および、-Wallオプション指定時のみ表示されます。
wchar_t array initialized by char string	●wchar_t型の初期化式をchar型の文字列で初期化しています。 ⇒初期化式の型を合わせてください。
zero divide in constant folding	●除算演算子、剰余算演算子において除数が0です。 ⇒除数は、0以外を使用してください。
zero divide,ignored	●除算演算子、剰余算演算子において除数が0です ⇒除数は、0以外を使用してください。
zero width for bitfield	●ビットフィールドの幅が0です。 ⇒ビット幅を1以上で記述してください。
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。
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。
#pragma DP[n]DATA format error,ignored	●-fDPO8オプションを併用しています。 ⇒#pragmaDP[n}DATAと-fDPO8オプションを併用した場合、#pragmaDP[n]DATAは、無効となります。-fDPO8オプションを削除してください。
	●#pragmaDP[n]DATAのフォーマットが間違っています。 ⇒正しく記述してください。

付録G
DPnDATA宣言ユーティリティ(utl79)

DPnDATA宣言ユーティリィティutl79の起動方法と起動オプションの機能を説明します。

G.1　utl79の 概 要

G.1.1　utl79の 処 理 概 要

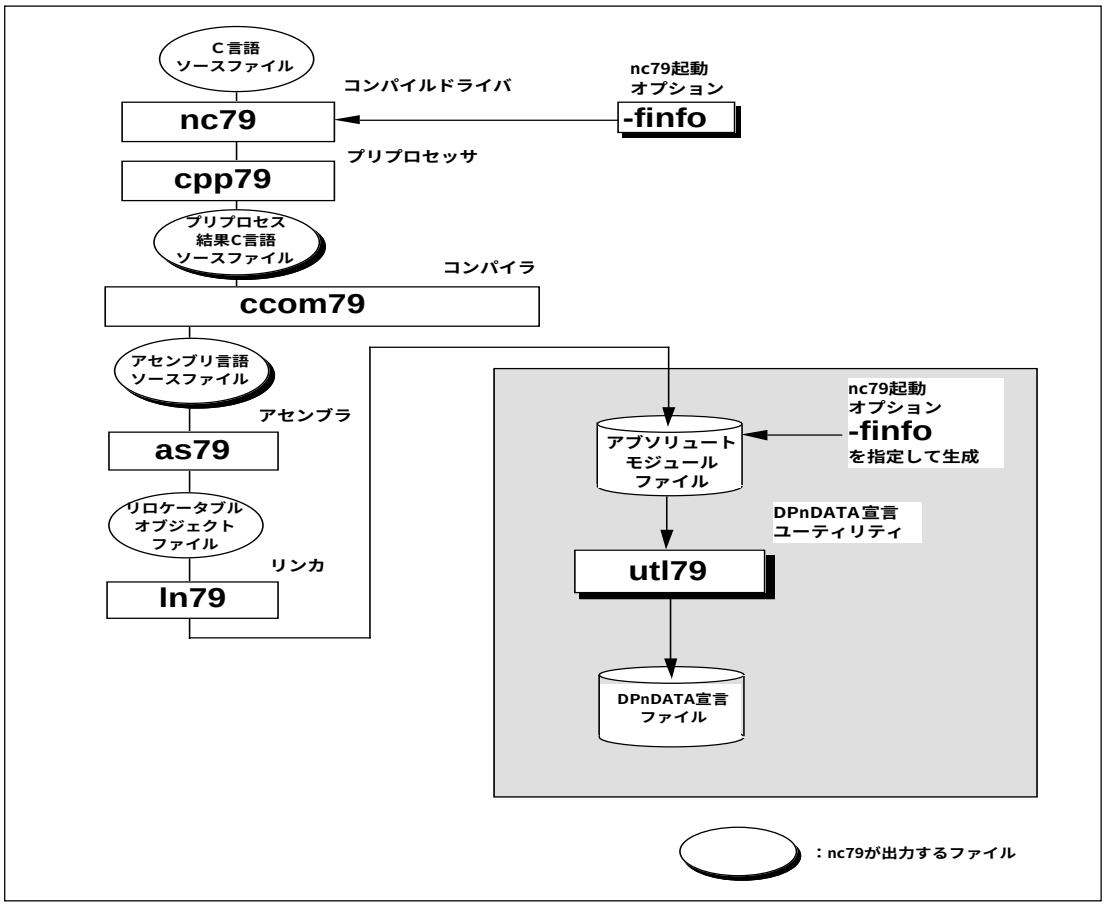
DPnDATA宣言ユーティリィティutl79は、アブソリュートモジュールファイル(拡張子 .x30)を処理して、

- DPnDATA宣言
  使用頻度の高い外部変数を#pragma DPnDATA(nは1～3)で宣言
  (#pragma　DPnDATA(nは1～3))

の結果を出力します。

utl79を起動するには、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 .x79)を生成してください。

NC79の処理フローを【図G.1】に示します。



図G.1 NC79の処理フロー

G.2　utl79の起動方法

G.2.1　入力書式

utl79を起動するためには、以下の【図G.2】に示す書式にしたがって、必要な情報、パラメータを指定する必要があります。

```
% utl79△[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 utl79コマンドの入力書式

utl79 を起動するために必要なファイル、

アブソリュートモジュールファイル(拡張子 .x79)

は、nc79の起動オプションに、
●アブソリュートモジュールファイル(拡張子 .x79)の出力-finfoオプション
●デバッグ情報の出力..... -gオプション
を指定することにより、生成されます。

以下の図に入力例を示します。入力例では utl79 に以下のオプションを指定しています。
●DPnDATA宣言ファイルの出力..... -oオプション
（デフォルトでは、標準出力への出力となっています。）
●DP1に割り当てるための指示.....-DP1オプション

```
% nc79 ncrt0.a79 -finfo sample.c<RET>
NC79 COMPILER for 7900 Series V.X.XX Release X
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a79
sample.c


% utl79 ncrt0.x79 -DP1 -o sample<RET>
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.


%
```

<RET>：リターンキーの入力を示します。

図G.3 utl79コマンドの入力例

G.2.2　オプションリファレンス

utl79 を起動するためには、以下の表に示す書式に従って、必要な情報、パラメータを指定する必要があります。

【表G.1】にutl79 の起動オプションを示します。

表G.1　utl30の起動オプション

オプション	短縮形	機　能
-DP/-DP1/-DP2/-DP	なし	DPnDATA宣言を行いたいダイレクトページレジスタ名を指定します。
-o	なし	DPnDATA宣言の結果をファイルに出力します。指定が無い場合は、ホストマシン(EWS又はパーソナルコンピュータ)の標準出力に出力します。拡張子は指定できません。 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。
-Tpointer	-TP	nearポインタ変数を対象にします。
-all	なし	全ての外部変数をDPnDATA宣言します。
-Wstdout	なし	エラー及びワーニングメッセージを標準出力へ出力します。
-fsection	なし	#pragma SECTIONでセクション変更された領域に配置された変数も処理の対象とします。
-fover_write	-fOW	-oオプションで指定された出力ファイルに対して強制的に上書きします。

-DP/-DP1/-DP2/-DP3

ダイレクトページレジスタの指定

[機 能]　DPnDATA宣言を行いたいダイレクトページレジスタ名を指定します。

[実 行 例]

```
% utl79 -DP1 sample.x79
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All  Rights  Reserved.

%
```

[注 意 事 項]　※本オプションは必須です。-DP、もしくは-DP1、-DP2、-DP3のいずれかを指定してください。
※-DPを指定した場合、ダイレクトページレジスタ1、ダイレクトページレジスタ2、ダイレクトページレジスタ3の順で変数を割り当てます。
※-DP1と-DP1、-DP2、-DP3の同時指定はできません。-DP1、-DP2、-DP3の同時指定は可能です。

-o*出力ファイル名*

DPnDATA宣言結果表示ファイルの生成

[機 能]　DPnDATA関数宣言の結果をファイルに出力します。指定が無い場合には表示をホストマシン（EWS又はパーソナルコンピュータ）の標準出力に出力します。
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。
-fover_write[-fOW]オプションを指定した場合には、強制的に上書きします。

[実行例]

```
% utl79 -osample    sample.x79
NC79 UTILITY UTL79 for 7900 Series V.1.00.00
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All  Rights  Reserved.

%
```

-Tpointer　　-TP

near型のポインタ変数に対する処理の指示

[機 能]　nearポインタ変数を utl79 で処理する変数の対象とします。

[実 行 例]

```
% utl79 -TP sample.x79
NC79 UTILITY UTL79 for for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All  Rights  Reserved.

%
```

-all

全変数のDPnDATA宣言の指示

[機 能]　すべてのグローバルデータをDPnDATA宣言します。

[実 行 例]

```
% utl79 -all sample.x79
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All  Rights  Reserved.

%
```

[注 意 事 項]　オプション-DP/-DP1/-DP2/-DP3で指定したダイレクトページレジスタ相対でアクセス可能な領域に入りきらない外部変数に対する宣言は、コメントアウトされた形式で出力されます。

-Wstdout

エラーメッセージの標準出力への表示

[機 能]　エラー、及び、ワーニングメッセージをホストマシンの標準出力に出力します。

[実 行 例]

```
% utl79 -osample.h sample.x79 -Wstdout
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

warning : cannot open file ‘ sample.x79 ’

%
```

-fsection

DPnDATAへの割り当て

[機 能]　#pragma SECTIONでセクション変更された領域に配置された変数も処理の対象にします。

[実 行 例]

```
% utl79 -osample -fsection sample.x79
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.



%
```

-fover_write　　-fOW

出力ファイルへの上書き

[機 能]　-oオプションで指定された出力ファイル名に対して強制的に上書きします。

[実 行 例]

```
% utl79 -osample -fover_write sample.x79
NC79 UTILITY UTL79 for 7900 Series V.X.XX.XX
Copyright 2000 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.



%
```

G.3　制限事項

utl79では、アセンブラで記述されたファイル中で宣言されている疑似命令.DPnSYMをカウントすることができません。したがって、アセンブリ言語で宣言された.DPnSYMがある場合には、utl79実行後に生成された結果に対して、ダイレクトページレジスタ相対でアクセス可能な領域に入るように調整する必要があります。

G.4　utl79が処理対象とする変数

utl79の処理対象となる変数は、以下の型を持つ外部変数に対してのみ処理を行います。

- unsigned char、signed char
- unsigned short、signed short
- unsigned int、signed int
- unsigned long、signed long

なお、上記の型を持つ変数でも下記の条件を満たす場合は、処理の対象外となります。

- #pragma SECTIONで変更されたセクションに配置された変数
- #pragma ADDRESSで宣言された変数
- #pragma ROMで宣言された変数

※near型のポインタ変数は、オプション-tpointer(-TP)を指定した場合に処理対象となります。

※すでにプログラム中で#pragma DPnDATAで宣言された変数が存在する場合は、utl79はその宣言を優先します。

G.5　utl79の使用例

G.5.1　DPnDATA宣言ファイルの生成

DPnDATA宣言ユーティリティutl79 にアブソリュートモジュールファイル（コンパイル時にオプション -finfo を使用）を処理させることにより、DPnDATAS宣言ファイルを出力することができます。【図G.4】にutl79の入力例を、【図G.5】にDPnDATA宣言ファイル例を示します。

```
% utl79 -DP1 sample.x79 -odpndata<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。
sample.x79          :マップファイル名です。
```

図G.4 utl79コマンドの入力例

```
/*
 *    #pragma DPnDATA Utility
  */
/*  DP1DATA  Size  [63]  */
#pragma DP1DATA  data3                  /*  size  =  (4)  /  ref  =  [ 1]   */
#pragma DP1DATA  data2                  /*  size  =  (2)  /  ref  =  [ 1]   */
#pragma DP1DATA  data1                  /*  size  =  (1)  /  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 DPnDATA宣言ファイル(dpndata.h)

上記方法により生成したDPnDATA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。DPnDATA宣言ファイルのインクルード例を【図G.6】に示します。

```
#include " dpndata.h "

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

図G.6 DPnDATA宣言ファイルのインクルード例

G.5.2　アセンブラでDPnDATA宣言がある場合の調整

アセンブリ言語プログラム中で、疑似命令.DPnSYMによりダイレクトページレジスタ相対でアクセス可能な領域を設定している場合は、utl79により生成されたファイルを調整する必要があります。

```
[アセンブリ言語プログラム]
            .DP1SYM       _sym
              :
            (省略)
              :
            .glp          _sym
      _sym:
            .blkb         2


[utl79生成ファイル]
      /*
       *      #pragma DPnDATA Utility
         */
      /*  DP1DATA  Size  [63]  */
      #pragma DP1DATA     data3                 /*  size  =  (4)  /  ref  =  [ 1]   */
      #pragma DP1DATA     data2                 /*  size  =  (2)  /  ref  =  [ 1]   */
      #pragma DP1DATA     data1                 /*  size  =  (1)  /  ref  =  [ 1]   */
      /*
       *      End  of  File
         */



アセンブリ言語ファイルにて２バイトデータ_symを.DP1SYMで宣言しているため、utl79が生成
したファイルから２バイト分のDPnDATA宣言を削除します。



削除例）
              :
            (省略)
              :
//コメントアウト
//#pragma DP1DATA       data2                 /*  size  =  (2)  /  ref  =  [ 1]   */
              :
            (省略)
              :
```

図G.7 utl79により生成されたファイルの調整例

G.6　utl79エラーメッセージ

G.6.1　エラーメッセージ

【表G.3】にDPnDATA宣言ユーティリィティutl79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.3 utl79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 '-?'	●utl79 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 'x79 ' 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 'x79 ' file 'ファイル名'	●マップファイルがオープンできません。 ⇒マップファイルを指定してください。
cannot close file 'ファイル名'	●該当する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。 ⇒コマンドラインより出力ファイルを指定してください。
not enough memory	●メモリが足りません。 ⇒メモリを増やしてください。
'-DP/-DP1/-DP2/-DP3' is missing	●オプション-DPまたは-DP1/-DP2/-DP3の指定がありません。 ⇒オプションを指定し直してください。
'-DP/-DP1/-DP2/-DP3' option multiple specified	●オプション-DPと-DP1/-DP2/-DP3が同時に指定されています。 ⇒オプションを指定し直してください。
since ' ファイル名 ' file exits.it makes a standard output	●-oオプションで指定された 'ファイル名 'が既に存在します。 ⇒出力ファイルを確認してください。 -fover_writeオプション指定で上書きになります。

G.6.2 ワーニングメッセージ

【表G.4】にDPnDATA宣言ユーティリィティutl79が出力するワーニングメッセージとその内容及び対処方法を示します。

表G.4 utl79ワーニングメッセージ一覧表

ワーニングメッセージ	ワーニング内容と対策
conflict declare of 変数名	●該当する変数が異なる記憶域クラス、型、#pragma宣言で重複定義を行っています。 ⇒変数を2度以上宣言する場合は、同じ記憶域クラス、型、#pragma宣言で行ってください。

付録H
NC79固有の注意事項

7900シリーズは、4本のダイレクトページレジスタ(DPR0～DPR3)を持っており、DPR0のみを使用するか、DPR0～DPR3を使用するかを「ダイレクトページレジスタ切り替えビット」で切り替えることができます。DPR0のみを使用する場合は256バイト、DPR0～DPR3を使用する場合は各64バイトのダイレクトページ領域を指定できます。

NC79は、自動変数、引数、および戻りアドレスなどをスタック上に確保するときのスタックフレームレジスタとしてDPR0、またはDPR0とDPR1を使用します。

ここでは、NC79のスタックフレーム、および標準ライブラリファイルについての注意事項を説明します。

H.1　スタックフレーム

H.1.1　スタックフレームサイズ

１関数で確保できるスタックフレームサイズの最大値は、DPR0のみを使用する場合とDPR0～DPR3を使用する場合とで異なります。

表H.1　スタックフレームサイズとダイレクトページレジスタ

DPR0のみを使用する場合	256バイト（実質255バイト）
DPR0～DPR3を使用する場合	64バイト（実質63バイト）、または128バイト（実質127バイト）

H.1.2　使用するDPR

NC79は、スタックフレームの最大バイト数をオプション指定の有無により切り替えることができます。

表H.2　スタックフレームサイズの切り替え（コンパイラのオプション）

-fDP_offset_8 [短縮系 -fDPO8]	DPR0のみを使用する場合のコードを生成します。使用するフレームレジスタはDPR0になります。スタックフレームの最大値は255バイトです。
-fauto_128 [短縮系 -fA1]	DPR0～DPR3を使用する場合のコードを生成します。使用するフレームレジスタはDPR0とDPR1になります。スタックフレームの最大値は127バイトです。
デフォルト	DPR0～DPR3を使用する場合のコードを生成します。使用するフレームレジスタはDPR0になります。スタックフレームの最大値は63バイトです。

-fDP_offset_8[短縮系 -fDPO8]オプションと-fauto_128[短縮系 -fA1]オプションは同時に指定することはできません。

なお、-fDP_offset_8[短縮系 -fDPO8]オプションを指定して作成したモジュールとと-fauto_128[短縮系 -fA1]オプション、または-fDP_offset_8[短縮系 -fDPO8]オプションを指定せずに作成したモジュールとをリンクした場合は、プログラムが誤動作する原因となります。

ファイル間でのオプションの整合性に注意してください。

H.2　標準ライブラリファイル

H.2.1　標準ライブラリファイルの種類

DPR0のみを使用する場合と、DPR0～DPR3を使用する場合とでは、使用できるニーモニックが異なります。このため、NC79では２種類の標準ライブラリを用意しています。

表H.3　標準ライブラリとダイレクトページレジスタ

nc79lib.lib	DPR0～DPR3を使用する場合に使用する標準ライブラリです。デフォルト、および -fauto_128[短縮系 -fA1]オプション指定時に使用します。
nc79m8.lib	DPR0のみを使用する場合に使用する標準ライブラリです。-fDP_offset_8[短縮系 -fDPO8]オプション指定時に使用します。

H.2.2　リンク時における標準ライブラリファイルの指定方法

標準ライブラリファイルは、リンクの仕方によりファイルの指定方法が異なります。

a.　NC79でリンクを制御する場合

標準ライブラリの指定はコンパイルオプションの指定により決定されます。

表H.3　ダイレクトページレジスタ関連のオプションと標準ライブラリ

デフォルト、および -fauto_128[短縮系 -fA1]オプション指定時	nc79lib.lib
-fDP_offset_8[短縮系 -fDPO8]オプション指定時	nc79m8.lib

b.　LN79でリンクを制御する場合

リンクオプション -l ライブラリファイル名 で標準ファイル名を指定する必要があります。この場合にはコンパイル時に指定したオプションとの整合性に注意してください。

H.3　ライブラリ関数のリエントラント性について

-fDP_offset_8[短縮系 -fDPO8]オプションの指定がない場合、1 関数のスタックフレームサイズは最大63バイト(-fauto_128[短縮系 -fA1]オプション指定時は127バイト)です。このため、スタックフレームサイズが64バイトを超えている以下の標準ライブラリ関数にはリエントラント性がありません。

abort	asin	atan	atan2	atof	atoi	bsearch
ceil	clearerr	cos	cosh	device	exp	fabs
feof	ferror	fflush	fgetc	fgets	floor	fmod
fprintf	fputc	fputs	fread	frexp	fscanf	fwrite
getc	getchar	gets	infinity	init	ldexp	log
log10	malloc	mblen	mbstowcs	mbtowc	modf	peeror
pow	print	printf	putc	putchar	puts	qsort
rand	scan	scanf	sin	sinh	sprint	sqrt
sscanf	strerror	strtod	strtod	tan	tanh	ungetc
vfprintf	vprintf	vsprintf	wcstombs	wctomb		

また、スタックフレームサイズが64バイトを超えているランタイムライブラリもリエントラント性がありません。このため、お客様作成の関数で浮動小数点データを扱っている場合は、その関数もリエントラント性がなくなる場合がありますのでご注意ください。

技術サポート連絡書

年　月　日（合計　　枚）

三菱電機セミコンダクタシステム株式会社
マイコンソフトツール部
開発ツールサポート窓口行

［電子メール］ support@tool.mesc.co.jp
［大阪地区］ FAX： 06-6398-6191
［東京地区］ FAX： 03-5783-7339
［中部地区］ FAX： 052-221-7318
［九州地区］ FAX： 092-452-1427

※インストーラが生成する以下のテキストファイルもサポート連絡書としてご利用できます。
Windows 98/95/Windows NT 4.0版の場合：\SUPPORT\製品名\SUPPORT.TXT
EWS版の場合：/support/製品名/toolinfo.txt

ご連絡先	製品情報
会社名：	ソフトウェア：
部署名：	バージョン番号：V.
担当者名：	ライセンスID：
電話番号：	– – – –
FAX番号：	ホストマシン：
電子メール：	OS： V.
通信欄：	

MEMO

NC79 V.4.10ユーザーズマニュアル

第1版：2000年12月16日発行
資料番号：MSD-NC79-U-001216

三菱電機株式会社
三菱電機セミコンダクタシステム株式会社

7900 シリーズ用　C コンパイラ
NC79 V.4.10　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