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

77XXシリーズ用 Cコンパイラ

NC77 V.5.20　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　1999.11

《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。

NC77 V.5.20ユーザーズマニュアル

目　次

目次

はじめに

NC77は、三菱16ビットマイクロコンピュータ7700用のCコンパイラです。NC77は、C言語で記述したプログラムを7700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

NC77 V.5.20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
nc77	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
NC77	NC77 V.5.20の製品名、又は製品の総称を意味します。
rasm77	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。
RASM77	RASM77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	NC77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

NC77 V.5.20 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。

NC77 V.5.20

ユーザーズマニュアル

第１章
NC77の処理概要

この章では、NC77が行うコンパイル処理の概要と、NC77を使用したプログラム開発の事例を説明します。

1.1　NC77の構成

NC77は、以下に示す４つの実行ファイルで構成されています。

1.nc77 ..... コンパイルドライバ
2.cpp77 ..... プリプロセッサ
3.ccom77 ..... コンパイラ
4.loop77 ..... ブランチオプティマイザ
5.s2ie IEEE-695アブソリュート形式ファイルコンバータ
6.stk77 ..... スタックサイズ算出ユーティリィティ

1.2　NC77の処理フロー

NC77の処理フローを【図1.1】に示します。

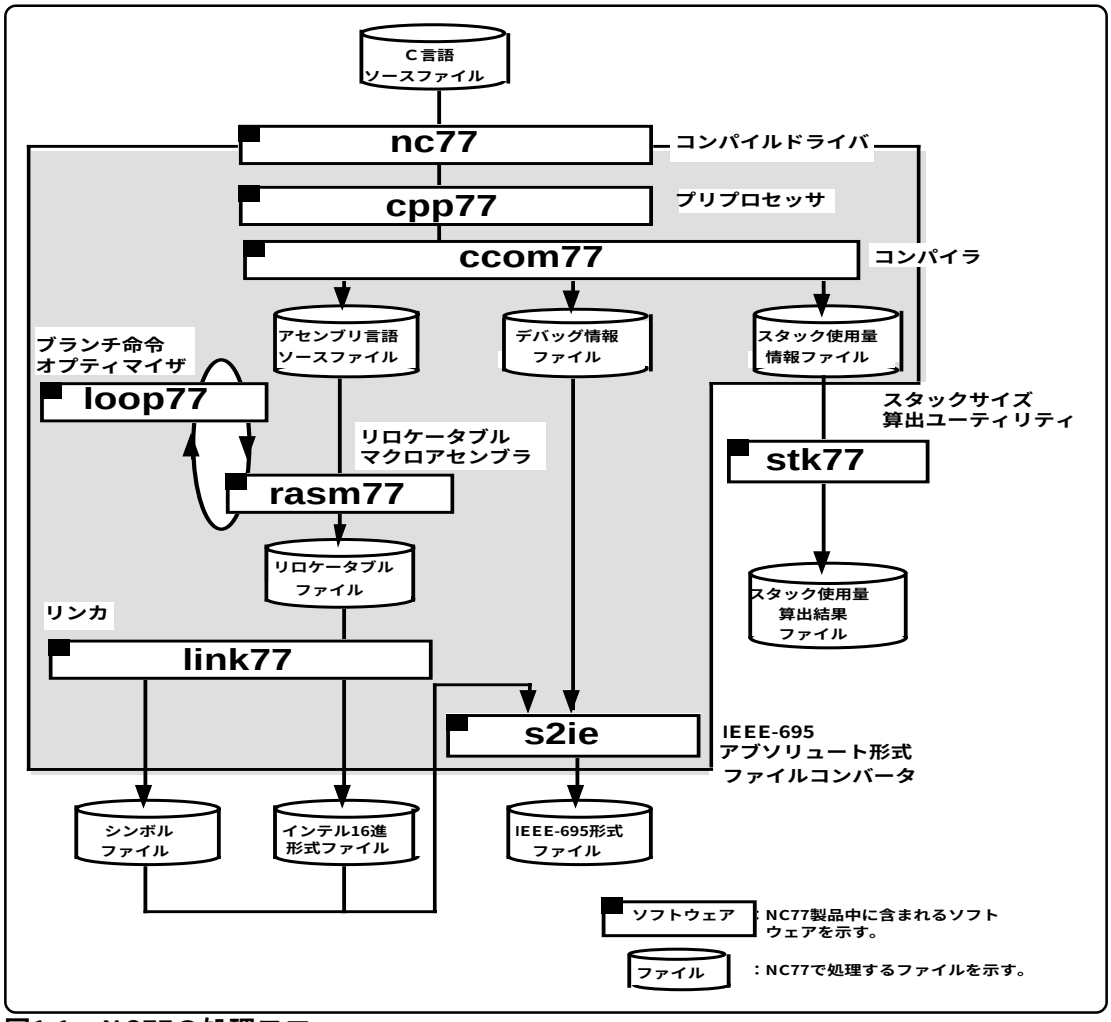


図1.1　NC77の処理フロー

1.2.1　nc77

nc77は、コンパイルドライバの実行ファイルです。nc77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、nc77の起動オプション-rasm77、-link77に続けてリロケータブルマクロアセンブラrasm77、リンケージエディタlink77のオプションを指定することができます。

1.2.2　cpp77

cpp77は、プリプロセッサの実行ファイルです。cpp77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　ccom77

ccom77は、コンパイラ本体の実行ファイルです。cpp77によって処理されたC言語ソースプログラムをrasm77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　loop77

loop77は、ブランチオプティマイザの実行ファイルです。ccom77によって変換されたアセンブリ言語ソースプログラム中のブランチ命令の最適化※1を行います。

1.2.5　s2ie

s2ieは、三菱シンボルファイル形式をIEEE-695アブソリュート形式※2へ変換するコンバータの実行ファイルです。

1.2.6　stk77

stk77は、スタックサイズ算出ユーティリィティの実行ファイルです。nc77の起動オプション-fSSU(-fshow_stack_usage)を指定することにより生成されるソースファイル単位のスタック使用量表示ファイル（拡張子.stk）を処理し、スタックサイズとC言語関数の呼び出し相互関係の情報を算出結果表示ファイル（拡張子.siz）に出力します。

※1.loop77は、ジャンプ先のアドレスが相対値の範囲を越える等でエラーとなった分岐命令(BRA、BCC、等)を、nc77のブランチオプティマイズオプション-OB1、-OB2、-OB3の変換規則によってジャンプ命令(JMP)等に変更し、再度アセンブルを行います。

※2.IEEE-695アブソリュート形式ファイルをサードパーティ製エミュレータ又はシミュレータ等に読み込ませた場合、IEEE-695に規定されていない情報等の相違等により、一部の機能が正常に動作しない、又は読み込めない等の障害が発生する可能性があります。万一、このような障害が発生した場合、弊社にて解決できない場合があることをご了承ください。なお、動作環境に関しては製品添付のリリースノートを参照してください。

1.3　プログラム開発例

NC77を使用したプログラム開発例の流れを【図1.2】に示します。このプログラムの概要を以下に示します（項目の[1]～[4]は【図1.2】の[1]～[4]に対応します）。

①C言語ソースプログラム(AA.c)をnc77でコンパイル、アセンブラrasm77でアセンブルし、リロケータブルオブジェクトファイル(AA.r77)を作成します。
②スタートアッププログラムncrt0.a77とセクション情報を記述したインクルードファイルsection.incを組み込むシステムに合わせて、セクションの配置/セクションサイズ/割り込みベクタテーブルの設定などを変更します。
③変更したスタートアッププログラムをアセンブルします。この結果、リロケータブルオブジェクトファイル(ncrt0.r77)を作成します。
④2つのリロケータブルオブジェクトファイル、AA.r77とncrt0.r77をnc77から実行されるリンケージエディタlink77でリンクし、16進数形式機械語データファイル(AA.hex)を作成します。nc77の起動オプション-gを指定した時には、シンボルファイル(AA.sym)を作成します。

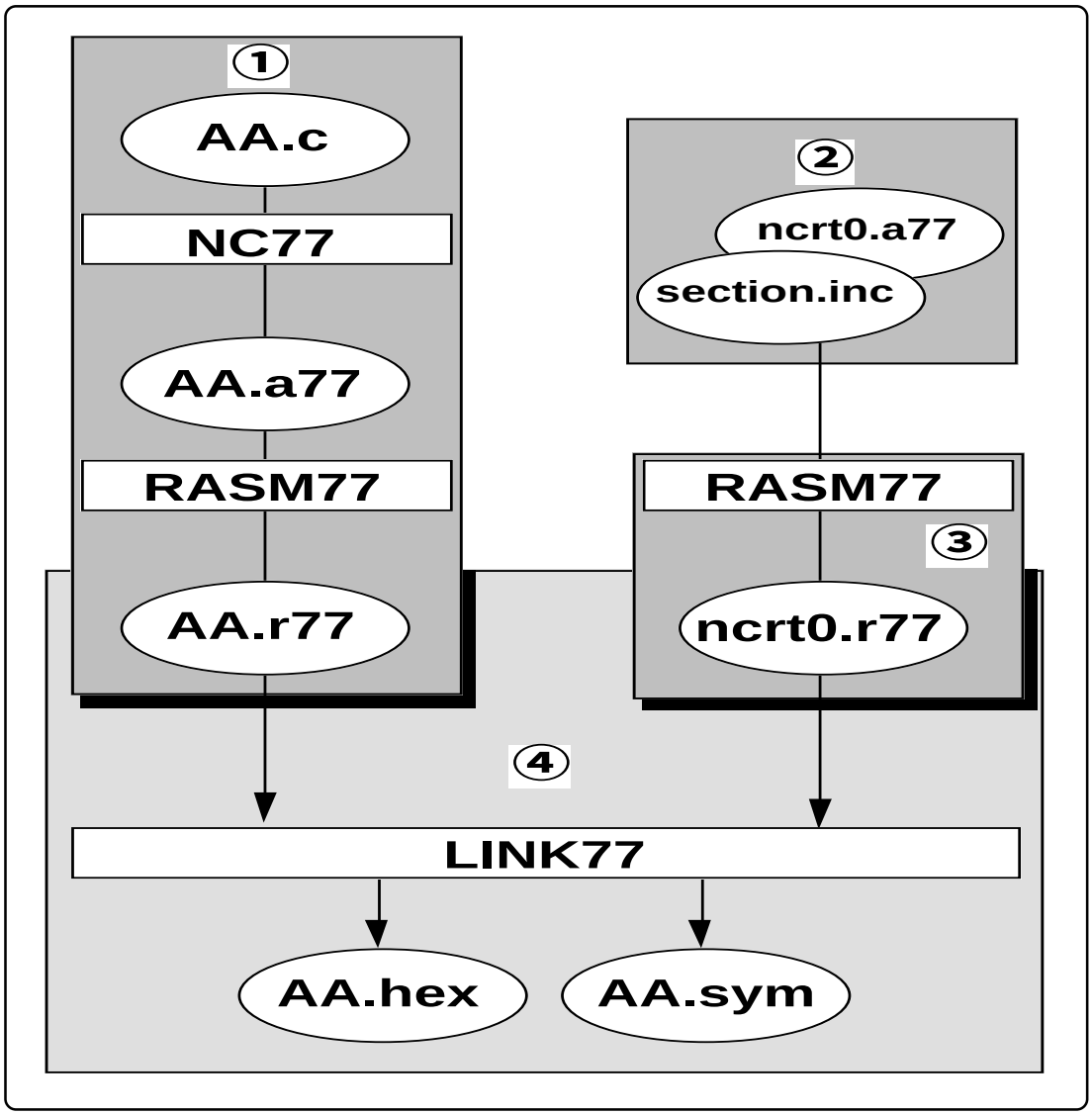


図1.2　プログラム開発フロー

【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.hex : ncrt0.a77 AA.r77
        nc77 -oAA -g ncrt0.r77 AA.r77

ncrt0.r77 : ncrt0.a77
        nc77 -c -g ncrt0.a77

AA.r77 : AA.c
        nc77 -c -g 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 nc77 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。

```
% nc77 -g -oAA ncrt0.a77 AA.c<RET>
```

%：プロンプトを示します。
<RET>：リターンキーの入力を示します。

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

図1.4　nc77コマンドの入力例

1.4　NC77の出力ファイル

サンプルプログラムsmp.cをNC77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nc77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、rasm77及びlink77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.r77）、プリントファイル（拡張子.prn）、マップファイル（拡張子.map）、等に関しては「RASM77ユーザーズマニュアル」を参照してください。

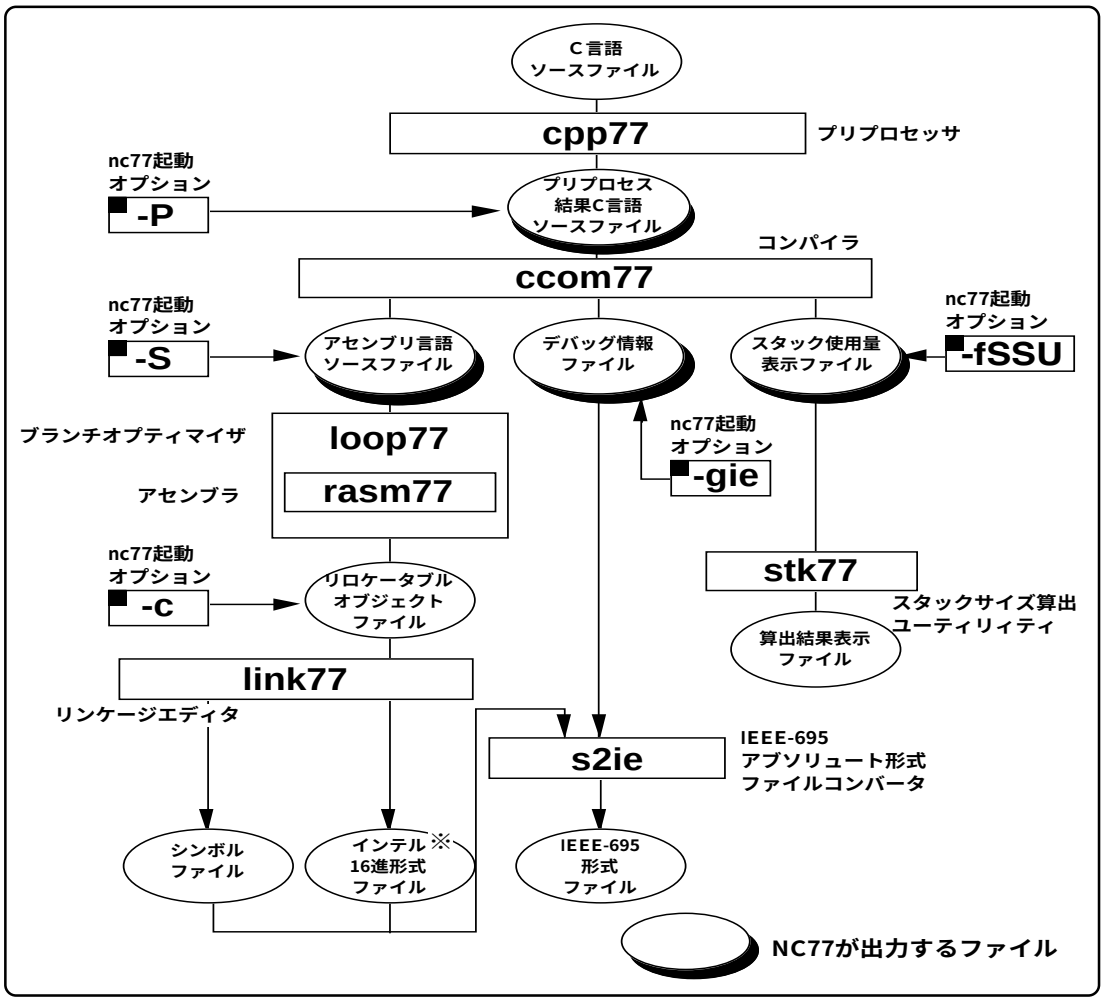


図1.5　nc77起動オプションと出力ファイルの相互関係図

```
#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cpp77は、#で始まるプリプロセスコマンドに対して、ヘッダファイルの内容、マクロの展開、及び条件コンパイルの判定、等の処理を行います。

プリプロセス結果 C 言語ソースファイルは、cpp77 が C 言語ソースファイルを処理した結果を格納しています。したがって、このファイルには、#pragma、#line 以外のプリプロセス行は出力されません。このファイルの内容を参照することによりコンパイラが処理を行うプログラムの内容を確認することができます。ファイルの拡張子は .i です。

ファイルの出力例を【図 1.7】、【図 1.8】に示します。

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

typedef unsigned int size_t;

extern FILE _iob[];
```

図1.7　プリプロセス結果C言語ソースファイル例(1)(smp.i)

```
                                                                    ①
int getc(FILE *st);
int getchar(void);
int putc(int c, FILE *st);
int putchar(int c);
int feof(FILE *st);
int ferror(FILE *st);
int fgetc(FILE *st);
char * fgets(char *s, int n, FILE *st);
int fputc(int c, FILE *st);
int fputs(const char *s, FILE *st);
size_t fread(void *ptr, size_t size, size_t nelem, FILE *st);
                :
             (省略)
                :
int ungetc(int c, FILE *st);
int printf(const char *format, ...);
int fprintf(FILE *st, const char *format, ...);
int sprintf(char *s, const char *format, ...);
                :
             (省略)
                :
extern  int     init_dev(FILE *, int);
extern  int     speed(int, int, int, int);
extern  int     init_prn(void);
extern  int     _sget(void);
extern  int     _sput(int);
extern  int     _pput(int);
extern  char    *_print( int(*)(), 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)(smp.i)

プリプロセス結果 C 言語ソースファイルの内容を以下に説明します。項目番号の①～④は【図 1.7】、【図 1.8】中の①～④にそれぞれ対応しています。

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

1.4.3　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ccom77がプリプロセス結果C言語ソースファイルをRASM77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.a77 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、nc77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。

```
     .language  c

;## NC77 Compiler for 7700 Family OUTPUT
;## ccom77 2.02.05
;## Copyright(c) 1999, MITSUBISHI ELECTRIC CORPORATION
;## and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;## All Rights Reserved
;## Compile Start Time Mon Feb  1 11:34:21 1999
;## COMMAND_LINE: ccom77  /home/kawajiri/tmp/2177xxx.i -o ./xxx.a77 -dS
;## Normal Optimize   OFF                                          ①
;## ROM size Optimize OFF
;## Speed Optimize    OFF
;## Default function is far
;## Default ROM is    near
;## Default RAM is    near
;## MCU type is            M37700
;## Branch instruction is"bra"

     .MCU        M37700
     .pointer  2
     .include  ./xxx.ext

_DT   .EQU        0     ; data bank
_STACK      .EQU        0     ; stack bank


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

     .source xxx.c
     .section  program_F
;## # C_SRC :    {
     .DT    __DT
     .DP    OFF
     .func_main
     .pub  _main
_main:
     phd
     pha
     tsa
     tad
     .cline    8
;## # C_SRC :          flag = CLR;
     ldm.W  #0000H,DP:1 ;    flag
```

図1.9　アセンブリ言語ソースファイル例(1)(smp.a77)

```
      .cline    11
;## # C_SRC :         printf("flag = %d\n", flag);        ←②
      pei   #1    ;   flag
      pea   #OFFSET__T0
      jsrl_printf
      plx
      plx
      .cline    13
;## # C_SRC :    }
      plx
      pld
      rtl
      .endfunc  _main

      .SECTION      rom_N0
__T0:
      .byte66H    ;     'f'
      .byte6cH    ;     'l'
      .byte61H    ;     'a'
      .byte67H    ;     'g'
      .byte20H    ;     ' '
      .byte3dH    ;     '='
      .byte20H    ;     ' '
      .byte25H    ;     '%'
      .byte64H    ;     'd'
      .byte0aH
      .byte00H
      .END

;## Compile End Time Mon Feb  1 11:34:22 1999
```

図1.10　アセンブリ言語ソースファイル例(2)(smp.a77)

アセンブリ言語ソースファイルの内容を以下に説明します。項目番号の①～②は【図 1.9】【図 1.10】中の①～②に対応しています。

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

② nc77 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 nc77 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、nc77 から起動できるアセンブラ rasm77 とリンケージエディタ link77 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 nc77コマンドの入力書式

コンパイルドライバ nc77 は、コンパイラの各コマンド(cpp77、ccom77、loop77、s2ie)とアセンブルコマンド rasm77 及びリンクコマンド link77 を起動し、機械語データファイルを生成します※1。この nc77 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.a77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.a77とsample.r77をリンク

を行い、機械語データファイル sample.hex 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、

●機械語データファイル名sample.hex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.prn)の出力指定 ..... -rasm77 "-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..... -link77 "-ms"オプション

を行っています。

※1.アセンブルコマンドrasm77はブランチオプティマイザloop77から起動されます。したがって、コンパイルドライバnc77は直接起動しません。

```
%  nc77△[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　nc77コマンドの入力書式

```
% nc77 -osample -rasm77 "-l" -link77 "-ms" ncrt0.a77 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　nc77コマンドの入力例

2.1.2　コマンドファイル

nc77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

```
%  nc77△[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

```
% nc77 -c @test.cmd -g<RET>

<RET>：リターンキーの入力を示します。
```

図2.4　コマンドファイルの入力例

コマンドファイルの記述は以下のようになります。

```
test.cmdの記述 ──────────▶  test.cmd<CR>
                            ncrt0.a77<CR>
                            sample1.c sample2.r77<CR>
                            -g -rasm77 -l<CR>
                            -o<CR>
                            sample<CR>

<CR>：改行を示します。
```

図2.5　コマンドファイルの記述例

b.　コマンドファイルの記述規定

コマンドファイルの記述には以下の規定があります。

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

c.　コマンドファイル使用時の注意事項

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

リンク時にファイルを指定するために拡張子 ".cm$" のlink77用コマンドファイルを自動生成します。このため、拡張子が ".cm$" 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　nc77起動オプションに関する注意事項

a.　nc77起動オプションの記述に関する注意事項

nc77 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.　nc77の制御に関するオプションの優先順位

nc77 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.r77)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.a77)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。

c.　最適化オプションの組み合わせ

最適化オプション中 -OS と -OB2、又は -OS と -OB3 を同時に指定した場合、BRA 命令を生成せずに JMP 命令又は JMPL 命令を生成します。

●-OS：ROM容量より速度を優先する最適化を行う。
●-OB2：速度を重視した、同一バンク内へのブランチ命令の最適化を行う。
●-OB3：速度を重視した、バンク外へのブランチ命令の最適化を行う。

2.1.4　nc77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.r77)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.a77)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。

b.　出力ファイル指定オプション

【表 2.2】に出力する機械語データ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*	link77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、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	link77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、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、及び-Sを指定しない場合、nc77はlink77まで制御を行い、機械語データファイル(拡張子.hex)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.sym）を出力します。本オプション指定時は、C 言語レベルのデバッグ情報は出力しません。
-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。
-gie	なし	C言語レベルのデバッグ情報(IEEE-695アブソリュート形式ファイル(拡張子.ie))を出力します。
-gie_no_local_symbl	-gILNS	アセンブリファイル中のローカルシンボルをIEEE-695アブソリュート形式ファイル（拡張子.ie）に出力しません。

e. 最適化オプション

【表2.5】にプログラムの実行速度及びROM容量を最小にする最適化を行う起動オプションを示します。

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	速度及びROM容量ともに最小にする最大限の最適化を行います。
-OR	なし	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。これによりROM容量を削減することができます。ただし使用するスタック量が多くなる可能性があります。
-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。

f.　生成コード変更オプション

【表 2.6】に nc77 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。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべてodd属性のセクションに配置します。
-fshow_stack_usage	-fSSU	スタックの使用状況をファイル(拡張子.stk)に出力します。
-ffar_RAM_data	-fFRAM	RAMデータのデフォルト属性をfarにします。
-ffar_ROM_data	-fFROM	ROMデータのデフォルト属性をfarにします。
-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jsrで呼び出し、rtsで戻ります。
-ffar_program_section	-fFPS	near関数・far関数を共に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=	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。
-fswitch_table	-fST	ROM効率が良くなる場合のみswitchu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。
-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	#pragmaADDRESS(#pragmaEQU)で指定した変数をvolatileで指定した変数とみなしません。
-fsmall_array	-fSA	far 型の配列を参照する場合、その総サイズが64Kバイト以内であれば、添字の計算を16ビットで行ないます。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。

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

g.　警告オプション

【表 2.7】に nc77 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2.7　警告オプション

オプション	短縮形	機　能
-Wnon_prototype	-WNP	プロトタイプ宣言されていない関数を使用した場合、警告を出します。
-Wunknown_pragma	-WUP	サポートしていない＃pragmaを使用した場合、警告を出します。
-Wno_stop	-WNS	エラーが発生してもコンパイル作業を停止しません。
-Wstdout	なし	エラーメッセージをホストマシンの標準出力（stdout）に出力します。
-Werror_file<file name>	-WEF	タグファイルを出力します。
-Wstop_at_warning	-WSAW	ワーニング発生時にコンパイル処理を停止します。
-Wnesting_comment	-WNC	コメント中に/*を記述した場合に警告を出します。
-Wccom_max_warnings	-WCMW	ccom77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。
-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。

h.　アセンブル・リンクオプション

【表 2.8】に RASM77 及び LINK77 のオプションを指定する起動オプションを示します。

表2.8　アセンブル・リンクオプション

オプション	機　能
-rasm77△<オプション>	アセンブルコマンドrasm77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、"(ダブルクオーテーション)で囲んでください。
-link77△<オプション>	リンクコマンドlink77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、"(ダブルクオーテーション)で囲んでください。

i.　7750/7751対応コード生成オプション

【表 2.9】に NC77 が生成する 7750/7751 対応コード生成オプション起動オプションを示します。

表2.9　7750/7751対応コード生成オプション

オプション	短縮形	機　能
-m7750	なし	7750/7751シリーズに対応したコードを生成します。

j.　その他のオプション

【表 2.10】に nc77 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。

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、NC77ではハードウェア(7700)の初期設定、セクションの配置、割り込みベクタアドレステーブル、等を設定するアセンブリ言語で記述したサンプルのスタートアッププログラムを製品に付属しています。スタートアッププログラムを組み込むシステムに合わせて変更する必要があります。

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

2.2.1　スタートアッププログラムのサンプル

NC77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.a77

   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。

2. section.inc

   このプログラムは、ncrt0.a77からインクルードされます。

ncrt0.a77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、section.incのソースプログラムリストを【図2.10】、【図2.11】、【図2.12】にそれぞれ示します。

```
;***********************************************************************
;
;    NC77 COMPILER for 7700 FAMILY
;    Copyright 1998, MITSUBISHI ELECTRIC CORPORATION
;    AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;    All Rights Reserved.
;
;    ncrt0.a77 : NC77 startup program
;
;    This program is applicable when using the basic I/O library
;
;
;***********************************************************************
    .include     section.inc                              ←①
;-----------------------------------------------------------------------
; section-size
;-----------------------------------------------------------------------
    .section     rom_NE
    .pub         bss_NEsz
    .pub         bss_NOsz
    .pub         bss_FEsz
    .pub         bss_FOsz
    .pub         data_NEsz
    .pub         data_NOsz
    .pub         data_FEsz
    .pub         data_FOsz
bss_NEsz:        .dword sizeof bss_NE
bss_NOsz:        .dword sizeof bss_NO
bss_FEsz:        .dword sizeof bss_FE
bss_FOsz:        .dword sizeof bss_FO
data_NEsz:       .dword sizeof data_NE
data_NOsz:       .dword sizeof data_NO
data_FOsz:       .dword sizeof data_FO
data_FEsz:       .dword sizeof data_FE


①section.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.a77)

```
;=====================================================================
; Initialize Macro declaration
;---------------------------------------------------------------------
BZERO:      .macro SIZE,TOP
    lda     A,DT:SIZE + 2
    pha
    lda     A,DT:SIZE
    pha
    pea     #bank TOP
    pea     #offset TOP
    .ext    _bzero
    jsrl    _bzero
    .endm

BCOPY:      .macro SIZE,TO,FROM
    lda     A,DT:SIZE + 2
    pha
    lda     A,DT:SIZE
    pha
    pea     #bank TO
    pea     #offset TO
    pea     #bank FROM
    pea     #offset FROM
    .ext    _bcopy
    jsrl    _bcopy
    .endm

;=====================================================================
; Libraly file name definition
;---------------------------------------------------------------------
    .lib    nc77lib

;=====================================================================
; Interrupt  section start
;---------------------------------------------------------------------
    .pub          start
    .section      interrupt
start:                                                  ←②
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
                                                        ←③
__DT .equ         00h
    ldt     #__DT               ; Initialize data bank register
    sem
    ldm.B   #24H,DT:5eH         ; set processor mode register          ←④
    clp     m,x,d
    lda.w   a, #offset stack_top - 1   ; Initialize stack pointer      ←⑤
    tas

;=====================================================================
; NEAR area initialize.
;---------------------------------------------------------------------
; bss_NE & bss_NO zero clear
;---------------------------------------------------------------------
                                                        ←⑥
    BZERO  bss_NEsz,bss_NE_top
    BZERO  bss_NOsz,bss_NO_top
;---------------------------------------------------------------------
; Copy data_NE(NO) section from data_INE(INO) section
;---------------------------------------------------------------------
                                                        ←⑦
    BCOPY  data_NEsz,data_NE_top,data_INE_top
    BCOPY  data_NOsz,data_NO_top,data_INO_top
    lda.w  a,#offset stack_top - 1
    tas
```

②リセット直後はこのラベルstartからスタートします。
③NC77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。
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。
⑥near領域のbssセクションのゼロクリア処理を行います。
⑦near領域のdataセクションの初期値をRAM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.a77)

```
;=====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss_FE & bss_F0 zero clear
;---------------------------------------------------------------------
    BZERO  bss_FEsz,bss_FE_top                          ←⑧
    BZERO  bss_F0sz,bss_F0_top
;---------------------------------------------------------------------
; Copy data_FE(F0) section from data_IFE(IF0) section
;---------------------------------------------------------------------
    BCOPY  data_FEsz,data_FE_top,data_IFE_top           ←⑨
    BCOPY  data_F0sz,data_F0_top,data_IF0_top
    lda.w  a,#offset stack_top - 1
    tas

;=====================================================================
; heap initialize
;---------------------------------------------------------------------
    .ext   __mbase, __mnext, __msize                    ←⑩
    lda.w  a, #offset heap_top
    lda.w  b, #bank heap_top
    sta    a, __mbase
    sta    b, __mbase + 0002h
    sta    a, __mnext
    sta    b, __mnext + 0002h
    lda.w  a, #offset HEAPSIZE
    lda.w  b, #bank HEAPSIZE
    sta    a, __msize
    sta    b, __msize + 0002h

;
;
;=====================================================================
; Initialize standard I/O
;---------------------------------------------------------------------
    .ext   _init
    jsrl   _init                                        ←⑪

;=====================================================================
; Call main() function
;---------------------------------------------------------------------
    .ext   _main
    jsrl   _main                                        ←⑫
```

⑧far領域のbssセクションのゼロクリア処理を行います。
⑨far領域のdataセクションの初期値をRAM領域に転送します。
⑩heap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。
⑫main関数の呼出しを行います。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.a77)

```
;===================================================================
; exit() function
;-------------------------------------------------------------------
    .func       _exit                                   ←⑬
    .pub        _exit
_exit:                          ; End program
    bra         _exit
    .endfunc    _exit
    .func       ?exit
    .pub        ?exit
?exit:                          ; End program
    bra         ?exit
    .endfunc    ?exit

;===================================================================
; dummy interrupt function
;-------------------------------------------------------------------
dummy_int:                                              ←⑭
    rti
    .end
;*******************************************************************************
;
;   NC77 COMPILER for 7700 FAMILY
;   Copyright 1998, MITSUBISHI ELECTRIC CORPORATION
;   AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;   All Rights Reserved.
;
;*******************************************************************************
```

⑬exit関数部です。
⑭ダミーの割り込み処理関数です。

図2.9　スタートアッププログラムリスト(4)(ncrt0.a77)

```
;***************************************************************************
;
;       NC77 COMPILER for 7700 FAMILY
;       Copyright 1998, MITSUBISHI ELECTRIC CORPORATION
;       AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;       All Rights Reserved.
;
;       section.inc     : section definition
;
;       This program is applicable when using the basic I/O library
;
;
;
;***************************************************************************
;
;---------------------------------------------------------------------
; Arrangement of section
;---------------------------------------------------------------------
    .section    data_NE
    .org   80H                                          ←①
data_NE_top:
;
    .section    data_NO
data_NO_top:
;
    .section    bss_NE
bss_NE_top:
;
    .section    bss_NO
bss_NO_top:
;
    .section    stack
        .blkb           300H                ; stack size                ←②
stack_top:
;
HEAPSIZE        .equ    300h                                            ←③
        .section        heap
heap_top:
        .blkb           HEAPSIZE
;
        .section        interrupt
        .ORG            8000H
;
        .section        program_N
;
        .section        rom_NE
rom_NE_top:
;
        .section        rom_NO
rom_NO_top:
;
        .section        data_INE
data_INE_top:
;
        .section        data_INO
data_INO_top:
;
```

①セクション開始アドレスを疑似命令.ORGを用いて設定します。
②使用するスタックサイズを定義します。
③使用するheapサイズを定義します。

図2.10　スタートアッププログラムリスト(5)(section.inc)

```
    .section     vector              ; Interrupt vector table
    .org   0ffd6H
ADCOMP:                                                          ←④
        .word   dummy_int
TRN1:
        .word   dummy_int
REC1:
        .word   dummy_int
TRN0:
        .word   dummy_int
REC0:
        .word   dummy_int
BS2I:
        .word   dummy_int
BS1I:
        .word   dummy_int
BS0I:
        .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
WDT:
        .word   dummy_int
RESERVED:
        .word   dummy_int
BRK:
        .word   dummy_int
DIV0:
        .word   dummy_int
;
RESET:
        .word   offset start
;
;
        .section        data_FE
        .org    12000H
data_FE_top:
;
```

④割込みベクタアドレステーブルの例です。

図2.11　スタートアッププログラムリスト(6)(section.inc)

```
        .section        data_F0
data_F0_top:
;
        .section        bss_FE
bss_FE_top:
;
        .section        bss_F0
bss_F0_top:
;
        .section        program_F
        .ORG            18000H
;
        .section        rom_FE
rom_FE_top:
;
        .section        rom_F0
rom_F0_top:
;
        .section        data_IFE
data_IFE_top:
;
        .section        data_IF0
data_IF0_top:

;******************************************************************************
;
;       NC77 COMPILER for 7700 FAMILY
;       Copyright 1998, MITSUBISHI ELECTRIC CORPORATION
;       AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;       All Rights Reserved.
;
;******************************************************************************
```

図2.12　スタートアッププログラムリスト(7)(section.inc)

2.2.2　スタートアッププログラムのカスタマイズ

a.　スタートアッププログラムの処理概要

ncrt0.a77 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。
●スタックポインタの初期化を行います。
●DTレジスタの初期化を行います。
●データのnear領域の初期化を行います。
　bss_NE、bss_NOセクションをゼロクリアします。また、初期値が格納されたＲＯＭ領域(ｄａｔａ_ＩＮＥ、ｄａｔａ_ＩＮＯ)の初期値をＲＡＭ領域(data_NE、data_NO)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。

※1.NC77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。

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2.13】に示します。

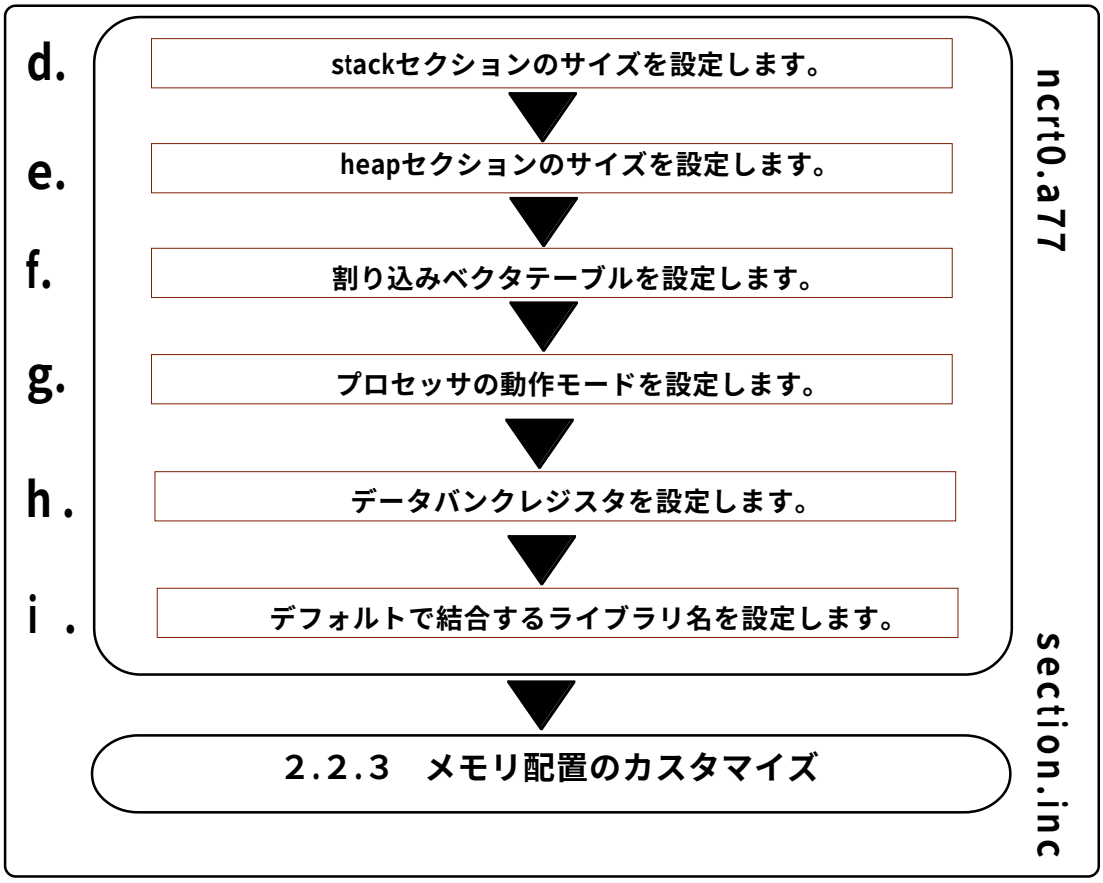


図2.13　スタートアッププログラムの変更手順例

c.　注意を要するスタートアップの変更例

(1)　標準入出力関数を使用しないときの設定

init関数は、7700ファミリの入出力の初期化を行います。init関数は、ncrt0.a77内でmain関数を呼び出す前に呼び出されます。【図2.14】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.a77内のinit関数の呼び出し部をコメントにしてください。

```
;===================================================================
; Initialize standard I/O
;-------------------------------------------------------------------
    .ext  _init
    jsrl  _init

;===================================================================
; Call main() function
```

図2.14　init関数呼び出し部（ncrt0.a77）

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.a77中で以下の設定を行っています。

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

【図2.15】にncrt0.a77内の初期化部を示します。

```
;=====================================================================
; heap initialize
;---------------------------------------------------------------------
        .ext      __mbase, __mnext, __msize
        lda.w     a, #offset heap_top
        lda.w     b, #bank heap_top
        sta       a, __mbase
        sta       b, __mbase + 0002h
        sta       a, __mnext
        sta       b, __mnext + 0002h
        lda.w     a, #offset HEAPSIZE
        lda.w     b, #bank HEAPSIZE
        sta       a, __msize
        sta       b, __msize + 0002h
```

図2.15　メモリ管理関数を使用するときの初期化部(ncrt0.a77)

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、DTレジスタの内容を変更しないでください。
(2)独自の初期化プログラムからC言語で記述されたサブルーチンを呼び出す場合は、以下の2項目に注意してください。
　　①M、X、Dフラグを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。

d.　stackセクションのサイズの設定

NC77は、関数単位で以下に示す内容でstackセクションの領域を使用します。

●記憶クラスautoの変数(自動変数)の格納領域
●複雑な演算等に使用するテンポラリ領域
●関数呼び出し元への戻りアドレスと旧フレームポインタ(DPR)アドレスの格納領域
●関数への引数の格納領域
●64ビット浮動小数点を格納する内部レジスタの領域

セクションの領域の確保は、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、スタックサイズ算出ユーティリティSTK77を使用して求めます。スタック使用量を求めるには、STK77のオプション"-e"を指定してください。
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。

```
;------------------------------------------------------------------------
; HEAP SIZE definition
;------------------------------------------------------------------------
HEAPSIZE        .equ    300h
```

図2.16　heapサイズの設定例(ncrt0.a77)

f.　割り込みベクタテーブルの設定

ncrt0.a77中の【図2.17】の部分において割り込みベクタテーブルの先頭アドレスを設定します。

```
    .section    vector                  ; Interrupt vector table
    .org  0ffd6H
```

図2.17　割り込みベクタテーブルの先頭アドレスの設定例(ncrt0.a77)

g.　プロセッサモードレジスタの設定

ncrt0.a77中の【図2.18】で示す部分において5EH番地(プロセッサモードレジスタ)※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。

```
    ldm.B  #24H,DT:5eH          ; set processor mode register
```

図2.18　プロセッサモードレジスタ設定例(ncrt0.a77)

h.　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.a77中の【図2.19】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nc77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.a77中の__DTに設定してください。

```
start:
;-----------------------------------------------------------------
; after reset,this program will start
;-----------------------------------------------------------------
__DT    .equ            00h
        ldt     #__DT                   ; Initialize data bank register
```

図2.19　データバンクレジスタの設定例(1)(ncrt0.a77)

【図2.20】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。リンクするすべてのC言語プログラムに対してをnc77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。

```
start:
;-----------------------------------------------------------------
; after reset,this program will start
;-----------------------------------------------------------------
__DT    .equ            02h
        ldt     #__DT                   ; Initialize data bank register
```

図2.20　データバンクレジスタの設定例(2)(ncrt0.a77)

※1.この設定例は、M37702グループに合わせて記述しています。したがってプロセッサモードレジスタのアドレス及びビットの設定内容は、使用する機種のマニュアルもしくはデータブックを参照してください。

i.　ライブラリファイルの指定

ncrt0.a77中にはNC77専用のライブラリファイルを読み込む処理を記述しています(【図2.21】参照)。このライブラリ以外に、ライブラリアンLIB77で作成した任意のライブラリファイルを使用する場合は、スタートアッププログラム中に記述します。読み込みの指定にはRASM77の.LIB疑似命令を使用します。この疑似命令のオペランドに記述できるライブラリファイル名の拡張子は.libです。なおライブラリファイルが存在するディレクトリは、環境変数LIB77で指定します。

```
;=====================================================================
; Libraly file name definition
;---------------------------------------------------------------------
        .lib            nc77lib
        .lib             usrlib          ←ユーザライブラリusrlib.libの指定
```

図2.21　ユーザライブラリファイル名指定例(ncrt0.a77)

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、NC77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。

NC77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7700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。

図2.22　セクション名

セクション名を【表2.11】に属性を【表2.12】に示します。

表2.11　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。
program	プログラムを格納します。

表2.12　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※1	data, bss, rom, program
	F...far属性※1	
E/O	E...データサイズが偶数	data, bss, rom
	O...データサイズが奇数	

※1. near、farとは、NC77固有の修飾子です。この修飾子を使用することによりアドレッシングモードを明示的に指定できます。
near.....アブソリュート系アドレッシングモード(アクセス範囲は64Kバイト以内)
far.....アブソリュートロング系アドレッシングモード(アクセス範囲は64Kバイト以上)

前述の命名規則に準じたセクション以外のセクションの内容を【表2.13】に示します。

表2.13　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7700ファミリの0バンクに配置して下さい。
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7700の任意の領域に配置できます。
vector	7700の割り込みベクタテーブルの内容を格納します。この割込みベクタ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	割込みプログラム(#pragma INTERRUPT, #pragma INTF, #pragma HANDLERで指定された関数)を格納します。このセクションは7700ファミリの0バンクに配置して下さい。

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。

サンプルのスタートアッププログラムのインクルードファイルsection.incのセクション配置を【図2.23】に示します。

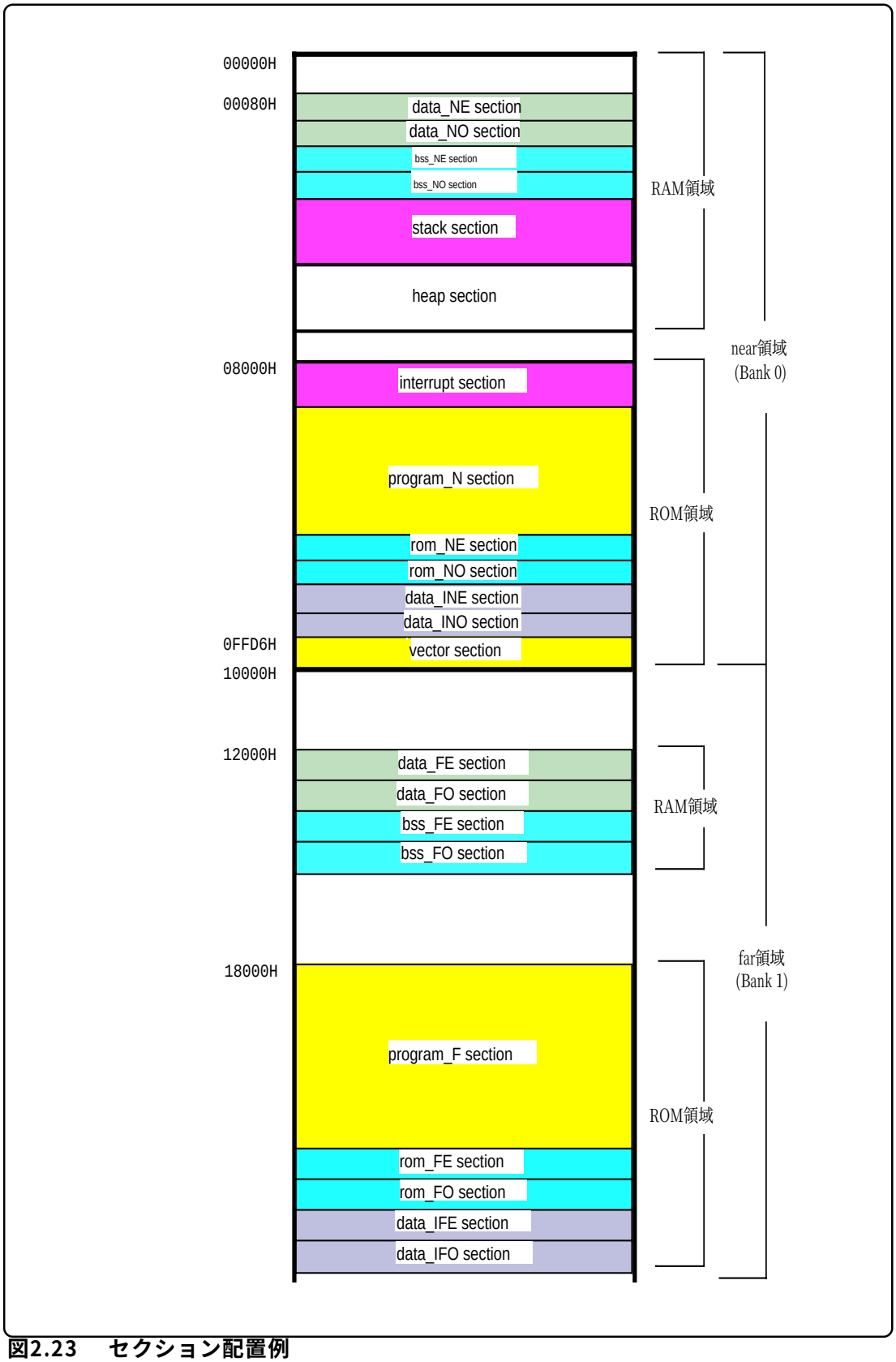


図2.23　セクション配置例

b.　メモリ配置設定用ファイルの概要

section.inc について

このプログラムは、ncrt0.a77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.　section.incの変更手順

スタートアッププログラムのメモリ配置設定用ファイルを、組み込むシステムに合わせて変更する方法の概略を【図2.24】に示します。

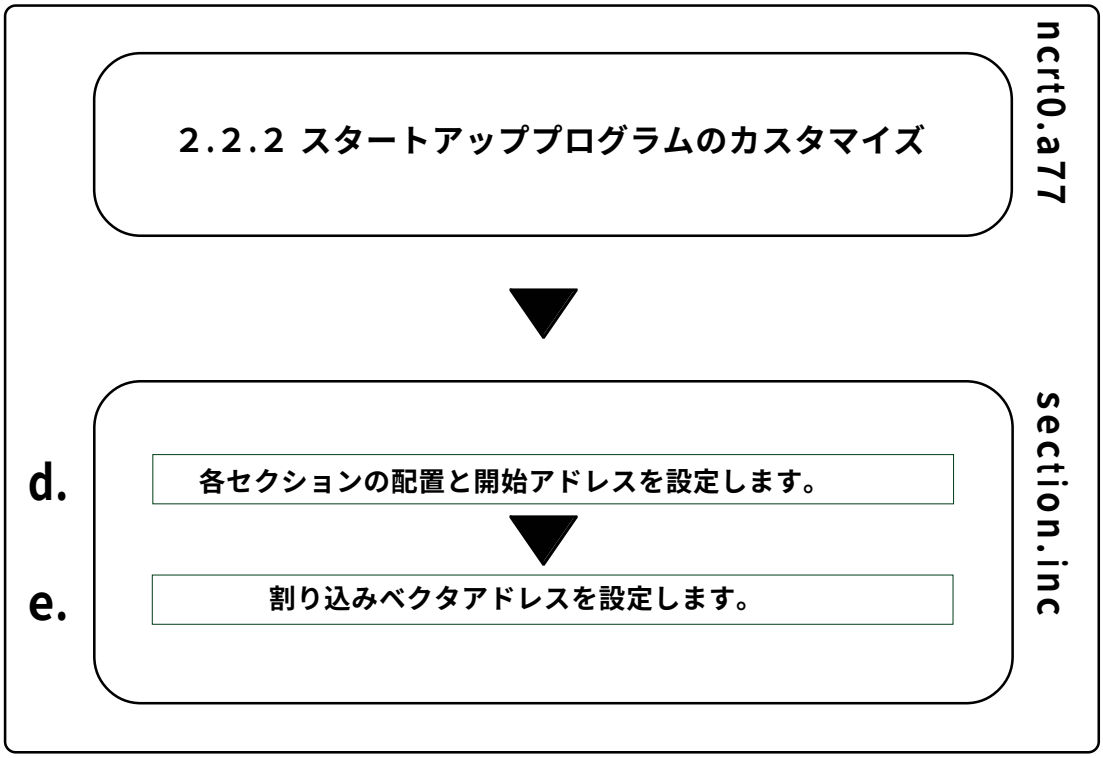


図2.24　メモリ配置の変更手順例

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section.incで行います。

セクションの配置は、section.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rasm77の疑似命令.ORGを用いて指定します。【図2.25】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2.25　セクション開始アドレスの設定例(section.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、7700ファミリのメモリ(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。セクションの配置は以下の規則に従ってください。

(a)RAM領域に配置するセクション

●data_NEセクション
●data_NOセクション
●data_FEセクション
●data_FOセクション

●bss_NEセクション
●bss_NOセクション
●bss_FEセクション
●bss_FOセクション

●stackセクション
●heapセクション

(b)ROMに配置するセクション

●rom_NEセクション
●rom_NOセクション
●rom_FEセクション
●rom_FOセクション

●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

●program_Nセクション
●interruptセクション

●program_Fセクション

また、セクションは7700ファミリ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。

(1)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**

(2)-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。

(3)7700ファミリ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。

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**
- **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**

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section.incの記述例を【図2.26】、【図2.27】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

```
;***************************************************************************
;
;       NC77 COMPILER for 7700 Family V.5.00
;       Copyright 1998 MITSUBISHI ELECTRIC CORPORATION
;       AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;       All Rights Reserved.
;
;       section.inc     : section definition
;
;       This program is applicable when using the basic I/O library
;
;***************************************************************************
;-------------------------------------------------------------------
; Arrangement of section
;-------------------------------------------------------------------
    .section    data_NE
    .org  80H
data_NE_top:
;
    .section    data_NO
data_NO_top:
;
    .section    bss_NE
bss_NE_top:
;
    .section    bss_NO
bss_NO_top:
;
    .section    stack
    .blkb 300H          ; stack size
stack_top:
;
    HEAPSIZE    equ   0h                ←メモリ管理関数を使用しませんのでheap
    .section    heap                      セクションのサイズを0にします。
heap_top:
    .blkb       HEAPSIZE
;
    .section    interrupt
    .ORG        8000H
;
    .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
    .section    rom_NE                    プモードにおいても定義が必要になりま
rom_NE_top:                             す。
;
```

図2.26　シングルチップモードにおけるsection.incリスト(1)

```
    .section    rom_NO
rom_NO_top:
;
    .section    data_INE
data_INE_top:
;
    .section    data_INO
data_INO_top:
;
    .section    vector                  ; Interrupt vector table
    .org  0ffd6H
ADCOMP:
    .word dummy_int
                :
              (省略)
                ;
RESET:
    .word offset start
;
;
    .section    data_FE
    .org  12000H
data_FE_top:
;
    .section    data_FO
data_FO_top:
;
    .section    bss_FE
bss_FE_top:
;
    .section    bss_FO
bss_FO_top:
;
    .section    program_F
    .ORG  18000H
;
    .section    rom_FE
rom_FE_top:
;
    .section    rom_FO
rom_FO_top:
;
    .section    data_IFE
data_IFE_top:
;
    .section    data_IFO
data_IFO_top:
;*****************************************************************************
;
;       NC77 COMPILER for 7700 Family V.5.00
;       Copyright 1998 MITSUBISHI ELECTRIC CORPORATION
;       AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
;       All Rights Reserved.
;
;*****************************************************************************
```

←セクションサイズが0であれば取り除きます。

またその場合、ncrt0.a77のfar領域の初期化プログラムも取り除く必要があります。

図2.27　シングルチップモードにおけるsection.incリスト(2)

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、section.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2.28】に割り込みベクタアドレステーブル例を示します。

```
 .section        vector                  ; Interrupt vector table
        .org    0ffd6H
ADCOMP:                                 ←A-D変換割り込み
        .word   dummy_int
TRN1:                                   ←UART1送信割り込み
        .word   dummy_int
REC1:                                   ←UART1受信割り込み
        .word   dummy_int
TRN0:                                   ←UART0送信割り込み
        .word   dummy_int
REC0:                                   ←UART0受信割り込み
        .word   dummy_int
BS2I:                                   ←タイマB2割り込み
        .word   dummy_int
BS1I:                                   ←タイマB1割り込み
        .word   dummy_int
BS0I:                                   ←タイマB0割り込み
        .word   dummy_int
TA4I:                                   ←タイマA4割り込み
        .word   dummy_int
TA3I:                                   ←タイマA3割り込み
        .word   dummy_int
TA2I:                                   ←タイマA2割り込み
        .word   dummy_int
TA1I:                                   ←タイマA1割り込み
        .word   dummy_int
TA0I:                                   ←タイマA0割り込み
        .word   dummy_int
INT2:                                   ←外部割り込みINT2
        .word   dummy_int
INT1:                                   ←外部割り込みINT1
        .word   dummy_int
INT0:                                   ←外部割り込みINT0
        .word   dummy_int
WDT:                                    ←監視タイマ割り込み
        .word   dummy_int
RESERVED:                               ←デバッガ専用割り込み
        .word   dummy_int
BRK:                                    ←BRK命令
        .word   dummy_int
DIV0:                                   ←0除算割り込み
        .word   dummy_int
;
RESET:                                  ←リセット
        .word   offset start


※dummy_intはダミーの割り込み処理関数です。
```

図2.28　周辺I/O割り込みベクタアドレステーブル(section.inc)

割り込みベクタの内容は、7700ファミリ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。

① 割り込み処理関数名をrasm77の疑似命令.EXTで外部参照宣言します。 NC77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 ベクタアドレステーブル中の該当する割込み名ラベル(ADCOMP:、TRN1: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2.29】にUART1送信割り込み処理関数uarttrnの登録例を示します。

```
        .section        vector                  ; Interrupt vector table
        .ext    _uarttrn                                    ←上記①の処理
        .org    0ffd6H
ADCOMP:
        .word   dummy_int
TRN1:
        .word   _uarttrn                                    ←上記②の処理
REC1:
        :
    (以下省略)
```

図2.29　割り込みベクタアドレスの設定例(section.inc)

第３章
プログラミング

この章では、NC77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、NC77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

NC77が生成する機械語命令（アセンブリ言語）は、コンパイル時に指定する起動オプション、バージョンアップの内容等により変化します。したがって、**起動オプションの変更又はコンパイラのバージョンアップを行った時には再度アプリケーションプログラムの動作評価を必ず行ってください。**

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

3.1.2　最適化について

a.　最適化の抑止方法

NC77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

NC77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、NC77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、NC77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。

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を指定すると、コード効率の向上の為にregister渡しとなる引数をauto領域に転送を行わずにregister変数として扱う場合があります。

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、配列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3.4】に示しますようにfar型配列のextern宣言においてサイズを明記するか、もしくは -fsmall_array (-fSA)※2 オプションを付加してコンパイルすることによりコード効率を良くすることができます。

```
extern int far array[];      ←サイズ不明なので添え字を32ビットで計算します

extern int far array[10];    ←サイズが64Kバイト以内のため効率の良いアクセスを行います
```

図3.4　far型配列のextern宣言例

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

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るためコード効率が良くありません。このような場合、配列の添え字になる変数をint型の変数にしてください。

d. プロトタイプ宣言の活用

NC77では、関数のプロトタイプ宣言を行なうことにより、効率の良い関数呼び出しを行なうことができます。
関数のプロトタイプ宣言を行なわない場合は、その関数を呼び出すときに引数を【表3.1】に示す規則によりスタック領域に積みます。

表3.1　引き数に関するスタックの使用規則

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

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

e.　nc77起動オプションに関して

nc77の起動オプションとして、速度とROM効率に関する最適化、ブランチ命令の選択オプションを用意しています。

1.速度とROM効率に関する最適化オプション
- ●-O ..... 速度とROM容量とに共通の最適化を行う
- ●-OR ..... 速度よりROM容量を優先する最適化を行う
- ●-OS ..... ROM容量より速度を優先する最適化を行う

2.ブランチ命令の選択オプション
- ●-OB1 ..... ROM容量を重視したブランチ命令の選択を行う
- ●-OB2 ..... 速度を重視した同一バンク内へのブランチ命令の選択を行う
- ●-OB3 ..... 速度を重視したバンク外へのブランチ命令の選択を行う

nc77起動時にこれら2種類の最適化オプションを組み合わせて指定することにより、メモリ効率及び速度に関するコードの生成を制御することができます。

例えば、バンク境界にまたがって配置された関数がないプログラムに対して-OB2又は-OB3と-OSオプションを組み合わせて使用した場合、【表3.2】に示すようにBRA命令を生成しません。この組み合わせを指定した場合、速度を重視したコードが生成されます。

表3.2　分岐命令の選択オプションと-OSオプションとの組み合わせ

オプション名	-OSオプションとの組み合わせで生成される命令
-OB2	JMP命令
-OB3	JMPL命令

ROM容量を重視する場合は、前述の6種類の最適化オプションに加え、生成コード変更オプションを組み合わせることで効率の良いコードを生成できます。

例えば、プログラムとデータが共に64Kバイト以内で同一バンクにある場合、次ページのオプションを指定することで、ROM容量、速度共に効率の良いコードを生成できます。

- ●-fnear_function ... near・far指定のない関数をすべてnear属性の関数として扱う
- ●-OS ..... ROM容量より速度を優先する最適化を行う
- ●-OB2 ..... 速度を重視した同一バンク内へのブランチ命令の最適化を行う

f.　関数のnear・far制御テクニック

関数に対してnear・far指定を行うことにより、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、現在のPGの値に依存したバンク参照になります。そのため局所的にnear関数を定義した場合、呼びだし側と定義側のプログラムバンクは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。

g.　16ビット長データ間の乗算結果を32ビット長で求める処理の高速化

NC77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, int i3)
{
    long l1,l2,l3;

    l1 = i1;
    l2 = i2;
    l3 = i3;
    l3 = l1 * l2;
    return l3;
}
```

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(1)

この【図3.5】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3.6】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3.6　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)

h.　その他

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

(1) 一回しか呼ばれない比較的小さな関数をinline関数にする。
(2) if-else文をswitch文で置き換える(判定対象の変数が配列、ポインタ、構造体などの単純な変数ではない場合に効果があります。
(3) ビットの比較を '&&'、'||' ではなく '|' で行う。
(4) char型の範囲でしか値を返さない関数の、戻り値の型を char型で宣言する。
(5) 関数呼び出しをまたいで使用する変数をレジスタ変数にしない。

3.2.2　スタートアップ処理を高速化する方法

スタートアッププログラムsection.inc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 area initialize.
;---------------------------------------------------------------------
; bss_NE & bss_NO zero clear
;---------------------------------------------------------------------
;       BZERO   bss_NEsz,bss_NE_top
;       BZERO   bss_NOsz,bss_NO_top
                    :
                 (省略)
                    :
;=====================================================================
; FAR area initialize.
;---------------------------------------------------------------------
; bss_FE & bss_FO zero clear
;---------------------------------------------------------------------
;       BZERO   bss_FEsz,bss_FE_top
;       BZERO   bss_FOsz,bss_FO_top
                    :
```

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。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示します。この例では、アセンブラ関数asm_funcをnear属性でプロトタイプ宣言しているためJSR命令で呼び出されます。

```
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(smp1.c)

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

        rtl
```

図3.10　smp1.cのコンパイル結果(抜粋)(smp1.a77)

※1.プロトタイプ宣言に記述されたnear又は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ときは、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。

b.　アセンブラ関数に対して引数を与える場合

アセンブラ関数に引数を渡す場合、拡張機能の #pragma PARAMETER を使用します。#pragma PARAMETER は、32bit データを 16bit レジスタの組み合わせ（ AB、XY ）、16bit 及び 8bit データをレジスタ（A、B、X、Y）を介してアセンブラ関数に引数を渡します。

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

① #pragma PARAMETER宣言を記述する前にアセンブラ関数のプロトタイプ宣言を行います。このときには、必ず引数の型宣言を行なってください。

② #pragma PARAMETERでアセンブラ関数の引数リストに使用するレジスタ名を宣言します。

【図 3.11】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
                                                  アセンブラ関数に渡します
void main()
{
        int         i = 0x02;
        int         j = 0x05;

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

図3.11　引数がある場合のアセンブラ関数の呼び出し例(smp2.c)

```
    .cline6
;## # C_SRC :           int         i = 0x02;
    ldm.W #0002H,DP:3 ;  i
    .cline7
;## # C_SRC :           int         j = 0x05;
    ldm.W #0005H,DP:1 ;  j
    .cline9
;## # C_SRC :           asm_func(i, j);
    ldy   DP:1  ;  j                        ←引数をX、Yレジスタを介して
    ldx   DP:3  ;  i                          アセンブラ関数に渡しています
    jsrl  _asm_func             ←アセンブラ関数の呼び出し('_'を付加しています。)
```

図3.12　smp2.cのコンパイル結果（抜粋）(smp2.a77)

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

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

●構造体型、共用体型の引数
●倍精度浮動小数点型(double)の引数

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

3.3.2　アセンブラ関数の記述方法

a.　呼び出されるアセンブラ関数の記述方法

アセンブラ関数の入り口処理の記述手順を以下に示します。

① アセンブラの疑似命令 .SECTION でセクション名を指定します。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 アセンブラの疑似命令 .FUNC で関数の開始を指定します。
③ 関数名ラベルをアセンブラの疑似命令 .PUB でグローバル指定します。
④ 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。
⑤ 関数内で DPR レジスタ及び DT レジスタの内容を書きかえる場合は、書き換えるレジスタをスタック上に退避してください。

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、全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時は、全てクリアした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。

⑥ 関数の入口処理で DPR レジスタまたは DT レジスタを退避した場合は、待避したレジスタを復帰してください。
⑦ m,x,D フラグを全てクリアします。
⑧ RTS 命令または RTL 命令を記述します。
⑨ RASM77 の疑似命令 .ENDFUNC で関数の終了を指定します。

【図3.13】にアセンブラ関数の記述例を示します。この例ではアセンブラ関数を格納するセクション名を、NC77がC言語プログラムに対して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
    .FUNC       _asm_func           ←②
    .PUB        _asm_func           ←③
_asm_func:                          ←④
    PHD                             ←⑤
    LDT         #10H
    STA         A, DT:MEM01
    STA         B, DT:MEM02
    SEM
    LDA.B       A, #7H
                :
              (省略)
                :
    PLT                             ←⑥
    CLP         m,x,D               ←⑦
    RTS                             ←⑧
    .ENDFUNC                        ←⑨
    .END
```

※①～⑥は、上記の手順に対応しています。

図3.13　アセンブラ関数の記述例

b.　アセンブラ関数からの戻り値の返し方

アセンブラ関数からＣ言語プログラムに値を返す場合、整数型、ポインタ型、浮動小数点型(float型のみ)については、レジスタ渡しで戻り値を返すことができます。【表3.3】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3.3　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
int型 nearポインタ型	Aレジスタ
float型 long型 farポインタ型	下位16ビットはAレジスタに、上位16ビットはBレジスタに格納して返します。
double型 long double型 構造体 共用体	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
        .FUNC    _asm_func
        .PUB     _asm_func
_asm_func:
                :
             (省略)
                :
        LDA.W   A, #01A0OH        ;32ビットデータの下位16ビット
        LDA.W   B, #0000H         ;32ビットデータの上位16ビット
        CLP     m,x,D
        RTS
        .ENDFUNC
        .END
```

図3.14　long型の戻り値を返すアセンブラ関数の記述例

c.　C言語の変数の参照方法

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

Ｃ言語の変数名をアセンブラ関数内で記述するときは、変数名の前に _(アンダースコア)を付加します。また、アセンブリ言語プログラムでは外部参照する変数をアセンブラの疑似命令 .EXT 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]
        .EXT            _counter            ←C言語プログラムの大域変数を
_asm_func:                                    外部参照宣言
                :
             (省略)
                :
        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ときの注意事項

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

1.関数の入口でレジスタ(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。

レジスタの待避と復帰は､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
    .FUNC      _int_func
    .PUB       _int_func
_int_func:
    CLP        m,x              ←m、xフラグをクリア
    PSH        A,B,X,Y,DPR,DT   ←レジスタの一括退避
    LDT        #__DT            ←割り込み処理プログラム
    SEM                           中で使用するDT値のセッ
    LDA.B      A,#7H              ト
          :
       (省略)
          :
    CLP        m,x              ←m、xフラグをクリア
    PUL        A,B,X,Y,DPR,DT   ←レジスタの一括退復帰
    RTI                         ←C言語プログラムへリターン
    .ENDFUNC
    .END
```

m、xフラグを同じ状態で退避／復帰します。

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ときの注意事項

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

(1)C 言語の関数名に _（アンダースコア）あるいは?(クエスチョン）を付加したラベル名で呼び出してください。
(2)m、x、D フラグは全てクリアされた状態で呼び出してください。
(3)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。
(4)DT（データバンクレジスタ）の値は、C 言語関数をコンパイルするときに指定した値をロードした上で呼び出してください。コンパイル時に指定しなかった場合、バンク値として 0 を DT にロードした上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フラグは、全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フラグ、xフラグ及びDフラグをクリア状態にしてください。

b.　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DPR及びDT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DPRの値を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.　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レジスタを介して渡すことができます。その書式を【図3.17】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(A, B)
```

図3.17　アセンブラ関数の記述例

#pragma PARAMETERは、16ビット及び8ビットレジスタ(A、B、X、Y)を介してアセンブラ関数に引数を渡します。また、16ビットレジスタを組み合わせて32ビットレジスタ(AB、XY)としてアセンブラ関数に引数を渡します。

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

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

- ●構造体型、共用体型の引数
- ●倍精度浮動小数点型(double)の引数

また、関数の戻り値として構造体、共用体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

NC77は、弊社製Cコンパイラ「NCxxx」とは、言語仕様レベル(拡張機能を含む)で基本的に互換性を有しています。ただし、以下の点について異なりますのでご注意ください。

a.　near / far のデフォルトの違い

NCシリーズの near / far のデフォルトは、以下の【表3.5】通りとなっています。このため、移植する時に、near / far 指定の調整を必要とする場合があります。

表3.5 NCシリーズの near / far デフォルト

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

3.4.2　7700ファミリの機種依存部に関する注意事項

7700ファミリの機種によってはハードウェア仕様が一部異なりますので、使用に際しては以下の注意事項を参照してください。

(1)SFR領域のレジスタに書き込む又は読み出す場合には特定の命令を使用しなければならないことがあります。この特定の命令は機種ごとに異なりますので、詳しくは各機種のユーザーズマニュアルを参照してください。この注意事項に関わる命令はasm関数を使用してプログラム中に直接命令を記述してください。
(2)M37700/M37701グループにおいて、プログラムをバンク0以外に配置する場合は、-OB2又は-OB3の起動オプションを必ず指定してください。
(3)RTS、JMP、JSR命令をバンクの最上位番地又はバンク境界にまたがって配置しないように注意してください。このような配置の可能性がある場合、リンク時にLINK77の警　告オプション(-C)を指定してください。この警告オプションは、RTS、JMP、JSR命令がバンク境界に配置される場合にワーニングメッセージを表示します。
(4)一部の機種において、スタック領域を最終バンク(バンク255)に配置する機能をもつものがあります。しかしNC77で使用する場合、この機能は使用できません。

3.4.3　移植全般に関する注意事項

Cコンパイラをバージョンアップした場合、一般に生成される機械語（アセンブリ言語）が変更されることがあります。したがって、以下の記述部に関して、新しいバージョンのCコンパイラが生成するアセンブリ言語の内容を必ず確認してください。

①処理速度に依存した記述部
②生成されるアセンブリ言語に依存した記述部

また、C77 V.2.10以前もしくはMR7700 V.2.12以前からの移植の場合、C言語ソースレベルにおいて多くの非互換項目があります。したがって、これらのバージョンから移植する場合、充分な移植期間をとってください。

3.4.4　C77 V.2.10以前からの移植に関する注意事項

a.　言語仕様に関する注意事項

(1)NC77 V.3.00以降では、C77 V.2.10以前で不可能であった符号付きの割算及び右シフトが可能になりました。したがって、これらの計算式を使用している場合、NC77 V.3.00以降とC77 V.2.10以前とで計算結果が異なる場合があります。

```
int         i, j;

i = -2;
j = i >> 1;
```

図3.18　符号付き演算の記述例

(2)NC77 V.3.00以降では、多次元配列のアドレスの扱いはANSI規格に準拠していますが、C77 V.2.10以前では独自の仕様で扱っています。したがって、多次元配列のアドレスに対する加減算を行った場合、結果が異なることがあります。C77 V.2.10以前の多次元配列のアドレスの取り扱いに関しては、C77 V.2.10ユーザーズマニュアルの19ページを参照してください。

(3)sizeof演算子を文字列に対して使用した場合、C77 V.2.10以前では文字列を格納する領域の先頭アドレスの大きさを返していました。NC77 V.3.00以降では、文字列を格納する領域の大きさを返すため結果が異なります。

```
sizeof("NC77");
```

戻り値	NC77 V.5.00以降	C77 V.2.10以前
sizeof("NC77");の返り値	5	スモールモデル：2
		ラージモデル：4

図3.19　sizeof演算結果の比較

(4)C77 V.2.10以前では、浮動小数点型をサポートしていないためfloat型、long double型をすべてlong int型として扱っています。NC77 V3.00以降では浮動小数点型をサポートしているため【図3.20】に示す記述をした場合、1.0は浮動小数点型（double型）ですので右辺の演算はdouble型で行われます。このため、コード効率が低下、又は実行速度が低下する場合があります。

```
i = j + 1.0;
```

図3.20　浮動小数点演算の記述例

(5)C77 V.2.10以前では構造体の配置において1バイトサイズのメンバーがあった場合は、1バイトのダミーを挿入していましたが、NC77では#pragma STRUCTで指定しない限りダミーの挿入は行わずに構造体の配置はパックされます。したがって、構造体上の配置の位置はC77 V.2.10以前とNC77とでは異なります。このため構造体へのポインタをchar ＊型などにキャストした上で演算する応用プログラム等は正常に動作しません。

(6)C77 V.2.10以前では【図3.21】に示す記述を行った場合、変数iの領域確保が行われていませんでした。NC77では標準C言語仕様に合わせて、領域確保が行われます。このため、V.2.10以前を用いて作られたプログラムをNC77に移植した場合、リンク時に多重定義エラーが発生することがあります。この場合には標準C言語仕様にあわせた記述に変更してください。

```
extern int i;

int i = 0;
```

図3.21　変数のextern宣言例

(7)本来はエラーとなるべき項目がC77 V.2.10以前ではエラーとならない項目があります。例えば、【図3.22】に示すソースプログラムでは、関数gfの再定義エラーを出力しなくてはなりませんがC77 V.2.10以前ではエラーになりませんでした。このような場合、NC77 V.3.00以降ではエラーとなりますので、必ず前もって関数のプロトタイプ宣言を行ってください。

```
func()
{
    unsigned int i;
    i = gf();
}

unsigned int gf()
{
    return  0;
}
```

図3.22　関数の再定義エラーを出力する記述例

b.　アセンブラ関数とのインタフェースに関する注意事項

NC77 V.3.00 以降と C77 V.2.10 以前では、以下の関数の戻り値の格納方法が異なっています。したがって、以下に示す関数に相当するアセンブラ関数は変更してください。

●構造体を返す関数
●double型を返す関数

戻り値の格納方法の詳細は、本書の第８章「アセンブリ言語プログラムと結合方法」と C77 V.2.10 ユーザーズマニュアルの 86 ページを参照してください。

c.　asm関数の使用に関する注意事項

(1)CLM等の命令を使用してデータ長選択フラグ(m)及びインデックスレジスタ長選択フラグ(x)を変更している記述は、必ず【図3.23】に示す記述に変更してください。

```
        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3.23　m、xフラグの切り換え記述形式

(2)記憶クラスautoの変数もしくは引数をダイレクトページレジスタ(DP)のオフセット値を使用して指定している場合、【図3.24】に示す記述に変更してください。

```
        asm("          オペコード      A, DP:$$", auto変数名);
```

図3.24　DPオフセット指定の記述書式

d.　#pragma EQUの互換性に関する注意事項

NC77 V.3.00以降では､絶対アドレスが指定された変数はそのアドレスに変数の実体が既に確保されているものとしてコンパイルします。したがって、その変数に対するextern宣言及びstatic宣言は無視されます。また､絶対アドレスのシンボルをグローバルシンボルとすることも行いません。

NC77 V.3.00以降でリンク時に#pragma EQUで指定したシンボルが解決しない旨のエラーが発生した場合、以下の方法で対処してください。

(1)C言語のプログラム中でエラーが発生した場合

#pragma EQU宣言を1つにまとめたヘッダファイルを作成し、そのヘッダファイルを各プログラムの先頭でインクルードしてください。

(2)アセンブラ関数中でエラーが発生した場合

C言語プログラム中で宣言した#pragma EQU変数をアセンブラ関数から呼び出す場合、呼び出す変数をすべてアセンブラ関数中で疑似命令.EQUを使用して指定してください。この疑似命令群を以下の方法で簡単に作成することができます。

①呼び出す変数をC言語のヘッダファイルとして作成し、それをNC77 V.3.00以降で-Sオプションを指定してアセンブリ言語ソースファイル出力までコンパイルします。

②そのアセンブリ言語ソースプログラムには.EQU疑似命令群が含まれますのでこれをアセンブリ言語ソースファイルの先頭にコピーします。

e.　C77 V.2.10以前でコンパイルしたプログラムの取り扱いに関する注意事項

NC77 V.3.00以降とC77 V.2.10以前とでは関数の呼び出し形式が異なります。そのため、C77 V.2.10以前でコンパイルしたライブラリ、オブジェクトファイルをNC77 V.3.00以降でコンパイルしたライブラリ、オブジェクトファイルとリンクした場合、正しく動作しません。このような場合、すべてのライブラリ、オブジェクトファイルをNC77 V.3.00以降で再コンパイルしてください。

f.　#pragma INTFで宣言した割り込み処理関数の取り扱いに関する注意事項

#pragma INTF宣言した割り込み処理関数の戻り値及び引数は必ずvoid宣言してください。

```
#pragma INTF     intfunc
intfunc()
{
          :
        (省略)
          :
}
```

```
#pragma INTF     intfunc
void  intfunc(void)
{
          :
        (省略)
          :
}
```

図3.25　割り込み処理関数の変更例

g.　標準入出力ライブラリ関数に関する注意事項

NC77 V.3.00以降では、以下に示す標準入出力ライブラリ関数で扱うポインタ変数は製品添付のライブラリではnear型でコンパイルしています。

表3.6　標準入出力ライブラリ関数一覧表

関数名	関数名
fgetc	fputs
getc	puts
fgets	fwrite
gets	printf
fread	fprintf
scanf	sprintf
fscanf	ungetc
sscanf	ferror
fputc	feof
putc	

従来のラージモデルを使用し、ポインタ変数をfar型として扱う場合は、標準ライブラリ関数のソースファイルのディレクトリにあるメイクファイルmake.far（MS-DOS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。

h　peek, pokeライブラリ関数に関する注意事項

NC77 V.3.00以降では、farポインタを使用することにより7700ファミリの全メモリ空間を自由にアクセスできるようになりました。このため、C77 V.2.10以前にあったpeek、pokeライブラリ関数は削除しています。

peek、pokeライブラリ関数は、memcpy又はbcopyライブラリ関数等に変更してください。

i.　divr, modrライブラリ関数に関する注意事項

NC77 V.3.00 以降では、符号付きの割算が可能になりました。このため、C77 V.2.10 以前にあった divr、modr ライブラリ関数は削除しています。

演算子 '/' 又は '%' を使用した記述に変更入力してください。

j.　-Zaオプションの廃止とchar型引き数の扱い変更に関する注意事項

C77 V.2.10 以前では、char 型の引き数をもつ関数を呼び出すときにその引き数を 8 ビットで積む、もしくは 16 ビットで積むかを -Za オプションで制御していました。NC77 V.3.00 以降では -Za オプションを廃止し、char 型引き数を 8 ビットで積む、もしくは 16 ビットで積むかはその関数の引き数のプロトタイプ宣言の有無により制御するようになりました。※1

このため、char 型引き数を持つ関数を使用しているファイルと、その関数を定義しているファイルとの両方に共通のその関数のプロトタイプ宣言（関数の引き数の型宣言）が必要になります。

k.　プロトタイプ宣言に関する注意事項

NC77 V.3.00以降では、関数呼び出し時の引数のスタックへの積み方がプロトタイプ宣言の有無により異なります。このため、C77 V.2.10 以前を使用して作成されたプロトタイプ宣言があるプログラム中で、プロトタイプ宣言された関数に対するすべての呼び出し側及び定義側で一貫してプロトタイプ宣言されていない場合、NC77 V.3.00以降では正常に動作しないことがあります。したがって、プロトタイプ宣言は必ず関数の呼び出し・定義を行っているすべてのファイルで宣言してください。

l.　セクション名に関する注意事項

NC77 V.3.00 以降が出力するセクション名と C77 V.2.10 以前が出力するセクション名は異なります。したがって、コンパイラが出力するセクション名を利用するアセンブリ言語プログラムはセクション名を変更してください。

3.4.5　NC77 V.3.00からの移植に関する注意事項

a.　-fext_const_set_rom_section(-fECSRS)オプションに関する注意事項

nc77 V.3.00 の起動オプション -fext_const_set_rom_section(-fECSRS)をデフォルト化しました。従来と同じデフォルト状態でコンパイルする場合は、nc77の起動オプション -fconst_not_ROM（-fCNR）を指定してコンパイルしてください。

b.　メモリ管理ライブラリ関数に関する注意事項

NC77 V.3.00 では、以下のメモリ管理ライブラリ関数は、near 領域からメモリの確保及び解放を行っていましたが、NC77 V.3.10 から far 領域に変更しました。したがって、メモリ管理ライブラリ関数が扱うポインタを far ポインタに変更してください。

※1.ANSI規格に合わせるために、この変更を行いました。

表3.7　メモリ管理ライブラリ関数一覧表

関数名
calloc
free
malloc
realloc

3.4.6　MR7700 V.2.12以前からの移植に関する注意事項

バージョン2.12以前のリアルタイムOS MR7700を使用して開発されたアプリケーションプログラムを移植する場合、タスク、ハンドラの指定には、NC77 V.3.00から新機能として追加しました＃ pragma TASK及び＃ pragma INTHANDLERを使用してください。この機能を使用することで､タスク及び割り込みハンドラの入口処理と出口処理に関するコードを自動的に生成します。

①タスクは、#pragma TASKで指定してください。
②#pragama INTHANDLERで割り込みハンドラを指定することでIntEntryマクロとret_intシステムコールを呼び出す必要はありまん。記述からは削除してください。

以上2点の変更例を【図3.26】に示します（図中、tasはタスク名、handは割り込みハンドラ名です)。

```
#pragma INTF hand

tas()
{
            :
          (省略)
            :
        ext_tsk();
}
hand()
{
        IntEntry();
            :
          (省略)
            :
        ret_int();
}

        ▼

#pragama TASK tas
#pragama INTHANDLER hand

void tas()
{
            :
          (省略)
            :
/*        ext_tsk(); */        ←コメント又は削除
}
void hand()
{
/*      IntEntry(); */        ←コメント又は削除
            :
          (省略)
            :
/*      ret_int();  */        ←コメント又は削除
}
```

図3.26　タスクと割り込みハンドラの書式変更例

③割り込みハンドラからの復帰とタスクの起床を行うret_wupシステムコールは、ret_intシステムコールと同等の処理と、指定したタスクの起床処理を行います。割り込みハンドラを#pragma INTHANDLERで指定している場合は、ret_intに相当するコードは自動的に生成されますので、指定タスクを起床するiwup_tskシステムコールに変更してください。ret_wupシステムコールの変更例を【図3.27】に示します（図中、tasはタスク名、handは割り込みハンドラ名です）。

```
#pragma INTF hand()

hand()
{
    IntEntry();
          :
       (省略)
          :
    ret_wup(ID_tas);
}

         ▼

#pragama TASK tas
#pragama INTHANDLER hand

void hand(void)
{
    /*      IntEntry(); */          ←コメント又は削除
          :
       (省略)
          :
    iwup_tsk(ID_tas);          ←iwup_tskシステムコールに変更
}
```

図3.27　ret_wupシステムコールの書式変更例

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nc77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、nc77から起動できるアセンブラrasm77とリンケージエディタlink77の起動オプションを併せて記載しています。

A.1　nc77コンパイルドライバの入力書式

```
%  nc77△[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　nc77コマンドの入力書式

```
% nc77 -osample -rasm77 "-l" -link77 "-ms" ncrt0.a77 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　nc77コマンドの入力例

A.2　nc77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.r77)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.a77)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、及び-Sを指定しない場合、nc77はlink77まで制御を行い、機械語データファイル(拡張子.hex)まで作成します。

-C

コンパイラドライバの制御

機　能：　リロケータブルオブジェクトファイル(拡張子.r77)を作成し、処理を終了します。

実行例：

```
%nc77 -c sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c

% ls sample.*
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
-rw-r-----  1 toolusr       450 Aug 17 11:28 sample.r77
%
```

注意事項：　このオプションを指定したときは、機械語データファイル(拡張子.hex)等、link77で処理した結果出力されるファイルは生成されません。

-D*識別子名*

コンパイラドライバの制御

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

書　式：　nc77△-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*

コンパイラドライバの制御

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：　nc77△-I*ディレクトリ名*△<C言語ソースファイル名>

実行例：

```
% nc77 -c -I./test/include -I./test/inc sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c

%
```

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

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

-E

コンパイラドライバの制御

機　能：　プリプロセスコマンドのみを起動し結果を標準出力に出力します。

実行例：

```
% nc77 -E sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

#line 1 "sample.c"
                    :
                (省略)
                    :
#line 1 "/usr3/tool/toolusr/work30/inc77/stdio.h"
                    :
                (省略)
                    :
```

注意事項：　このオプションを指定したときは、アセンブリ言語ソースファイル(拡張子.a77)、リロケータブルオブジェクトファイル(拡張子.r77)、機械語データファイル(拡張子.hex)等、ccom77、rasm77、及びlink77で処理した結果出力されるファイルは生成されません。

-P

コンパイラドライバの制御

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

実行例：

```
% nc77 -P sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

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

注意事項：
1. このオプションを指定したときは、アセンブリ言語ソースファイル(拡張子.a77)、リロケータブルオブジェクトファイル(拡張子.r77)、アブリュートモジュールファイル(拡張子.hex)等、ccom77、rasm77、及びlink77で処理した結果出力されるファイルは生成されません。
2. このオプションにより生成されるファイル(拡張子.i)には、プリプロセッサが生成する#lineは含まれません。#lineを含む結果を得る場合には、-Eオプションを指定し、リダイレクトしてください。

-S

コンパイラドライバの制御

機　能：　アセンブリ言語ソースファイル(拡張子.a77)を作成し、処理を終了します。

実行例：

```
% nc77 -S sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
% ls sample.*
-rw-r-----  1 toolusr      2059 Aug 17 11:30 sample.a77
-rw-r--r--  1 toolusr      2835 Aug 17 11:28 sample.c
%
```

注意事項：　このオプションを指定したときは、リロケータブルオブジェクトファイル(拡張子.r77)、機械語データファイル(拡張子.hex)等、rasm77及びlink77で処理した結果出力されるファイルは生成されません。

-U*プリデファインドマクロ名*

コンパイラドライバの制御

機　能：　プリデファインドマクロ定数を未定義にします。

書　式：　nc77△-U*プリデファインドマクロ名*△<C言語ソースファイル名>

実行例：

```
% nc77 -c -UNC77 sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c

%


※この例では、2つのマクロ定義NC77とM16Cを未定義にしています。
```

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

-silent

コンパイラドライバの制御

機　能：　起動時のコピーライトメッセージを出力しません。

実行例：

```
% nc77 -c -silent sample.c
sample.c

%
```

A.2.2　出力ファイル指定オプション

【表A.2】に出力する機械語データ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	link77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、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	link77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、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：　link77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、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：　nc77△-oファイル名△<C言語ソースファイル名>

実行例：

```
% nc77 -o./test/sample ncrt0.a77 sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a77
sample.c
% cd test
% ls
total 65
drwxr-x---  2 toolusr       512 Aug 17 16:13 ./
drwxrwxrwx 11 toolusr      3584 Aug 17 16:14 ../
-rw-r-----  1 toolusr     44040 Aug 17 16:14 sample.hex

%
```

※この例では、ディレクトリ./testに機械語データファイルsample.hex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：　nc77△-dir　*ディレクトリ名*

実行例：

```
% nc77 -dir./test/sample ncrt0.a77 sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a77
sample.c
% cd test/sample
% ls
total 65
drwxr-x---  2 toolusr       512 Aug 17 16:13 ./
drwxrwxrwx 11 toolusr      3584 Aug 17 16:14 ../
-rw-r-----  1 toolusr     44040 Aug 17 16:14 ncrt0.hex

%
```

※この例では、ディレクトリ./test/sampleに機械語データファイルncrt0.hex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：

```
% nc77 -c -v sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
cpp77 sample.c -o sample.i -DMELPS -DMELPS7700 -DNC77
ccom77 sample.i -o ./sample.a77
rasm77 -. -N sample.a77
loop77 -zopt77 -. test.a77 -o.

%
```

注意事項：　このオプションは、小文字のvを記述します。

-V

バージョン情報の表示

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

実行例：

```
C:\USR\>nc77 -V
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

NC77 Compile Driver for 7700 Family Version 4.XX.XX
NC Preprocessor                 Version 4.XX.XX
NC77 Compiler for 7700 Family   Version 2.XX.XX (NC_CORE Version 2.XX.XX)
Relocatable Macro Assembler for 7700 Family Version V.5.XX.XX
NC77 Branch Optimizer for 7700 Family Version 1.XX.XX
7700 Family LINKER V.2.XX.XX
NC77 IEEE-695 Object Format Converter for 7700 Family Version 1.XX.XX

C:\USR>
```

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「NC77WA V.5.10 導入ガイド」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】にデバッグ関連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
-gie	なし	C言語レベルのデバッグ情報(IEEE-695アブソリュート形式ファイル(拡張子.ie))を出力します。
-gie_no_local_symbl	-gILNS	アセンブリファイル中のローカルシンボルをIEEE-695アブソリュート形式ファイル(拡張子.ie)に出力しません。
-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。
-g	なし	デバッグ時に必要なシンボルファイル(拡張子.sym)を出力します。本オプション指定時は、C言語レベルのデバッグ情報は出力しません。

-gie

IEEE-695アブソリュート形式ファイルの作成

機　能：　C言語レベルのデバッグ情報(IEEE-695アブソリュート形式ファイル(拡張子.ie))を出力します。

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

IEEE-695アブソリュート形式ファイルをサードパーティ製エミュレータ又はシミュレータ等に読み込ませた場合、IEEE-695に規定されていない情報等の相違により、一部の機能が正常に動作しない、又は読み込めない等の障害が発生する可能性があります。
万一、このような障害が発生した場合、弊社にて解決できない場合があることをご了承ください。なお、動作環境に関しては製品添付のリリースノートを参照してください。

-gie_no_local_symbol　　-gINLS

IEEE-695アブソリュート形式ファイルの作成

機　能：　IEEE-695アブソリュート形式ファイル(拡張子.ie)を出力します。
ただし、アセンブリ言語ファイル中のローカルシンボルをIEEE-695ファイルには出力しません。

注意事項：　NC77 V.3.20と同形式のIEEE-695ファイルを生成するためのオプションです。通常は指定する必要はありません。

-genter

スタックフレームの生成コード出力

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。

-g

シンボルファイルの作成

機　能：　デバッグ時に必要なシンボルファイル(拡張子.sym)を作成します。

注意事項：　本オプション指定時は、auto変数の参照等、C言語レベルのデバッグ情報を出力しません。

A.2.5　最適化オプション

【表A.5】にプログラムの実行速度及びROM容量を最小にする最適化を行う起動オプションを示します。

表A.5　最適化オプション

オプション	短縮形	機　能
-O[1～5]	なし	デバッグに影響を与えない最適化のみ実行します(数字が大きいほど最適化が強くなります)。
-OR	なし	速度よりもROM容量を重視した最大限の最適化を行います。
-OS	なし	ROM容量よりも速度を重視した最大限の最適化を行います。
-Oconst	-OC	const修飾子で宣言した、外部変数の参照を定数で置き換える最適化を行います。
-Ono_bit	-ONB	ビット操作をまとめる最適化を抑止します。
-Ono_break_source_debug	-ONBSD	ソース行情報に影響する最適化を抑止します。
-Ono_float_const_fold	-ONFCF	浮動小数点の定数畳み込み処理を抑止します。
-Ono_stdlib	-ONS	標準ライブラリ関数のインライン埋め込みやライブラリ関数の変更等を抑止します。
-Osp_adjust	-OSA	スタック補正コードを取り除く最適化を行います。これによりROM容量を削減することができます。ただし、使用するスタック量が多くなる可能性があります、
-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。

最適化オプション効果一覧

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

良：良く(もしくは同じ)なることを意味します。
悪：悪く(もしくは同じ)なることを意味します。
同：変化がな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、-gieオプションと同時に指定することができます。
数字(最適化レベル)を指定しない場合は、-O3と等価です。

補足説明：速度には効果があるが、コードサイズが増える。あるいはコードサイズは増えるが、速度は低下するような最適化は行いません。

-O1：　-O3、-Ono_bit、-Ono_break_source_debug、-Ono_float_const_fold、-Ono_stdlibを有効にします。

-O2：　-O1と同じです。

-O3：　速度及びROM容量ともに効果のある最適化を行います。

-O4：　-O3及び-Oconstを有効にします。

-O5：　-O4を有効にします。かつ、共通部分式の最適化等を強化した最大限の最適化を行います。

※-O5の最適化は、異なるポインタ変数が同時に同じ領域を指さないことを前提にした最適化を行います。このため下例のようなプログラムに対しては正常なコードを出力できません。

・ポインタ等により異なる変数が、同時に同じ箇所を指す。
・かつ上記変数が同一関数内で使用する。

```
例)
int a=3, b;
int *p = &a;
main()
{
    test1();
}
test1()
{
    *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 = 10)
```

-OR

最適化

機　能：　速度は低下する場合がありますが、ROM容量を最小にする最大限の最適化を行います。このオプションは、-g、-gieオプションと同時に指定することができます。

注意事項：　このオプションを使用した場合、ソース行情報の一部を変更する最適化を行なう可能性があります。ソース行情報を変更したくない場合、-Ono_break_source _debug(-ONBSD)オプションを使用して最適化を抑止してください。

-OS

最適化

機　能：　ROM容量は増大する場合がありますが、速度重視の最大限の最適化を行います。このオプションは、-g、-gie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に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

最適化の行われる記述を以下に示します。

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：-O(もしくは-OR、-OS)オプションを指定した場合には、ソース行情報に影響する最適化を行う可能性があります。本オプションは、ソース行情報に影響する最適化を抑止する場合に使用します。

注意事項：-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_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：NC77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、浮動小数点で表現できる値を限界まで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：　オプション-Osp_adjustによりROM容量を削減することができます。ただし、使用するスタック量が多くなる可能性があります。

-Ostack_frame_align　　-OSFA

スタックフレームのアライメント

機　能：　スタックフレームの、偶数アライメントを行います。

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

注意事項：　(1)以下の#pragmaで指定した関数はアライメントを行いません。

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

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

※1.割り込みが発生したタイミングで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　分岐命令の選択オプション

【表A.6】にnc77が生成するアセンブリ言語ソースファイル中の分岐命令の選択を行う起動オプションを示します。

表A.6　分岐命令の選択オプション

オプション	機　能
-OB1	コードサイズのみを考慮した分岐命令の生成を行います(デフォルト)。 ブランチ命令の変換順序：bra→bral→jmpl
-OB2	速度を考慮した分岐命令の生成を行います(同一バンク内)。 ブランチ命令の変換順序：bra→jmp
-OB3	速度を考慮した分岐命令の生成を行います(バンク外)。 ブランチ命令の変換順序：bra→jmpl

※-OSオプションと組み合わせて-OB2又は-OB3オプションを使用した場合は、bra命令を出力しません。

-OB1

分岐命令の選択

機　能：　コードサイズのみを考慮した分岐命令の選択を行います。分岐命令を以下の規則で選択します。
分岐命令の選択順序：BRA→BRAL→JMPL

注意事項：　分岐命令の選択オプションを指定しない場合は、-OB1オプションと全く同じ規則で分岐命令が選択されます。

-OB2

分岐命令の選択

機　能： 速度を考慮した分岐命令の選択を行います。ただし、選択された分岐命令により分岐できる範囲は同一バンク内に限られます。選択された分岐命令で分岐できない場合には、リンク時にエラーとなります。分岐命令を、以下の規則で選択します。
分岐命令の選択順序：BRA→JMP

注意事項： -OSオプションと組み合せて指定した場合、JMP命令のみを選択します。

-OB3

分岐命令の選択

機　能： 速度を考慮した分岐命令の選択を行います。また、選択された分岐命令は7700ファミリの全メモリ空間に対して分岐できます。分岐命令を、以下の規則で選択します。
分岐命令の変換順序：BRA→JMPL

注意事項： -OSオプションと組み合せて指定した場合、JMPL命令のみを選択します。

A.2.7　生成コード変更オプション

【表A.7】にnc77が生成するアセンブリ言語を制御する起動オプションを示します。

表A.7　生成コード変更オプション

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。
-fno_even	-fNE	データ出力時に奇数データと偶数データを分離しないで、すべてodd属性のセクションに配置します。
-fshow_stack_usage	-fSSU	スタックの使用状況をファイル(拡張子.stk)に出力します。
-ffar_RAM_data	-fFRAM	RAMデータのデフォルト属性をfarにします。
-ffar_ROM_data	-fFROM	ROMデータのデフォルト属性をnearにします。
-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jsrで呼び出し、rts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。
-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=	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。
-fswitch_table	-fST	ROM効率が良くなる場合のみswitchu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。
-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	#pragmaADDRESS(#pragmaEQU)で指定した変数をvolatileで指定した変数とみなしません。
-fsmall_array	-fSA	far 型の配列を参照する場合、その総サイズが64Kバイト以内であれば、添字の計算を16ビットで行ないます。
-fenable_register	-fER	レジスタ記憶クラスを有効にします。
-fuse_DIV	-fUD	除算に対するコード生成を変更します。

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

-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生成を行います。

-fnot_reserve_asm　　-fNRA

生成コードの変更

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

-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;

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

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

-fchar_enumerator　-fCE

生成コードの変更

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

注意事項：　型デバッグ情報には型のサイズ情報は含まれていません。
このため、このオプションを指定した場合、デバッガによっては正しくenum型を参照できない場合があります。

-fno_even　　-fNE

生成コードの変更

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

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

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

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

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

-fshow_stack_usage　　-fSSU

生成コードの変更

機　能：　スタックの使用状況をファイル（拡張子.stk）に出力します。

補足説明：本オプションで生成されるファイルを用いて、スタックサイズ計算ユーティリティstk77が、プログラムで使用するスタックサイズを算出します。

注意事項：asm関数内で使用されるスタックの使用状況は出力されません。また、stk77を使用した場合も、同様にasm関数内で使用されるスタックは計算されません。

-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。

補足説明：　RAMデータ（変数）は、デフォルトでnear領域に配置されます。

-ffar_ROM_dara　　-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。

補足説明：　ROMデータ（const指定された変数等）は、デフォルトでnear領域に配置されます。

-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。

-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、7700ファミリ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、NC77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： nc77△-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.a77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。

-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： 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。

-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。本オプションを指示することにより、この「array」をRAM領域に配置することができます。

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

通常の用途では、本オプションを使用する必要はありません。

-fsmall_array　-fSA

生成コードの変更

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

補足説明: far型配列の要素を参照する場合に、配列の総サイズが不明であれば、64Kバイト以上の配列にも対応できる様に添字の計算を32bit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ので、添字「j」の計算を32bitで行います。
本オプションを指定することにより、配列「array」の総サイズを64Kバイト以下と仮定して、添字「j」の計算を16bitで行います。この結果処理速度の向上、コードサイズの削減が可能となります。
一つの配列のサイズが64Kバイトを超えないのであれば、常に本オプションを使用することを推奨します。

-fenable_register　-fER

レジスタ記憶クラス

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

補足説明：　auto変数をレジスタに割り当てる最適化において必ずしも全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ました。
本オプションを指定することにより、register指定された、

1. 整数型変数
2. ポインタ変数

を強制的にレジスタに割り当てます。

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

-fuse_DIV　　-fUD

生成コードの変更

機　能：　除算に対する生成コードを変更します。

補足説明：除算時に被除数が4バイト値、除数が2バイト値で、かつ演算結果が2バイト値の場合に7700ファミリのDIV命令及びDIVS命令(775xシリーズのみ)を生成します。

注意事項：7700ファミリのDIV命令は、over flowする場合の演算結果は不定値です(詳細は各マイコンのマニュアルをご覧ください)。演算結果を保証するために、通常は被除数が4バイト値、除数が2バイト値で、かつ演算結果が2バイト値の場合でもランタイムライブラリを呼び出します。
本オプションを指定した場合に除算結果がover flowすると、ANSIの規定とは異なる動作を行います。

A.2.8　警告オプション

【表A.8】にnc77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。
-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	ccomnocc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。
-Wmake_tagfile	-WMT	タグファイルに生成メッセージの内容を出力します。
-Wuninitialize_variable	-WUV	初期化されていない auto変数に対してワーニングを出力します。
-Wlarge_to_small	-WLTS	大きいサイズから、小さいサイズへの暗黙の転送に対して、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：

```
A> nc77 -c -Wstdout sample.c > err.doc

A> type err.doc
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999, MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

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

A>
```

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

注意事項：コンパイルドライバから呼び出されるrasm77、link77のエラー出力は、本オプションに関係なくWindows版(パソコン版)であれば標準出力に出力されます。

-Werror_file <file name>　　-WEF

警告オプション

機　能：　エラーメッセージを指定したファイルに出力します。

書式：　nc77△-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、ワーニングが発生した場合にコンパイル処理を停止したいときに使用します。

-Wnesting_comment　-WNC

警告オプション

機　能：　コメント内に"/*"を記述している場合に警告を発生します。

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

-Wccom_max_warnings=最大ワーニング数　　-WCMW

警告オプション

機　能：　コンパイラ本体が出力するワーニングの上限回数を指定します。

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

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

-Wall

警告オプション

機　能：　検出可能な警告をすべて表示します。
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、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を生成してメッセージの内容を出力します。

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

-Wuninitialize_variable　-WUV

警告オプション

機　能：　初期化されていない auto変数に対してワーニングを出力します。

-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。

A.2.9　アセンブル / リンクオプション

【表A.9】にrasm77及びlink77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
-rasm77△<*オプション*>	アセンブルコマンド rasm77 のオプションを指定します。2個以上のオプションを渡す場合は、" (ダブルクオーテーション)で囲んでください。
-link77△<*オプション*>	リンクコマンドlink77 のオプションを指定します。2個以上のオプションを渡す場合は、 " (ダブルクオーテーション)で囲んでください。

-rasm77″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rasm77 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：　nc77△-c△-rasm77△″*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：

```
% nc77 -c -v -rasm77 "-l -s" sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c
cpp77 sample.c -o sample.i -DMELPS -DMELPS7700 -DNC77
ccom77 sample.i -o ./sample.a77
loop77 -zopt77 -. -l -s sample.a77
% ls sample.*
-rw-r-----  1 toolusr      2059 Aug 17 15:43 sample.a77
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr       597 Aug 17 15:43 sample.ext
-rw-r-----  1 toolusr     10508 Aug 17 15:43 sample.prn     ←
-rw-r-----  1 toolusr       587 Aug 17 15:43 sample.r77
%
```

※この例では、アセンブルコマンドのオプションとして、プリントファイル(拡張子.prn)を生成する指定を行っています。

注意事項：1. ブランチオプティマイザloop77がrasm77を起動したときに、rasm77の起動オプション-.と-Cを自動的に指定します。
2. RASM77の-B、-E、-Oオプションおよび-Qオプションは指定しないでください。

参考としてRASM77 V.5.00 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RASM77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。
-B	ビット長の整合性をチェックします。この指定をした場合、疑似命令".BYTE"、".WORD"、".BLKB"、".BLKW"で宣言したローカルラベルの参照時に、".DATA"、".INDEX"で宣言しているビット長とが一致していない場合にワーニング6を出力します。 ※このオプションは指定しないでください。
-C	ソースデバッグ情報をオブジェクトファイルに出力します。 デバッグ時にソースデバッグを行う場合、アセンブル時にこのオプションを指定してください。
-D	シンボルに数値を設定します。コマンドで設定されたシンボルは、疑似命令.EQUで定義されたシンボルと同等に扱われます。
-E	タグファイル(拡張子.tag)の生成及びエディタの起動を行います。 ※このオプションは指定しないでください。
-L	プリントファイル(拡張子.prn)を生成します。この指定がない場合プリントファイルは生成しません(ただし、-Mオプションを指定した場合はこの指定がなくてもプリントファイルを生成します)。
-LC	.IF命令による条件アセンブル部の条件偽の部分もプリントファイルへ出力します。この指定がない場合、プリントファイルへは条件偽の部分は出力しません。
-M	マクロ展開部をプリントファイルへ出力し、プリントファイルを生成します。この指定がない場合、プリントファイルへはマクロ展開部は出力しません。
-Q	疑似命令".EQU"による再設定時にワーニングを出力します。この指定がない場合は、同一シンボルに対して異なる値を設定した場合にもエラーとなりません。 ※このオプションは指定しないでください。
-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。 ※このオプションは指定しないでください。
-S	ローカルシンボル情報をオブジェクトファイルに出力します。
-U	ラベル直後の':'(コロン)の省略を許可　します。
-X	アセンブル終了後、クロスリファレンサCRF77を起動します。

※ NC77は、オプション-rasm77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RASM77のオプション-B、-E、-Oおよび-Qを指定しないで下さい。

●NC77は、アセンブル時にワーニングメッセージを出力しないようにコードを生成します。しかし、RASM77のオプション-B、-E、-Oおよび-Qを指定した場合には、NC77では利用されないワーニングメッセージをRASM77が出力するため、コンパイルが途中で終了してしまいます。

-link77"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link77 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：　nc77△-c△-link77△"*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：

```
% sun:toolusr(361)-> nc77 -g -v -osample -link77 -ms ncrt0.a77 sample.c
NC77 COMPILER for 7700 FAMILY V.5.10 Release 1
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

ncrt0.a77
loop77 -zopt77 -. -C ncrt0.a77
sample.c
cpp77 sample.c -o sample.i -DMELPS -DMELPS7700 -DNC77
ccom77 sample.i -o ./sample.a77
loop77 -zopt77 -. -C sample.a77


link77 ncrt0.r77 sample.r77 , , , -. -s -M -ms -o. -fsample ,
processing "ncrt0.r77"
processing "sample.r77"
processing "Libraries"
processing "ncrt0.r77"
processing "sample.r77"
processing "/usr3/tool/toolusr/work77/lib77/nc77lib.lib ( fprintf.r77 )"
          :
        (省略)
          :
% ls sample.*
-rw-r--r--  1 toolusr      2850 Aug 17 14:51 sample.c
-rw-r-----  1 toolusr     44040 Aug 17 15:47 sample.hex
-rw-r-----  1 toolusr      8310 Aug 17 15:47 sample.map     ←
-rw-r-----  1 toolusr      8819 Aug 17 15:47 sample.sym
%
```

※この例では、リンクコマンドのオプションとして、マップファイル(拡張子.map)を生成する指定を行っています。

注意事項：　nc77がlink77を起動したときに、link77の起動オプション-o、-fを自動的に指定します。
このとき-oは、出力ファイルのディレクトリを指定します。また–fは、nc77のコマンドラインで一番最初に入力されたファイル名(拡張子.a77、.r77、.c)を指定します。任意のディレクトリ名とファイル名を指定する場合、nc77の起動オプション-oで出力先のディレクトリ名とファイル名指定してください。

参考としてRASM77 V.5.00の製品パッケージに含まれているLINK77のオプションを以下に示します。

オプション	機　能
-A	同名の絶対属性セクションのオーバーラップを許可します。グローバルなメモリ領域を共有結合する場合に利用できます。
-C	特定の分岐命令がバンク境界にあった場合にワーニングを出力します。ただし、命令の機械語と同じ値のデータが存在した場合もワーニングメッセージを出力します。
-F	出力ファイル名を指定します。 ※このオプションは指定しないでください。
-M	マップファイル(属性.map)の出力を行います(セクション情報のみ)。
-MS	マップファイルを、グローバルラベル及びグローバルシンボルリスト付きで出力します。
-N	ソースファイル中の疑似命令.OBJ、.LIBにより指定されたリロケータブルファイル(属性.R77)、ライブラリファイル(属性.LIB)の参照指定情報を無視します。
-O	出力ファイルのディレクトリを指定します。 ※このオプションは指定しないでください。
-S	シンボルファイル(属性.sym)の出力を行います。
-V	リロケータブルファイル間のバージョン整合性の確認を行います。ただし、確認を行なうためには、アセンブリ言語ソース中で疑似命令.VERによりバージョンの指定を行なっておく必要があります。
-W	セクションの配置をワードアライメントで行います。

※ NC77は、オプション-link77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link77のオプション-Fおよび-Oを指定しないで下さい。

A.2.10　775x対応コード生成オプション

【表A.10】に775x対応コード生成オプションを示します。

表A.10　775x対応コード生成オプション

オプション	短縮形	機　能
-m7750	なし	7750/7751シリーズに対応したコードを生成します。

-m7750

7750/7751シリーズ対応コードの生成

機　能：　7750/7751シリーズに対応したコードを生成します。

注意事項：　このオプションを使用する場合は、必ず7750/7751シリーズ対応のライブラリファイルを使用してください。

A.2.11　その他のオプション

【表A.11】にnc77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起動オプションを示します。

表A.11　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言語ソースリストをコメントとして出力します。

-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。
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。

A.3　nc77起動オプションに関する注意事項

A.3.1　nc77起動オプションの記述に関する注意事項

nc77の起動時オプションは、アルファベットの大文字と小文字を区別します。
誤って入力した場合、そのオプションによる機能は取り消されます。

A.3.2　nc77の制御に関するオプションの優先順位

nc77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.77)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.a77)を作成して処理を終える

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

付録B
拡張機能リファレンス

NC77は、7700ファミリを用いたシステムへの組み込みを容易にするために独自の拡張機能を追加しています。

付録Bでは、言語仕様に関する機能以外の拡張機能の使用方法を説明します。

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、アブソリュート系又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 ..... 同一バンク内(64Kバイト以内の領域)のアクセス far ..... バンク外(64Kバイトを越える領域)のアクセス 2.関数の呼び出しにJSR命令又は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フラグの切り換えをコンパイラに指示することができます。 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。 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]); 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.文字列中に漢字文字を使用することができます。 記述例) L"漢字" 2.漢字文字の文字定数を使用することができます。 記述例) L'漢' 3.コメント中に漢字文字を記述することができます。 記述例) /＊ 漢字 ＊/ ※シフトJISコード及びEUCコードをサポートしています。

表B.2　拡張機能(2)

拡張機能	機能の内容
関数のデフォルト引数宣言	1.関数の引数にデフォルト値を定義できます。 記述例1)　extern　int　func(int=1, char=0); 記述例2)　extern　int　func(int=a, char=0); ※デフォルト値として変数を記述する時は、関数を宣言するよりも前にデフォルト値として使用する変数の宣言を行ってください。 ※デフォルト値は引数の後ろから順に埋めてください。
inline記憶クラスのサポート	1.inline記憶クラス指定子により関数をインライン展開することができます。 記述例) inline　func(int i); ※インライン関数を使用する前に必ずインライン関数の実体定義を行ってください。
C++風コメント	1.C++言語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、7700ファミリシリーズハードウェア仕様を効率良く活かすための拡張機能を使用できます。
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　dadd_b(char　val1,　char　val2); 記述例)　char　dadd_w(char　val1,　char　val2);

B.1　near / far修飾子

7700ファミリは、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使用されるアドレッシングモードが変わります。NC77は、near・far修飾子によりアドレッシングモードの切り換えを制御できます。
この章では、near・far修飾子の仕様について説明します。

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)

NC77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、NC77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、NC77はコンパイルドライバnc77の起動オプションにより、このデフォルトの属性を変更することができます。

B.1.2　変数の宣言書式

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

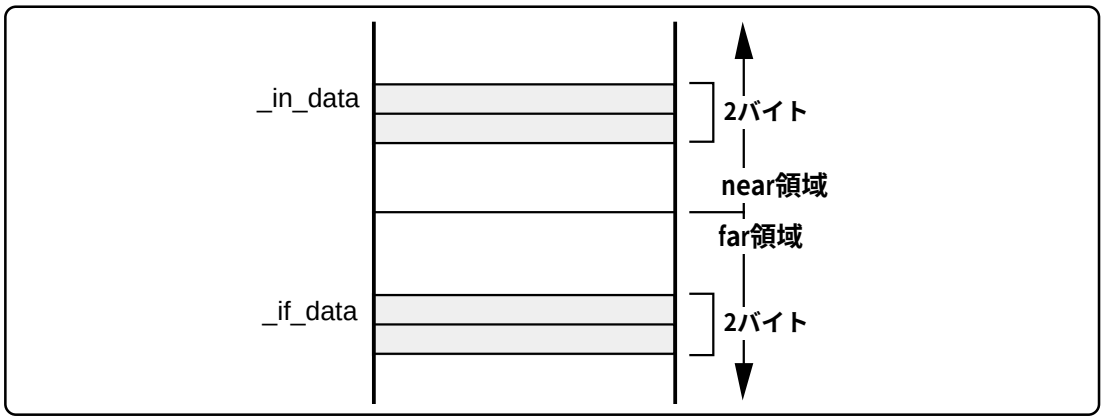
```
型指定子△near又はfar△変数;
```

図B.1　near / far修飾子を付加した変数の宣言書式

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

```
int near in_data;
int far  if_data;

func()
{
      (以下省略)
     :
```

図B.2　変数の宣言例



図B.3　変数のメモリ配置

B.1.3　ポインタ型変数の宣言書式

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

```
●例
  int   *   ptr;
```

図B.4　ポインタ型変数の宣言例(1)

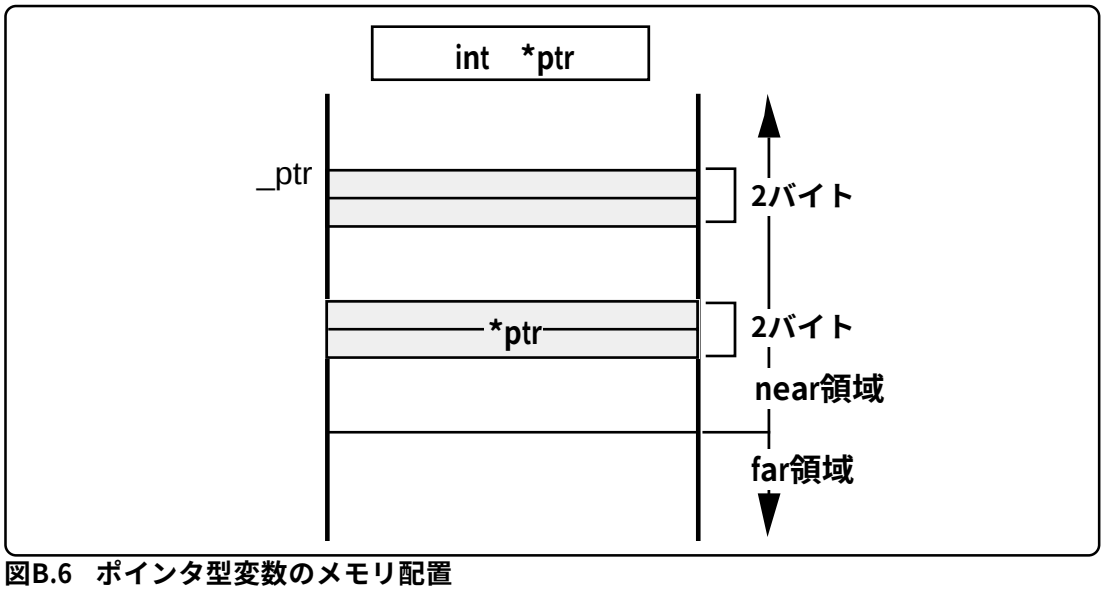
デフォルトでは、変数の配置は near 、型は near 型となるため、【図B.4】の記述は、【図B.5】のように解釈されます。

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

図B.5　ポインタ型変数の宣言例(2)

変数ptr は、near領域にあるint型変数を指し示す 2byte の変数です。ptr 自身はnear領域に配置されます。

上記例のメモリ配置を【図B.6】に示します。



図B.6　ポインタ型変数のメモリ配置

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

```
●例1
  int  far   *  ptr1;

●例2
  int  *   far    ptr2;
```

図B.7　アドレスを扱うポインタ型変数の宣言例(1)

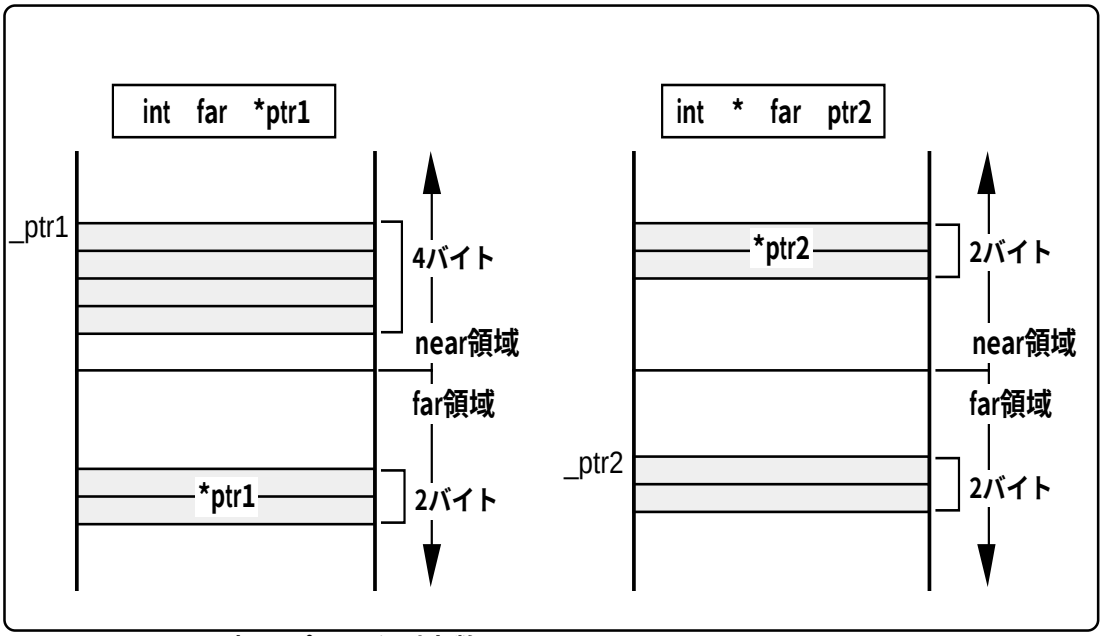
先にも説明したように near / far の指定がない場合には、変数の配置を“ near ”、変数の型を“ near ”として扱います。したがって、例１、例２はそれぞれ、【図B.8】のように解釈されます。

```
●例1
  int  far   *  near   ptr1;

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

図B.8　アドレスを扱うポインタ型変数の宣言例(2)

例１では、変数ptr1 は far領域にある int型変数を指し示す 4byte型の変数で、変数自身はnear領域 に配置されます。
例2では、変数ptr2 は near領域にある int型変数を指し示す 2byte型の変数で、変数自身はfar領域 に配置されます。

例１、例２のメモリ配置を【図B.9】に示します。



図B.9　アドレスを扱うポインタ型変数のメモリ配置

B.1.4　関数の宣言

【図B.10】に宣言時の書式を、【図B.11】に宣言例を示します。

```
型指定子△near又はfar△関数;
```

図B.10　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;
}
```

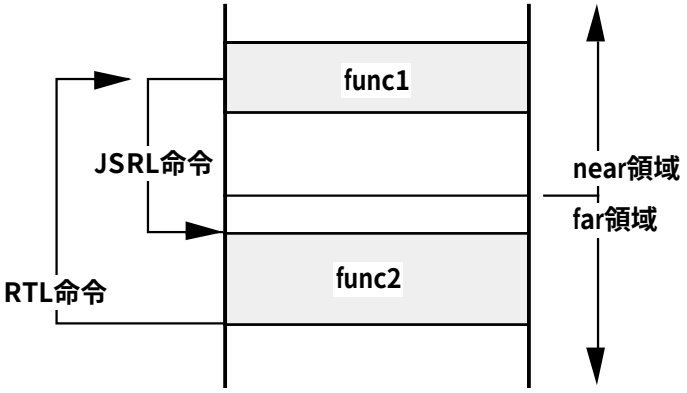
図B.11　変数・関数の宣言例

【図B.11】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.11】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.12】に示します。



図B.13　関数メモリ配置と呼び出し・リターン関係図

B.1.5　nc77の起動オプションによるnear / farの制御

NC77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。NC77の起動オプションには、変数(データ)のデフォルトを変更するオプションを用意しています。(【表B.3】)

表B.3　nc77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.14】に示すプログラムの記述において、nearからfarへの型変換が行われます。

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

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

図B.14　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、nc77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ポインタへの代入の検査機能

コンパイル時に、【図B.15】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.15　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.16】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)
        :
```

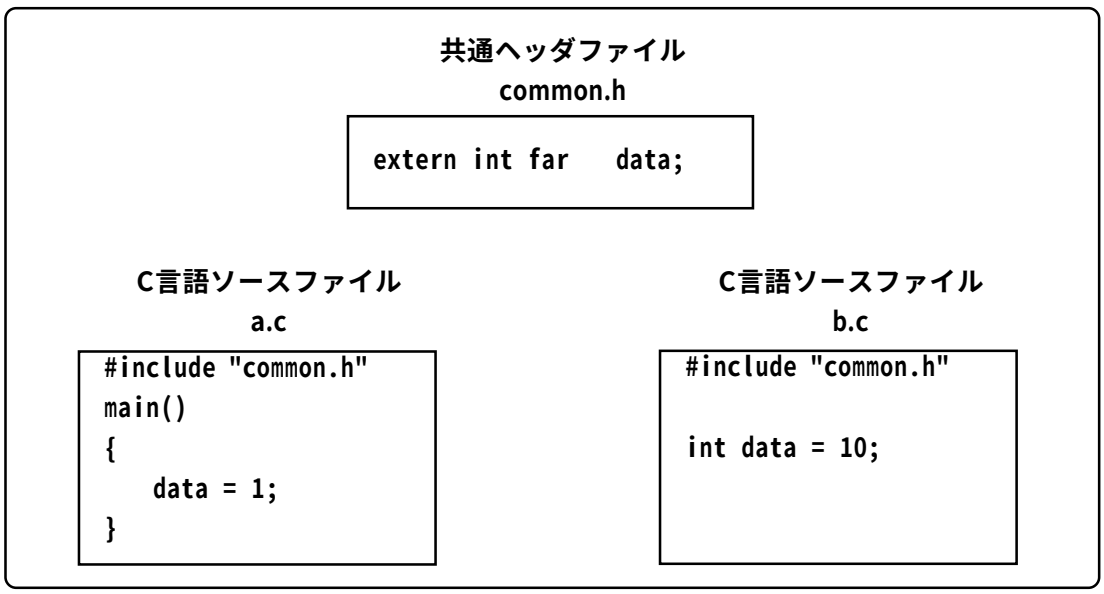
この宣言は、以下の宣言として解釈されます。

```
extern int far idata = 10;

func()
{
     (以下省略)
        :
```

図B.16　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。



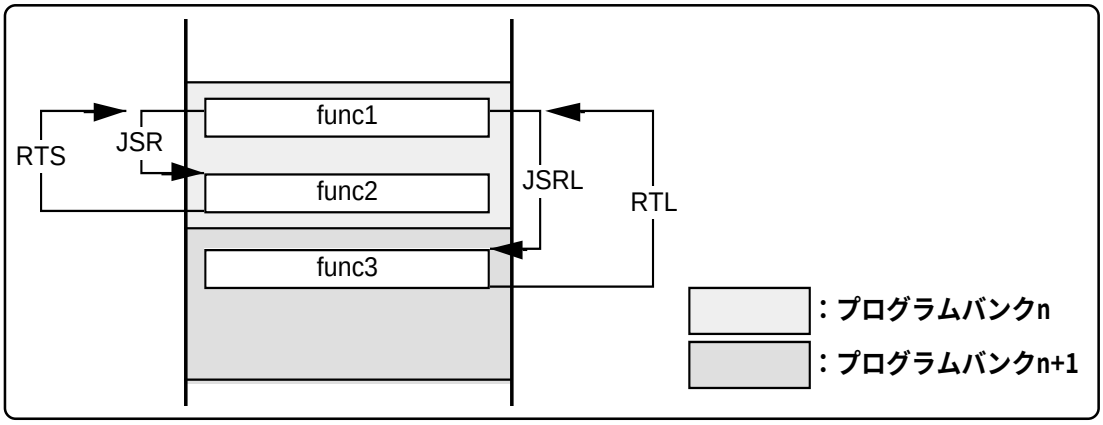
図B.17　共通ヘッダファイルの宣言例

※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。

B.1.9　関数に対するnear/far宣言

a　関数のnear・far属性

【図3.18】に示しますように、同一プログラムバンク内に配置されるnear関数を呼び出すときはJSR命令を使用し、関数から戻るときはRTS命令を使用します。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ときはJSRL命令を使用し、関数から戻るときはRTL命令を使用します。



図B.18　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

このように、呼び出す関数がプログラムバンク内／外によって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。この不整合はコンパイル時及びリンク時にはエラーとして検出できませんので不整合が発生しないようにご注意ください。

呼び出し命令及びリターン命令は、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ない場合、関数はfar属性として扱われます)。【図B.18】を基に、以下に例を示します。ここでは、プログラムバンクnを仮にnear領域とします。

●func1からfunc2を呼び出す場合
　func1に対してfunc2はnear属性となります。 func2のプロトタイプ宣言ではnearを指定します。
●func1からfunc3を呼び出す場合
　func1に対してfunc3はfar属性となります。func3のプロトタイプ宣言では、farを指定します。
●func3からfunc1を呼び出す場合
　func1に対してfunc3はfar属性となります。このような呼び出しがある場合func1のプロトタイプ宣言では、farを指定する必要があります。

func1、func2、func3のプロトタイプ宣言を共通ヘッダファイル等に記述する場合は、以下の書式で記述してください。この記述を共通ヘッダファイル等に記述することにより、呼び出し元の関数と呼び出される関数との整合性を一致させることができます。

```
extern int far      func1(void);
extern int near     func2(void);
extern int far      func3(void);
```

図B.19　プロトタイプ宣言例

b　関数のアドレス値の取り扱い

関数及び変数の宣言にnear・far属性を指定しない場合、関数はfar属性として、変数はnear属性として扱われます。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 ＊型又は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、変数のアドレスを32ビット長で扱うためにchar far ＊型又はvoid far ＊型として宣言してください。

```
func()
{
        int       (*func_ptr)();
        char far  *ptr;

        func_ptr = func;
        ptr = func_ptr;
}
```

図B.20　関数のアドレスと変数のアドレス値の相互関係

B.1.10　near / far属性に関する注意事項

a.　near / far修飾子の文法上の注意事項

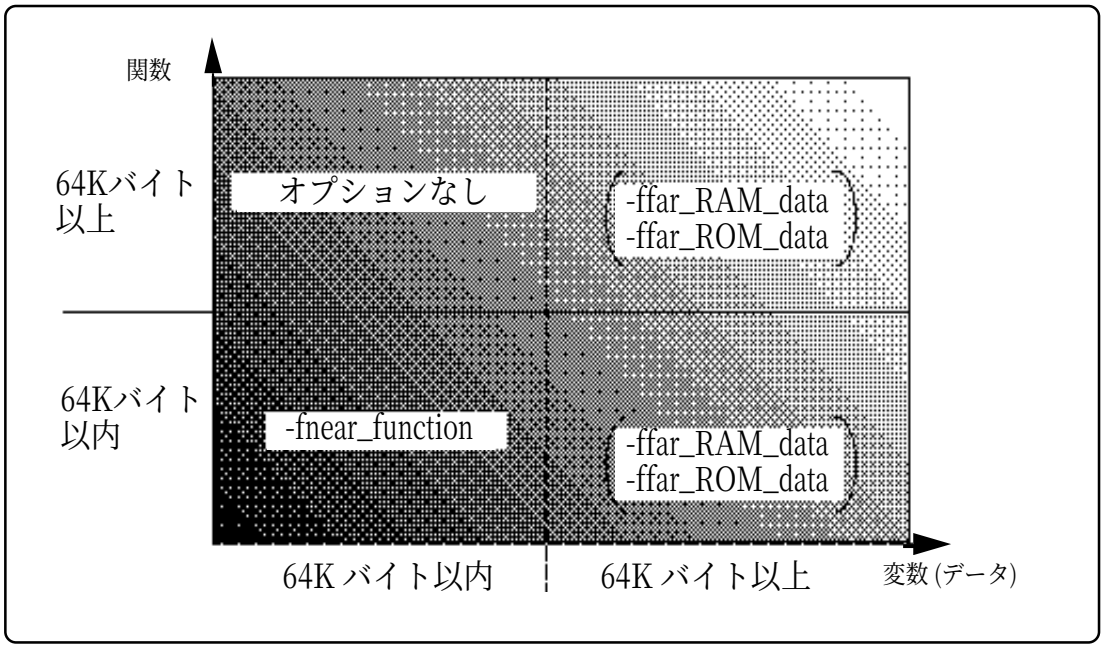
near / far修飾子は文法上、const修飾子と全く同じに扱われます。したがって、以下に示す記述はエラーとなります。

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

              ▼

        int   i;
        int   far   j;
```

図B.21　変数の宣言例

B.1.11　near/far属性のデフォルト指定のオプションに関する注意事項

NC77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。NC7の起動オプションには、関数及び変数(データ)のデフォルトを変更するオプションを用意しています。関数・変数(データ)のサイズとデフォルト指定オプションの関係を【図B.22】に示します。



図B.22　関数・変数のサイズと起動オプションの関係

【図B.22】に示した起動オプションの内容を以下に示します。

表B.4　nc77起動オプション

起動オプション	機　能
-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。
-ffar_ROM_data(-fFROM)	ROMデータのデフォルトをfarにします。
-ffar_RAM_data(-fFRAM)	RAMデータのデフォルトをfarにします。

B.1.12　near領域のバンク値に関する注意事項

NC77の起動オプション中、

●near領域のバンク値を設定する.....-bank=

を指定した場合、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こと、又は関数の引数として指定することは行わないでください。
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を、-bank=オプション付き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.a77中の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値と同じバンク値を設定してください。また、プロセッサモードレジスタの初期化は、【図B.23】に示す方法で設定してください。

```
start:
;---------------------------------------------------------------------
; after reset,this program will start
;---------------------------------------------------------------------
__DT    .equ            02h
        ldt     #__DT                   ; Initialize data bank register
        sem
        lda.W   A,#24H                  ; set processor mode register
        sta.W   A,LG:5eH
```

※この例では、nc77の起動オプション-bank=2でnear領域のバンク値を2に設定したときの設定です。

図B.23　データバンクレジスタとプロセッサモードレジスタの設定例(ncrt0.a77)

B.1.13　farのビットフィールド構造体に関する注意事項

以下に示す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メンバの内、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した場合、リンクエラー(Expression is out of DT range)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

NC77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、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.24】に示すようにasm("　　");のダブルクォーテーションの中にRASM77の言語仕様に準じたアセンブリ言語の命令を記述します。

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

void func()
{
          :
      (省略)
          :
        ta0_int = 0x07;         ←タイマA0割り込みを許可
        asm("       CLI");      ←割り込み禁止フラグのクリア
}
```

図B.24　asm関数の記述例(1)

また、【図B.25】に示す記述によりステートメントの前後関係を用いたコンパイラの最適化処理を部分的に抑止することができます。

```
        asm();
```

図B.25　asm関数の記述例(2)

NC77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、分岐情報をコンパイラに通知しませんので、分岐以降の生成コードを確認してください。

B.2.2　m、xフラグの切り換え機能

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.26】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.26　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.26】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.27】に示します。

●C言語ソースファイル

```
void near func()
{
        asm( 0, 0 );        ←m、xを共にクリア
            :
          (省略)
            :
        asm( 1, 1 );        ←m、xを共にセット
            :
          (省略)
            :
        asm( 2, 0 );        ←xフラグのみをクリア
            :
          (省略)
            :
        asm( 0, 2 );        ←mフラグのみをクリア
          (以下省略)
            :
```

●アセンブリ言語ソースファイル(コンパイル結果)

```
;### 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.27　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.28】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, DP:$$", 変数名);
```

図B.28　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.29　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.30】に示します。

●C言語ソースファイル

```
void near func()
{
        int idata;                              ←auto変数(DPオフセットは1)
        asm("       LDA.W   A,DP:$$", idata);
        asm("       CMP.W   A, #1000H ");

   (以下省略)
      :
}
```

●アセンブリ言語ソースファイル(コンパイル結果)

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

;#### ASM START
       LDA.W   A,DP:1
    .cline  6               ←DPオフセット1の内容をAレジスタに転送
;### C_SRC :        asm("       CMP.W   A, #1000H ");
       CMP.W   A, #1000H

;#### ASM END

   (以下省略)
      :
```

図B.30　auto変数の参照例

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。

【図B.31】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.31　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.32】に記述例を示します

```
/* 記述例 */

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

図B.32　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.33】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("seb    $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
        .source bit.c
        .cline  3
        .DT     __DT
        .DP     OFF
        .func   _func
        .pub    _func
_func:
        phd
        pht
        tsa
        tad
        .cline  10
;## ASM START
seb #02H,DP:1   ;  s
;## ASM END
        .cline  11
        sem
        pla
        clm
        pld
        rtl
```

図B.33　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(引数を含む)は、レジスタ上で管理されます。

【図B.34】に示す書式で記述することにより、asm関数中でレジスタ変数を使用することができます。※1

```
asm("　　オペコード　　$$, #00Hレジスタ変数名);
```

図B.34　レジスタ変数の記述書式

NC77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.35】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
        .source reg.c
        .cline  3
        .DT     __DT
        .DP     OFF
        .func   _func
        .pub    _func
_func:
        .cline  4
        lda.W   A,#0001H ;  i
        .cline  6
;## ASM START
lda.w   A,#0000H        ;  i
;## ASM END
```

図B.35　register変数の参照例

※1 register修飾子を有効にするためには、コンパイル時にオプション-fenable_register(-fER)を指定してください。

B.2.5　global,extern変数及びstatic変数のシンボル名の指定

C言語で記述した記憶クラスglobal,extern及びstaticの変数は、シンボルとして参照されます。

【図B.36】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, <DT: or LG:>$$", e変数名);
asm("      オペコード       B, <DT: or LG:>$$", 変数名);
```

図B.36　シンボル名指定の記述書式

この記述形式で指定できる変数名は1つです。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  i;
         int  k;
} s;
void func()
{
               :
        asm("       LDA.W   A, DT:$$", idata);
               :
        asm("       LDA.W   A, DT:$$", a[2]);
               :
        asm("       LDA.W   A, DT:$$", s.i);
           (以下省略)
              :
}
```

図B.37　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.38】示します。

●C言語ソースファイル

```
extern far      int   ext_val;          ←extern変数

func()
{
    static near int s_val;              ←static変数

    asm("lda    A,LG:$$", ext_val);
    asm("lda    B,DT:$$, s_val;
}
```

●アセンブリ言語ソースファイル(コンパイル結果)

```
        .pub    _func
_func:
        .cline  8
;## ASM START
        lda     A,LG:_ext_val
        .cline  9
        lda     B,DT:___S0_s_val
;## ASM END
        .cline  10
        rtl
        .endfunc        _func

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

図B.38　extern変数及びstatic変数の参照例

また、【図B.39】に示す書式で記述することにより、asm関数中でextern変数及びstatic変数の1ビットのビットフィールドを使用することができます。

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.39　シンボル名指定の記述書式

この記述形式で指定できる変数名は1つです。【図B.40】に記述例を示します。

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

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

図B.40　シンボルのビット位置指定の記述例

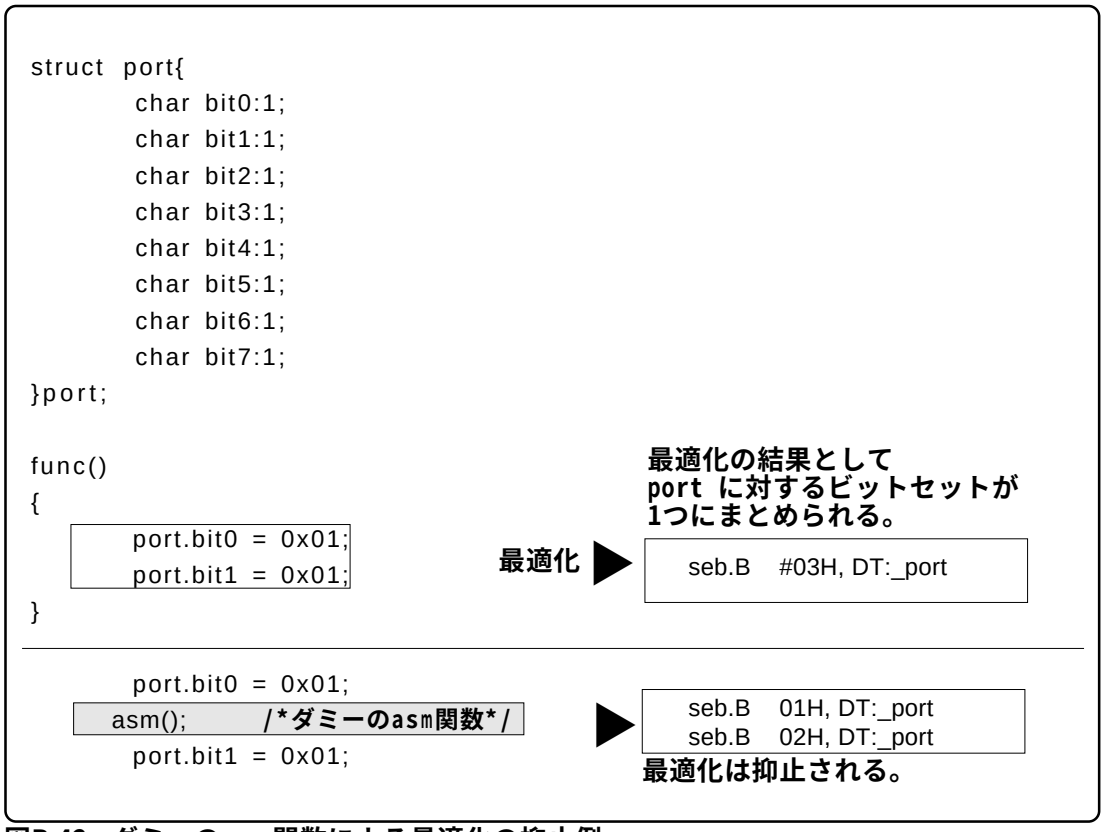
【図B.40】のCソースファイルのコンパイル結果を【図B.41】に示します。

```
;## #   FUNCTION func
;## #   ARG Size(0)     Auto Size(0)    Context Size(3)

        .section        program_F
        .source   kk.c
        .cline  10
        .DT     __DT
        .DP     OFF
        .func   _func
        .pub    _func
_func:
        .cline  11
;## ASM START
seb #02H,DT:_s              ←構造体sのビットフィールドbit0を参照
;## ASM END
        .cline  12
        rtl
        .endfunc        _func

        .SECTION        bss_NO
        .pub    _s
_s:
        .blkb   1
```

図B.41　シンボルに対するビットフィールドの参照例

B.2.6　最適化の部分的な抑止方法

【図B.42】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.42　ダミーのasm関数による最適化の抑止例

B.2.7　asm関数に関する注意事項

a.　asm関数の拡張機能

asm関数※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)記憶クラスがautoの変数、引数もしくは、1ビットのビットフィールドを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。autoの変数、もしくは引数を指定する場合には、【図B.33】に示す書式で記述してください。

```
asm("LDA.W      #01H,DP:$$",  i);←記憶クラスautoの変数を参照する書式です。
asm("SEB  $$",   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.43　asm関数の記述例(1)

(2)NC77では、register記憶クラスを指定することができます。 register記憶クラスで指定した変数でかつオプション-fenable_register(-fER)を指定してコンパイルした場合にasm関数でregister変数を記述する時は、【図B.44】に示す書式で記述してください。

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

図B.44　asm関数の記述例(2)

また、オプション-O,-OR,-OSを指定すると、コード効率の向上の為にregister渡しとなる引数をauto領域に転送を行わずにregister変数として扱う場合があります。

この場合に、asm関数で引数を指定すると変数のDPRオフセット値でなくレジスタ名でアセンブリ言語を出力するので、ご注意ください。

(3)asm関数では、clm、clp、sem、sepなど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下さい。m、xフラグの変更は、【図B.45】に示す書式で記述してください。

```
asm("clp      m,x");この書式は使わないで下さい。
  ▼
asm(0,  0);          ←asm(mフラグの内容,xフラグの内容)を指定する書式です。
```

図B.45　asm関数の記述例(3)

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

b.　DTレジスタについて

(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.46】に示すDTを元に戻す記述を行ってください。また、変更している間に呼び出す関数や、その間に発生する割り込み処理について十分考慮してください。

```
asm("    .DT    2”);                ←DT変更
asm("    LDT    #2H");
asm("    LDA    A, DT:_port");
                 :
               (省略)
                 :
asm("    LDT    #__DT");            ←DTを元に戻す
```

図B.46　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

NC77が生成するアセンブルソースファイルでは、【図B.47】に示す形式の内部ラベルが出力されます。したがって、asm関数を用いてこの規定と重複する可能性のあるラベルを記述しないでください。

```
●アルファベットの大文字１文字と数字で構成されるラベル名
例）A1:
     C9830:
●_(アンダースコア)で始まる２文字以上のラベル名
例）__LABEL:
    ___START:
```

図B.47　asm関数で記述できないラベル名の形式

d.　アセンブラコメントの記述に関する注意事項

NC77が生成するアセンブルソースファイルでは、【図B.48】に示す規定でコメントを付加します。したがって、asm関数を用いてこの規定と同じコメント記述をしないでください。

```
●;(セミコロン)と#が２文字で構成：コンパイラのための制御情報
例）;## Comment

●;(セミコロン)と#が３文字で構成：デバッグのための制御情報
例）;### Comment

●;(セミコロン)と#が４文字で構成： loop77のための制御情報
例）;#### Comment
```

図B.48　アセンブリ言語ソースファイル中のコメントの規定

B.3　日本語文字サポート

NC77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.49】に示します。

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

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

図B.49　環境変数NCKINとNCKOUTの設定例

NC77は入力漢字コードをプリプロセッサcpp77で処理します。cpp77でEUCコードに変換し、その後コンパイラccom77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.50】に示します。

```
L"漢字文字列"
```

図B.50　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。

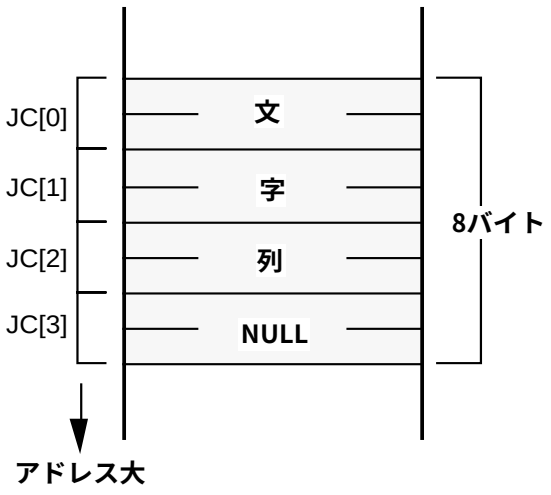
2バイト文字として扱う場合は、文字列の先頭にLを付加してwchar_t型へのポインタ型として使用します。wchar_t型は、標準ヘッダファイルstdlib.hの中でunsigned short型にtypedefしています。

【図B.51】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.51　日本語文字列の記述例

【図B.52】中の①の初期化された文字列のメモリ配置を【図B.40】に示します。



図B.52　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.53】に示します。

```
        L'漢'
```

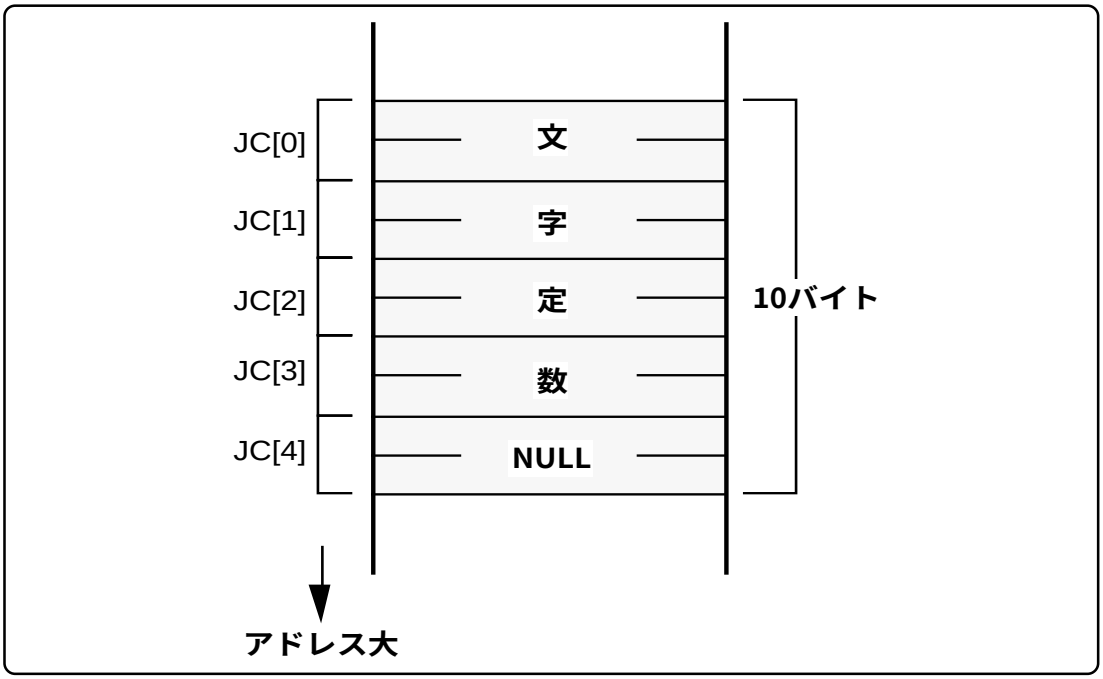
図B.53　文字列中の漢字コード記述書式

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.54】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.54　日本語文字定数の記述例

【図B.55】中の文字定数を代入した配列のメモリ配置を【図B.44】に示します。



図B.55　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

NC77では、C++の機能と同様に関数の引数にデフォルト値を定義できます。
この章では、関数のデフォルト引数宣言機能について説明します。

B.4.1　関数のデフォルト引数宣言の概要

NC77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.56】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.56　関数のデフォルト引数宣言書式

デフォルト引数宣言の例を【図B.57】に、【図B.57】で示すサンプルプログラムのコンパイル結果を【図B.58】に示します。

```
int func( int i=1 , int j=2 );                ←関数funcに仮引数のデフォルト値を
                                                第1引数：1、第2引数：2に宣言しています。
void main(void)
{
        func();                               ←実引数は第1引数：1、第2引数：2になります。
        func(3);                              ←実引数は第1引数：3、第2引数：2になります。
        func(3,5);                            ←実引数は第1引数：3、第2引数：5になります。
}
```

図B.57 関数のデフォルト引数の宣言例(smp1.c)

```
_main:
    .cline      5
;## # C_SRC :        func();
    pea         #0002H                  ←第2引数：2
    lda.W       A,#0001H                ←第1引数：1
    jsrl        ?func
    plx
    .cline      6
;## # C_SRC :        func(3);
    pea         #0002H                  ←第2引数：2
    lda.W       A,#0003H                ←第1引数：3
    jsrl        ?func
    plx
    .cline      7
;## # C_SRC :        func(3,5);
    pea         #0005H                  ←第2引数：5
    lda.W       A,#0003H                ←第1引数：3
    jsrl        ?func
    plx
    .cline      8
;## # C_SRC :   }
    rtl
          :
        (省略)
          :
```

注：NC77での引数を積む順序は、関数で宣言した引数の後ろからです。また、この例では引数をレジスタ渡しで処理しています。

図B.58 smp1.cのコンパイル結果(smp1.a77)

関数の引数には、変数を記述することができます。

デフォルト引数の変数指定の例を【図B.59】に、【図B.59】で示しましたサンプルプログラムのコンパイル結果を【図B.60】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だしを行ないます。
}
          :
        (省略)
          :
```

図B.59　デフォルト引数の変数指定例(smp2.c)

```
_main:
    .cline      6
;## # C_SRC :        func();
    lda    A,DT:_sym                    ←変数(sym)を引数として関数呼びだしを行ないます。
    jsrl   ?func
    .cline      7
;## # C_SRC :   }
    rtl
```

図B.60 smp2.cのコンパイル結果(smp2.a77)

関数のデフォルト引き数の宣言を行なう時には、以下の点に注意してください。

(1)複数の引数にデフォルト値を指定する時

関数の引き数が複数ある場合にデフォルト値を指定する時には、必ず引き数の後ろから埋めてください。【図B.61】に、不適切な記述の例を示します。

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

図B.61 プロトタイプ宣言の記述例

(2)変数をデフォルト値を指定する時

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

B.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

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.62】に宣言時の書式を示します。

```
inline△型指定子△関数;
```

図B.62　inline記憶クラスの宣言書式

関数の宣言例を【図B.62】に、コンパイル結果【図B.63】を示します。

```
extern int     i;

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

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

図B.63　インライン関数のサンプルプログラム(smp.c)

```
;## #        FUNCTION func
;## #        ARG Size(0)  Auto Size(0)  Context Size(0)

     .source test.c
     .section        program_F
;## # C_SRC :    {
     .DT     __DT
     .DP     OFF
_func:       .MACRO                          ←インライン関数をマクロ定義しています。
;## # C_SRC :            return ++i;
     inc     DT:_i
     lda     A,DT:_i
     .ENDM

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

;## # C_SRC :    {
     .DT     __DT
     .DP     OFF
     .func   _main
     .pub    _main
_main:
     phd
     pha
     tsa
     tad
     .cline  11
;## # C_SRC :            s = func();
     _func                                   ←マクロ_funcを呼び出しています。
     sta     A,DP:1        ; s
     .cline  12
;## # C_SRC :    }
     plx
     pld
     rtl
     .endfunc       _main
     .END
```

図B.64 サンプルプログラムのコンパイル結果(smp.a77)

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

                        .DT    __DT
                        .DP    OFF
                        .func  _main
                        .pub   _main
000000                  _main:
000000 0B                      phd
000001 48               pha
000002 3B                      tsa
000003 5B                      tad
                        .cline 11
000004                  _func
000004 EE0000  E +      inc    DT:_i                ←関数func(_func)がインライン展開
000007 AD0000  E +      lda    A,DT:_i                (マクロ展開)されています。
               +        .ENDM
00000A 8501                    sta    A,DP:1       ; s
                        .cline 12
00000C FA                      plx
00000D 2B                      pld
```

図B.65 smp.a77のマクロ展開結果(smp.prn)

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RASM77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なってください。実体定義は必ず、同一ファイル内に記述してください。【図B.66】の記述はNC77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.66 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、NC77は警告を発します。(【図B.67】)

```
int func(int i);

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

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

【ワーニングメッセージ】

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

図B.67 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。そのため、インライン関数に対して&演算子を使用した場合には、エラーになります。(【図B.68】)

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

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

        f = &func;
}
```

【エラーメッセージ】

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

図B.68 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。そのため、複数のファイルにまたがったインライン関数ではアクセスする領域が異なります。インライン関数内で使用するstaticデータは関数外で宣言してください。NC77では、インライン関数内でstatic宣言をした場合には、警告を発します。 また、インライン関数内のstatic宣言は、推奨しません。(【図B.69】)

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

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

【ワーニングメッセージ】

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

図B.69 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

　NC77ではインライン関数に対するC言語レベルのデバッグ情報を出力しません。従いまして、インライン関数のデバッグはアセンブリ言語レベルで行なうことになります。

B.6　コメント "//"の概要

NC77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の以降の記述は、コメントとして扱われます。
【図B.70】に書式を示します。

```
// コメント
```

図B.70　コメントの記述書式

コメントの記述例を【図B.71】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.71　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.　メモリ配置に関する拡張機能

表B.3　メモリ配置に関する拡張機能

拡張機能	機能の内容
#pragmaROM	指定した変数をromセクションに配置します。 記述形式)#pragma ROM 変数名 記述例)#pragma ROM val ※通常はconst修飾子を用いてromセクションに配置してください。
#pragmaSECTION	NC77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
#pragmaSTRUCT	1. 指定したタグを持つ構造体のパックを禁止します。 記述形式)#pragma STRUCT 構造体のタグ名 unpack 記述例)#pragma STRUCT TAG1 unpack 2. 指定したタグを持つ構造体のメンバの並べ替えを行い、偶数サイズのメンバを先に配置します。 記述形式)#pragma STRUCT 構造体のタグ名 arrange 記述例)#pragma STRUCT TAG1 arrange

b. 組み込み機器に関する拡張機能の使用方法

表B.4　組み込み機器に使用するための拡張機能

拡張機能	機能の内容
#pragma ADDRESS (#pragma EQU　)	変数の絶対アドレスを指定します(near型変数に対して指定した場合は、16ビットアドレスになります)。 記述形式)#pragma ADDRESS 変数名 絶対アドレス 記述例)#pragma ADDRESS port0 2H ※C77との互換のため#pragma EQUも使用できます。
#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△[/E]△割り込み処理関数名 記述例)#pragma INTERRUPT int_func 記述例)#pragma INTERRUPT /E int_func ※C77との互換のため#pragma INTFも使用できます。
#pragma PARAMETER	アセンブラで記述された関数を呼び出す際に、その引数をレジスタを介して渡すことを宣言します。 記述形式) #pragma PARAMETER△関数名(レジスタ名) 記述例)#pragma PARAMETER asm_func(R0, R1) ※本宣言を行う前に、必ず関数のプロトタイプ宣言を行ってください。

c.　MR7700に関する拡張機能の使用方法

表B.5　MR7700サポートに関する拡張機能

拡張機能	機能の内容
#pragma ALMHANDLER	MR7700のアラームハンドラ名を宣言します。 記述形式)#pragma ALMHANDLER 関数名 記述例)#pragma ALMHANDLER alm_func
#pragma CYCHANDLER	MR7700の周期起動ハンドラ名を宣言します。 記述形式)#pragma CYCHANDLER 関数名 記述例)#pragma CYCHANDLER cyc_func
#pragma INTHANDLER #pragma HANDLER	MR7700の割り込みハンドラ名を宣言します。 記述形式1)#pragma INTHANDLER 関数名 記述形式2)#pragma HANDLER 関数名 記述例)#pragma INTHANDLER int_func
#pragma TASK	MR7700のタスクの開始関数名を宣言します。 記述形式)#pragma TASK タスクの開始関数名 記述例)#pragma TASK task1

補足: 上記の拡張機能は通常コンフィグレータが生成しますので、気にする必要はありません。

d.　DTレジスタの操作に関する拡張機能の使用方法

表B.6　DTレジスタの操作に関する拡張機能

拡張機能	機能の内容
#pragmaLOADDT	コンパイル時にオプション-bank=で指定したデータバンクレジスタの値を、関数の先頭でロードする関数名を指定します。 記述形式) #pragma LOADDT func

e.　関数呼出し規則に関する拡張機能の使用方法

表B.7　関数呼出し規則に関する拡張機能

拡張機能	機能の内容
#pragmaM1FUNCTIUON	mフラグを1の状態にして呼び出す関数名を指定します。必ず関数の呼出し側と定義側の双方で本宣言を行ってください。 記述形式) #pragma M1FUNCTION func

f.　その他の拡張機能の使用方法

表B.8　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, #00H ADC.w　A, #02H #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

B.7.2　メモリ配置に関する拡張機能

NC77は、以下に示すメモリの配置に関する拡張機能を持っています。

#pragma ROM

romセクションへの配置機能

[機　能] 指定データ(変数)をromセクションに配置します。

[書　式] #pragma ROM△変数名

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

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

[規　定] 1.変数名以外を指定した場合、無効となります。
2.#pragma ROM宣言の二重定義はエラーとなりません。
3.初期化式を記述しない場合は初期値として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
___S0_i:                 ;### C's name is i  ←②の条件を満たす変数i
        .word   0014H
        .pub    _i
_i:                                          ←①の条件を満たす変数i
        .byte   00H
        .byte   00H
```

図B.72　#pragma ROM宣言の使用例

[備　考]　通常はconst修飾子を用いてromセクションに配置してください。

#pragma SECTION

セクション名変更機能

[機　能] コンパイラが生成するセクション名を変更します。

[書　式] #pragma SECTION△既定セクション名△変更セクション名

[解　説] この宣言をprogramセクションに対して行った場合は、その#pragma宣言以降に記述された関数のセクション名を変更します。
この宣言をデータ(data、bss及びrom)セクションに対して行った場合は、そのファイル中で実体を定義した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 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_Nセクションに配置
        ._file  'smp.c'
        ._line  9
        .pub    _func
_func:
```

図B.73　#pragma SECTION宣言の使用例

[備　考] セクションの名称を変更するときには、セクション名の後ろにセクションの配置属性(_N、_NE、_INE等)が付加されるのでご注意ください。

#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] NC77では、構造体はパックされます。例えば、【図B.74】に示す構造体のメンバは、宣言された順にパディング(透き間)を入れずに配置されます。

[使用例]

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

メンバ名	データ型	データサイズ	配置位置(ｵﾌｾｯﾄ)
i	int	16ビット	0
c	char	8ビット	2
j	int	16ビット	3

図B.74　構造体メンバの配置例(1)

①パックの禁止

NC77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.74】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.75】に示します。

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

メンバ名	データ型	データサイズ	配置位置(ｵﾌｾｯﾄ)
i	int	16ビット	0
c	char	8ビット	2
j	int	16ビット	3
ﾊﾟﾃﾞｨﾝｸﾞ	(char)	8ビット	—

図B.75　構造体メンバの配置例(2)

【図B.75】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。

[解　説] ②メンバ配置の並び換え

NC77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.74】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.76】に示します。

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

メンバ名	データ型	データサイズ	配置位置(ｵﾌｾｯﾄ)
i	int	16ビット	0
j	int	16ビット	2
c	char	8ビット	4

図B.76　構造体メンバの配置例(3)

パックの禁止及びメンバ配置の並び替えのための#pragma STRUCTは、必ず構造体のメンバ定義を行う前に宣言してください。

[使用例]

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

図B.77　#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

NC77は、以下に示す組み込みに関する拡張機能を持っています。

#pragma ADDRESS(#pragma EQU)

入出力変数の絶対アドレス指定機能

[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] #pragmaADDRESS△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宣言とnear/far属性の管理は異なります。near領域がバンク1(コンパイルオプションｓで-bank=1を指定)のとき、【図B.78】に示すプログラムをコンパイルした場合、変数ioはnear属性のデータであるためにioの絶対アドレスはDT値が加算されて10024Hになります。

[使用例]

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

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

図B.78　#pragma ADDRESS宣言とnear/far属性(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 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。

[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;
}
```

図B.79　#pragma ADDRESS宣言とnear/far属性(2)

[使用例]

```
#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;
}
```

図B.80　#pragma ADDRESS宣言の使用例

#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、7700ファミリ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。
- ●#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。

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。

[規　定]1.引数を持つ割り込み処理関数を記述した場合、コンパイル時に警告を出力します。
2.戻り値を返す割り込み処理関数を記述した場合、コンパイル時に警告を出力します。関数の戻り値は、必ずvoid型であるように宣言してください。
3.#pragma INTERRUPT宣言以降に関数の実体を定義した関数のみ有効となります。
4.関数名以外を指定した場合、何の効果も及ぼしません。
5.#pragma INTERRUPT宣言の二重定義はエラーとなりません。
6.#pragma INTERRUPTにより宣言された関数に対して、near/far宣言を行わないで下さい。

[使用例]

```
#pragma INTERRUPT /B i_func


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

図B.81　#pragma INTERRUPT宣言の使用例

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

#pragma PARAMETER

レジスタ渡しのアセンブラ関数宣言機能

[機　能] 引数をレジスタに格納して渡すアセンブラ関数を宣言します。

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

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

●float型、long型、farポインタ　(16ビットレジスタの組み合わせ)
　AB, XY
●int型、nearポインタ(16ビットレジスタ)
　A、B、X、Y
●char型　(8ビットレジスタ)
　A、B、X、Y
※レジスタ名を記述する際、大文字、小文字の区別はしません。

[規　定] 1.#pragma PARAMETER宣言を行う前には、必ずアセンブラ関数のプロトタイプ宣言を行ってください。プロトタイプ宣言を行わない場合は警告を出力し、この宣言を無効にします。

2.プロトタイプ宣言では、以下の項目を守ってください。

a. プロトタイプ宣言の引数の数と#pragma PARAMETER宣言の引数の数が一致している必要があります。

b. アセンブラ関数の引数の型として、以下の型は宣言できません。
●構造体型、共用体型
●double型

c. アセンブラ関数のリターン値の型として以下の関数は宣言できません。
●構造体、共用体を返す関数
●double型

3.#pragmaPARAMETERで指定した関数の実体をC言語で記述した時はエラーとなります。

[使用例]

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

}
```

図B.82　#pragma PARAMETER宣言の使用例

B.7.4　MR7700に関する拡張機能の使用方法

NC77は、以下に示すリアルタイムOS MR7700サポートに関する拡張機能を持っています。

#pragma ALMHANDLER

アラームハンドラの記述機能

[機　能] MR7700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。
- ●入り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
- ●#pragmaINTHANDLER
- ●#pragmaHANDLER
- ●#pragmaCYCHANDLER
- ●#pragmaTASK

[使用例]

```
#include   <mr7700.h>
#include   "id.h"

#pragma ALMHANDLER      alm

void alm()              ←必ずvoid型で宣言します。
{
          :             ←IntEntryシステムコールを記述する必要はありません。
         (省略)
          :             ←ret_intシステムコールを記述する必要はありません。
}
```

図B.83　#pragma ALMHANDLER宣言の使用例

#pragma CYCHANDLER

周期起動ハンドラ関数の記述機能

[機　能] MR7700の周期起動ハンドラを宣言します。

[書　式] #pragma CYCHANDLER△周期起動ハンドラ名

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

- ●システムクロック割り込みからはJSR命令で呼び出します。復帰はRTS命令でリターンします。
- ●入り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、interrupte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   <mrXXX.h>
#include   "id.h"

#pragma CYCHANDLER      cyc

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

図B.84　#pragma CYCHANDLER宣言の使用例

#pragma INTHANDLER(#pragma HANDLER)

割り込みハンドラ関数の記述機能

[機　能] MR7700のOS依存割り込みハンドラを宣言します。

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

[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)へ待避します。
- ●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宣言により、従来のMR7700(V.2.12以前)のプログラムの書式と比較して以下の点が異なります。
- ●割込みハンドラの先頭でIntEntryマクロを呼び出す必要が無くなりました。
- ●割込みハンドラ関数に記憶クラスautoの変数を使用できるようになりました。
- ●割込みハンドラ関数で一時領域を使用するような式を記述できるようになりました。

#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. IntEntryマクロを使用しないで下さい。
3. C言語からret_int、ret_wupシステムコールを使用しないでください。ret_wupシステムコールの代わりにiwup_skシステムコールを使用してください。
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
   - ●#pragmaHANDLER
   - ●#pragma ALMHANDLER
   - ●#pragmaCYCHANDLER
   - ●#pragmaTASK

［使用例］

```
#include   <mr7700.h>
#include    "id.h"

#pragma INTHANDLER    hand

void hand()                          ←必ずvoid型で宣言します。
{
     /* IntEntry() */                ←IntEntryシステムコールを記述しないで下さい。
      :
          (省略)
      :
     /* ret_int(); */                ←ret_intシステムコールを記述しないで下さい。
}
```

図B.85　#pragma INTHANDLER宣言の使用例

#pragma TASK

タスク開始関数の記述機能

[機　能] MR7700のタスク開始関数を宣言します。

[書　式] #pragma TASK△タスクの開始関数名

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

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する必要はありません。
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   <mrXXX.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)
```

図B.86　#pragma TASK宣言の使用例

B.7.5　DTレジスタの操作に関する拡張機能の使用方法

NC77は、以下に示すダイレクトページレジスタ(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。

[規　定] 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。

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7700ファミリ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]
;###    FUNCTION func

    .source sor2.c
    .section    program_F
    .DT __DT
    .DP OFF
    .func   _func
    .pub    _func
_func:
    pht
    ldt #__DT             ←コンパイル時のDT値をロード
    .cline  7
      :
  (以下省略)
      :
```

図B.87　#pragma LOADDT宣言の使用例

B.7.6　関数の呼出し規則に関する拡張機能の使用方法

NC77は、以下に示す関数の呼出し規則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、同一の指定を行ってください。

[使用例]

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

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


char func(char c)
{
   return c;
}

[アセンブリ言語ソースプログラム]
_main:
    .cline    6
;## # C_SRC :      func(1);
    sem                    ←Mフラグを1にセット
    lda.BA,#01H
    jsrl  ?func
       :
  (以下省略)
       :
```

図B.88　#pragma M1FUNCTION宣言の使用例

B.7.7　その他の拡張機能の使用方法

NC77は、前述以外に以下の拡張機能を持っています。

#pragma ASM(ENDASM)

インラインアセンブラ指定機能

[機　能] アセンブリ言語で記述を行なう領域を指定します。

[書　式] #pragmaASM
アセンブリ言語記述
#pragmaENDASM

[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。
#pragmaASMと対になる#pragmaENDASMがない場合にはNC77は処理を中断します。

[使用例]

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

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

#pragma ASM
   sem
LOOP1:
   ldx.W      #0000H
        :
      (省略)
        :
   clm
#pragma ENDASM

}
```

この領域をアセンブリ言語ファイルにそのまま出力します。

図B.89　#pragma ASM(ENDASM)記述例

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

#pragma PAGE

.PAGE疑似命令出力機能

[機　能] アセンブラで出力するリストファイルでの改行位置を宣言します。

[書　式] #pragma PAGE

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

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

[使用例]

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

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

#pragma PAGE

   i++;

}
```

図B.90　#pragma PAGE 記述例

B.8　アセンブラマクロ関数

B.8.1　アセンブラマクロ関数の概要

NC77では、アセンブラ命令の一部をC言語の関数として記述することができます。
この機能を使用することにより、特定のアセンブラの命令を直接的にC言語のプログラム上に記述できるので、プログラムのチューンナ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、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
*/
char dest[20];
char src[20];

func()
{
    mvn(dest,src,20); /* アセンブラマクロ関数(mvn命令) */
}
```

図B.91　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);          /* 8bit での演算の場合 */
int   asl_w(int  val);          /*  16bit  での演算の場合  */
```

asr

[機　能] valを1ビット算術右シフトした値を返します。
※この機能は775xシリーズのみで使用できます。

[書　式] #include <asmmacro.h>

```
char  asr_b(char val);          /* 8bitでの演算の場合 */
int   asr_w(int  val);          /*  16bit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する場合、演算結果は不定値です。

[書　式] #include <asmmacro.h>

```
int     div_w(long val1, int val2);          /* 16bit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する場合、演算結果は不定値です。
※この機能は775xシリーズのみで使用できます。

[書　式] #include <asmmacro.h>

```
int    divs_w(long val1, short val2);          /* 16bit での演算の場合 */
```

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);          /* 8bit での演算の場合 */
int   lsr_w(int  val);          /*  16bit  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);          /* 8bitでの演算の場合 */
int   rol_w(int  val);          /*  16bit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);          /* 8bitでの演算の場合 */
int   ror_w(int  val);          /*  16bit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

NC77は、7700ファミリ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7700ファミリ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　NC77性能概要

以下にNC77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Solaris2.5
	HP 9000 700シリーズ	HP-UX V.10.20
スワップ領域の空き容量	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
パーソナルコンピュータの環境	IBM PC/AT互換機	Windows 95, Windows NT 4.0
搭載CPUの種類	Pentium	
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NC77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.　NC77の仕様

【表C.4】にNC77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　NC77の仕様

項　　目	仕 様
NC77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
nc77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
nc77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
nc77の起動オプション-rasm77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
nc77の起動オプション-link77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

NC77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

NC77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	0XF又は0xf
		0123456789abcdef	

整数定数の型は、その値の大小により以下の順で決定されます。

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

NC77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	?:
	–	カンマ演算子	,
	＊	アドレス演算子	&
	/	ポインタ演算子	＊
	%	ビット演算子	<<
代入演算子	=		>>
	＋=		&
	－=		¦
	＊=		^
	/=		~
	%=		&=
関係演算子	>		¦=
	<		^=
	>=		<<=
	<=		>>=
	==	sizeof演算子	sizeof
	!=		

f.　区切り子

NC77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

NC77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

NC77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.9】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	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

NC77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

NC77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

NC77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

NC77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

NC77は、以下の記憶クラス指定子をサポートしています。

- ●extern
- ●static

(2)型宣言子

NC77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

NC77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

NC77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

NC77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NC77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)
```

また、NC77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。

```
NC77
MELPS
MELPS7700
```

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

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

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

#define

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

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

#undef

[機　能]　マクロを未定義にします。

[書　式]　#undef△識別子

[解　説]　●マクロとして定義された識別子を無効にします。

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

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

#include

[機　能]　指定したファイルの取り込みを行います。

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

[解　説]　①nc77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INC77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INC77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換えます。

[書　式]　#line△整数△"ファイル名"

[解　説]　●ファイルの行番号及びファイル名を設定します。
●ソースファイル名及び行番号を変更することができます。
●行番号は最大9999です。9999を越えた場合、デバッグ情報としてソース行情報を出力することはできません。

#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(cpp77.82):x.c, line xx]assertion warning
```

#pragma

[機　能]　NC77の拡張機能の処理を指示します。

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

[解　説]　①romセクションへの配置機能
②セクションベース名変更機能
③構造体配列制御機能
④入出力変数の絶対アドレス指定機能
⑤割り込み関数の記述機能
⑥レジスタ渡しのアセンブラ関数宣言機能
⑦アラームハンドラ関数の記述機能
⑧周期起動ハンドラ関数の記述機能
⑨割り込み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レジスタの操作機能
⑫関数呼出し規則変更機能
⑬インラインアセンブラ記述機能
⑭改ページ指定機能

●#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。
nc77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
　　：
[解　説]　#endif

●#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

NC77では、予め以下のマクロが定義されています。

●NC77
●MELPS
●MELPS7700

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、NC77以外のC言語プログラム中でマシン依存部をプリプロセスコマンドを使用して切り換えるとき等に使用します。

```
#ifdef  NC77
#pragma ADDRESS       port0     2H
#pragma ADDRESS       port1     3H

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

#endif
```

図C.14　プリデファインドマクロの使用例

付録D
C言語実装仕様

NC77が扱うデータの内部構造 / 配置、演算時等における符号拡張規則と、関数の呼び出し、および関数からの戻り値に関する規則を説明します。

D.1　データの内部表現

D.1.1　整数型

【表D.1】に整数型のデータが使用するバイト数を示します。

表D.1　整数型のデータサイズ

型　名	符号の有無	ビットサイズ	表現できる数値
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。

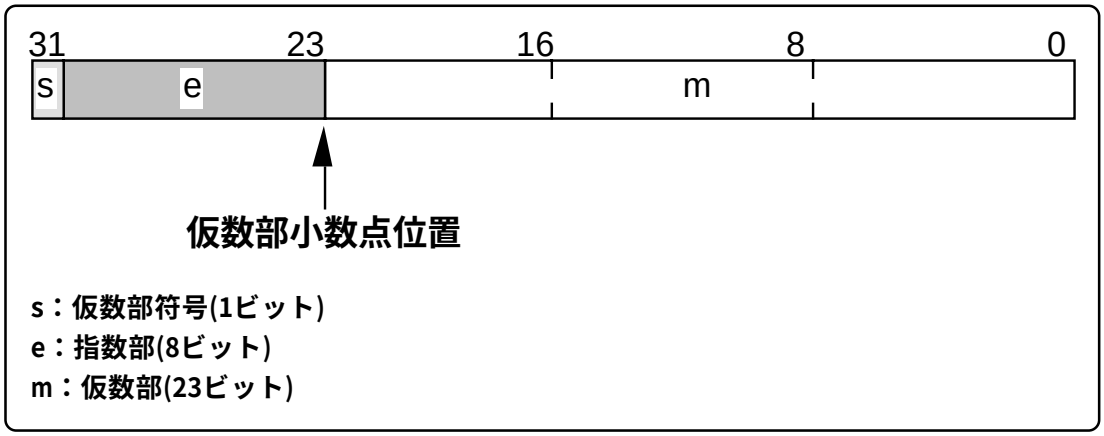
表D.2　浮動小数点型のデータサイズ

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

NC77機種依存データ有り()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

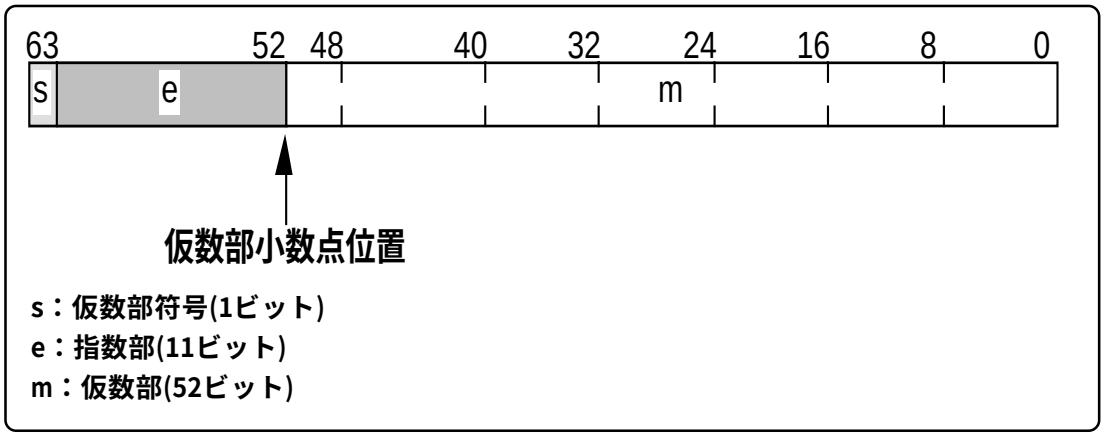
【図D.1】に示すデータ形式で2進数の浮動小数点(float)データを表現します。



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、nc77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。

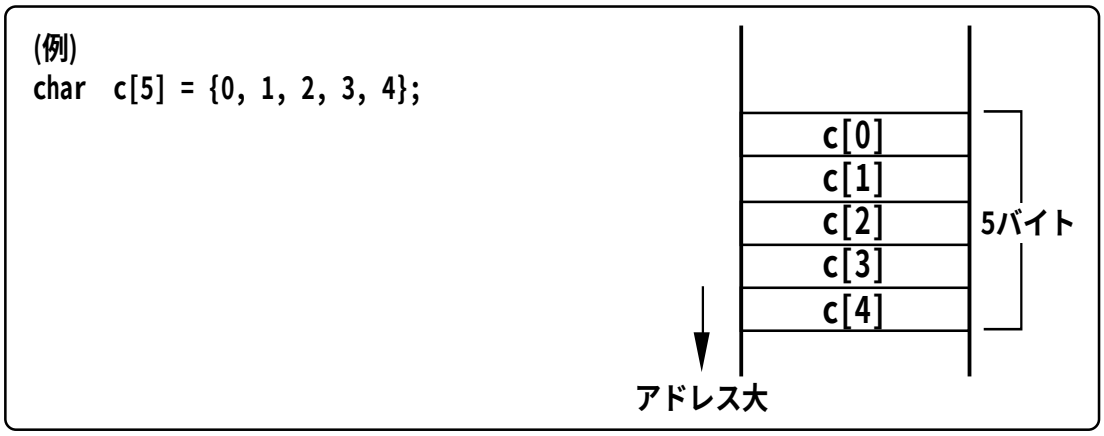
表D.3　ポインタ型のデータサイズ

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

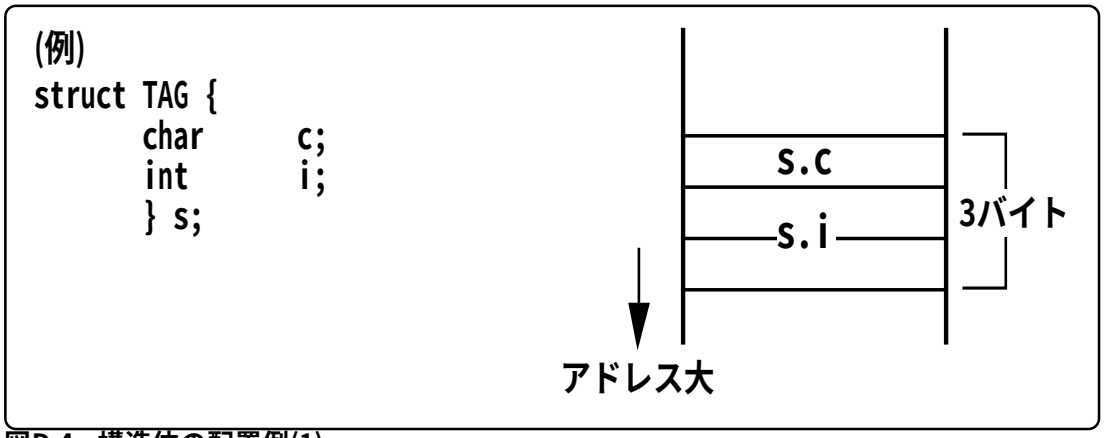
配列型は、要素のサイズ（バイト数）と要素数との積で表す領域に連続して配置されます。要素の出現順にメモリに配置されます。【図D.3】に配置例を示します。



図D.3　配列の配置例

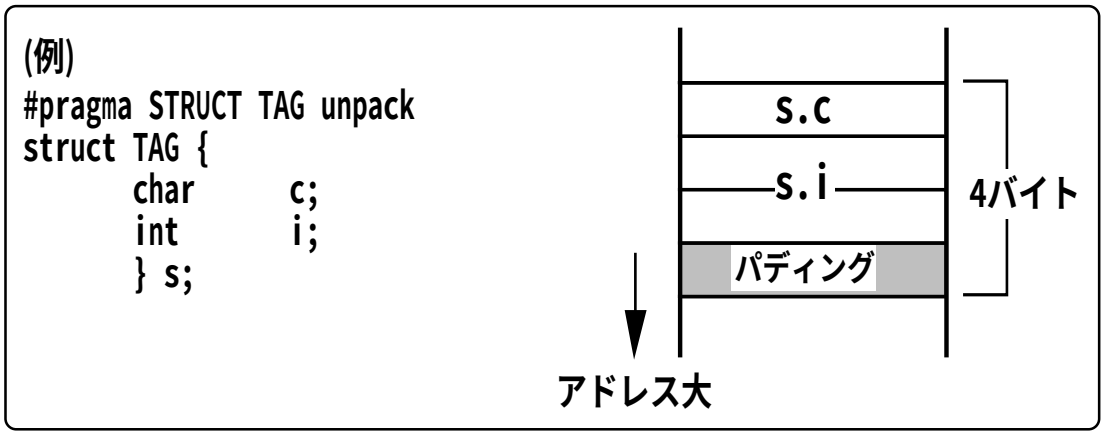
D.1.6　構造体型

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)

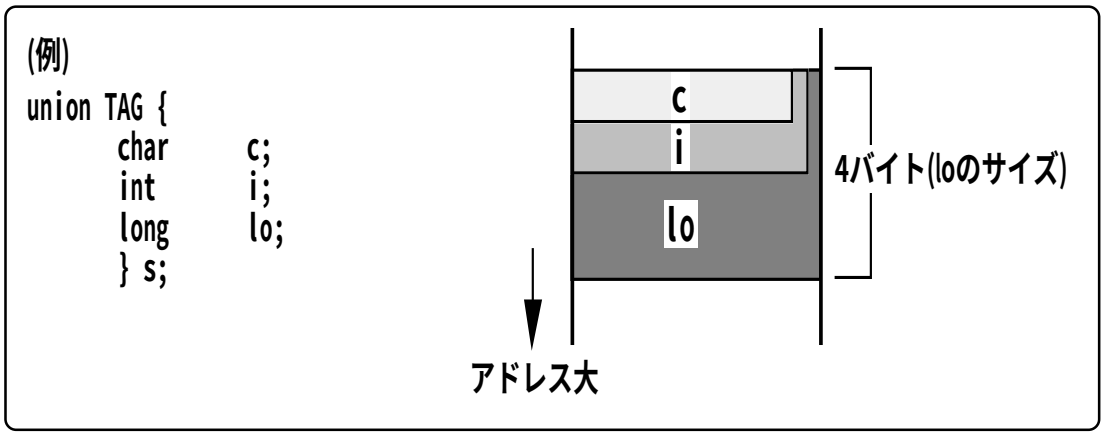
構造体は通常の場合、ワードアライメントを行いません。複数の構造体のメンバは連続して配置されます。ワードアライメントを行う場合は、拡張機能の#pragma STRUCTを使用します。#pragma STRUCTで宣言することにより、メンバのサイズの合計が奇数バイトであるときに1バイトのパディングを付加します。【図D.5】に配置例を示します。



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。



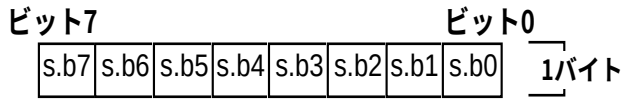
図D.6　共用体の配置例

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;
```

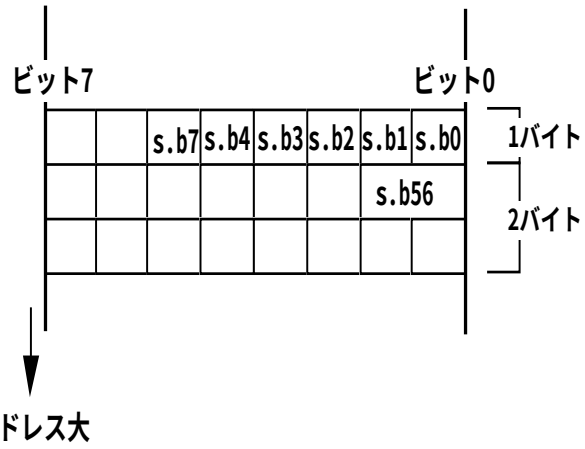


図D.7　ビットフィールドの配置例(1)

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

NC77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、整数型、ポインタ型、浮動小数点型については、レジスタ渡しで戻り値を返します。【表D.4】に戻り値に関する呼び出し規則を示します。

表D.4　戻り値に関する呼び出し規則

戻り値の型	規　則
char型	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	下位16ビットはAレジスタに、上位16ビットはBレジスタに格納して返します。
double型 long double型 構造体型 共用体型	呼び出しを行う直前に、戻り値を格納するための領域を指すfarアドレスをスタックに積みます。呼び出された関数はリターンする前にスタックに積まれたfarアドレスで指す領域に戻り値を書き込みます。

※1.NC77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

NC77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※1を行ない、関数呼びだし時に引数の型が確定している。
●プロトタイプ宣言に可変引数"..."を使用していない。
●関数の引数の型として、【表 D.5】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 nearポインタ型	Aレジスタ

(2)引数のスタック渡し

レジスタ渡しの条件を満たさない引数はすべて、スタック渡しになります。
引き数の渡し方をまとめると、【表 D.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 nearポインタ型	Aレジスタ	スタック
その他の型	スタック	スタック

※1.NC77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。
NC77では、上記の理由によって、プロトタイプ宣言を行なう記述形式に統一してC言語ソースファイルを記述することを推奨しています。

D.3.3　関数のアセンブリ言語シンボルへの変換規則

C言語ソースファイルでの関数定義時の関数名は、アセンブラソースファイルでの関数の先頭ラベルとして使用します。

アセンブラソースファイルでの関数の先頭ラベルは、C言語ソースファイルでの関数名の先頭に_(アンダースコア)あるいは?(クエスチョン)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
?(クエスチョン)	一つでも引数がレジスタ渡しとなる関数
_(アンダースコア)	上記条件以外の関数

【図 D.10】に示すプログラムは、関数の引数がレジスタ引数を持つものと、関数の引数をスタック渡しのみで扱う例です。

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

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

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

void                                                           ④
main(void)
{
        int sum;
        sum = func_proto(1,2,3);        ←⑤
        sum = func_no_proto(1,2,3);     ←⑥
}
```

①関数func_protoのプロトタイプ宣言です。
②関数func_protoの実体です。(プロトタイプ宣言を行なっています。(新しい形式))
③関数func_no_protoの実体です。(K&R形式(旧形式)の記述です。)
④関数mainの実体です。
⑤関数func_protoを呼んでいます。
⑥関数func_no_protoを呼んでいます。

図D.10 関数呼び出しのサンプルプログラム(sample.c)

上記サンプルプログラムのコンパイル結果について、関数func_protoの定義(②の部分)を【図D.11】に、関数func_no_protoの定義(③の部分)を【図D.12】に、関数func_protoと関数func_no_protoの呼び出しを(④の部分)を【図D.13】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)

        .source test.c
        .section    program_F
;## # C_SRC :    {
        .DT    __DT
        .DP    OFF
        .func ?func_proto
        .pub  ?func_proto
?func_proto:     ←⑩
        phd
        pha    ; Register Argument
        tsa
        tad
        .cline      6
;## # C_SRC :            return i + j + k;
        lda    A,DP:8       ;   j
        clc
        adc    A,DP:1       ;   i
        clc
        adc    A,DP:10      ;   k
        plx
        pld
        rtl
        .endfunc    ?func_proto
```

⑦第3引数kをスタック渡しにしています。
⑧第2引数jをスタック渡しにしています。
⑨第1引数iをレジスタ渡しにしています。
⑩関数func_protoの先頭アドレスです。

図D.11 サンプルプログラム(sample.c)のコンパイル結果(1)

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

関数の第一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?(クエスチョン)を付加して"?func_proto"になります。

```
;## #  __  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)

;## # C_SRC :    {
      .DT    __DT
      .DP    OFF
      .func _func_no_proto
      .pub   _func_no_proto
_func_no_proto:                    ←⑫
      phd
      tsa
      tad
      .cline       14
;## # C_SRC :             return i + j + k;
      lda    A,DP:8       ;   j
      clc
      adc    A,DP:6       ;   i
      clc
      adc    A,DP:10      ;   k
      pld
      rtl
      .endfunc     _func_no_proto
```

⑪すべての引き数をスタック渡しにしている。
⑫関数func_no_protoの先頭アドレスです。

図D.12 サンプルプログラム(sample.c)のコンパイル結果(2)

図D.10　のサンプルプログラム(sample.c)のコンパイル結果(2)では、関数func_no_protoは、K&R形式の記述を行っているので全引数がスタック渡しになります。

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_proto"の前に_(アンダースコア)を付加して"_func_proto"になります。

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

;## # C_SRC :    {
      .DT    __DT
      .DP    OFF
      .func _main
      .pub   _main
_main:
      phd
      pha
      tsa
      tad
      .cline       21
;## # C_SRC :  __ __ __ __sum = func_proto(1,2,3); __ __ ┐  ⑪
|     pea    #0003H                                     |
|     pea    #0002H                                     |
|     lda.W  A,#0001H                                   |
|     jsrl   ?func_proto                                |
|     plx                                               |
|     plx                                               |
|     sta    A,DP:1       ;  sum                        |
└ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ ┘
      .cline       22
;## # C_SRC :            sum = func_no_proto(1,2,3);
┌ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ ┐  ⑫
|     pea    #0003H                                     |
|     pea    #0002H                                     |
|     pea    #0001H                                     |
|     jsrl   _func_no_proto                             |
|     tax                                               |
|     tda                                               |
|     tas                                               |
|     txa                                               |
|     sta    A,DP:1       ;  sum                        |
└ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ ┘
      .cline       23
;## # C_SRC :    }
      plx
      pld
      rtl
      .endfunc     _main
```

図D.13 サンプルプログラム(sample.c)のコンパイル結果(3)

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

D.3.4　関数間のインターフェース

【図D.14】に示すプログラムにおいて、スタックフレームの構築及び解放の処理を【図D.16】～【図D.18】に示します。なお、【図D.15】は、【図D.14】のプログラムをコンパイルした結果、出力されたアセンブリ言語プログラムです。

```
 int          func( int, int ,int);
void main(void)
{
        int             i = 0x1234;       ←funcへの引数
        int             j = 0x5678;       ←funcへの引数
        int             k = 0x9abc;       ←funcへの引数
         k = func( i, j ,k);
}

int func( int x,int   y,int   z )
{
     int  sum;
       sum = x + y + z ;

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

図D.14　C言語サンプルプログラム

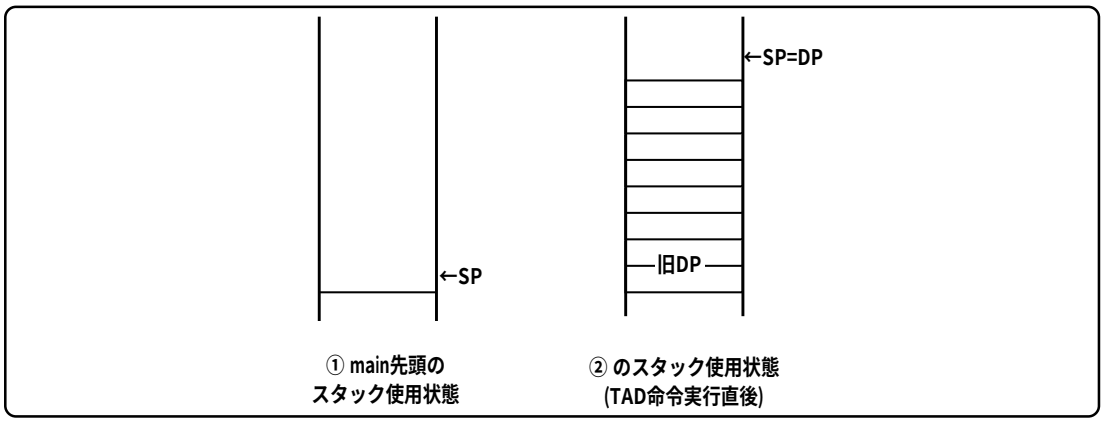
```
;## #       FUNCTIONmain
;## #       FRAMEAUTO (        k) size   2,   offset 1
;## #       FRAMEAUTO (        j) size   2,   offset 3
;## #       FRAMEAUTO (        i) size   2,   offset 5
;## #       ARG Size(0) Auto Size(6)      Context Size(5)

      .source test.c
      .section    program_F
;## # C_SRC :    {
      .DT   __DT
      .DP   OFF
      .func _main
      .pub  _main
_main:                                  ←①
      phd
      pha
      pha
      pha
      tsa
      tad                               ←②
      .cline      4
;## # C_SRC :           int            i = 0x1234;
      ldm.W #1234H,DP:5 ;   i
      .cline      5
;## # C_SRC :           int            j = 0x5678;
      ldm.W #5678H,DP:3 ;   j
      .cline      6
;## # C_SRC :           int            k = 0x9abc;
      ldm.W #9abcH,DP:1 ;   k
      .cline      7
;## # C_SRC :            k = func( i, j ,k);
      pei   #1    ;   k                 ←③
      pei   #3    ;   j                 ←④
      lda   A,DP:5      ;   i           ←⑤
      jsrl  ?func                       ←⑥
      plx
      plx                               ←⑩
      sta   A,DP:1      ;   k           ←⑪
      .cline      8
;## # C_SRC :    }
      plx
      plx
      plx
      pld
      rtl
      .endfunc    _main
```

図D.15 アセンブリ言語サンプルプログラム(1)

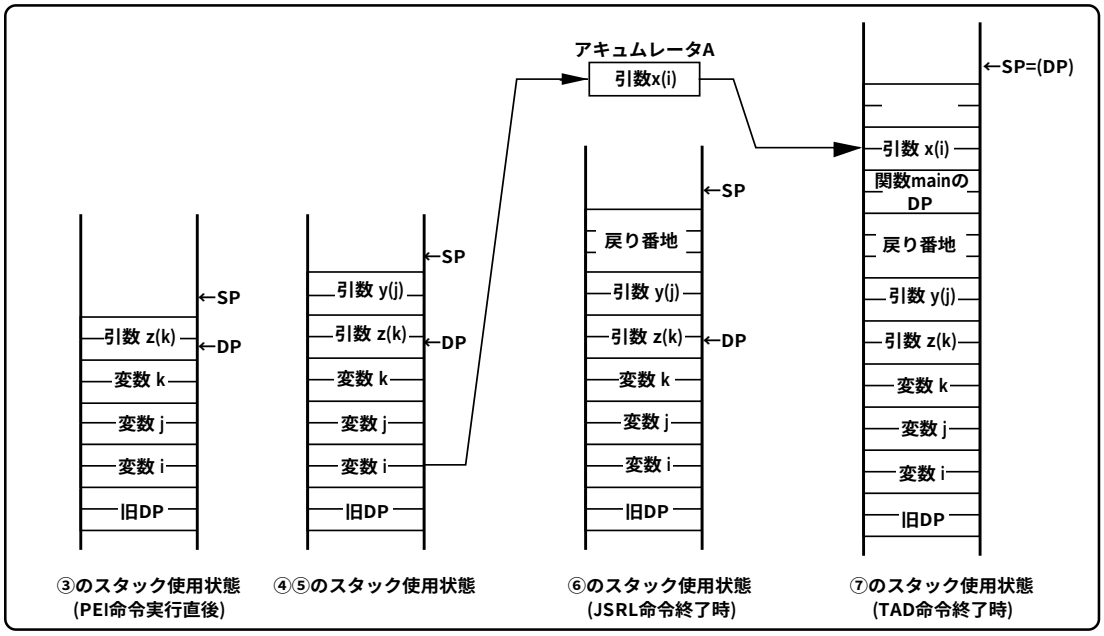
```
;## #       FUNCTION func
;## #       FRAME AUTO  (        x)  size    2,   offset 3
;## #       FRAME AUTO  (      sum)  size    2,   offset 1
;## #       FRAME ARG   (        y)  size    2,   offset 10
;## #       FRAME ARG   (        z)  size    2,   offset 12
;## #       REGISTER ARG      (        x)  size    2,   REGISTER A
;## #       ARG Size(4) Auto Size(4)      Context Size(5)

;## # C_SRC :    {
      .DT    __DT
      .DP    OFF
      .func ?func
      .pub  ?func
?func:
      phd
      pha    ; Register Argument
      pha
      tsa
      tad                                  ←⑦
      .cline       13
;## # C_SRC :           sum = x + y + z ;
      lda    A,DP:10      ;  y
      clc
      adc    A,DP:3       ;  x
      clc
      adc    A,DP:12      ;  z
      sta    A,DP:1       ; sum
      .cline       15
;## # C_SRC :           return  sum;
      lda    A,DP:1       ; sum          ←⑧
      plx
      plx
      pld
      rtl                                  ←⑨
      .endfunc     ?func
```

図D.16 アセンブリ言語サンプルプログラム(2)

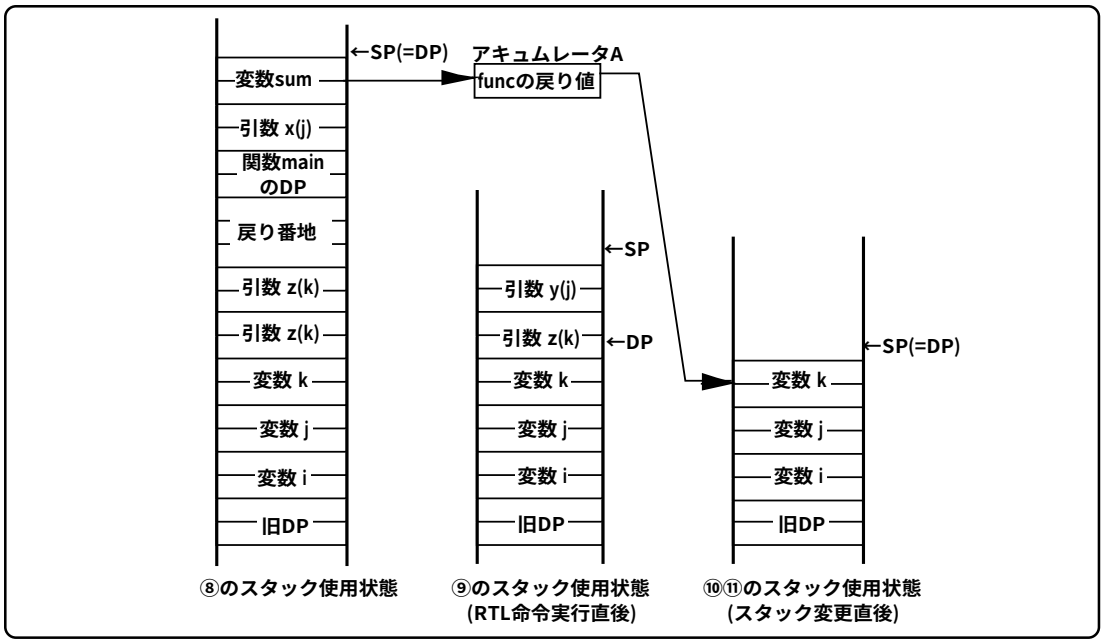
【図D.15】中の①→②の処理(関数mainの入り口処理)を【図D.17】に、③→④→⑤→⑥→⑦の処理(関数funcの呼び出し及び関数funcで使用するスタックフレームの構築処理)を【図D.18】に、⑧→⑨→⑩→⑪の処理(関数funcから関数mainへの戻り処理)を【図D.19】に、各々のスタック及びレジスタの遷移を示します。



図D.17　関数 mainの入り口処理



図D.18　関数funcの呼び出し及び、入り口処理



図D.19　関数funcの出口処理

D.4　auto変数の領域確保

記憶クラスautoの変数は、7700ファミリのスタック上に配置されます。【図D.20】に示すようなC言語ソースプログラムでは、記憶クラスautoの変数が有効となる領域が互いに重ならない場合、1つの領域のみ確保を行い複数の変数でその領域を共有します。

```
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.20　C言語ソースプログラム例

この例では、3つのauto変数i、j、kは有効となる範囲が重ならないため、同じ2バイトの領域(DPからのオフセット位置)を共有します。【図D.20】をコンパイルして生成されたアセンブリ言語ソースファイルを【図D.21】に示します。

```
;## #      FUNCTIONfunc
;## #      FRAME AUTO (        k)  size    2,   offset 1                    ←①
;## #      FRAME AUTO (        j)  size    2,   offset 1                    ←②
;## #      FRAME AUTO (        i)  size    2,   offset 1                    ←③
;## #      ARG Size(0) Auto  Size(2)      Context  Size(5)

     .source  test.c
     .section     program_F
;## # C_SRC :    {
     .DT    __DT
     .DP    OFF
     .func _func
     .pub   _func
_func:
     phd
     pha
     tsa
     tad
     .cline       7
;## # C_SRC :         for( i=0 ; i<=0 ; i++ ){
          :
     (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.21　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の仕様の詳細を説明します。

E.1.1　標準ヘッダファイルの概要

　NC77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NC77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NC77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。

NC77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。

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

math.h

[機　能]　数学関数のプロトタイプを宣言しています。プロトタイプを宣言している関数を以下に示します。

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

setjmp.h

[機　能]　分岐関数のプロトタイプ宣言と、その関数で使用する構造体を定義しています。プロトタイプを宣言している関数を以下に示します。

関数名	機　能
longjmp	大域ジャンプ
setjmp	大域ジャンプのためのスタック環境の設定

signal.h

[機　能]　非同期割り込みを処理するためのマクロと関数を定義 / 宣言しています。

stdarg.h

[機　能]　可変長の引数リストを処理するためのルーチンを定義しています。

stddef.h

[機　能]　各標準ヘッダファイルで共通に使用するマクロ名を定義しています。

stdio.h

[機　能]　FILE構造体 / ストリーム名の定義と、入出力関数のプロトタイプを宣言しています。プロトタイプを宣言している関数を以下に示します。

種別	関数名	機　　能
初期化	init	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

NC77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

NC77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	7700ファミリの入出力の初期化	○
	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NC77が提供する標準関数の詳細仕様を説明します。関数は、アルファベット順に掲載しています。

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

abort

実行制御関数

[機　能]　プログラムを異常終了します。

[書　式]　#include　<stdlib.h>

　　void abort(void);

[実現方法]　関数

[変　数]　引数はありません。

[戻り値]　●戻り値はありません。

[解　説]　●プログラムを異常終了します。

[注　意]　●実際には、abortプログラム内部で無限ループとなります。

abs

整数算術関数

[機　能]　整数の絶対値を計算します。

[書　式]　#include　<stdlib.h>

　　int abs(n);

[実現方法]　関数

[変　数]　int n;..... 整数

[戻り値]　●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

[変　数] double x; ..... 実数

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

asin

数学関数

[機　能] 逆サインを計算します。

[書　式] #include　<math.h>

double _far asin(x);

[実現方法] 関数

[変　数] double x; ..... 実数

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

atan

数学関数

［機　能］ 逆タンジェントを計算します。

［書　式］ #include　<math.h>

double _far atan(x);

［実現方法］ 関数

［変　数］ double x; ..... 実数

［戻り値］ ●-π/2からπ/2ラジアンの範囲の値を返します。

atan2

数学関数

［機　能］ "x"と"y"の商の逆タンジェントを計算します。

［書　式］ #include　<math.h>

double _far atan2(x , y);

［実現方法］ 関数

［変　数］ double x; ..... 実数
double y; ..... 実数

［戻り値］ ●-πからπラジアンの範囲の値を返します。

atof

文字列数値変換関数

[機　能] 文字列を浮動小数に変換します。

[書　式] #include　<stdlib.h>

double _far atof(s);

[実現方法] 関数

[変　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

[変　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

［変　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

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

［戻り値］ ●戻り値はありません。

［解　説］ ●dtopで示される領域に、srcで示される領域の先頭からsizeで指定されたバイト数分の内容を複写します。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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

［変　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（MS-DOS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。

［注　意］ ●昇順にソート済みの配列内から指定された項目を検索します。

bzero

メモリ操作関数

［機　能］ メモリ領域の初期化(ゼロクリア)を行います。

［書　式］ #include　<string.h>

void _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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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(MS-DOS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

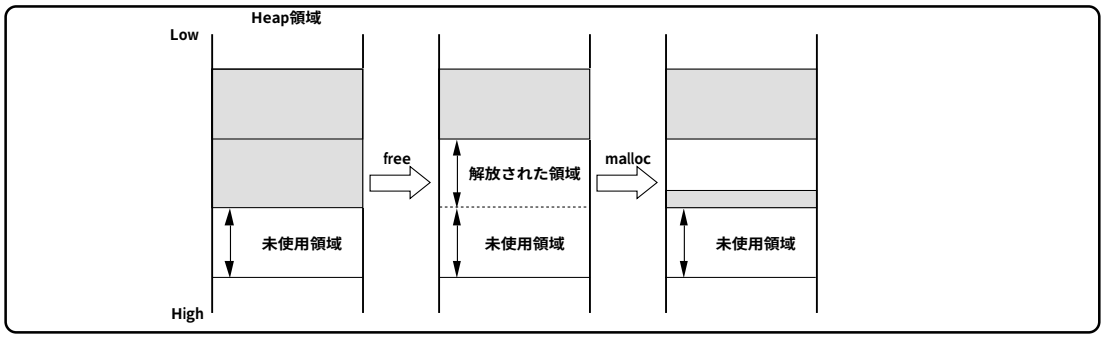
［戻り値］ ●確保した領域の先頭アドレスを返します。
●指定した大きさの領域が確保できなかった場合には戻り値としてNULLを返します。

［解　説］ ●動的にメモリ領域を割り当てます。

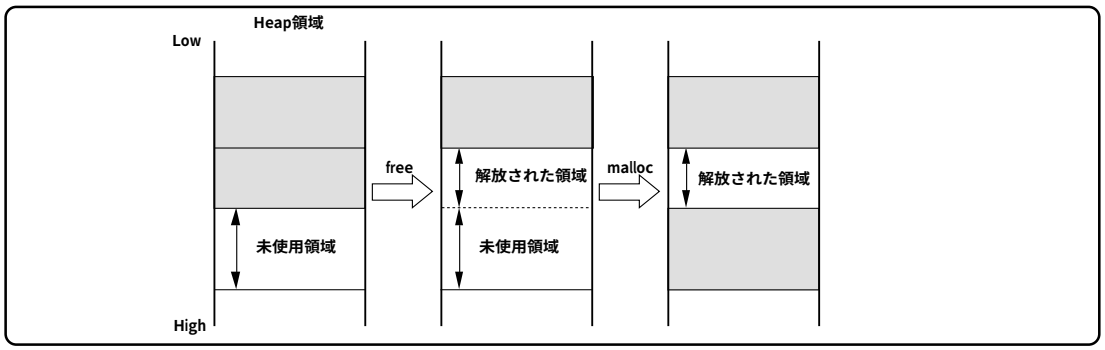
［規　則］ ●mallocを行う場合、以下の(1)～(2)の順にチェックを行い、適合する箇所でメモリを　確保します。

(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。

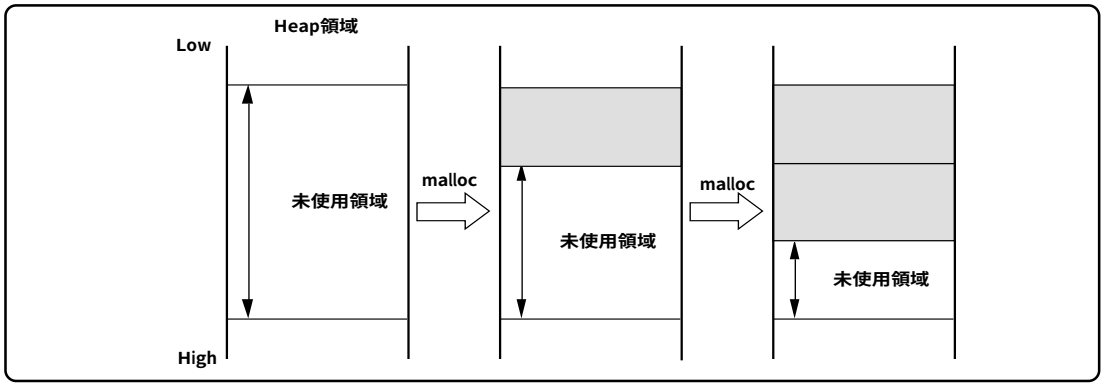


malloc

(2)freeで解放された領域がない場合

(2-1)確保可能な未使用領域が存在する場合

未使用領域の下位番地から上位番地方向に領域が確保されます。



(2-2)確保可能な未使用領域が存在しない場合

メモリは確保せずに、NULLを返します。

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

mblen

多バイト文字・多バイト文字列操作関数

[機　能]　マルチバイト文字列の長さを計算します。

[書　式]　#include　<stdlib.h>

int _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。
●メモリ領域中に文字"c"が見つからない場合にはNULLを返します

[解　説]　●"s"で示されるアドレスから始まる"n"で指定された大きさのメモリ領域から、"c"で示される文字を検索します。
●-D*NEAR_LIB*オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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

[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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。

perror

入出力関数

［機　能］　エラーメッセージをstderrに出力します。

［書　式］　#include　　<stdio.h>

　　void _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; ..... 被乗数

［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（MS-DOS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.ddddddde±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。NC77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(MS-DOS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(MS-DOS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(MS-DOS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（MS-DOS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の書式指定

1.変換指定記号

●d

　符号付きの10進数を変換します。対応する引数は整数へのポインタでなければいけません。

●i

　符号付きの10進数、8進数、16進数の入力を変換します。8進数は先頭が0、16進数は先頭が0x、0Xで始まります。対応する引数は整数へのポインタでなければいけません。

●u

　符号なし10進数を変換します。対応する引数は、符号なし整数へのポインタでなければいけません。

●o

　符号付き8進数を変換します。対応する引数は整数へのポインタでなければいけまん。

●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の書式指定

2.＊(データ格納の抑止指定)

＊が指定されている場合、変換したデータを引数に格納することを抑止します。

3.最大フィールド幅

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

4.l、L、又はh

- lの場合、d、i、o、u、xの変換結果をlong int又はunsigned long intとして格納します。また、e、E、f、g、Gの変換結果をdoubleとして格納します。
- hの場合、d、i、o、u、xの変換結果をshort int又はunsigned short intとして格納します。
- l、hをd、i、o、u、x変換以外で指定した場合、指定が無視されます。
- Lの場合、e、E、f、g、Gの変換結果をfloatとして格納します。

setjmp

実行制御関数

[機　能]　関数呼び出し時の環境の退避を行います。

[書　式]　#include　　<setjmp.h>

　　int _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>

　　char _far ＊setlocale(category,locale);

[実現方法]　関数

[引　数]　int category;..... ロケール情報、検索部情報
　　const char ＊locale;..... ロケール情報文字列へのポインタ

[戻り値]　●ロケール情報文字列へのポインタを返します。
　　●設定、検索できない場合はNULLを返します。

sin

数学関数

[機　能]　サインを計算します。

[書　式]　#include　<math.h>

double _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(MS-DOS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(MS-DOS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。
●-D*NEAR_LIB*※2オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※2オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※1.s1はs1とs2が充分に入る大きさが必要です。
※2.-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

strerror

文字列操作関数

[機　能]　エラー番号を文字列に変換します。

[書　式]　#include　<string.h>

char *_far strerror(errcode);

[実現方法]　関数

[引　数]　int errcode; ..... エラーコード

[戻り値]　●エラーコードに対するメッセージ文字列へのポインタを返します。

[注　意]　●stderrは静的な配列に対してポインタを返します。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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
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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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。
●-D*NEAR_LIB*※オプションを指定してコンパイルした場合は、この関数のポインタ引数をnear属性で扱う高速版ライブラリが選択されます。
●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。

※-Dオプションで指定するNEAR_LIBは、標準ヘッダファイルstring.hの中でライブラリを選択するために使用する識別子です。

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]　マクロ

[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(MS-DOS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

[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(MS-DOS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(MS-DOS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。

●引数をfarポインタで扱う場合は、メイクファイルmake.far(MS-DOS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】の条件を満たす時、関数のインライン埋め込みを行ないます。

表E.13　標準ライブラリ関数に対する最適化条件

関数名	最適化条件	記述例
strcpy	第1引数：nearポインタ 第2引数：文字列定数	strcpy(str, "sample");
memcpy	第1引数：nearポインタ 第2引数：文字列定数 第3引数：定数	memcpy(str ,"sample", 6);

(2)高速ライブラリの選択（NC30 のみ）

通常NC30は、標準ライブラリ関数において、ポインタを引数に持つ関数はポインタをfarポインタとして扱います。そのため、引数がnearポインタであるとき効率の良いコードが生成されません。そこで、引数がnearポインタの時には、near用のライブラリ関数を選択する最適化を行ないます。最適化の対象となる関数は以下の通りです。

表E.14　最適化の対象となるライブラリ関数

関数名	関数名	関数名	strspn
bcopy	strcat	strnicmp	strstr
bzero	strchr	strlen	strspn
memchr	strcmp	strncat	strtod
memcmp	strcoll	strncmp	strtok
memcpy	strcpy	strncpy	strtol
memicmp	strcspn	strnicmp	strtoul
memmove	strerror	strpbrk	strxfrm
memset	stricmp	strrchr	

c.　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、src77/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。

E.3　標準入出力関数ライブラリのカスタマイズ方法

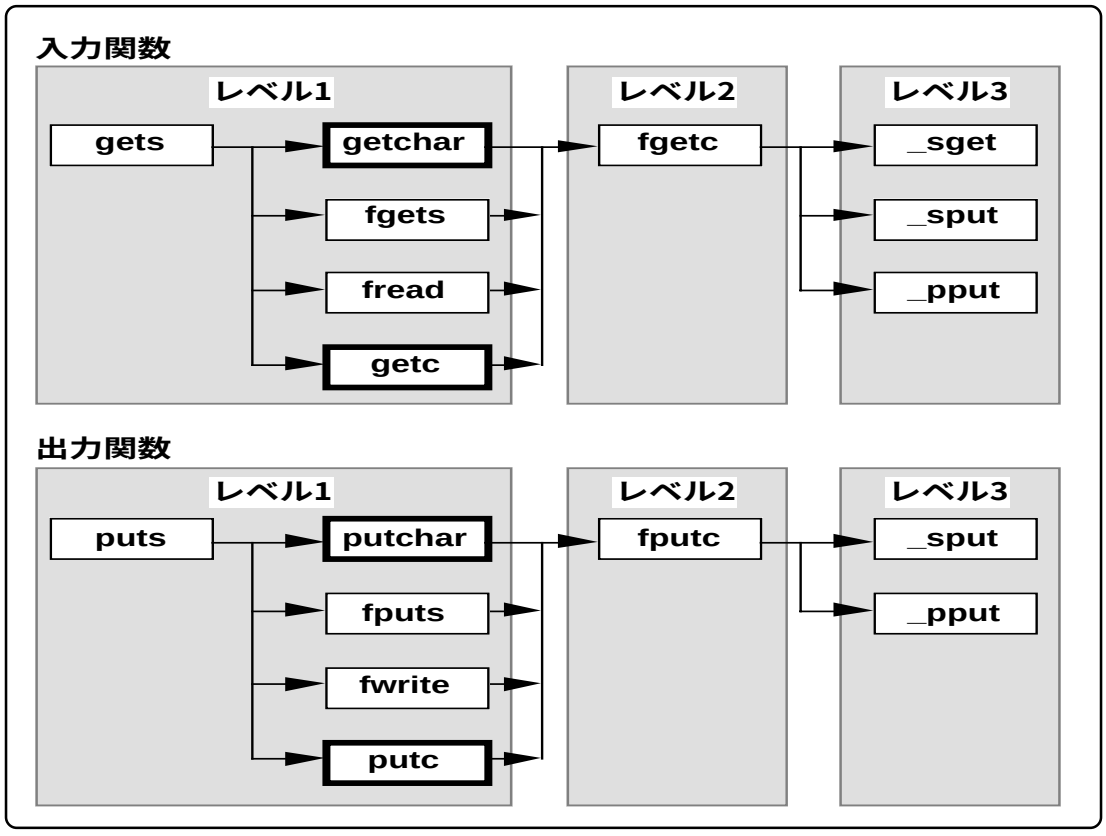
入出力関数としてscanf、printf等の高機能な関数ライブラリを用意しています。これらの関数は一般的に高水準入出力関数と呼ばれます。高水準入出力関数は、ハードウェア環境に依存した低水準入出力関数の組み合わせで実現しています。

7700ファミリ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.1】に示すようにレベル1の関数から下位の関数(レベル2→レベル3)を呼び出すことで機能を実現しています。例えば、fgetsはレベル2のfgetcを呼び出し、更にfgetcはレベル3の関数を呼び出します。

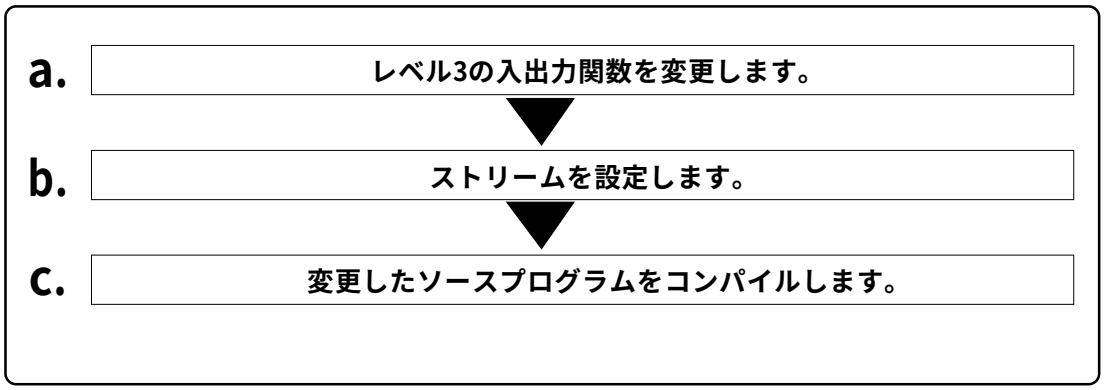
7700ファミリのハードウェア(入出力ポート)に依存しているのは最下位のレベル3の関数のみです。入出力関数をアプリケーションプログラムで使用する場合、必要に応じてレベル3の関数のソースファイルを書き換えることで、システムに順応した関数に変更できます。



図E.1　入出力関数の呼び出し関係図

E.3.2　入出力関数の変更手順

入出力関数を組み込むシステムに合わせて変更する方法の概略を【図E.2】に示します。



図E.2　入出力関数の変更手順例

a.　レベル3の入出力関数の変更方法

レベル3の入出力関数は、7700ファミリの入出力ポートに対して1バイトの入出力を行う関数です。レベル3の入出力関数は、シリアル通信回路(UART)に対して入出力を行う_sget、_sputと、セントロニクス仕様の通信回路に対して入出力を行う_pputがあります。

[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.15】に示します。

表E.15　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、デフォルトでは7700ファミリが持つ2本のUARTの内UART1を設定しています。device.cでは、条件コンパイルコマンドを用いることでUART1の代わりにUART0を選択することができます。UART0を選択するためには、

●UART0を使用する場合 ..... #define UART0 1

をdevice.cファイルの先頭で記述するか、あるいは

●UART0を使用する場合 ..... -DUART0

をコンパイル時に指定してください。

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

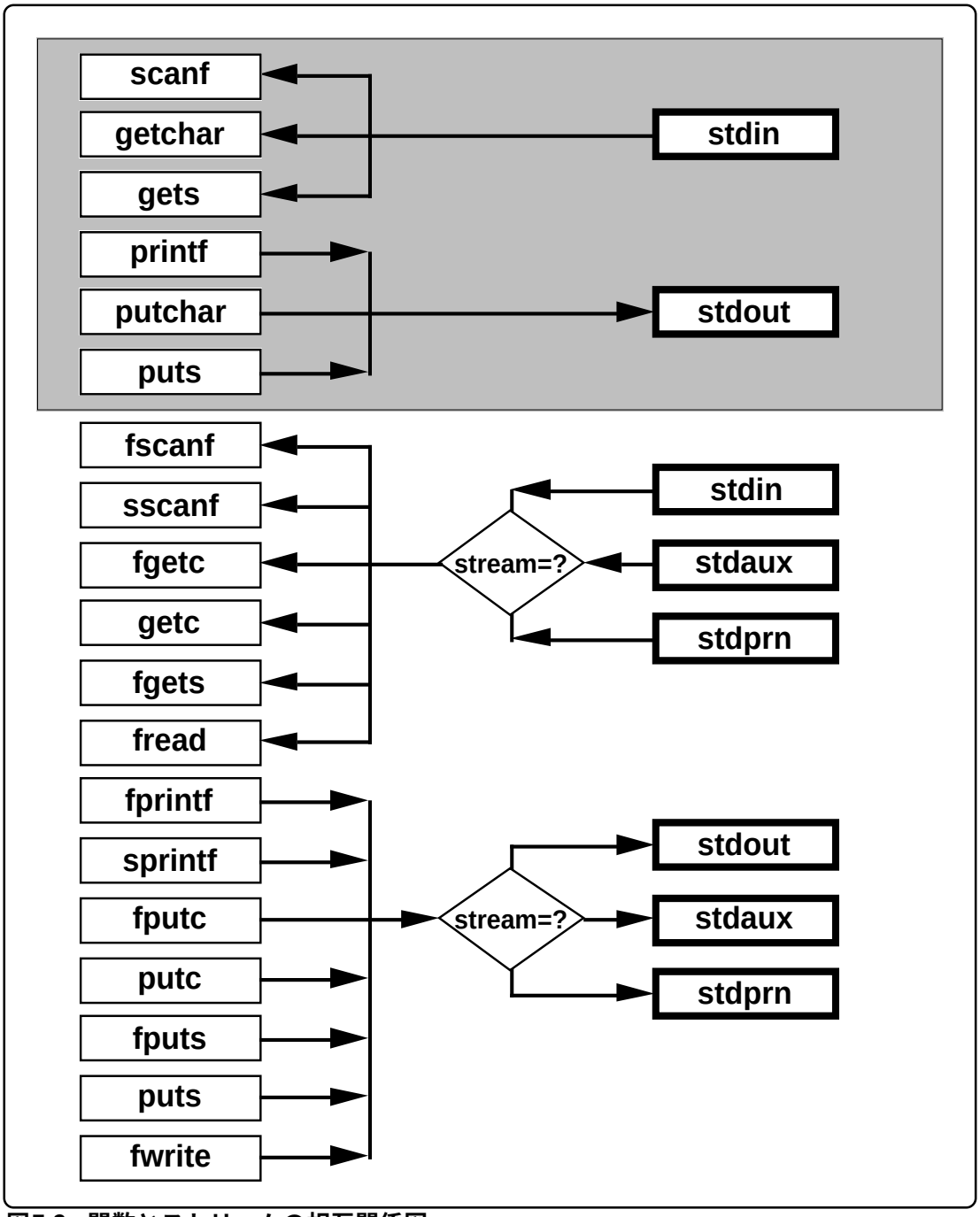
NC77の標準ライブラリはstdin、stdout、stderr、stdaux、stdprnの5種類のストリーム情報を外部構造体として持っています。この外部構造体は標準ヘッダファイルstdio.h内で定義されており、各ストリームのモード情報(入力ストリームか出力ストリームを表すフラグ)やステータス情報(エラー又はEOFを表すフラグ)等を管理しています。

表E.16　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

NC77の標準ライブラリ関数中、【図E.3】の網掛部に示す関数の対応するストリームは標準入力(stdin)又は標準出力(stdout)に固定しています。これらの関数に関しては、ストリームを変更することはできません。なお、stderrはstdoutに#defineで定義されています。

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



図E.3　関数とストリームの相互関係図

【図E.4】にstdio.h内のストリーム定義部を示します。

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

#define  stderr  stdout

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

   (以降省略)
        :
```

図E.4　stdio.h内のストリーム定義部(stdio.h)

【図E.4】に示しましたファイル構造体の要素を以下に説明します。説中の①～⑥は、【図E.4】中の①～⑥に対応しています。

①char _buff

関数scanf、fscanfでは入力の際に1文字分の先読みを行っています。先読みした文字が不用な場合は関数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。NC77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.a77から呼び出されています。

【図E.5】にinit関数のソースプログラムを示します。

```
#include <stdio.h>

FILE _iob[4];

void init( void );

void init( void )
{
    stdin->_cnt = stdout->_cnt = stdaux->_cnt = stdprn->_cnt = 0;
    stdin->_flag = _IOREAD;
    stdout->_flag = _IOWRT;
    stdaux->_flag = _IORW;
    stdprn->_flag = _IOWRT;

    stdin->_mod = _TEXT;
    stdout->_mod = _TEXT;
    stdaux->_mod = _BIN;
    stdprn->_mod = _TEXT;

    stdin->_func_in = _sget;
    stdout->_func_in = NULL;
    stdaux->_func_in = _sget;
    stdprn->_func_in = NULL;

    stdin->_func_out = _sput;
    stdout->_func_out = _sput;
    stdaux->_func_out = _sput;
    stdprn->_func_out = _pput;

#ifdef UART0
    speed(_96, _B8, _PN, _S2);
#else
    speed(_96, _B8, _PN, _S2);
#endif
    init_prn();
}
```

図E.5　init関数のソースファイル(init.c)

7700ファミリの2本のUARTを使用するシステムでは、init関数を以下の手順で変更します。前節では、device.cソースファイル中でUART0用の関数を仮に_sget0、_sput0、speed0と設定しました。

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

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

```
void init( void )
{
          :
       (省略)
          :
    stdaux->_flag = _IORW;              ←②(読み書き両用モードに設定)
          :
       (省略)
          :
    stdaux->_mod = _TEXT;               ←②(テキストモードに設定)
          :
       (省略)
          :
    stdaux->_func_in = _sget0;          ←③(UART0用のレベル3の入力関数を指定)
          :
       (省略)
          :
    stdaux->_func_out = _sput0;         ←③(UART0用のレベル3の出力関数を指定)
          :
       (省略)
          :
    speed0(_96, _B8, _PN, _S2);         ←④(UART0用のspeed関数を指定)
    speed(_96, _B8, _PN, _S2);
    init_prn();
}
```

※図中の②～④は、上記の設定手順の項番に対応しています。

図E.6　init関数の変更例

c.　変更したソースプログラムの組み込み

変更したソースファイルをシステムに組み込む方法として、以下の2通りの方法があります。

①変更した関数のソースファイルのオブジェクトファイルをリンク時に指定します。
②製品に同封の実行手順ファイル(makefile（Windows版（パソコン版）はmakefile.dos）を使用してライブラリファイルを更新します。

手順①の場合、リンク時に指定した関数が有効となり、ライブラリファイル内の同一名の関数は組み込まれません。
①の方法を【図E.7】に、②の方法を【図E.8】にそれぞれ示します。

```
% nc77 -c -g -osample  ncrt0.a77 device.r77 init.r77 sample.c<RET>

※この例は、device.cとinit.cを変更したときの記述方法です。
```

図E.7　変更したソースプログラムを直接リンクする方法

```
% make <RET>
```

図E.8　変更したソースプログラムを基にライブラリを更新する方法

付録F
エラーメッセージ一覧表

NC77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

NC77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。

```
nc77:[エラーメッセージ]
```

図F.1　コンパイルドライバnc77のエラー出力形式

```
[Error(cpp77.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(cpp77.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。cpp77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　nc77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nc77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　nc77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NC77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 '-?'	●NC77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	●-rasm77 や-link77 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　nc77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'	●NC77が認識できないファイル拡張子 (.c,.i,.a77,.r77,.hex 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] ●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-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。
Permission denied	●各処理系が実行できません。 ⇒各処理系のパーミッションを確認してください。又、パーミッションが正しい場合は、各処理系が格納されているディレクトリを環境変数で正しく設定しているかを確認してください。
can't open command file	●@で指定されたコマンドファイルがオープンできません。 ⇒正しいファイル名を指定してください
too many options	●指定されたオプションの数が100を超えています。 ⇒オプションは99文字までにしてください。
Result too large	●通常は発生しません(内部エラーです)。 ⇒弊社までご連絡ください。
Too many open files	●通常は発生しません(内部エラーです)。 ⇒弊社までご連絡ください。

F.3　cpp77エラーメッセージ

【表F.3】～【表F.6】にプリプロセッサcpp77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　cpp77エラーメッセージ一覧表(1)

番号	エラーメッセージ	エラー内容と対策
1	illegal command option	●入力ファイル名を2回指定しています。 ⇒入力ファイル名の指定を1つにしてください。
		●入力ファイル名と出力ファイル名が同じ名前です。 ⇒出力ファイル名は入力ファイル名と違う名前を指定してください。
		●出力ファイル名を2回指定しています。 ⇒出力ファイル名の指定を1つにしてください。
		●コマンドラインが-oオプションで終了しています。 ⇒-oオプションの後に出力ファイル名を指定してください。
		●インクルードファイルのパスを指定する-Iオプションが制限値を越えました。 ⇒-Iオプションを8個以下にしてください。
		●コマンドラインが-Iオプションで終了しています。 ⇒-Iオプションの後に出力ファイル名を指定してください。
		●-Dオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は__)でありません。不当なマクロ名定義を行っています。 ⇒正しいマクロ名、正しいマクロ定義を指定してください。
		●コマンドラインが-Dオプションで終了しています。 ⇒-Dオプションの後に出力ファイル名を指定してください。
		●-Uオプションの次の文字列がマクロ名で使用できる文字タイプ(英字又は__)でありません。 ⇒正しいマクロ名定義を指定してください。
		●cpp77で使用できないオプションを指定しています。 ⇒正しいオプションを指定してください。
11	cannot open input file	●入力ファイルが見つかりません。 ⇒正しいファイル名を指定してください。
12	cannot close input file	●入力ファイルをクローズできません。 ⇒入力ファイルを確認してください。

表F.4　cpp77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] ⇒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-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。

表F.5　cpp77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
28	input buffer over flow	●ソースファイルリード中に入力行バッファがオーバーフローしました。又は、マクロ変換中にバッファがオーバーフローしました。 ⇒ソースファイルの1行を1023文字以下にしてください。マクロ変換を予想される場合は、変換結果が1023文字以下になるように変更してください。
29	EOF in comment	●コメント中にファイルが終了しています。 ⇒ソースファイルを正しく記述してください。
31	EOF in preprocess command	●プリプロセスコマンド中にファイルが終了しています。 ⇒ソースファイルを正しく記述してください。
32	unknown preprocess command	●不当なプリプロセスコマンドを使用しています。 ⇒CPP30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 commandusage	●#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。

表F.6　cpp77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
53	elseif / else sequence error	●#if～#ifdef～#ifndefがないのに#elseif又は#elseを使用しています。 ⇒#elseif又は#elseは#if～#ifdef～#ifndefの後に使してください。
54	endif not exist	●#if～#ifdef～#ifndefに対応した#endifがありません。 ⇒#endifをソースファイル中に記述してください。
55	endif sequence error	●#if～#ifdef～#ifndefがないのに#endifを使用しています。 ⇒#endifは#if～#ifdef～#ifndefの後に使してください。
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　cpp77ワーニングメッセージ

【表F.7】にプリプロセッサcpp77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　cpp77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	●cpp77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　ccom77エラーメッセージ

【表F.8】～【表F.19】にコンパイラccom77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　ccom77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redefined	●#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	●#pragmaPARAMETERで指定した関数の定義が、その関数の呼び出しの後に記述されてます。 ⇒関数の呼び出しを行う前に宣言してください。
#pragma INTCALL function's argument on stack	●#pragma INTCALL で宣言した関数の実体をC言語で記述した場合に引き数がスタック渡しになっています。 ⇒#pragma INTCALL で宣言した関数の実体をC言語で記述する場合には引き数にはレジスタ渡しとなる型を指定してください。
#pragma PARAMETER functions register not allocated	●#pragmaPARAMETERで指定した関数で指定したレジスタは、記述できません。 ⇒レジスタを正しく記述してください。
'const' is duplicate	●constを2回以上記述しています。 ⇒型修飾子を正しく記述してください。
'far' & 'near' conflict	●同じ変数(関数)に対してnear/farの宣言が一致していません。 ⇒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。

表F.9　ccom77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
'volatile' is duplicate	●volatileを2回以上記述しています。 ⇒型修飾子を正しく記述してください。
(can't read C source from filename line 行数 for error message)	●エラーが発生したソースラインを表示できません。filenameで示されるファイルがないか、行番号が、ファイルに存在しません。 ⇒ファイルの存在を確認してください。
(can't open C source filename for error message)	●エラーが発生したソースファイルがオープンできません。 ⇒ファイルの存在を確認してください。
argument type given both places	●関数定義中の引数の宣言において、引数リストと重複して引数の宣言を行っています。 ⇒引数リストか、引数の宣言のどちらかで引数の宣言を行ってください。
array of functions declared	●配列宣言において関数のポインタ配列ではなく関数自身の配列を宣言しています。 ⇒関数のポインタ配列等に変更してください。
array size is not constant integer	●配列の宣言において要素数が定数ではありません。 ⇒要素数を定数で記述してください。
asm()'s string must have 1 $$	●asmステートメントで$$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。

表F.10　ccom77エラーメッセージ一覧(3)

エラーメッセージ	エラー内容と対策
can't 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。
conflict declare of 変数名	●1度目と2度目で記憶域クラスの異なる重複定義を行っています。 ⇒変数を2度宣言する場合は、同じ記憶域クラスで行ってください。
conflict function argument type of 変数名	●引数リストに同じ変数名があります。 ⇒変数名を変更してください。
declared register parameter function's body declared	●#pragma PARAMETER で宣言した関数をC言語で実体の定義を行っています。 ⇒#pragma PARAMETER で宣言した関数はC言語での実体記述を行わないでください。
default function argument conflict	●プロトタイプ宣言において、引数のデフォルト値を２回以上宣言しています。 ⇒引数のデフォルト値は、１回だけ宣言してください。
default: is duplicated	●defaultの値を重複して使用しています。 ⇒1つのswitch文で、defaultは1つにしてください。
do while(struct/union) statement	●do while文の式にstruct/union型を使用しています。 ⇒do while文の式は、スカラ型を記述してください。
do while(void) statement	●do while文の式にvoid型を使用しています。 ⇒do while文の式は、スカラ型を記述してください。
duplicate frame position defind 変数名	●auto変数が2回以上記述しています。 ⇒正しく記述してください。
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 type's bitfield	●不当な型のビットフィールドを宣言しています。 ⇒ビットフィールドは、整数型を使用してください。
for(; struct/union;) statement	●for文の2番目の式にstruct/union型を使用しています。 ⇒for文の2番目の式は、スカラ型を記述してください。

表F.11　ccom77エラーメッセージ一覧(4)

エラーメッセージ	エラー内容と対策
for(; void ;) statement	●for文の2番目の式にvoid型を使用しています。 ⇒for文の2番目の式は、スカラ型を記述してください。
function initialized	●関数の宣言に対して初期化式を記述しています。 ⇒初期化式を削除してください。
function member declared	●構造体,共用体のメンバで関数型を指定しています。 ⇒メンバを正しく記述してください。
function returning a function declared	●関数の宣言においてリターン値の型が関数型になっています。 ⇒戻り値の型を関数へのポインタ型等に変更してください。
function returning an array	●関数の宣言においてリターン値の型が配列型になっています。 ⇒戻り値の型を関数へのポインタ型等に変更してください。
identifier (変数名) is duplicated	●変数が重複して定義されています。 ⇒変数の定義を正しく指定してください。
if(struct/union) statement	●if文の式にstruct/union型を使用しています。 ⇒if文の式は、スカラ型を記述してください。
if(void) statement	●if文の式にvoid型を使用しています。 ⇒if文の式は、スカラ型を記述してください。
illegal 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 memberaccess	●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。

表F.12　ccom77エラーメッセージ一覧(5)

エラーメッセージ	エラー内容と対策
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 ' ? : ' operand	●?: 演算子の記述に誤りがあります。 ⇒演算子の各式を確認してください。また、:の左右の式の型は、互換型である必要があります。
invalid '!=' operands	●!=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '&&' operands	●&&演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '&' operands	●&演算子の記述に誤りがあります。 ⇒演算子の右辺式を確認してください。
invalid '&=' operands	●&=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '()' operand	●()の左辺式が関数ではありません。 ⇒()の左辺式は関数又は関数へのポインタを記述してください。
invalid '＊' operands	●乗算の場合＊演算子の記述に誤りがあります。ポインタ演算子の場合、右辺式がポインタ型でありません。 ⇒乗算の場合、演算子の左辺式,右辺式を確認してください。ポインタの場合、右辺の型を確認してください。
invalid '＊=' operands	●＊=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '+' operands	●+演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '+=' operands	●+=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '-' operands	●-演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '-=' operands	●-=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '/=' operands	●/=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<<' operands	●<<演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<<=' operands	●<<=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '<=' operands	●<=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。

表F.13　ccom77エラーメッセージ一覧(6)

エラーメッセージ	エラー内容と対策
invalid '=' operand	●=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '==' operands	●==演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>=' operands	●>=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>>' operands	●>>演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '>>=' operands	●>>=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '[ ]' operands	● [ ] の左辺式が配列,ポインタ型ではありません。 ⇒ [ ] の左辺式は配列,又はポインタ型を記述してください。
invalid '^=' operands	●^=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '|=' operands	● |=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '||' operands	● || 演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid '%=' operands	● %=演算子の記述に誤りがあります。 ⇒演算子の左辺式,右辺式を確認してください。
invalid ++ operands	● ++単項演算子又は後置演算子の記述に誤りがあります。 ⇒単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。
invalid -- operands	● --単項演算子又は後置演算子の記述に誤りがあります。 ⇒単項演算子の場合、右辺式を確認してください。後置演算子の場合、左辺式を確認してください。
invalid -> used	● ->の左辺式が、構造体,共用体型ではありません。 ⇒ ->の左辺式を、構造体,共用体型で記述してください。
invalid (? ;)'s condition	●三項演算子の記述に誤りがあります。 ⇒三項演算式を確認してください。
Invalid #pragma OS拡張機能 interrupt number	●#pragma OS拡張機能 で記述したINT番号は、指定することができません。 ⇒正しく指定してください。
Invalid #pragma INTCALL interrupt number	●#pragma INTCALL で記述したINT番号は、指定することができません。 ⇒正しく指定してください。
Invalid #pragma SPECIAL special page number	●#pragma SPECIAL で記述した番号は指定することができません。 ⇒正しく指定してください。

表F.14　ccom77エラーメッセージ一覧(7)

エラーメッセージ	エラー内容と対策
invalid CAST operand	●cast演算子に誤りがあります。void型を他の型にキャスト及び、構造体,共用体からもしくは他の構造体、共用体へのキャストはできません。 ⇒正しく記述してください。
invalid asm()'s argument	●asmステートメントに使用できる変数は、auto変数と引数です。 ⇒auto変数か引数で記述してください。
invalid bitfield declare	●ビットフィールドの宣言で誤りがあります。 ⇒正しく記述してください。
invalid break statements	●break文を記述できないところで使用しています。 ⇒switch, while, do-while, forのなかで記述してください。
invalid case statements	●switch文に誤りがあります。 ⇒switch文を正しく記述してください。
invalid case value	●caseの値に誤りがあります。 ⇒整数型,列挙型の定数を記述してください。
invalid cast operator	●cast演算子の記述に誤りがあります。 ⇒正しく記述してください。
invalid continue statements	●continue文を記述できないところで使用しています。 ⇒while, do-while, forのなかで記述してください。
invalid default statements	●switch文に誤りがあります。 ⇒switch文を正しく記述してください。
invalid enumerator initialized	●列挙子の初期値に変数名を記述するなど誤った指定を行っています。 ⇒列挙子の初期値を正しく記述してください。
invalid function argument	●関数定義中の引数の宣言において、引数リストに含まれない引数を宣言しています。 ⇒引数リストに存在する変数を宣言してください。
invalid function's argument declaration	●関数の引数の宣言に誤りがあります。 ⇒正しく記述してください。
invalid function's default argument	●関数のデフォルト引数に誤りがあります。 ⇒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。

表F.15　ccom77エラーメッセージ一覧(8)

エラーメッセージ	エラー内容と対策
invalid 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。
invalid sign specifier	● signed/unsignedを2回以上記述しています。 ⇒型指定子を正しく記述してください。
invalid strage class for data	●記憶クラスの指定に誤りがあります。 ⇒正しく記述してください。
invalid struct or union type	●列挙型のデータに対して構造体,共用体のメンバ参照しています。 ⇒正しく記述してください。
invalid truth expression	●条件式(?:)の1つめの式でvoid, struct, union型を使用してします。 ⇒スカラ型で記述してください。
invalid type specifier	● int　int　i; 等のように同じ型指定子を2回以上記述しているか、float　int　i; 等のように矛盾した型指定子を記述しています。 ⇒型指定子を正しく記述してください。
invalid type's bitfield	●不当な型のビットフィールドを宣言しています。 ⇒ビットフィールドは整数型を使用してください。
invalid unary '!' operands	●!単項演算子の記述に誤りがあります。 ⇒演算子の右辺式を確認してください。

表F.16　ccom77エラーメッセージ一覧(9)

エラーメッセージ	エラー内容と対策
invalid 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。
Not enough memory	[UNIX版] ●スワップ領域が不足しています。 ⇒スワップ領域を増やしてください。 [MS-Windows 95 / NT版] ●メモリ空間が不足しています。 ⇒メモリを増やすか、Windows 95 / 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。
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。

表F.17　ccom77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
parameter function's body declared	●#pragma PARAMETERで指定された関数名と同一の名前で関数が定義されています。 ⇒#pragma PARAMETERで指定した関数はアセンブリ言語で記述してください。また、アセンブリ言語で記述した関数名と重複している場合は関数名を変更してください。
redeclare of 列挙子	●列挙子が重複して定義されています。 ⇒どちらかの列挙子の名前を変更してください。
redefine function 関数名	●関数名で示される関数が2度定義されています。 ⇒関数は1度しか定義できません。どちらかの関数名を変更してください。
redefinition tag of enum タグ名	●列挙を二重に定義しています。 ⇒列挙の定義は1回にしてください。
redefinition tag of struct タグ名	●構造体を二重に定義しています。 ⇒構造体の定義は1回にしてください。
redefinition tag of union タグ名	●共用体を二重に定義しています。 ⇒共用体の定義は1回にしてください。
reinitialized of 変数名	●同じ変数に対して初期化式を2度指定しています。 ⇒初期化式を1つにしてください。
Sorry stack fram memory exhaust, max. 128 bytes but now nnn bytes (NC30, NC308 のみ)	●スタックフレーム上に確保可能な引数は最大128バイトまでです。現在 nnnバイト使用しています。 ⇒引数のサイズあるいは引数の個数減らしてください。
Sorry 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 (NC79)、 255バイト(NC30、NC308、NC77、および、 NC79で起動オプション -fDPO8 使用時) までです。 現在nnnバイト使用しています。 ⇒auto変数、引数などのスタックフレーム領域に確保される変数を減らしてください。
Sorry, compilation terminated because of these errors in関数名	●関数名で示される関数でエラーが発生しました。コンパイルを中止します。 ⇒このメッセージが出力される以前のエラーを修正してください。
Sorry, compilation terminated because of too many errors.	●ソースファイル中のエラーがエラーの上限(50個)を超えました。 ⇒このメッセージが出力される以前のエラーを修正してください。
struct or enum's tag used for union	●構造体,列挙型のタグ名を共用体のタグ名として使用しています。 ⇒タグ名を変更してください。
struct or union's tag used for enum	●構造体,共用体のタグ名を列挙型のタグ名として使用しています。 ⇒タグ名を変更してください。

表F.18　ccom77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
struct or union,enum does not have long or sign	●struct/union/enum型指定子にlongやsignedの型指定子を記述しています。 ⇒型指定子を正しく記述してください。
switch's condition is floating	●switch文の式に浮動小数点型を使用しています。 ⇒整数型,列挙型を使用してください。
switch's condition is void	●switch文の式にvoid型を使用しています。 ⇒整数型,列挙型を使用してください。
switch's condition must integer	●switch文の式に整数型,列挙型以外の型を使用しています。 ⇒整数型,列挙型を使用してください。
syntax error	●文法エラーです。 ⇒正しく記述してください。
System Error...	●通常は発生しません(内部エラーです)。 ⇒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。

表F.19　ccom77エラーメッセージ一覧表(12)

エラーメッセージ	エラー内容と対策
unknown structure idetifier "変数名"	●.の左辺式が、構造体,共用体型ではありません。 ⇒．の左辺式を、構造体,共用体型で記述してください。
unknown variable "変数名" used in asm()	●asmステートメントにおいて、未定義の変数名を使用しています。 ⇒変数を定義してください。
unknown variable 変数名	●未定義の変数名を使用しています。 ⇒変数を定義してください。
unknown variable 変数名 used	●未定義の変数名を使用しています。 ⇒変数を定義してください。
void array is invalid type, int array assumed	●void型の配列は宣言できません。int型の配列として処理を継続します。 ⇒型指定子を正しく記述してください。
void value can't return	●voidでキャストされた値を関数の戻り値に記述しています。 ⇒正しく記述してください。
while(struct/union) statement	●while文の式にstruct/union型を使用しています。 ⇒while文の式は、スカラ型を記述してください。
while(void) statement	●while文の式にvoid型を使用しています。 ⇒while文の式は、スカラ型を記述してください。
multiple #pragma EXT4MPTR's pointer, ignored (NC30 のみ)	●#pragma EXT4MPTR が２個以上宣言されています。 ⇒宣言を一つにしてください。
zero size array member	●サイズがゼロの配列です。 ⇒サイズを明確にしてください。

F.6　ccom77ワーニングメッセージ

【表F.20】～【表F.29】にコンパイラccom77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.20　ccom77ワーニングメッセージ一覧表(1)

ワーニングメッセージ	ワーニング内容と対策
#pragma プラグマ名 & HANDLER both specified	●１つの関数に#pragma プラグマ名 と#pragma INTERRUPTの両方を指定しています。 ⇒#pragma プラグマ名と#pragma INTERRUPT は、排他的に指定してください。
#pragma プラグマ名 & INTERRUPT both specified	●１つの関数に#pragma プラグマ名 と#pragma INTERRUPTの両方を指定しています。 ⇒#pragma プラグマ名と#pragma INTERRUPT は、排他的に指定してください。
#pragma プラグマ名 & TASK both specified	●１つの関数に#pragmaプラグマ名と#pragmaTASKの両方を指定しています。 ⇒#pragma プラグマ名と#pragma TASKは、排他的に指定してください。
#pragma プラグマ名 format error	●#pragma プラグマ名 の記述に誤りがあります。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 の記述に誤りがあります。この行を無視します。 ⇒正しく記述してください。
#pragma プラグマ名 not function, ignored	●#pragma プラグマ名 において関数でない名前を記述しています。 ⇒関数名で記述してください。
#pragma プラグマ名's function must be pre-declared, ignored	●#pragma プラグマ名で指定した関数が、宣言されていません。 ⇒#pragma プラグマ名で指定する関数は、予めプロトタイプ宣言を行ってください。
#pragma プラグマ名's function must be prototyped,ignored	●#pragma プラグマ名で指定した関数が、プロトタイプ宣言されていません。 ⇒#pragma プラグマ名で指定する関数は、予めプロトタイプ宣言を行ってください。
#pragma プラグマ名's function return type invalid,ignored	●#pragmaプラグマ名で指定された関数のリターン値の型が不当です。 ⇒リターン値の型は、struct, union, double型以外を指定してください。
#pragma プラグマ名 unknown switch, ignored	●#pragmaプラグマ名で不正な switch を記述しています。 ⇒正しい switch を指定してください。
#pragma プラグマ名 format error, ignored	●#pragma プラグマ名 の記述に誤りがあります。この行を無視します。 ⇒正しく指定してください。

表F.21　ccom77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 variable initialized, ADDRESS ignored	●#pragma ADDRESSで指定した変数を初期化しています。#pragma ADDRESSの指定を無視します。 ⇒#pragma ADDRESSか、初期化式のどちらかを削除してください。
#pragma ASM line too long, then cut	●#pragma ASM で記述できる一行の文字数1024バイトを越えています。 ⇒1024バイト以内で記述してください。
#pragma directive conflict	●一つの関数に対して、異なる機能の#pragma を指定しています。 ⇒正しく記述してください。
#pragma DP[n]DATA format error, ignored (NC79のみ)	●#pragma DP[n]DATA と -fDPO8オプションを併用しています。 ⇒#pragma DP[n]DATA と -fDPO8オプションを併用した場合、#pragma DP[n]DATA は無効となります。-fDPO8オプションを削除してください。 ●#pragma DP[n]DATA のフォーマットが間違っています。 ⇒正しく記述してください。
#pragma JSRA illegal location ,ignored (NC30, NC308 のみ)	●関数のスコープ内に#pragma JSRAを記述しています。 ⇒#pragma JSRAは、関数外に記述してください。
#pragma JSRW illegal location ,ignored (NC30, NC308 のみ)	●関数のスコープ内に#pragma JSRWを記述しています。 ⇒#pragma JSRWは、関数外に記述してください。
#pragma PARAMETER function's address used	●#pragma PARAMETERで指定された関数のアドレスをポインタ変数に代入しています。 ⇒代入しないで、正しく記述してください。
#pragma control for function duplicate, ignored (NC30, NC308 のみ)	●#pragmaで同じ関数に対して、INTERRUPT, TASK, HANDLER, CYCHANDLER, ALMHANDLER を重複して指定しています。 ⇒INTERRUPT,TASK,HANDLER,CYCHANDLER, ALMHANDLERのうち１つを指定してください。
'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。
-OR,-OS duped option	●-OR、-OSを重複して指定しています。 ⇒オプションを正しく指定してください。
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。

表F.22　ccom77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
assignment from const pointer to non-const pointer	●constポインタ変数をconstではないポインタに代入しています。 ⇒データの型を確認してください。
assignment from volatile pointer to non-volatile pointer	●volatileポインタ変数をvolatileではないポインタに代入しています。 ⇒データの型を確認してください。
block level extern variable initialize forbid,ignored	●関数内のextern変数宣言で初期化式を記述しています。 ⇒初期化式を削除するか、記憶域クラスを変更してください。
can't get address from register storage class variable	●register記憶域クラスの変数に&演算子を記述しています。 ⇒register記憶域クラスの変数に&演算子を記述しないでください。
can't get size of bitfield	●sizeof演算子のオペランドにビットフィールド型を記述しています。 ⇒sizeof演算子のオペランドを正しく記述してください。
can't get size of function	●sizeof演算子のオペランドに関数名を記述しています。 ⇒sizeof演算子のオペランドを正しく記述してください。
can't get size of function,unit size 1 assumed	●関数へのポインタを++,--しています。増分、減分の値を1として処理を継続します。 ⇒関数へのポインタを++,--しないでください。
char array initialized by wchar_t string	●char型の初期化式をwchar_t型の文字列で初期化しています。 ⇒初期化式の型を合わせてください。
case value is out of range	●caseの値がswitchの引数の範囲を越えてます。 ⇒正しく記述してください。
character buffer overflow	●文字列のサイズが512文字を超えました。 ⇒511文字以下で記述してください。
character constant too long	●文字定数(シングルクォートに囲まれた文字)の文字数が多すぎます。 ⇒正しく記述してください。
constant variable assignment	●const型修飾子で指定した変数に対して代入しています。 ⇒代入先の宣言部を確認してください。
cyclic or alarm handler always Bank 0 (NC77, NC79 のみ)	●#pragma CYCHANDLER又はALMHANDLERで指定した関数は、常にバンク0(アドレスが10000H未満)の領域にコンパイルされます。 ⇒バンク0以外に配置しないで下さい。

表F.23　ccom77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
cyclic or alarm handler always load DT (NC77, NC79 のみ)	●#pragmaCYCHANDLER又はALMHANDLERで指定した関数は、#pragma LOADDT をする必要はありません。 ⇒#pragma LOADDT を削除してください。
cyclic or alarm handler function has argument	●#pragma CYCHANDLER又はALMHANDLERで指定した関数が、引数を使用しています。 ⇒#pragma CYCHANDLER又はALMHANDLERで指定した関数は、引数を使用できません。引数を削除してください。
enumerator value overflow size of unsigned char	●-fCEオプション使用時において、列挙子の値が255を越えました。 ⇒255以下で表現できるように記述してください。
enumerator value overflow size of unsigned int	●列挙子の値が65535を越えました。 ⇒65535以下で表現できるように記述してください。
enum's bitfield	●ビットフィールドのメンバに列挙型を用いて定義しています。 ⇒違う型のメンバを用いてください。
external variable initialized,change to public	●externで宣言した変数に対して、初期化式を記述しています。externを無視します。 ⇒externを削除してください。
far pointer (implicitly) casted by near pointer	●farポインタが、nearポインタに変換されました。 ⇒データの型、near/farを確認してください。
function must be far	●関数をnear型で宣言しています。 ⇒正しく記述してください。
handler function called	●#pragma HANDLERで指定した関数を呼び出しています。 ⇒ハンドラ関数は呼び出さないようにしてください。
handler function can't return value	●#pragma HANDLERで指定した関数が、戻り値を使用しています。 ⇒#pragma HANDLERで指定した関数は、戻り値を使用できません。戻り値を削除してください。
handler function has argument	●#pragma HANDLERで指定した関数が、引数を使用しています。 ⇒#pragma HANDLERで指定した関数は、引数を使用できません。引数を削除してください。
hex character is out of range	●文字定数においてHEX文字が長すぎます。また、\の後に16進数以外の文字が入っています。 ⇒HEX文字を短くしてください。
identifier (メンバ名) is dupilicated, this declare ignored	●メンバ名が重複して定義されています。この宣言を無視します。 ⇒メンバ名の宣言を1つにしてください。
identifier (変数名) is dupilicated	●変数名が重複して定義されています。この宣言を無視します。 ⇒変数名の宣言を1つにしてください。

表F.24　ccom77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
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。
init elements overflow,ignored	●初期化式が初期化しようとする変数のサイズを超えました。 ⇒初期化式の数が、初期化する変数のサイズを超えないようにしてください。
inline function is called as normal funciton before, change to static function	●記憶クラスinline で宣言された関数が通常の関数として呼び出されています。 ⇒inline関数は使用する前に必ず定義を行ってください。
integer constant is out of range	●整数定数の値がunsigned longで表現できる値を超えました。 ⇒定数の値をunsigned longで表現できる値で記述してください。
interrupt function called	●#pragma INTERRUPTで指定した関数を呼び出しています。 ⇒割り込み処理関数は呼び出さないようにしてください。
interrupt function can't return value	●#pragma INTERRUPTで指定した割り込み処理関数が、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	●#pragma INTERRUPTで指定した割り込み処理関数が、引数を使用しています。 ⇒割り込み関数では引数を使用できません。引数を削除してください。
interrupt function has argument	●#pragma INTERRUPTで指定した割り込み処理関数が、引数を使用しています。 ⇒割り込み関数では引数を使用できません。引数を削除してください。
invalid #pragma EQU	●#pragma EQUの記述に誤りがあります。この行を無視します。 ⇒正しく記述してください。
invalid #pragma SECTION, unknown section base name	●#pragma SECTIONにおいてセクション名に誤りがあります。指定できるセクション名は data, bss, program, rom, interruptです。この行を無視します。 ⇒正しく記述してください。

表F.25　ccom77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
invalid #pragma operand,ignored	●#pragmaのオペランドの記述に誤りがあります。この行を無視します。 ⇒正しく記述してください。
invalid function argument	●関数引数の型に誤りがあります。 ⇒正しく記述してください。
invalid asm's M flag (NC77, NC79 のみ)	●asmステートメントにおいて、Mフラグに設定する値に誤りがあります。 ⇒整数型の定数 (0, 1, 2)で記述してください。
invalid asm's MX flag, ignored (NC77, NC79 のみ)	●asmステートメントにおいて、MXフラグに設定する値が整数値の定数ではありません。 ⇒整数型の定数 (0, 1, 2)で記述してください。
invalid asm's X flag (NC77, NC79 のみ)	●asmステートメントにおいて、Xフラグに設定する値に誤りがあります。 ⇒整数型の定数 (0, 1, 2)で記述してください。
invalid return type	●return文の式が関数の型と異なっています。 ⇒リターン値を関数の型に合わせるか、関数の型をリターン値の型に合わせてください。
invalid storage class for function, change to extern	●関数宣言において、不当な記憶域クラスを使用しています。externとして処理します。 ⇒記憶域クラスをexternにしてください。
Kanji in #pragma ADDRESS	●#pragma ADDRESSの記述に漢字コードが含まれています。この行を無視します。 ⇒この宣言では漢字コードを記述しないでください。
keyward (キーワード) are reserved for future	●将来のために予約されているキーワードを使用しています。 ⇒別の名前に変更してください。
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 assign-ment	●レジスタ引数の関数のアドレスを、レジスタ引数でない(プロトタイプされていない)関数のポインタ変数へ代入しています。 ⇒関数のポインタ変数の宣言をプロトタイプ形式にしてください。
multi-character character constant	●2文字以上の文字定数を使用しています。 ⇒2文字以上のときはワイド文字(L'xx')を使用してください。
near/far is conflict beyond over typedef	●near/far を指定して typedef した型を参照時に再度、near/farを指定して宣言しています。 ⇒型指定子を正しく記述してください。

表F.26　ccom77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
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 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。
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。
no meaning statement	●意味のないプログラムを記述しています。 ⇒プログラムを正しく記述してください。
non-prototyped function used	●プロトタイプ宣言していない関数を呼び出しています。-Wnon_prototypeオプションを指定した時のみ出力されます。 ⇒プロトタイプ宣言を行うか、-Wnon_prototypeオプションを削除してください。
non-propototyped function declared	●定義されている関数のプロトタイプ宣言が存在しません(-WNPオプション指定時のみ表示)。 ⇒プロトタイプ宣言を行ってください。
octal constant is out of range	●8進数の定数に8進数で使用できない文字が含まれています。 ⇒8進数の定数は0から7で記述してください。
octal_character is out of range	●8進数の定数に8進数で使用できない文字が含まれています。 ⇒8進数の定数は0から7で記述してください。
overflow in floating value converting to integer	●浮動小数点の値が制限を越えています。 ⇒制限以下で記述してください。
old style function declaration	●K&R形式の関数宣言が行われています。 ⇒ANSI形式で記述してください。
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 name of (xxx)	●同じtypedefの定義を2度行っています。 ⇒typedefの定義は1回にしてください。
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、#pragma SECTION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。

表F.27　ccom77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
sorry, get stack's address, but DT not 0 (NC77, NC79 のみ)	●オプション-bankを指定した時に発生します。auto変数のアドレスをポインタに代入し、そのポインタでオブジェクトを参照した場合、DTがバンク0以外を示しているため、バンク0を参照できません。 ⇒ポインタを far で宣言してください。
size of incomplete type	●sizeof演算子のオペランドに定義されていない構造体、共用体を記述しています。 ⇒構造体、共用体を先に定義してください。
	●sizeof演算子のオペランドに定義されている配列の要素数が決定していません。 ⇒構造体、共用体を先に定義してください。
size of void	●void型変数に対して、sizeof演算子を使用しています。 ⇒正しく記述してください。
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	●#pragma TASKで指定した関数を呼び出しています。 ⇒タスク関数は呼び出さないようにしてください。
task function can't return value	●#pragma TASKで指定した関数が、戻り値を使用しています。 ⇒#pragma TASKで指定した関数は、戻り値を使用できません。戻り値を削除してください。
task function has invalid argument	●#pragma TASK で指定した関数が、引数を使用しています。 ⇒#pragma TASK で指定した関数は、引数を使用できません。引数を削除してください。
this comparison is always false	●常に偽になる比較を行っています。 ⇒条件式を確認してください。
this comparison is always true	●常に真になる比較を行っています。 ⇒条件式を確認してください。

表F.28　ccom77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
this feature not supported now, ignored	●文法エラーです。この記述は、将来拡張用の文法ですので使用しないでください。 ⇒正しく記述してください。
this function used before with non-default argument	●使用されたことのある関数をデフォルト引数を持つ関数として宣言しています。 ⇒関数を使用する前にデフォルト引数を宣言してください。
this interrupt function is called as normal function before	●使用したことのある関数を #pragma INTERRUPT で宣言しています。 ⇒割り込み関数は呼び出せません。#pragma の内容を確認してください。
too big octal character	●文字定数、文字列中の8進数の定数が、限界値（10進数で255）を超えました。 ⇒255以下の値で記述してください。
too few parameters	●プロトタイプ宣言で宣言した時より引数が足りません。 ⇒引数の数を確認してください。
too many parameters	●プロトタイプ宣言で宣言した時より引数が多すぎます。 ⇒引数の数を確認してください。
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。
uncomplete return type	●不完全な型を戻り値に記述しています。 ⇒戻り値を確認してください。
unknown #pragma STRUCT xxx	●#pragma STRUCTxxxは、処理できません。この行を無視します。 ⇒正しく記述してください。
Unknown debug option (-dx)	●オプション-dxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown function option (-Wxxx)	●オプション-Wxxxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown function option (-fx)	●オプション-fxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown function option (-gx)	●オプション-gxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown optimize option (-mx)	●オプション-mxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown optimize option (-Ox)	●オプション-Oxは、指定できません。 ⇒オプションを正しく指定してください。
Unknown option (-x)	●オプション-xは、指定できません。 ⇒オプションを正しく指定してください。

表F.29　ccom77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
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。

付録G
スタックサイズ計算ユーティリティ（stk77）

スタックサイズ算出ユーティリィティstk77の起動方法と起動オプションの機能を説明します。

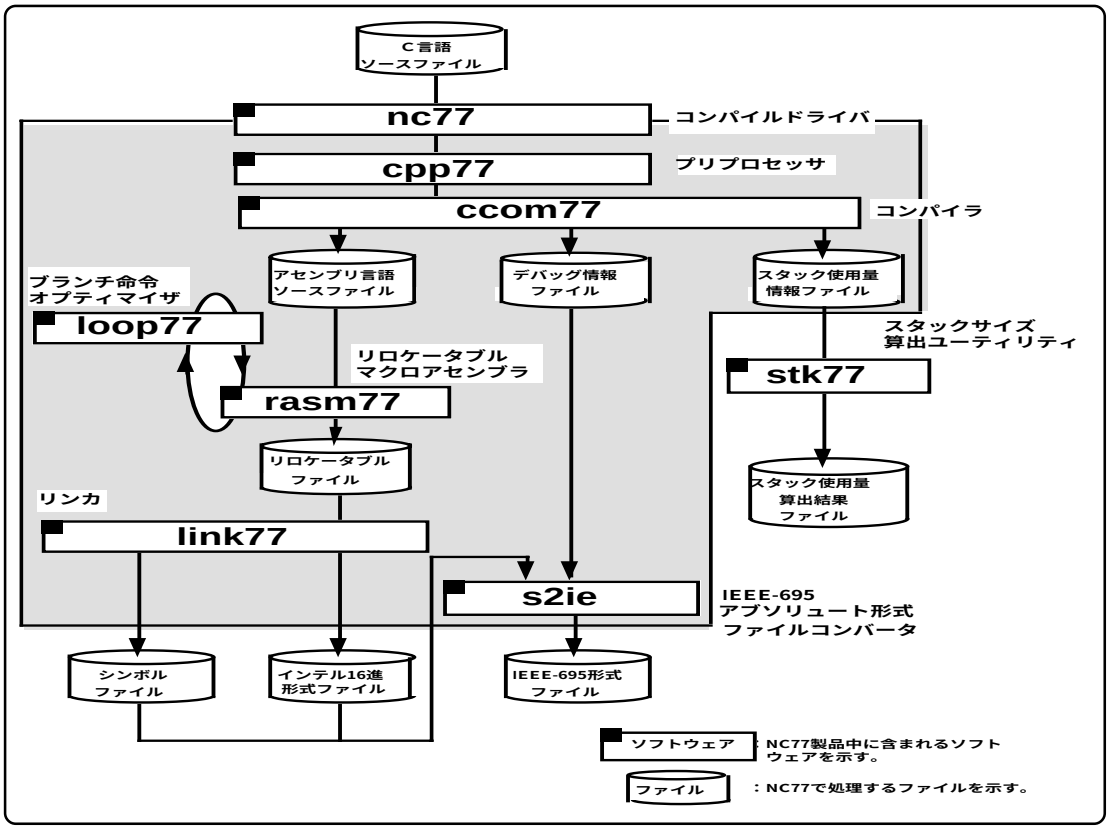
G.1　stk77 の概要

G.1.1　stk 77の処理概要

スタックサイズ算出ユーティリィティstk77は、コンパイルドライバの起動オプション-fshow_stack_usage（-fSSU）を指定してコンパイルしたときに生成されるスタック使用量表示ファイル（拡張子.stk）を処理して、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（Cフロー）を求めます。また、stk77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.スタック使用量表示ファイル（必要項目）
2.シンボルファイル※1（必要に応じて記述する項目）
3.起動オプション（必要に応じて記述する項目）

NC77の処理フローを【図G.1】に示します。



図G.1　NC77の処理フロー

※1. stk77はシンボルファイル（拡張子.sym）名の入力により、ソースファイルに対応したスタック使用量表示ファイルの入力を省略することができます。この場合、そのマップファイルを作成したすべてのソースファイルに対応したスタック使用量表示ファイルが必要になります。

G.1.2　スタック使用量表示ファイル

このファイルは、コンパイル時にnc77の起動オプション-fshow_stack_usage(-fSSU)を指定することで出力されるスタックの使用状況を格納したファイルです。ファイルの属性は.stkです。NC77に同封していますスタックサイズ算出ユーティリィティstk77は、スタック使用量表示ファイルから指定したファイル単位で使用するスタックサイズを算出します。ファイルの出力例を【図G.2】に示します。

```
FUNCTION main                                    ←①
    context        5 bytes                       ←②
    auto           2 bytes                       ←③
    f8regSize      0 bytes                       ←④
           6 bytes PUSH &   CALL printf          ←⑤
           6 bytes PUSH (MAX)                    ←⑥
=========================================================
```

図G.2　スタック使用量表示ファイル例(smp.stk)

スタック使用量表示ファイルの内容を以下に説明します。項目番号の①～⑥は【図G.2 】中の①～⑥に対応しています。

① 関数名を示しています。
② その関数を呼ばれたときにスタック上に積まれる戻りアドレス、及び旧のフレームポインタ(DPR)のために使用するサイズを示しています。
③ 記憶クラスauto又はテンポラリ領域として使用するスタックサイズを示しています。
④64ビット浮動小数点演算用の内部レジスタの総バイト数を示しています。
⑤ 関数呼び出し時にスタックにプッシュしたバイト数と呼び出した関数名を示しています。
⑥ 関数内でプッシュする最大のバイト数を示しています。

関数を呼び出すときの引き数のプッシュもすべて呼ぶ側が使用するスタックとして計算します。またポインタ間接で呼び出された関数はどの関数が呼び出されるか特定することができません(0 byte PUSH & CALL(indirect call)と表示されます)。

G.2　stk77の起動方法

G.2.1　入力書式

stk77を起動するためには、以下の図に示す書式に従って、必要な情報、パラメータを指定する必要があります。

●スタック使用量表示ファイルを直接指定する場合
% stk77△[起動オプション]△<スタック使用量表示ファイル名>

●マップファイルからスタック使用量表示ファイルを指定する場合
% stk77△[起動オプション]△-s<シンボルファイル名>

%：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。
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.3　stk77コマンドの入力書式

stk77を起動するために、必要なファイル、

1.スタック使用量表示ファイル（必要項目）
2.シンボルファイル（必要に応じて記述する項目）

は、nc77の起動オプションに、

●スタック使用量表示ファイル（拡張子.stk）の出力 ..... -fSSUオプション
●リロケータブルオブジェクトファイル（拡張子.r77）の出力 ..... -c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stk77に以下のオプションを指定しています。
●算出結果表示ファイル（拡張子.siz）の出力 ..... -oオプション
●計算開始関数 ..... -eオプション
●ライブラリ関数用スタック使用量表示ファイル ..... -lオプション

```
%nc77 -c -fSSU sample.c<RET>
7700 NC77 COMPILER V.5.10
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
and MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

sample.c


%
%stk77 -etimer_a0int -o sample.stk -lnc77lib.stk<RET>
NC77 STACK UTILITY stk77 for 7700 V.1.10.01
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

          292 bytes

%
```

<RET>：リターンキーの入力を示します。
※1.stk77の計算開始関数名は、timer_a0intです。
※2.ライブラリ関数用スタック使用量表示ファイル名は、nc77lib.stkです。

図G.4　stk77コマンドの入力例

G.2.2　オプションリファレンス

stk77を起動するためには、以下の図に示す書式に従って、必要な情報、パラメータを指定する必要があります。

【表G.1】にstk77の起動オプションを示します。

表G.1　stk77の起動オプション

オプション	機　能
-e<*関数名*>	スタック使用量の計算を開始する関数名を指定します。この起動オプションを省略した場合は、main関数からのスタックサイズを計算します。
-s<*シンボルファイル名*>	シンボルファイル名を指定します。
-o	スタックサイズと関数の呼び出し関係の表示を算出結果表示ファイル(拡張子.siz)に出力します。
-c	関数の呼び出し関係の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。
-l<ファイル名>	ライブラリ関数用のスタック使用量表示ファイルを指定します。

-e *関数名*

関数の指定

機　能：　スタック使用量の計算を開始する関数名を指定します。この起動オプションを省略した場合は、main関数からのスタック使用量を計算します。

書　式：　stk77△-e*関数名*△[起動オプション]△<スタック使用量表示ファイル名>

実行例：

```
% stk77 -efunc1 sample.stk
NC77 STACK UTILITY stk77 for 7700 V.1.10.XX
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

            514 bytes

%
```

注意事項：　main関数を含まないソースファイルに対するスタック使用量表示ファイルを指定するときは、必ず先頭の関数名を指定してください。

-s*シンボルファイル名*

シンボルファイルの指定

機　能：　スタックサイズを算出するソースファイルを含むシンボルファイルを指定します。これにより、スタック使用量表示ファイルの入力を省略することができます。

書　式：　stk77△[起動オプション]△-s<シンボルファイル名>

実行例：

```
% stk77 -ssample.sym
NC77 STACK UTILITY stk77 for 7700 V.1.10.XX
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

            514 bytes

%
```

注意事項：　1.シンボルファイルの数は、1つのみ指定できます。
2.オプション-sを指定する時には、nc77コンパイル時にリンカの起動オプション"-link77△-s"を指定してシンボルファイルを作成してください。

-O

算出結果表示ファイルの生成

機　能：　スタックサイズと関数の呼び出し関係の表示を、算出結果表示ファイル(拡張子.siz)に出力します。

実行例：

```
% stk77 -o func1 sample.stk
NC77 STACK UTILITY stk77 for 7700 V.1.10.XX
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

            514 bytes

%
```

-C

関数呼び出し関係の表示

機　能：　関数の呼び出し関係の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。

実行例：

```
%stk77 -c sample.stk
NC77 STACK UTILITY stk77 for 7700 V.1.10.XX
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

            514 bytes

*** C Flow ***

main(sample.stk)
        func1(sample.stk)
                func2(sample.stk)
%
```

-l*ライブラリ関数用スタック使用量表示ファイル名*

ライブラリ関数用スタック使用量表示ファイルの指定

機　能：　ライブラリ関数用のスタック使用量表示ファイルを指定します。

書　式：

stk77△[起動オプション]△-l<ライブラリ関数用スタック使用量表示ファイル名>

実行例：

```
% stk77 -lnc77lib.stk sample.stk
NC77 STACK UTILITY stk77 for 7700 V.1.10.XX
Copyright 1999 MITSUBISHI ELECTRIC CORPORATION
AND MITSUBISHI ELECTRIC SEMICONDUCTOR SYSTEMS CORPORATION
All Rights Reserved.

*** Stack Size ***

            514 bytes

%
```

注意事項：
1. ライブラリファイルに含まれる関数を使用する場合（nc77コンパイル時に-lオプションでライブラリファイルを指定する場合）には、必ずライブラリ関数用のスタック使用量計算ファイルを作成してください。
2. NC77ではライブラリファイル(nc77lib.LIB)に対応するスタック使用量計算ファイルを用意しています。ライブラリファイル(nc77lib.LIB)を使用したときのスタック使用量を計算する際には、-lオプションで"-lnc77lib.stk"を指定してください。

G.3　制限事項

stk77では、スタックサイズの計算中に【表G.2】に示す関数呼び出しに対してスタックサイズを算出することができません。これらの関数呼び出しがある場合、【表G.2】に示すメッセージを画面及び算出結果表示ファイルに出力します。この場合に表示されるスタックサイズは、計算できる範囲の最大値(【表G.2】中のXXbytes)を表示します。

表G.2　関数呼び出し関係とメッセージ

関数呼び出し関係	メッセージ
プログラム中に再帰呼び出しがある場合	XX bytes ＋∗'関数名'
プログラム中に間接呼び出しがある場合	XX bytes ＋ Indirect Call
プログラムを構成している関数の情報が入力ファイル中に存在しない場合 ※main関数又は-eオプションで指定した関数がない場合は0 bytes ＋ '関数名'となります。	XX bytes ＋ '関数名'

NC77ではアセンブラ関数に対してスタック使用量表示ファイルを生成することはできません。したがって、プログラム中にアセンブラ関数がある場合、別途スタックサイズを算出して、スタック使用量表示ファイルを作成してください。

また、asm関数を用いて呼び出される関数により使用されるスタックをstk77を用いて計算することはできません。

G.4　stk の使用例

G.4.1　ユーザースタックの計算

スタックサイズ算出ユーティリィティstk77にスタック使用量表示ファイルを処理させることにより、スタックの使用量を求めることができます。【図G.5】にstk77の入力例を、【図G.6】に算出結果表示ファイル例示します。

```
% stk77 -o smp.siz -lnc77lib.stk<RET>

 %                    ：プロンプトを示します。
  smp.stk             ：スタック使用量表示ファイル名です。
-lnc77lib.stk  ：オプション"-l"でライブラリ用スタック使用量表示ファイル
                 nc77lib.stkを指定しています。
```

図G.5　STK77コマンドの入力例

```
*** Stack Size ***

          84 bytes      ←スタックサイズを示しています。

*** C Flow ***

main(smp.stk)           ←C言語関数の呼び出し関係を示しています。
        func1(smp.stk)
        func2(smp.stk)
                _i4mod(C:\lib77\nc77lib.stk)
                _i4div(C:\lib77\nc77lib.stk)
```

図G.6　算出結果表示ファイル(smp.siz)

上記方法により算出したスタックサイズをスタートアップファイルに設定します。スタックサイズの設定例を【図G.7】に示します。

```
;------------------------------------------------------------------
; STACK SIZE definition
;------------------------------------------------------------------
STACKSIZE       .equ    54h
```

図G.7　ユーザースタックサイズの設定例

G.4.2　割り込み関数使用時のスタックサイズの計算

通常stk77はmain関数を基点にして、再帰的に各関数を追跡して最大スタックサイズを求めます。このため割り込み関数や間接呼び出しの関数等が使用するスタックの量は別途求める必要があります。

割り込み関数が使用するスタックサイズを求める方法を以下に解説します。

```
#pragma INTERRUPT func3     /* func3が割り込み関数であることの宣言    */

void far  main();
int       func1(int, int);
int       func2(int, long, int);
void      func3(void);
int       func4(int);

int s = 0;
int ss = 0;

void far main()             /* 関数main */
{
    int   i, j, k;          /* 自動変数 6バイト(使用) */

    k = func1(i, k);
    k = func2(i, j, k);
}
                :
                :
                (省略)
                :
                :

void func3( void )          /* 割り込み関数func3 */
{
    s = func4(ss);
}

int func4(int a)                  /*関数func4 */
{
    a++;
    return a;
}
```

図G.8 C言語サンプルプログラム(smp2.c)

a.　stk77を用いたスタック使用量の計算方法

スタックサイズ算出ユーティリティstk77は、任意の関数からスタック使用量を求めることができます。【図G.8】のサンプルプログラムの割り込み関数はfunc3ですので、関数func3からスタック使用量を求める必要があります。関数func3からスタック使用量を求めるには、stk77のオプション"-e"によりfunc3を指定してください。【図G.9】にstk77の入力例を、【図G.10】に算出結果表示ファイル例を示します。

```
%stk77 -c -o -efunc3 smp2.stk

 %        ：プロンプトを示します。
 smp2.stk ：スタック使用量表示ファイル名です。
```

図G.9　stk77のコマンド実行例

```
** Stack Size ***

           23 bytes


*** C Flow ***

func3(smp2.stk)
        func4(smp2.stk)
```

図G.10　算出結果表示ファイル(smp2.stk)

上記方法により算出した割り込み関数が使用するスタックサイズを、main関数から求めた総スタックサイズに追加した値をスタートアップファイルに設定します。

注）多重割り込みが発生する可能性がある場合は、多重割り込み時に使用する関数の必要とするスタックサイズを追加してください。

G.5　stk77のエラーメッセージ

G.5.1　エラーメッセージ

【表G.3】にスタックサイズ算出ユーティリティstk77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　stk77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
Usage : stk77 [option...] filename... <ret>	●コマンド入力の書式が間違っています。 ⇒コマンドの入力書式を確認して再入力してください。
not enough memory.	●ホストマシンの使用可能なメモリが不足しています。 ⇒不要なドライバを削除する等、使用できるメモリを増やしてください。
target file not found.	●該当するファイルが見つかりません。 ⇒処理に必要なスタック使用量表示ファイルが存在するか、確認してください。
invalid file format	●ファイルフォーマットに誤りがあります。 ⇒ファイルのフォーマットが正しいかを確認してください。

G.5.2　ワーニングメッセージ

【表G.4】にスタックサイズ算出ユーティリティstk77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　stk77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
cannot open 'ファイル名'.	●該当するファイルをオープンできません。 ⇒ファイルを確認してください。
cannot close 'ファイル名'.	●該当するファイルをクローズできません。 ⇒ファイルを確認してください。
invalid option 'xxx'.	●オプションの指定に誤りがあります。 ⇒正しくオプションを入力してください。
Ignore option 'xxx'.	●stk77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入力してください。

付録H

IEEE-695アブソリュート形式ファイルコンバータ

IEEE-695アブソリュート形式ファイルコンバータs2ieの起動方法と起動オプションの機能を説明します。

H.1　s2ieの概要

IEEE-695アブソリュート形式ファイルコンバータs2ieは、以下のファイルを結合してIEEE-695形式のデバッグ情報ファイル(拡張子.ie)を作成します。三菱製デバッガ及びシミュレータデバッガで、auto変数や構造体等のC言語情報を参照する際にはIEEE-695アブソリュート形式ファイルが必要です。

●s2ieが結合するファイル

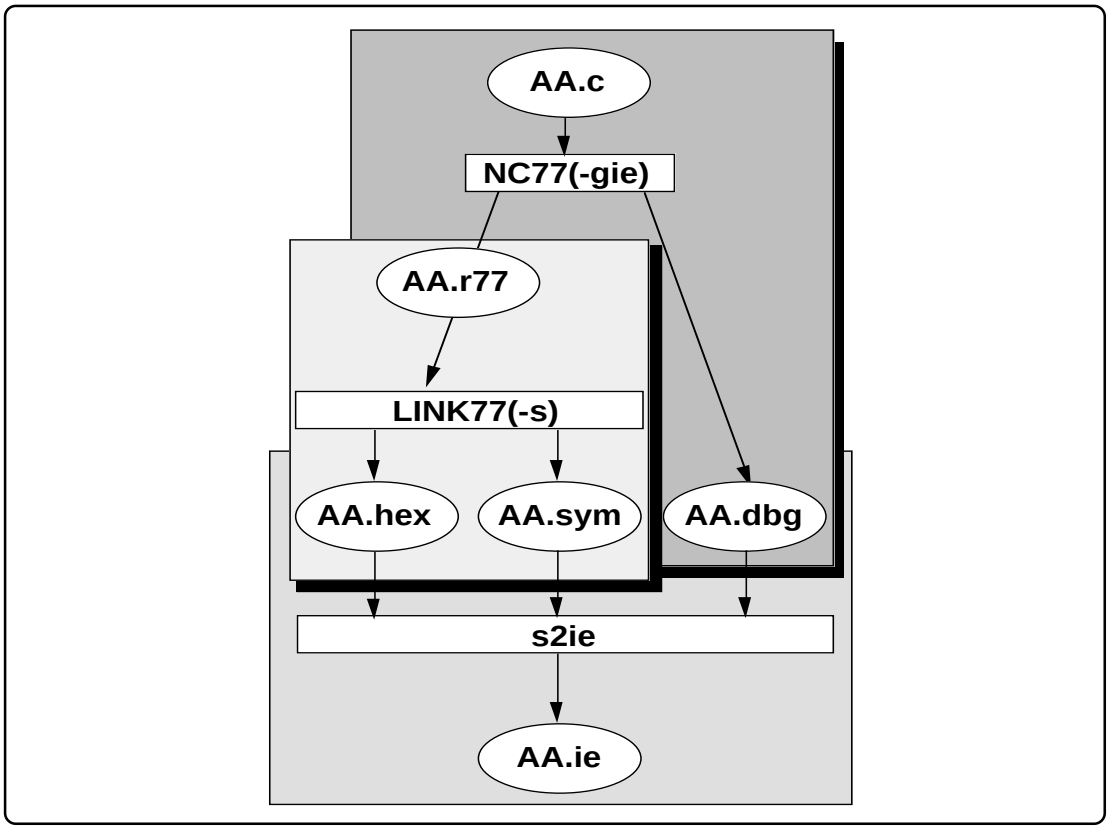
1.C言語デバッグ情報ファイル(コンパイルオプション-gieで作成されます：拡張子.dbg)

2.シンボルファイル(リンクオプション-sで作成されます：拡張子.sym)

3.16進機械語ファイル(拡張子.hex)

s2ieはコンパイルオプション-gieを指定することにより、コンパイラから自動的に起動されます。※1

s2ieの処理フローを【図H.1】に示します。



図H.1　s2ieの処理フロー

※1.s2ieはコンパイルの最終段階(link77動作後)に起動されます。したがってオプション-E、-P、-S、-cにより動作を途中で停止した場合は、s2ieは起動されません。

H.2　s2ieの起動方法

H.2.1　入力書式

s2ieを起動するためには、以下の図に示す書式に従って必要な情報、パラメータを指定する必要があります。

```
% s2ie△[起動オプション]△<シンボルファイル名>

%：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。
複数の起動オプションを記述する場合はスペースで区切ってください。
```

図H.2　s2ieコマンドの入力書式

H.2.2　オプションリファレンス

【表H.1】にs2ieの起動オプションを示します。

表H.1　s2ieの起動オプション

オプション	機　能
-V(大文字)	起動メッセージ(バージョン)を表示します。.ieファイルは作成しません。
-NLS	アセンブラローカルシンボルをIEEE-695ファイル中に出力しません。 ※これはNC77 V.3.20互換のための機能です。通常は指定する必要はありません。
-o出力ファイル名	出力先のファイル名を指定します。 ※本オプションを指定しない場合は、「シンボルファイル名.ie」の名称でIEEE-695ファイルが作成されます。
-Wstdout	エラーメッセージを標準出力に出力します。

H.3　注意事項

s2ieはコンパイラが生成するC言語デバッグ情報ファイル、及びリンカが生成するシンボルファイルを結合します。このためmakeプログラム等により部分的に再コンパイルした場合、C言語デバッグ情報とシンボル情報の整合性が取れなくなることがあります。

s2ieが処理中にエラーを発生する場合は上記可能性が高いため、全ソースファイルを再コンパイル・再リンクしてください。

H.4　s2ie の使用例

H.4.1　コンパイルドライバnc77から制御する例

s2ieは、コンパイルオプション-gieにより自動的に起動されます。コンパイル終了後には16進機械語ファイルと同名の、拡張子.ieのファイルが作成されます。

```
% nc77 -gie ncrt0.a77 sample.c<RET>
          :
       (省略)
          :
%ls
    ncrt0.a77    ncrt0.hex    section.inc    test.dbg
    ncrt0.ie     ncrt0.sym    test.c
```

図H.3　コンパイルドライバnc77から制御する例

H.4.2　s2ieを単独で起動する例

makeプログラム等を使用する場合はコンパイラ・リンカを個別に動作させるため、s2ieを直接起動する必要があります。

```
% nc77 -c -g ncrt0.a77<RET>
          :
       (省略)
          :
% nc77 -c -gie sample.c<RET>
          :
       (省略)
          :
% link77 ncrt0.r77 test.r77 , , ,-s<RET>
          :
       (省略)
          :
% s2ie ncrt0.sym
```

図H.4　s2ieを直接起動する例

H.5　s2ieのエラーメッセージ

H.5.1　エラーメッセージ

【表H.3】にIEEE-695アブソリュート形式ファイルコンバータs2ie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H.3　s2ieエラーメッセージ一覧表

番号	エラーメッセージ	エラー内容と対策
100	Ignore symbol file name 'ファイル名'	●該当するファイル名が不正です。 ⇒ファイル名を確認してください。
101	Can't open symbol file 'ファイル名'	●該当するファイルが存在しません。 ⇒ファイルを確認してください。
102	Can't read symbol file 'ファイル名'	●該当するファイルが読めません。 ⇒ファイルを確認してください。
103	Can't seek symbol file 'ファイル名'	●該当するファイルがシークできません。 ⇒ファイルを確認してください。
104	Can't malloc	●メモリが確保できません。 ⇒メモリを拡張してください。
105	The file has no data 'ファイル名'	●該当するファイルにデータが存在しません。 ⇒ファイルを確認してください。
200	Illegal symbol file format.	●シンボルファイルのフォーマットが不正です。 ⇒シンボルファイルを再生成してください。
201	Illegal symbol file SECTION format.	
202	Illegal symbol file FUNCTION format.	
203	Illegal symbol file LOCAL LABEL format.	
204	Illegal symbol file GLOBAL LABEL format.	
205	Illegal symbol file SOURCEformat.	
206	Illegal symbol file LANGUAGE format.	
207	I found ._end before ._Func.	
208	I found new SCOPE out of functions	●デバッグファイルのフォーマットが不正です。 ⇒再コンパイルしてください。
209	I found unknown type '関数型'	
210	Illegal index number 'インデックス番号'	
211	I don't know this type '型番号	
212	Illegal function variable '変数型'	
300	Can't open IEEE file	●IEEEフォーマットファイル(属性.ie)が生成できません。 ⇒ファイルを確認してください。

H.5.2　ワーニングメッセージ

【表H.4】にIEEE-695アブソリュート形式ファイルコンバータs2ie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　s2ie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
500	Can't file address 'シンボル名'	●シンボル名に対応したアドレスが決定できません。 ⇒再コンパイルしてください。
501	Can't open .dbg file 'シンボル名'	●デバッグ情報ファイル(属性.dbg)が存在しません。 ⇒再コンパイルしてください。

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

NC77 V.5.20ユーザーズマニュアル
第1版：1999年11月1日発行
資料番号：MSD-NC77-U-991101

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

77XX シリーズ用　C コンパイラ
NC77 V.5.20　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