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

■本ドキュメントに記載されているURLは、以下のとおり読み替えをお願いいたします。
http://www.necel.com/
http://www2.renesas.com/
↓
開発環境トップページ　http://japan.renesas.com/tools
ダウンロードポータル　http://japan.renesas.com/tool_download

■技術問合せについては、以下のページをご覧ください。
http://japan.renesas.com/tech_inquiry

■ツールユーザ登録については、以下のページをご覧ください。
http://japan.renesas.com/myrenesas

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。

ユーザーズ・マニュアル

CC78K0S

Cコンパイラ　Ver.1.30以上

言語編

対象デバイス
78K/0Sシリーズ

資料番号　U14872JJ1V0UM00（第1版）
発行年月　November 2000 N　CP(K)

〔メ　モ〕

目　次　要　約

Windows**および**WindowsNT**は，米国**Microsoft Corporation**の米国およびその他の国における登録商標または商標です。**

PC/AT**は，米国**IBM Corp.**の商標です。**

i386**は，米国**Intel Corporation**の商標です。**

UNIX**は，**X/Open**カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。**

SPARCstation**は，米国**SPARC International, Inc.**の商標です。**

SunOS, Solaris**は，米国サン・マイクロシステムズ社の商標です。**

HP9000**シリーズ**700, HP-UX**は，米国**Hewlett-Packard Corp.**の商標です。**

- **本資料の内容は予告なく変更することがありますので，最新のものであることをご確認の上ご使用ください。**
- **文書による当社の承諾なしに本資料の転載複製を禁じます。**
- **本資料に記載された製品の使用もしくは本資料に記載の情報の使用に際して，当社は当社もしくは第三者の知的財産権その他の権利に対する保証または実施権の許諾を行うものではありません。上記使用に起因する第三者所有の権利にかかわる問題が発生した場合，当社はその責を負うものではありませんのでご了承ください。**
- **本資料に記載された回路，ソフトウエア，及びこれらに付随する情報は，半導体製品の動作例，応用例を説明するためのものです。従って，これら回路・ソフトウエア・情報をお客様の機器に使用される場合には，お客様の責任において機器設計をしてください。これらの使用に起因するお客様もしくは第三者の損害に対して，当社は一切その責を負いません。**

M7A 98.8

巻末にアンケート・コーナを設けております。このドキュメントに対するご意見をお気軽にお寄せください。

は じ め に

CC78K0S Cコンパイラ（以下本Cコンパイラとする）は，Draft Proposed American National Standard for Information Systemsー Programming Language C（December 7, 1988）中の2. ENVIRONMENTと3. LANGUAGEを元に作成されています。したがって，ANSIに準拠しているCソース・プログラムであれば，本Cコンパイラによってコンパイルすることにより，78K/0Sシリーズ応用製品の開発が可能となります。

CC78K0S Cコンパイラ　言語編（本マニュアル）は，本Cコンパイラを用いてソフトウエアの開発を行われる方に，本Cコンパイラの基本機能と，言語仕様を理解していただくことを目的として書かれています。

本マニュアルでは，本Cコンパイラの操作に関する解説はいたしません。したがって，本マニュアルをご理解されたあと，本Cコンパイラをお使いの際には，CC78K0S Cコンパイラ　操作編（U14871J）をお読みください。

また，78K/0Sシリーズのアーキテクチャについては，78K/0Sシリーズの各ユーザーズ・マニュアルを参照してください。

【ターゲット・デバイス】

本Cコンパイラでは，78K/0Sシリーズのマイクロコンピュータのソフトウエア開発が可能です。各ターゲット・デバイスに対応させるためには，ターゲットの種類に応じたデバイス・ファイル（別売）が必要となります。

【対 象 者】

本マニュアルは，開発対象となるマイクロコンピュータのユーザーズ・マニュアルを一読された方で，ソフトウエア・プログラミングの経験がある方を対象とします。Cコンパイラ やC言語の知識は特に必要ありませんが，ソフトウエアに関する用語は理解されているものとして説明します。

【構　　成】

本マニュアルの構成を次に示します。

第1章　概　　説

Cコンパイラの一般的な機能と，本Cコンパイラの性能および特徴を説明します。

第2章　C言語の基本構成

C言語プログラムの構成と，その構成要素を説明します。

第3章　型，記憶域クラスの宣言

C言語で使用される型とその宣言，および記憶クラスについて説明します。

第4章　型の変換

本Cコンパイラによって自動的に行われる型の変換について説明します。

第5章　演算子と式

C言語で使用可能な演算子の記述方法や優先度を説明します。

第6章　C言語の制御構造

C言語の制御構造を，制御の流れや使用方法をあげて説明します。

第7章　構造体と共用体

構造体と共用体の概念と使用方法を説明します。

第8章　外部定義

外部定義の種類とその使用方法を説明します。

第9章　前処理指令

前処理指令の種類と使用方法を説明します。

第10章　ライブラリ関数

C言語におけるライブラリ関数の使い方と個々のライブラリ関数を説明します。

第11章　拡張機能

ターゲット・デバイスを活用するための拡張機能を説明します。

第12章　アセンブラとの相互参照

C言語プログラムからアセンブラ・プログラムを呼び出す方法などを説明します。

第13章　効率の良いコンパイラの活用法

本Cコンパイラを効率良く使用するための手段を説明します。

付　　録

saddr領域のレーベル一覧，セグメント名一覧，ランタイム・ライブラリ一覧，ライブラリ消費スタック一覧，索引があります。

【読 み 方】

本書の読み方を説明します。

■Cコンパイラおよび C言語初心者の方

Cコンパイラおよび C言語について初心者の方は，第1章から順を追ってご覧ください。本マニュアルでは，C言語プログラムの制御構造から拡張機能にいたるまでを順を追って説明しています。なお，**第1章　概　説**ではCソース・プログラム例を使い，本マニュアルの参照箇所を示していますのであわせてご利用ください。

■Cコンパイラおよび C言語経験者の方

本Cコンパイラの言語仕様は，ANSIに準拠しています。したがって，Cコンパイラおよび C言語経験者の方は，本Cコンパイラ特有の機能を示した**第11章　拡張機能**からお読みください。なお，**第11章　拡張機能**をお読みになる場合には，ターゲット・デバイスである78K/0Sシリーズ付属のユーザーズ・マニュアルを参照してください。

【関連資料】

本マニュアルに関連する資料（ユーザーズ・マニュアル）を紹介します。

資 料 名	資料番号
CC78K0S Cコンパイラ　操作編　ユーザーズ・マニュアル	U14871J

【参考資料】

「Draft Proposed American National Standard for Information Systems - Programming Language C（December 7, 1988）」

【用　　語】

RTOS = 78K/0シリーズ用 リアルタイムOS　RX78K0

【凡　　例】

本マニュアル中で共通に使用される記号などの意味を示します。

…	：同一の形式を繰り返す
“　”	：“　”で囲まれた文字そのもの
‘　’	：‘　’で囲まれた文字そのもの
⋮	：プログラム記述の省略形
／	：区切り記号
＼	：バック・スラッシュ
[　]	：かっこ内は省略可能

〔メ　モ〕

目　　次

図の目次

図番号　　タイトル，ページ

表の目次（1/2）

表の目次（2/2）

〔メ　モ〕

第1章　概　　説

CC78K0Sシリーズ　Cコンパイラは，78K/0SシリーズのC言語またはANSI-Cで記述されたソース・プログラムを機械語に変換する言語処理プログラムです。CC78K0Sシリーズ　Cコンパイラにより，78K/0Sシリーズのオブジェクト・ファイルまたはアセンブラ・ソース・ファイルが得られます。

1.1　C言語とアセンブリ言語

マイクロコンピュータに仕事をさせるには，プログラムやデータが必要です。これを人間がプログラミングして，マイクロコンピュータのメモリ部に記憶させます。マイクロコンピュータが扱えるプログラムやデータは2進数の集まりで，これを機械語といいます。

この機械語に英語の略記号を1対1で対応させたものがアセンブリ言語です。アセンブリ言語は，機械語と1対1で対応しているためコンピュータに対して詳細な指示を与えられます（たとえば，入出力時の処理速度の向上など）。しかし，このことはコンピュータのあらゆる動作を1つ1つ指示しなければならないことを意味しています。そのためにプログラムの論理構造が，一目見ただけでは理解しにくく，またエラーなども発生しやすいものです。

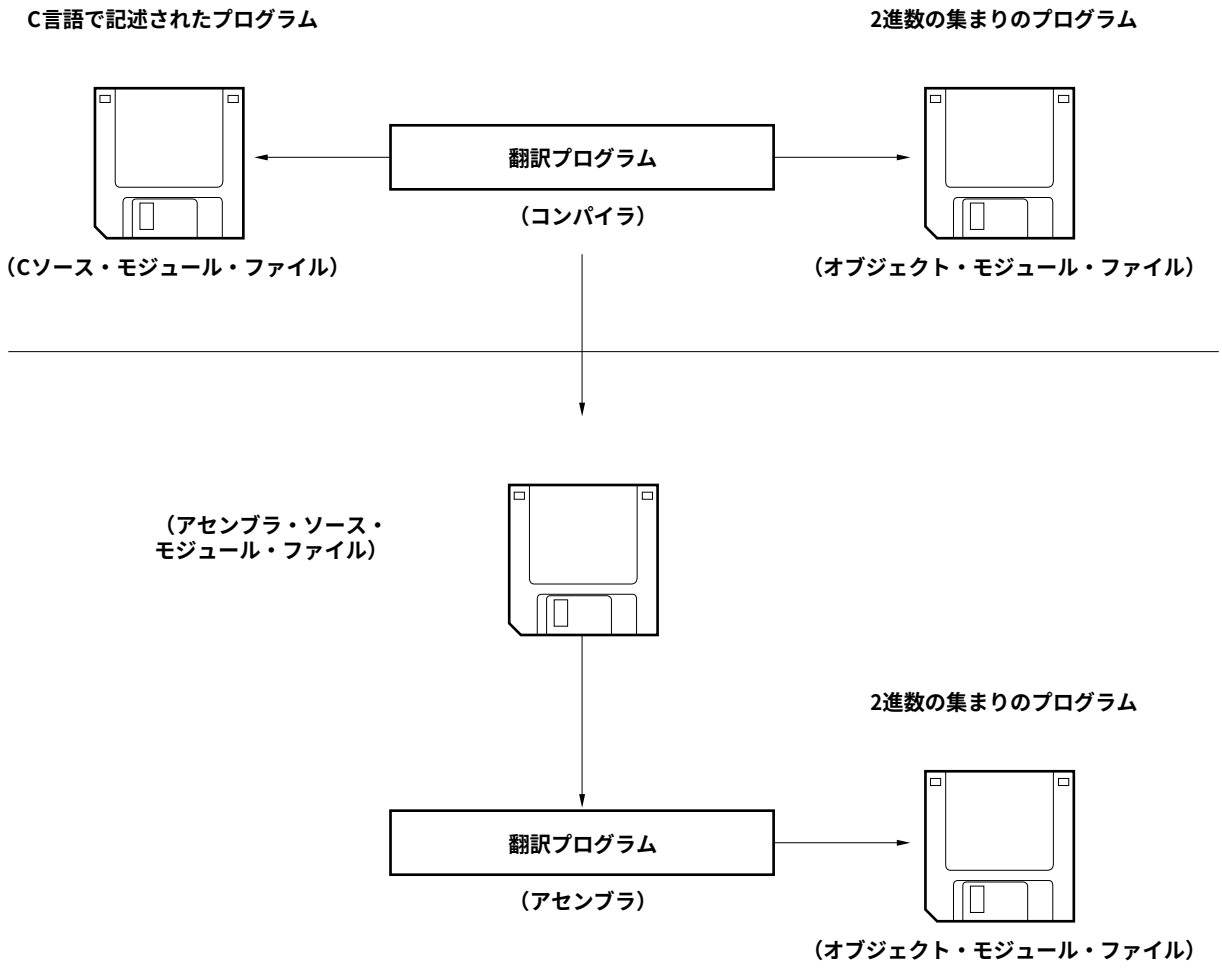
このようなアセンブリ言語に代わるものとして高級言語が開発されました。その中の1つにC言語があります。これによりプログラマは，コンピュータのアーキテクチャを気にせずにプログラミングでき，プログラム自体もアセンブリ言語に比べて論理構造などが理解しやすくなったといえます。

また，C言語ではプログラムを作成するための多くの部品（関数）が用意されているので，プログラマはこれらを組み合わせてプログラムを作成できます。

C言語は，人間にとって理解しやすいという特徴を持っています。しかし，C言語で書かれたプログラムのままでは，マイクロコンピュータは理解できません。C言語を理解させるには，それに相当する機械語に翻訳するプログラムが必要となります。このC言語を機械語に翻訳する翻訳プログラムをCコンパイラと呼びます。

本Cコンパイラは，Cソース・モジュールを入力しオブジェクト・モジュールとアセンブラ・ソース・モジュールを出力します。したがって，プログラマはC言語を用いてプログラムを作成し，プログラムの実行の細部まで指示したい場合にはアセンブリ言語でプログラムを修正できます。本Cコンパイラの翻訳の流れを，**図1－1 コンパイルの流れ**に示します。

図1－1　コンパイルの流れ



1.2 Cコンパイラによる開発手順

Cコンパイラによる製品開発には，Cコンパイラによって生成されたオブジェクト・モジュール・ファイルを連結するためのリンカや，ライブラリ・ファイルの作成を行うライブラリアン，また，プログラムのバグ取りのためのディバッガが必要になります。

本Cコンパイラに関連して必要となるソフトウエアを次に示します。

・エディタ……ソース・モジュール・ファイルの作成
・RA78K0Sアセンブラ・パッケージ

アセンブラ ……	アセンブラ・ソース・モジュール・ファイルのアセンブル
リンカ ……	オブジェクト・モジュール・ファイルの結合 リロケータブル・セグメントの配置アドレス決定
オブジェクト・コンバータ……	HEXファイルへの変換
ライブラリアン ……	ライブラリ・ファイルの作成

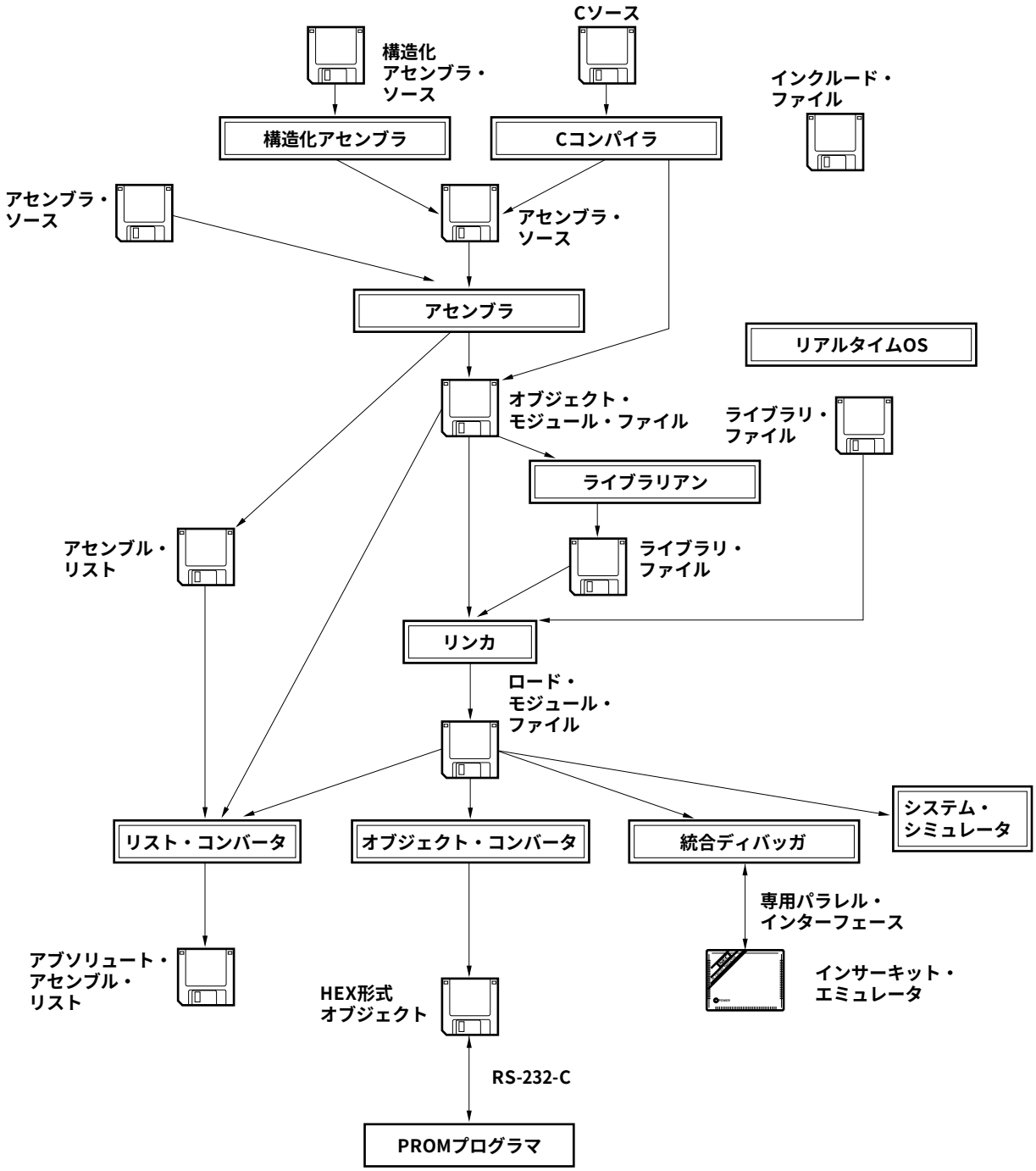
・ソース・ディバッガ（78K/0S用）……Cソース・モジュール・ファイルのバグ取り

Cコンパイラによる製品開発手順は次のようになります。

① 製品の機能分けを行う
② 機能ごとにCソース・モジュールを作成する
③ 各モジュールをコンパイルする
④ 使用頻度の高いモジュールをライブラリ化する
⑤ 各モジュールをリンクする
⑥ モジュールのディバグを行う
⑦ オブジェクト・コンバータによりHEXファイルに変換する

本Cコンパイラは，Cソース・モジュール・ファイルをコンパイルしてオブジェクト・モジュール・ファイルまたはアセンブラ・ソース・モジュール・ファイルを生成します。生成されたアセンブラ・ソース・モジュール・ファイルにより手作業による最適化（ハンド・オプティマイズ）が行え，効率のよいモジュールを作成できます。特に，高速な処理を必要とする場合，またはモジュールをコンパクトにしたい場合などに有効です。

図1－2　本Cコンパイラを含めたプログラム開発手順

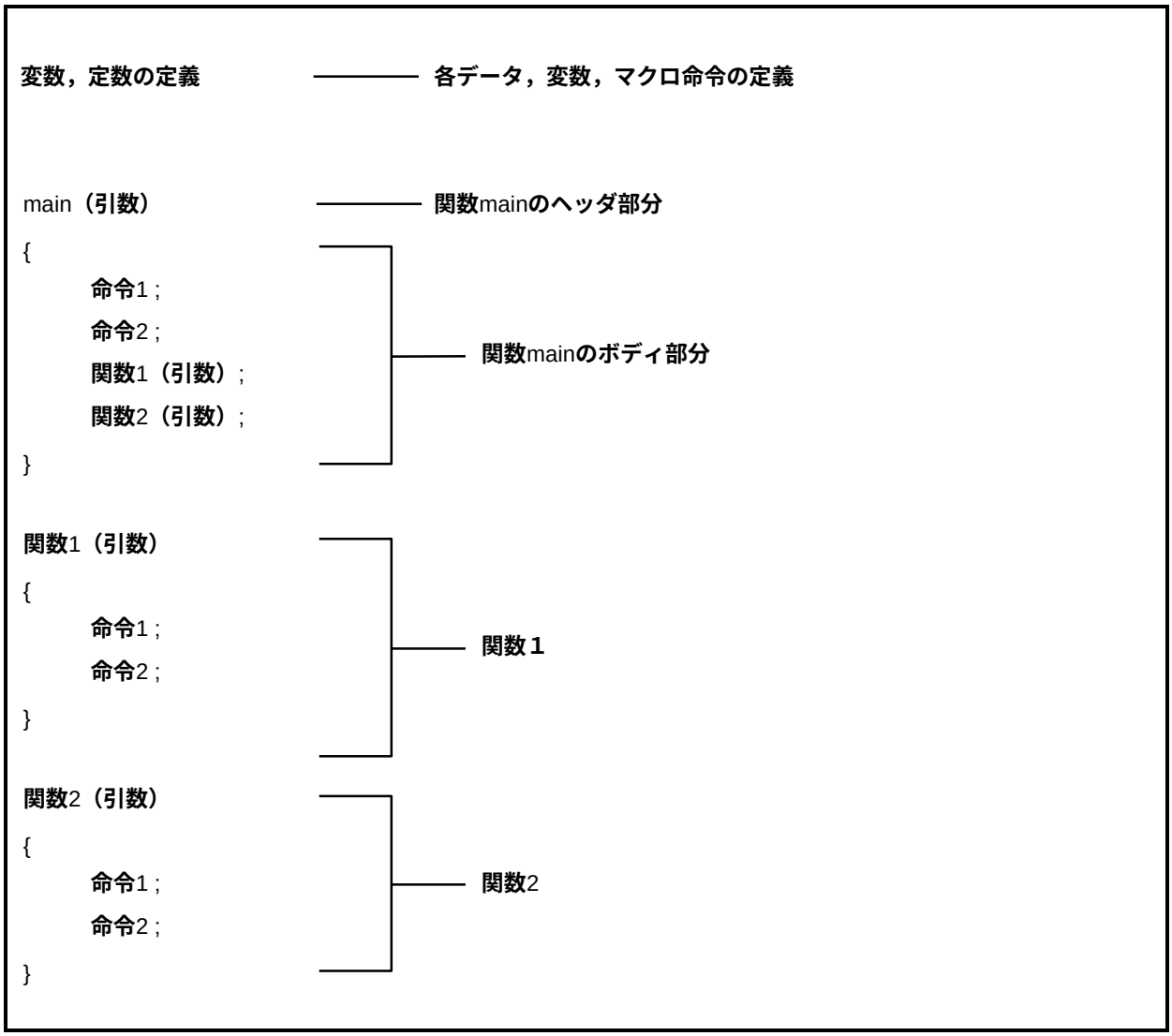


1. 3　Cソース・プログラムの基本構成

1. 3. 1　プログラム形式

C言語のプログラムは，関数の集まりです。関数は，それぞれ独立した機能を持つように作成します。そして，関数‘ main ’によって1つのプログラムにまとめます。C言語のメイン・ルーチンは，関数‘ main ’になります。

関数は，関数名と引数を定義するヘッダ部分と，プログラムの本体を示すボディ部分からなります。次にC言語のプログラム形式を示します。



実際のCソース・プログラムでは，次のようになります。

```
#define TRUE   1
#define FALSE  0
#define SIZE   200

void printf(char*, int);
void putchar(char);

char    mark[SIZE+1];
main()
{
   int  i, prime, k, count;

   count = 0;

   for(i = 0;i <= SIZE;i++)
      mark[i] = TRUE;

   for(i = 0;i <= SIZE;i++){
      if(mark[i]){
               prime = i + i + 3;
               printf("%6d", prime);

               count++ ;
               if((count%8) == 0) putchar('¥n') ;
               for(k = i + prime ; k <= SIZE ; k += prime)
                   mark[k] = FALSE;
      }
   }
   printf("¥n%d primes found.",count) ;------xxx(xxx) ;
}
```

#define xxx xxx ···· **前処理指令（マクロ定義）⑥**

xxx xxxx(xxx,xxx)···· **関数プロトタイプ宣言 ⑦**

char　xxx ·····**型宣言 ①　外部定義⑤**

xx [xx] ·····**演算子 ②**

int xxx·····**型宣言 ①**

xx = xx ·····**演算子 ②**

for (xx;xx;xx) xxx; ····**制御構造 ③**

xxx = xxx + xxx + xxx·····**演算子②**

xxx(xxx) ;·····**演算子②**

if(xxx)xxx;···**制御構造③**

演算子②

```
 void printf(char *s, int i)
 {
    int  j ;
    char *ss ;

    j = i ;
    ss = s ;
}

 void putchar(char c)
{
    char d ;
    d = c ;
 }
```

① 型，記憶クラスの宣言

オブジェクトを示す識別子の型，および記憶クラスの宣言です。型，記憶クラスの詳細については，**第3章　型，記憶域クラスの宣言**を参照してください。

② 演算子，式

算術演算，論理演算，代入などを行います。演算子と式の詳細については，**第5章　演算子と式**を参照してください。

③ 制御構造

プログラムの流れを指定します。C言語の制御構造には，選択，繰り返し，分岐それぞれ数個の命令が用意されています。制御構造の詳細については，**第6章　C言語の制御構造**を参照してください。

④ 構造体，共用体

構造体，または共用体を宣言します。構造体は，異なる型の連続した領域を持つオブジェクトで，共用体は異なる型の重なり合う領域を持つオブジェクトです。構造体と共用体の詳細については，**第7章　構造体と共用体**を参照してください。

⑤ 外部定義

関数，または外部オブジェクトを定義します。関数は，C言語プログラムを機能別に分けたときの1つの要素です。C言語のプログラムは，関数の集まりによって構成されます。外部定義の詳細については，**第8章　外部定義**を参照してください。

⑥ 前処理指令

コンパイラに対する命令です。‘ #define ’は，Cコンパイラに対してプログラム中に第1オペランドと同じものが現れたら第2オペランドに置き換えることを指令します。前処理指令の詳細は，**第9章　前処理指令（コンパイラに対する指令）**を参照してください。

⑦ 関数プロトタイプ宣言

関数の戻り値と引数の型を宣言します。

1. 4　プログラム開発をはじめる前に

実際にプログラム開発をはじめる前に，次のことを頭に入れておかなければなりません。

表1－1　本Cコンパイラの最大性能

項　番	項　　目	制 限 値
1	複文，繰り返し制御文，選択制御文のネスト	45
2	条件コンパイルのネスト	255
3	1つの宣言中の1つの算術型，構造体型，共用体型または不完全型を修飾するポインタ，配列及び関数宣言子（の任意の組み合わせ）の個数	12
4	式中のかっこのネスト	32
5	マクロ名で意味を持つ文字数	256
6	内部，外部シンボル名で意味を持つ文字数	249
7	1ソース・モジュール・ファイル中のシンボル数	1024注1
8	1ブロックでブロック・スコープを持つシンボル数	255注1
9	1ソース・モジュール・ファイル中のマクロ数	10000注2
10	関数定義，関数呼び出しのパラメータ	39
11	1つのマクロ定義，マクロ呼び出しのパラメータ	31
12	1つの論理ソース行の文字数	2048
13	結合後の文字列リテラル内の文字数	509
14	1つのオブジェクト・サイズ（データを示します）	65535バイト
15	＃includeのネスト	8
16	switch文のcaseレーベル数	257
17	1コンパイル単位のソース行数	約30000
18	テンポラリ・ファイルを作成せずにコンパイルできるソース行数	約300
19	関数コールのネスト	40
20	1関数内のレーベル数	33
21	1オブジェクト・モジュールあたりのコード，データ，スタック・セグメントのトータル・サイズ	65535バイト
22	1つの構造体，または共用体のメンバ数	256
23	1つの列挙の列挙定数の数	255
24	1つの構造体，共用体における構造体，または共用体のネスト	15
25	初期化子要素のネスト	15
26	1ソース・モジュール・ファイル中の関数定義数	1000
27	1つの完全宣言子におけるかっこで囲まれた宣言子の入れ子のレベル	591
28	マクロのネスト	200
29	-Iインクルード・ファイル・パス指定数	64

注1. テンポラリ・ファイルを使用せずに，メモリ・スペースのみで処理できる制限値を示します。メモリ・スペースで処理しきれない場合は，テンポラリ・ファイルを使用し，そのときの制限値はファイル・サイズにより変わります。

2. コンパイラの予約マクロ定義を含みます。

1.5 本Cコンパイラの特長

本Cコンパイラは，ANSIにないCPUのコードを生成する拡張機能を備えています。本Cコンパイラの拡張機能には，78K/0Sシリーズの特殊機能用レジスタをC言語レベルで記述可能にするものや，オブジェクト・コードを短縮し実行速度の向上を図るものがあります。拡張機能の詳細については，第11章　拡張機能を参照してください。

オブジェクト・コードを短縮し，実行速度を向上させる方法としては，次のものがあります。

- callt領域を利用して関数を呼び出す ····· callt / _ _callt関数
- 変数をレジスタに割り当てる····· レジスタ変数
- saddr領域に変数を割り当てる····· sreg / _ _sreg
- sfr名を使用できる ····· sfr領域
- 前後処理（スタック・フレーム）のない関数を生成する ····· noauto関数，
  norec / _ _leaf関数
- Cソース・プログラム中にアセンブリ言語を記述する ····· ASM文
- saddr，sfr領域へのビット・アクセスを行う ····· bit型変数，
  boolean/_ _boolean型変数
- ビット・フィールドを unsigned char 型で指定できる ····· ビット・フィールド宣言
- 乗算するコードを直接インライン展開して出力する····· 乗算関数
- 除算するコードを直接インライン展開して出力する····· 除算関数
- ローテートするコードを直接インライン展開して出力する····· ローテート関数
- メモリ空間の特定番地のアクセスを行う····· 絶対番地アクセス関数
- 特定のデータや命令を直接コード領域に埋め込む····· データ挿入関数
- 使用スタックの修正を呼ばれた関数側で行う····· _ _pascal関数

次に本コンパイラの拡張機能の概要を示します。各拡張機能の詳細は第11章を参照してください。

(1) callt / _ _callt関数

呼び出される関数のアドレスがcalltテーブル領域に置かれ，関数が呼び出されます。通常の呼び出し命令callに比べ，オブジェクト・コードを短縮できます。

(2) レジスタ変数

レジスタ，またはsaddr領域に変数をとることができ，通常の変数を使用した場合と比べ実行速度が向上します。また，オブジェクト・コードを短縮できます。

(3) saddr領域利用

変数をsaddr領域に割り当てることができ，通常の変数を使用した場合と比べ実行速度が向上します。また，オブジェクト・コードを短縮できます。変数はオプションによってもsaddr領域に割り当てることができます。

(4) sfr領域

特殊機能レジスタ（sfr）を，sfrの略号（sfr名）によってCソース・ファイル中で使用できます。

(5) noauto関数

前後処理（スタック・フレーム）のない関数を生成します。noauto関数の呼び出しで引数は，レジスタ渡しになります。これにより，実行速度が向上しオブジェクト・コードを短縮できます。この関数は，引数，オートマティック変数に制限があります。詳細は，11. 5 (5) noauto関数を参照してください。

(6) norec/ _ _leaf関数

前後処理（スタック・フレーム）のない関数を生成します。norec / _ _leaf関数の呼び出しで，引数はレジスタ渡しになります。また，norec / _ _leaf関数内で使用するオートマティック変数は，レジスタあるいはsaddr領域に割り当てられます。これにより，実行速度の向上およびオブジェクト・コードの短縮ができます。この関数は，引数，オートマティック変数に制限があります。また，この関数から関数呼び出しはできません。詳細は，11. 5 (6) norec関数を参照してください。

(7) bit型変数 / boolean / _ _boolean型変数

1ビットの記憶領域を持つ変数を生成します。bit型変数，boolean / _ _boolean型変数を使用することにより，saddr領域へのビット・アクセスができます。

なお，boolean / _ _boolean型変数は，機能，使用方法とも bit型変数と同じです。

(8) ASM文

Cコンパイラが出力したアセンブラ・ソース・ファイルにユーザが記述したアセンブラ・ソースが埋め込まれます。

(9) 漢　字

Cソース・ファイルのコメント文中に漢字を記述できます。漢字コードには，シフトJISコード，EUCコードを選択できます。また，漢字コードなしも選択できます。

(10) 割り込み関数

ベクタ・テーブルを生成し，割り込みに対応したオブジェクト・コードを出力します。これにより，Cソース・レベルで割り込み関数の記述が可能となります。

(11) 割り込み関数修飾子

この修飾子により，ベクタ・テーブルの設定と割り込み関数定義を別ファイルに記述できます。

(12) 割り込み機能

オブジェクトに割り込み禁止命令，割り込み許可命令を埋め込みます。

(13) CPU制御命令

オブジェクトに次の各命令を埋め込みます。

halt用の値をSTBCレジスタに設定する命令

stop用の値をSTBCレジスタに設定する命令

nop命令

(14) 絶対番地アクセス関数

オブジェクトに通常のメモリ空間をアクセスするコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。

(15) ビット・フィールド宣言

ビット・フィールドをunsigned char型で指定することにより，メモリの節約，オブジェクト・コードの短縮，実行速度の向上が図れます。

(16) コンパイラ出力セクション名の変更機能

コンパイラ出力セクション名を変更することにより，リンカでそのセクションを独立に配置できます。

(17) 2進定数の記述機能

Cソース中で，2進数を記述できます。

(18) モジュール名変更機能

Cソース中で，オブジェクト・モジュール名を任意に変更できます。

(19) ローテート関数

オブジェクトに式の値をローテートするコードを直接インライン展開して出力します。

(20) 乗算関数

オブジェクトに式の値を乗算するコードを直接インライン展開して出力します。この関数により，オブジェクト・コードを短縮し，実行速度を向上できます。

(21) 除算関数

オブジェクトに式の値を除算するコードを直接インライン展開して出力します。この関数により，オブジェクト・コードを短縮し，実行速度を向上できます。

(22) BCD演算関数

オブジェクトに式の値をBCD演算するコードを直接インライン展開して出力します。

BCD演算は，10進数1桁を2進数4ビットで表現するための演算です。

(23) データ挿入関数

カレント・アドレスに定数データを挿入します。アセンブラ記述を使用せずに，特定のデータや命令をコード領域に埋め込めます。

(24) スタティック・モデル

コンパイル時に-SMオプションを指定することにより，オブジェクト・コードの短縮，実行速度の向上，割り込み処理の高速化，メモリの節約が可能となります。

(25) 型 変 更

-ZIオプションや-ZLオプションを指定することにより，int型/short型をchar型とみなしたり，long型をint型とみなします。

(26) パスカル関数（_ _pascal）

関数呼び出し時に引数の積み込みによって使用したスタックの修正を関数呼び出し側では行わずに，呼ばれた関数側で行うことにより，関数呼び出し箇所が多い場合に，オブジェクト・コードを短縮できます。

(27) 関数呼び出しインタフェースの自動パスカル関数化

コンパイル時に-ZRオプションを指定することにより，norec/_ _interrupt/可変長引数の関数を除くすべての関数に対して_ _pascal属性を付加します。

(28) 引数／返り値のint拡張抑制方法

コンパイル時に-ZBオプションを指定することにより，オブジェクト・コードの短縮，実行速度の向上が図れます。

(29) 配列オフセット計算簡略化方法

コンパイル時に-QW2, -QW3, -QW4, -QW5オプションを指定することにより，オフセット計算コードが簡略化され，オブジェクト・コードの短縮，実行速度の向上が図れます。

(30) レジスタ直接参照関数

関数呼び出しと同様の形式でソース中に記述したり，モジュールの#pragma realregister指令によりレジスタ直接参照関数の使用を宣言することにより，C記述によるレジスタへのアクセスを簡単に行えます。

(31) メモリ操作関数

#pragma inline指令により，標準ライブラリ関数memcpy, memsetを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。これにより，実行速度の向上が図れます。

(32) 絶対番地配置指定

絶対番地に配置する変数を定義したいモジュール中で_ _directmap宣言を行うことにより，任意のアドレスに変数を配置でき，同じアドレスに複数の変数を重ねて配置できます。

(33) スタティック・モデル拡張仕様

コンパイル時に-ZMオプションを指定することにより，既存スタティック・モデルの制限事項が緩和され，記述性が向上します。

(34) テンポラリ変数

コンパイル時に-SM, -ZMオプションを指定し，引数とオートマティック変数に対して_ _temp宣言を行うことにより，引数，オートマティック変数領域を節約できます。また，引数，オートマティック変数の生存区間が明確に分かっていて，関数呼び出しの前後で値の一致が保証される必要がない変数に対して適用すると，メモリを節約できます。

(35) プロローグ／エピローグ対応ライブラリ

コンパイル時に-ZDオプションを指定することにより，プロローグ／エピローグ・コードがライブラリに置換され，オブジェクト・コードを短縮できます。

〔メ　モ〕

第2章　C言語の基本構成

この章では，Cソース・モジュール・ファイルの構成要素の説明を行います。Cソース・モジュール・ファイルは，次の字句から構成されます。

キーワード	識別子	定数
文字列リテラル	演算子	区切り子
ヘッダ名	前処理数	コメント

次に，Cプログラム記述例で使用されている字句を示します。

```
#include "expand.h"
extern void testb(void);                    extern ・・・・・・・・・・・・・・・・・キーワード
extern void chgb(void);
extern bit data1;                           data1, data2 ・・・・・・・・・・・・識別子
extern bit data2;

void main ( )                               void ・・・・・・・・・・・・・・・・・・キーワード
{
      data1 = 1 ;                           1 ・・・・・・・・・・・・・・・・・・・・定数
      data2 = 0 ;                           0 ・・・・・・・・・・・・・・・・・・・・定数

      while(data1){                         while ・・・・・・・・・・・・・・・・・キーワード
            data1 = data2 ;                 {}・・・・・・・・・・・・・・・・・・・・区切り子
            testb() ;                       = ・・・・・・・・・・・・・・・・・・・・演算子
      }
      if(data1 && data2){                   if・・・・・・・・・・・・・・・・・・・・キーワード
            chgb( ) ;                       && ・・・・・・・・・・・・・・・・・・・演算子
      }                                     ( )・・・・・・・・・・・・・・・・・・・・演算子
}

void lprintf(char *s, int i)                lprintf・・・・・・・・・・・・・・・・・識別子
{                                           char, int・・・・・・・・・・・・・・・・キーワード
      int   j ;                             s, I ・・・・・・・・・・・・・・・・・・識別子
      char   *ss ;                          * ・・・・・・・・・・・・・・・・・・・・演算子
      j = i ;
      ss = s ;
}
                    ⋮
```

2. 1　文字セット

(1) 文字集合

Cプログラムで使用する文字集合には，ソース・ファイルを記述するソース文字の集合と実行環境で解釈される実行文字の集合があります。

実行文字集合中の文字の値はJISコードです。

ソース文字集合および実行文字集合中では，次の文字を使用できます。

26個の英大文字

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

26個の英小文字

a b c d e f g h i j k l m

n o p q r s t u v w x y z

10個の10進数

0 1 2 3 4 5 6 7 8 9

29個の図形文字

! " # % & ' ( ) * + , - . / :

; < = > ? [¥] ^ _ { | } ~

およびスペース，水平タブ，垂直タブ，改ページなどを示す制御文字

備考　文字定数，文字列リテラルおよびコメント中では，この他の文字を使用できます。

(2) 多バイト文字

ソース文字集合は，拡張文字集合中（コメント等）で多バイト文字を使用できます。また，実行文字集合はシフトJIS漢字コードまたはEUC漢字コードの多バイト文字が使用できます。

(3) 英字拡張表記（エスケープ・シーケンス）

警報や改ページなどの非図形文字は，英字拡張表記によって表現します。英字拡張表記は，円記号‘￥’とアルファベット1文字からなります。

非図形文字を表現する英字拡張表記を次に示します。

表2－1　英字拡張表記一覧

英字拡張表記	意　味	文字コード
￥a	警報	07H
￥b	バックスペース	08H
￥f	改ページ	0CH
￥n	改行	0AH
￥r	復帰	0DH
￥t	水平タブ	09H
￥v	垂直タブ	0BH

(4) 3文字表記（トライグラフ・シーケンス）

次に示す左側の三つの文字の並び（“3文字表記”という）がソース・ファイル中にある場合，その3つの文字の並びを右側の対応する1文字に置き換えます。

表2－2　3文字表記一覧

3文字表記	意　味
??=	#
??(	[
??/	＼
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

2. 2　キーワード

(1) ANSI-C**キーワード**

次の字句は，コンパイラによってキーワードとして使用されるので，レーベルや変数名として使用できません。

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

(2) CC78K0S**用に追加されたキーワード**

本Cコンパイラでは，拡張機能を実現するために次の字句をキーワードとして追加しています。これらの字句もANSIキーワードと同様，レーベルや変数名として使用できません（大文字が含まれる場合は，キーワードとみなされません）。

ANSI-C言語仕様のみを許可するオプション（-ZA**）指定により，“_ _”で始まらないキーワードを無効にできます。**

callf, _ _callf, _ _banked1〜15, _ _rtos_interrupt, _ _interrupt_brk**は，**CC78K0**との互換性を考慮してキーワードとします。**

_ _callt /callt	callt**関数の宣言**
_ _callf / callf	callf**関数の宣言**
_ _sreg / sreg	sreg**変数の宣言**
noauto	noauto**関数の宣言**
_ _leaf / norec	norec**関数の宣言**
bit	bit**型変数の宣言**
_ _boolean / boolean	boolean**型変数の宣言**
_ _interrupt	**ハードウエア割り込み関数**
_ _interrupt_brk	**ソフトウエア割り込み関数**
_ _banked1〜15	**バンク関数**
_ _asm	asm**文**
_ _rtos_interrupt	RTOS**用割り込みハンドラ**
_ _pascal	**パスカル関数**
_ _directmap	**絶対番地配置指定**
_ _temp	**テンポラリ変数**
_ _mxcall	_ _mxcall**関数**[注]

注　MX**とのインタフェース用に予約しているキーワードです。ユーザは使用しないでください。**

2.3 識 別 子

識別子は，次のものを示します。

- 関数
- オブジェクト
- 構造体，共用体および列挙のタグ
- 構造体，共用体および列挙のメンバ
- typedef名
- レーベル名
- マクロ名
- マクロ仮引数

識別子は，アンダスコアを含めた英大文字と英小文字および数字で表します。識別子として使用できる文字を次に示します。

なお，識別子の最大の長さに関して制限はありません。ただし，本コンパイラで認識できるのは，最初の249文字です（表1－1　本Cコンパイラの最大性能を参照してください）。

_（アンダスコア）	a	b	c	d	e	f	g	h	i	j	k	l	m	
n	o	p	q	r	s	t	u	v	w	x	y	z		
A	B	C	D	E	F	G	H	I	J	K	L	M		
N	O	P	Q	R	S	T	U	V	W	X	Y	Z		
0	1	2	3	4	5	6	7	8	9					

識別子の先頭に数字を使用できません。また，識別子はキーワードと同じ名前にしてはなりません。

2. 3. 1　識別子の有効範囲

識別子は，宣言された場所によりその識別子が使用できる有効範囲が決まります。

識別子の有効範囲には，次のものがあります。

- 関数有効範囲
- ファイル有効範囲
- ブロック有効範囲
- 関数プロトタイプ有効範囲

```
extern _ _boolean  data1, data2 ;  ―――――  data1, data2 ･･･････ ファイル有効範囲
void testb(int x);                 ―――――  x ･･･････････････････ 関数プロトタイプ有効範囲
void main(void)
{
      int cot ;                    ―――――  cot ･････････････････ ブロック有効範囲
        data1 = 1 ;
        data2 = 0 ;

        while(data1){
                data1 = data2 ;
             j1 :                  ―――――  j1 ･･･････････････････ 関数有効範囲
                testb(cot) ;
        }
}

void testb(int x)                  ―――――  x ･･･････････････････ ブロック有効範囲
{
        ⋮
```

(1) 関数有効範囲

関数有効範囲は，関数内全体を指します。関数有効範囲を持つ識別子は，指定された関数内のどこからでも参照できます。

関数有効範囲を持つ識別子は，レーベル名だけです。

(2) ファイル有効範囲

ファイル有効範囲は，コンパイル単位全体を指します。

ブロックまたはパラメータ・リストの外で宣言された識別子は，ファイル有効範囲を持ちます。ファイル有効範囲を持つ識別子は，プログラム中のどこからでも参照できます。

(3) ブロック有効範囲

ブロック有効範囲は，対になったブロック（中かっこ‘ { } ’で囲まれたところ）を閉じるまでの範囲を示します。

ブロックまたはパラメータ・リストの中で宣言された識別子は，ブロック有効範囲を持ちます。ブロック有効範囲を持つ識別子は，指定したブロック内で有効です。

(4) 関数プロトタイプ有効範囲

宣言された関数の終わりまでの範囲を指します。

関数プロトタイプ内のパラメータ・リストの中で宣言された識別子は，関数プロトタイプ有効範囲を持ちます。関数プロトタイプ有効範囲を持つ識別子は，指定された関数内で有効です。

2. 3. 2　識別子の結合

異なった，または同一の有効範囲内で2回以上宣言された識別子が，同じオブジェクトあるいは関数として参照できるようになることを識別子の結合といいます。識別子は，結合されることにより同一のものであるとみなされます。

識別子の結合には，外部結合と内部結合および無結合があります。

(1) 外部結合

外部結合は，プログラム全体を構成するコンパイル単位およびライブラリの集まりで結合されるものです。

外部結合の例を次にあげます。

- 記憶クラスを指定せずに宣言された関数
- extern宣言されたオブジェクトあるいは関数で，参照する識別子に記憶クラスの指定がない場合
- ファイル有効範囲を持ち，記憶クラスの指定がないオブジェクト

(2) 内部結合

内部結合は，1つのコンパイル単位内で結合されるものです。

内部結合の例を次にあげます。

- ファイル有効範囲を持ち，記憶クラス指定子staticを含むオブジェクトまたは関数

(3) 無 結 合

無結合は，固有な実体です。

無結合の例を次にあげます。

- オブジェクト，あるいは関数以外の識別子
- 関数のパラメータを宣言する識別子
- ブロック内で記憶クラス指定子externを持たないオブジェクトの識別子

2.3.3　識別子の名前空間

すべての識別子は，次に示す‘ 名前空間 ’に分類されます。

・レーベル名‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥レーベルの宣言により区別されます。
・構造体，共用体，列挙のタグ名‥‥‥‥キーワードstruct，union，enumによって区別されます。
・構造体，共用体のメンバ名‥‥‥‥‥‥‥演算子‘ ． ’，‘ －＞ ’によって式中で区別されます。
・普通の識別子（上記以外の識別子）‥‥通常の宣言子，または列挙定数として宣言されます。

2.3.4　オブジェクトの記憶域期間

各オブジェクトは，そのライフタイムを決定する‘ 記憶域期間 ’を持っています。記憶域期間には，静的（static）記憶域期間と自動（automatic）記憶域期間の2つがあります。

（1）静的記憶域期間

静的記憶域期間を持つオブジェクトは，実行前に領域が確保されます。確保される領域は1回だけ初期化されます。静的オブジェクトは，プログラムの実行中存在して，最後に格納された値を保持します。

静的記憶域期間を持つオブジェクトを次に示します。

・外部結合を持つオブジェクト
・内部結合を持つオブジェクト
・記憶クラス指定子staticで宣言されたオブジェクト

（2）自動記憶域期間

自動記憶域期間を持つオブジェクトは，宣言されるブロック内に入るときにオブジェクトの領域が確保されます。ブロックに頭から入るときに，初期化の指定があるとオブジェクトの初期化が行われます。ブロック内のレーベルにジャンプして入った場合は，初期化されません。

自動記憶域期間を持つオブジェクトの領域は，宣言されたブロックの実行が終わると，保証されません。自動記憶域期間を持つオブジェクトを次に示します。

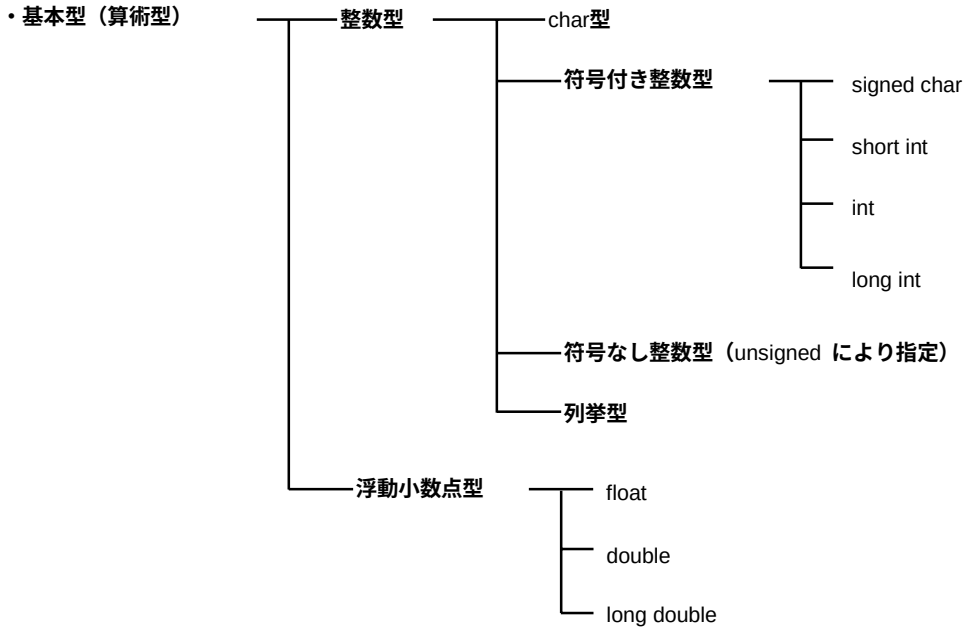
・無結合のオブジェクト
・記憶クラス指定子staticで宣言されていないオブジェクト

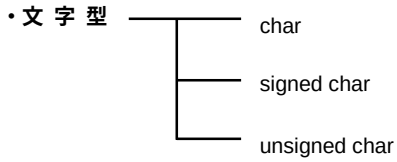
2.3.5　型

型は，オブジェクトに格納される値の性質を決定し，次の3種類に分けられます。

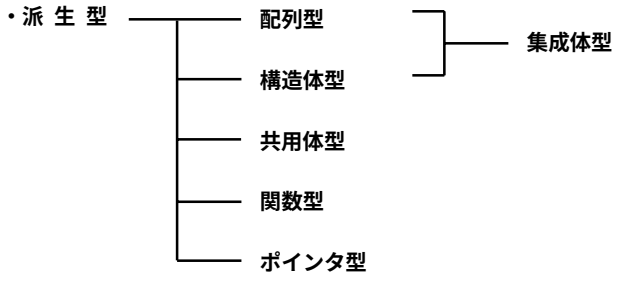
・オブジェクト型‥‥オブジェクトを表す型
・関数型‥‥‥‥‥‥‥関数を表す型
・不完全型‥‥‥‥‥‥サイズに関する情報を持たないオブジェクトを表す型

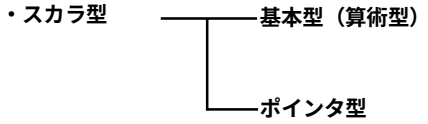
型の分類を次に示します。





・不完全型 ―― オブジェクトの大きさが確定しない配列や構造体，共用体とvoid型





(1) 基 本 型

基本型は，算術型とも呼ばれ，整数型と浮動小数点型からなります。また整数型は，char型，符号付き整数型，符号なし整数型，列挙型に分類されます。

(a) 整 数 型

整数型には次の4種類の型があります。整数型の値は2進数0と1によって表現されます。

- char型
- 符号付き整数型

・符号なし整数型
・列挙型

(i) char型

char型は，実行文字集合の任意の文字を格納するのに十分な大きさを持っています。charオブジェクトに格納される文字の値は，正になります。文字以外のものは，符号付き整数として扱われます。格納する際，あふれが生じるとあふれた部分は無視されます。

(ii) 符号付き整数型

符号付き整数型には，次の4種類の型があります。

- signed char
- short int
- int
- long int

signed char 型で宣言されるオブジェクトは，修飾子がないcharと同じ大きさの領域を持ちます。

修飾子がないintオブジェクトは，実行環境のCPUアーキテクチャにとって自然な大きさを持ちます。

符号付き整数型には，それに対応する符号なし整数型があり，ともに同じ大きさの領域を使用します。符号付き整数型の正の数は，符号なし整数型の部分集合です。

(iii) 符号なし整数型

符号なし整数型はキーワードunsignedで示されるものです。

符号なし整数型を含む計算ではオーバフローしません。符号なし整数型を含む計算の場合，整数型で表現できない値になると，計算結果は符号なし整数型で表現できる最大数に1を加算した値で割った余りに置き換わるからです。

(iv) 列 挙 型

列挙は，名付られた整数定数の集合です。列挙の並びにより，構成されます。

(b) 浮動小数点型

浮動小数点型には次の3種類の型があります。

- float
- double
- long double

なお，本コンパイラでは，double，long double型は，floatと同様にANSI / IEEE 754-1985で規定されている，単精度正規化数に対する浮動小数点表現としてサポートします。つまり，float，double，long double型の値の範囲は同じとなります。

表2－3　基本型一覧

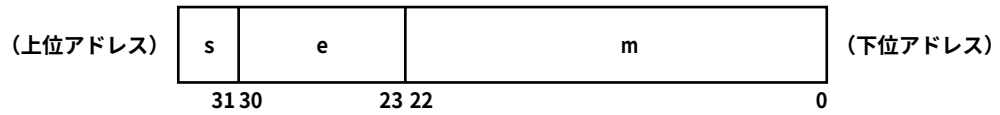
型	値の範囲
（signed） char	－128 ～ ＋127
unsigned char	0 ～ 255
（signed） short int	－32768 ～ ＋32767
unsigned short int	0 ～ 65535
（signed） int	－32768 ～ ＋32767
unsigned int	0 ～ 65535
（signed） long int	－2147483648 ～ ＋2147483647
unsigned long int	0 ～ 4294967295
float	1.17549435E－38F ～ 3.40282347E＋38F
double	1.17549435E－38F ～ 3.40282347E＋38F
long double	1.17549435E－38F ～ 3.40282347E＋38F

- signedは省略できます。ただしchar型の signed を省略した場合，コンパイル時の条件（オプション）によりsigned charまたはunsigned charと判断されます。
- short intとintは，同じ値の範囲を持ちますが，異なる型として扱われます。
- unsigned short intとunsigned intは，同じ値の範囲を持ちますが，異なる型として扱われます。
- float，double，long doubleは，同じ値の範囲を持ちますが，異なる型として扱われます。

(i) 浮動小数点数（float型）の仕様

- **フォーマット**

浮動小数点数のフォーマットを次に示します。



この形式の数値は，次のようになります。

（サイン部値）	（指数部値）
（－1）	＊（仮数部値）＊2

s ：サイン部（1ビット）

正数の場合0，負数の場合1をとります。

e ：指数部（8ビット）

底2の指数を1バイトの整数型（負の場合2の補数表現）で表し，この値にさらに7FHのバイアスを加えた値を用いています。これらの関係を，表2－4に示します。

表2－4　指数部の関係

指数部（16進）	指数部の値
FE	127
⋮	⋮
81	2
80	1
7F	0
7E	－1
⋮	⋮
01	－126

m：仮数部（23ビット）

仮数部は絶対値で表現され，仮数部のビット位置22-0が2進数の小数点第1位-第23位に相当します。仮数部値は浮動小数点値が0になる場合を除いて，常に1-2の範囲になるように指数部の値を調整します（正規化）。そのため1の位（1の値を意味する）は常に1となり，この形式では省略した形で表現しています。

・ゼロの表現

指数部 = 0，かつ仮数部 = 0のとき，次のように±0を表現します。

```
      （サイン部値）
(－1)          ＊0
```

・無限大の表現

指数部 = FFH，かつ仮数部 = 0のとき，次のように±∞を表現します。

```
      （サイン部値）
(－1)          ＊∞
```

・非正規化数

指数部 = 0，かつ仮数部 ≠ 0のとき，次のように非正規化数を表現します。

```
      （サイン部値）                －126
(－1)          ＊（仮数部値）＊2
```

備考　ここでの仮数部値は，1未満の数値であり，仮数部のビット位置22-0がそのまま小数点第1位-23位を表現します。

・非数（NaN）の表現

指数部 = FFH，かつ仮数部 ≠ 0のとき，サイン部にかかわらず非数を表現します。

・演算結果の丸め処理

最近偶数への不偏丸めを行います。演算結果が上記の浮動小数点のフォーマットで表現できない場合，表現可能な最も近い値に丸めます。

丸め以前の値に対して等差の表現可能な値が2つある場合，偶数（2進表現の最下位ビットが0となる数）に丸めます。

・演算例外

演算例外には，次の5種類があります。

表2－5　演算例外一覧

例　　外	返　　値
アンダフロー	非正規化数
消滅（INEXACT）	±0
オーバフロー	±∞
ゼロ除算	±∞
演算不能	非数（NaN）

各例外発生時の警告は，matherr関数を呼び出すことによって行います。

(2) 文 字 型

文字型には，次の3種類の型があります。

- char
- signed char
- unsigned char

(3) 不完全型

不完全型には，次の4つがあります。

- オブジェクトの大きさが確定しない配列
- 構造体
- 共用体
- void型

(4) 派 生 型

派生型には次の型があります。

- 配列型
- 構造体型
- 共用体型
- 関数型
- ポインタ型

(a) 集成体型

配列型と構造体型を総称して集成体型と呼びます。集成体型はメンバ・オブジェクトが連続して取られます。

(i) 配 列 型

配列型は，要素型と呼ばれるメンバ・オブジェクトの集まりを連続して割り付けることを示します。メンバ・オブジェクトは，すべて同じ大きさの領域を持ちます。要素型および配列の要素の個数を指定します。なお，不完全型の配列は作れません。

(ii) 構造体型

構造体型は，大きさの異なるメンバ・オブジェクトの集まりを連続して割り付けることを示します。個々のメンバ・オブジェクトは，名前によって指定できます。

(b) 共用体型

共用体型は重なり合うメンバ・オブジェクトの集まりを示します。個々のメンバ・オブジェクトは異なる大きさと名前を持ち，個別に指定できます。

(c) 関 数 型

関数型は，指定された型の返り値を持つ関数を示します。返り値の型，パラメータの数，およびパラメータの型を指定します。返り値の型がTであれば，その関数はTを返す関数と呼ばれます。

(d) ポインタ型

ポインタ型は，被参照型と呼ばれる関数型オブジェクト型，および不完全型から作られます。ポインタ型は，オブジェクトを表します。オブジェクトが示す値は，被参照型の実体を参照するために使用されます。

被参照型Tから作られるポインタ型は，Tへのポインタと呼ばれます。

(5) スカラ型

基本型（算術型）と，ポインタ型を総称してスカラ型といいます。スカラ型には，次のものがあります。

- char型
- 符号付き整数型
- 符号なし整数型
- 列挙型
- 浮動小数点型
- ポインタ型

2.3.6　適合型と合成型

(1) 適 合 型

2つの型が同じものであればそれは適合型と呼ばれます。たとえば，別々のコンパイル単位で宣言された2つの構造体，共用体または列挙型は，メンバ数，メンバ名が同じで，メンバの型が一致すれば適合型です。このとき2つの構造体，共用体は個々のメンバが同じ順序で並び，2つの列挙では個々のメンバは同じ値を持たなければなりません。

同じオブジェクト，または関数に関係するすべての宣言は，適合型を持たなければなりません。

(2) 合 成 型

合成型は，適合する2つの型から作られます。

合成型では次の事が成り立ちます。

- 片方が型の大きさが決まった配列であれば，その合成型は同じ大きさを持つ配列です。
- 片方だけがパラメータ型リスト（関数プロトタイプ）を持つ関数型であれば，その合成型はパラメータ型リストを持つ関数プロトタイプです。
- 両方の型がパラメータ型リストを持てば，合成パラメータ型リストのパラメータの型は対応するパラメータの合成型です。

【合成型の例】

ファイル有効範囲を持つ2つの宣言が次のようであると仮定します。

```
int f(int(*)(), double(*)[3]);
int f(int(*)(char *), double(*)[]);
```

このとき関数の合成型は，次のようになります。

```
int f(int(*)(char *), double(*)[3]);
```

2.4 定　　数

定数は，あらかじめ設定しておく値です。個々の定数は，指定した形式および値によって型が決定されます。定数には次の4種類があります。

- 浮動小数点定数
- 整数定数
- 列挙定数
- 文字定数

2.4.1 浮動小数点定数

浮動小数点定数は，有効数字部，指数部，浮動小数点接尾語から構成されます。

有効数字部　　：整数部，小数点，小数部
指数部　　：eまたはE，符号付き指数
浮動小数点接尾語：f / F（float）
　　l / L（long double）
　　省略時（double）

指数部の符号付き指数と浮動小数点接尾語は，省略できます。

なお，有効数字部は，整数部または小数部のいずれかがなくてはなりません。また，小数点または指数部のいずれかはなくてはなりません（例　1.23F，2e3）。

2.4.2 整数定数

整数定数は，数字ではじまり，小数点および指数部を持ちません。整数定数が符号なし（unsigned）であることを示すのに符号なし接尾語を，longであることを示すのに長語接尾語を整数定数の後ろに付けることができます。

整数定数には次の3種類があります。

- 10進定数　：0以外の数字から始まる10進数字
  10進数字 = 1 2 3 4 5 6 7 8 9
- 8進定数　：整数接尾語0 ＋ 8進数字
  8進数字 = 0 1 2 3 4 5 6 7
- 16進定数　：整数接尾語 0xまたは0X ＋16進数字
  16進数字 = 0 1 2 3 4 5 6 7 8 9
  a b C d e f A B C D E F

符号なし接尾語
u U
長語接尾語
l L

(1) 10進定数

10進定数は10を基数とする整数値です。指定方法は0以外の数字を先頭にして，その後ろに0-9の数字を続けます（**例**　56UL）。

(2) 8進定数

8進定数は8を基数とする整数値です。指定方法は0を先頭にして，その後ろに0-7の数字を続けます（**例** 034U）。

(3) 16進定数

16進定数は16を基数とする整数値です。指定方法は0xまたは0Xを先頭にして，その後ろに10進数字および10から15の値を表すa（または A）からf（または F）を続けます（**例**　0xF3）。

整数定数の型は，次に示す“表現できる型”の最初のものとみなされます。

本コンパイラでは，添字なしの定数の型は，コンパイル時の条件（オプション）により，charまたはunsigned charに変更できます。

（整数定数）	（表現できる型）
・接尾語なし10進数	int，long int，unsigned long int
・接尾語なし8進数，16進数	int，unsiged int，long int，unsigned long int
・u または U の接尾語付き	unsiged int，unsigned long int
・l または L の接尾語付き	long int，unsigned long int
・u または U の接尾語および l または L の接尾語付き	unsigned long int

2.4.3　列挙定数

列挙定数は，列挙型変数の要素，つまり列挙型変数の値を示すために使用します。列挙型変数は，識別子で示される特定の値のみを持てます。

列挙型（enum）は次の型の中ですべての列挙定数を表現できる最初のものとなります。列挙定数は，識別子で示します。

- signed char
- unsigned char
- signed int

記述方法は，‘ enum　列挙型 {列挙型定数の並び} ’となります。

例
```
enum months { January=1, February, March, April, May};
```

= と整数の指定がある場合は，列挙型変数はその整数値を持ち，その後ろに続く列挙型変数の値は，その値＋1の値を持ちます。上記の例では，それぞれ順に1, 2, 3, 4, 5の値を持ちます。‘ = 1 ’がない場合は，0, 1, 2, 3, 4の値を持ちます。

2.4.4　文字定数

文字定数は‘ X ’または‘ ab ’のようにシングル・クォートで囲まれる1文字以上の文字列です。

文字定数には，シングル・クォート’，バック・スラッシュ（￥あるいは＼），および改行文字（¥n）は含まれません。それらの文字を表す場合は，拡張表記（エスケープ・シーケンス）を使用します。拡張表記は次の3種類があります。

- 単純拡張表記　：　\’　\”　\?　\¥
  \a　\b　\f　\n　\r　\t　\v
- 8進拡張表記　：　\8進数字　[8進数字　8進数字]
  （例. \012, \0[注1]）
- 16進拡張表記　：　\x 16進数字
  （例. \xFF[注2]）

注1.　ナル文字を表します。

2.　本コンパイラでは，－1の値を表します。ただし，charをunsigned charとみなす条件（オプション）を付けた場合は，＋255の値を表します。

2.5　文字列リテラル

文字列リテラルは，“ｘｘｘ”のようにダブル・クォートで囲まれる0個以上の文字の並びです（例　“xyz”）。

シングル・クォート’は，それ自身また拡張表記＼’で表現します。また，ダブル・クォート”は，拡張表記＼”で表現します。

配列要素は，char型を持ち，与えられた文字で初期化されます（例　char array[ ] = “abc”;）。

2.6　演 算 子

演算子を次に示します。

```
[ ]  ( )  .  ->
++  --  &  *  +  -  ~  !  sizeof
/  %  <<  >>  <  >  <=  >=  ==  !=
^  |  &&  ||
? :
=  *=  /=  %=  +=  -=  <<=  >>=
&=  ^=  |=
, #  ##
```

‘[]’，‘()’および‘?:’演算子は必ずペアで使用します。またこの間には式を書くこともできます。

‘#’および‘##’は前処理指令のマクロ定義にのみ使用できます（記述方法については，第5章　演算子と式を参照してください）。

2.7　区切り子

区切り子は独立した文法または意味を持つ記号ですが，値の生成は行いません。
区切り子を次に示します。

```
[ ]  ( )  { }  *  ,:  =  ;  ...  #
```

区切り子‘[ ]’，‘( )’，‘{ }’は，間に式，宣言または文を書けます。ただし，これらは必ずペアで使用します。
区切り子‘#’は前処理指令だけに使用します。

2.8　ヘッダ名

ヘッダ名は，‘#include’前処理指令でのみ使用されます。ヘッダ名の #include指令により，外部ソース・ファイルとの対応付けが行われます。
ヘッダ名の #include指令の例を示します（#include各指令の詳細については，9.2　**ソース・ファイルの取り込み**を参照してください）。

```
#include  <ヘッダ名>
#include  “ ヘッダ名 “
```

2.9　コメント

コメントは，Cソース・モジュールに入れる注釈文のことです。コメント文は，先頭を“/*”で示し最後を“*/”で閉じます。
なお，本Cコンパイラはマルチバイト文字を識別でき，漢字を使用できます。オプションまたは環境変数により，漢字コードを指定できます。また，-ZPオプションにより，“//”以降改行までをコメント文として認識できます。

```
例      /* コメント文 */
        // コメント文
```

〔メ　モ〕

第3章　型，記憶域クラスの宣言

この章では，C言語で使用されるデータや関数の型と宣言，またその有効範囲について説明します。宣言とは，識別子または識別子の集まりに，解釈および属性を指定することです。識別子によって名付けられたオブジェクトや関数に対して，記憶域も確保する宣言は‘定義’といいます。

宣言の例を次に示します。

```
#define  TRUE   1
#define  FALSE  0
#define  SIZE   200
void main （void）
{
   auto int  i, prime, k ;                /* オートマティック変数の宣言 */
   for （i = 0 ; i <= SIZE ; i++）
      mark [i] = TRUE ;
          ⋮
```

宣言は，記憶域クラス指定子，型指定子，初期化宣言子などで構成されます。記憶域クラス指定子，型指定子は，結合，記憶の持続期間および宣言子が示す実体の型を指定します。初期化宣言子はコンマで区切られる宣言子で，各々の宣言子は付加的な型情報，初期化子またはその両方を持てます。

あるオブジェクトを表す識別子が無結合で宣言されている場合，そのオブジェクトの型はその宣言子の終わりまでに，または初期化子を持っていればその初期化宣言子の終わりまでに完全（サイズに関する情報を持つオブジェクト）になっていなければなりません。

3. 1　記憶域クラス指定子

記憶域クラス指定子は，オブジェクトの記憶域クラスを指定するものです。記憶域クラスは，オブジェクトが持つ値の格納場所や，オブジェクトのスコープ（有効範囲）を示します。1つの宣言中で，記憶域クラス指定子は1つまでしか記述できません。記憶域クラス指定子には，次の5つがあります。

- typedef
- extern
- static
- auto
- register

（1）typedef

typedef指定子は，指定した型に対する同義語を宣言します。

typedef指定子の詳細は，3. 6　typedefを参照してください。

（2）extern

extern指定子は，外部変数であることを示します。

（3）static

static 指定子は，オブジェクトが静的持続期間を持つことを示します。

静的持続期間を持つオブジェクトは，プログラムの実行前に領域を確保され，格納される値は1回だけ初期化されます。オブジェクトは，プログラムの実行中存在して，最後に格納された値を保持します。

（4）auto

auto指定子は，オブジェクトが動的持続期間を持つことを示します。

動的持続期間を持つオブジェクトは，実行時に領域が確保されます。ブロックに頭から入るときに，初期化の指定があるとオブジェクトの初期化が行われます。ブロック内のレーベルにジャンプして入った場合は，初期化されません。

動的持続期間を持つオブジェクトの領域は，宣言されたブロックの実行が終わると保証はされません。

（5）register

register指定子は，オブジェクトがCPUのレジスタに割り当てられることを示します。本Cコンパイラでは，レジスタ，saddr領域に割り当てられます。レジスタ変数の詳細については，第11章　拡張機能を参照してください。

3. 2　型指定子

型指定子は，オブジェクトの型を指定します。型指定子には，次のものがあります。

- void
- char
- short
- int
- long
- float
- double
- long double
- signed
- unsigned
- **構造体／共用体指定子**
- **列挙型指定子**
- typedef**名**

また，本Cコンパイラでは，次の型指定子が追加されています。

- bit / boolean / _ _boolean

各型指定子の意味と，本コンパイラで表現できる値の限界値（（）内の数値）について次に示します。本コンパイラは，浮動小数点演算についてIEEE Std 754 -1985の単精度のみをサポートするため，double，long doubleは，floatと同じフォーマットを持つものとします。

- void ････････････････････････････････････ 空の値の集合
- char ････････････････････････････････････ 基本文字セットを格納できる大きさ
- signed char ･･････････････････････････････ 符号付き整数（－128 ～ ＋127）
- unsigned char ････････････････････････････ 符号なし整数（0-255）
- short / signed short / short int / signed short int ････････････････････････ 符号付き整数（－32768 ～ ＋32767）
- unsigned short / unsigned short int ･･ 符号なし整数（0-65535）
- int / signed / signed int ･･････････････ 符号付き整数（－32768 ～ ＋32767）
- unsigned / unsigned int ････････････････ 符号なし整数（0-65535）
- long / signed long / long int / signed long int ････････････････････････ 符号付き整数（－2147483648 ～ ＋2147483647）
- unsigned long / unsigned long int･････ 符号なし整数（0-4294967295）
- float ･･･････････････････････････････････ 単精度の浮動小数点数（1.17549435E－38F ～ 3.40282347E＋38F）
- double･･･････････････････････････････････ 倍精度の浮動小数点数（1.17549435E－38F ～ 3.40282347E＋38F）
- long double･･･････････････････････････････ 拡張精度の浮動小数点数（1.17549435E－38F ～ 3.40282347E＋38F）
- 構造体／共用体指定子 ････････････････････････ メンバ・オブジェクトの集合
- 列挙指定子 ･････････････････････････････････ int型定数の集合
- typedef名 ･････････････････････････････････ 指定した型の同義語
- bit / boolean / _ _boolean ･･････････････ 1ビットで表現できる整数（0-1）

スラッシュで区切られた型指定子は，同じ大きさです。

3.2.1　構造体指定子と共用体指定子

構造体指定子と共用体指定子は，名前付きメンバ（オブジェクト）の集まりを示します。個々のメンバは，それぞれ異なった型を持てます。

(1) 構造体指定子

構造体指定子は，複数の異なった型の集まりを1つのオブジェクトとして宣言します。個々の型のオブジェクトはメンバと呼ばれ，それぞれに名前を付けられます。また，宣言された順に，メンバ用の連続した領域が確保されます。

ただし，78K/0Sシリーズは奇数番地からワード・データのリード／ライトができない制約があるため，デフォルトではコード・サイズを優先して，2バイト以上のメンバが偶数番地に配置されるようアライン・データを挿入します。したがって，メンバ間に関しては，アライン・データによる隙間が生じる場合があります。

-RCオプションを指定することによって，アライン・データの挿入を抑制し，構造体をパッキングできます。この場合，データ・サイズは小さくなりますが，奇数番地に配置された2バイト以上のメンバのリード／ライトが1バイト単位のリード／ライトのコードに展開されるため，コード・サイズが増加します。

構造体は，次のように宣言します。この宣言は，後ろに構造体変数の並びがないため，まだメモリ割り当てを行いません。構造体変数の定義については，**第7章　構造体と共用体**を参照してください。

```
struct 識別子 { メンバ宣言並び };
```

構造体宣言の例

```
struct tnode {
        int count ;
        struct tnode *left, *right ;
} ;
```

(2) 共用体指定子

共用体指定子は，複数の異なった型の集まりを1つのオブジェクトとして宣言します。

個々の型のオブジェクトはメンバと呼ばれ，それぞれに名前を付けられます。

共用体のメンバ用に確保された領域は，すべて重なり合った領域です。

共用体は，次のように宣言します。この宣言は，後ろに共用体変数の並びがないため，まだメモリ割り当てを行いません。共用体変数の定義については，**第7章　構造体と共用体**を参照してください。

```
union 識別子 { メンバ宣言並び };
```

共用体宣言の例

```
union u_tag {
      int var1 ;
      long var2 ;
} ;
```

メンバの型は，不完全型または関数型以外であればどのような型でもかまいません。また，メンバはビット数の指定付きで宣言できます。ビット数が指定されたメンバをビット・フィールドと呼びます。

また，本コンパイラでは，ビット・フィールド宣言に関する拡張機能を加えています。詳細は，11. 5 (15) **ビット・フィールド宣言**を参照してください。

(3) ビット・フィールド

ビット・フィールドは，指定したビット数からなる整数型の領域です。ビット・フィールドには，int型，unsigned int型およびsigned int型を指定できます[注1]。

修飾子のないintビット・フィールド，およびsigned intビット・フィールドの最上位ビットは，符号ビットとして判断されます[注2]。

ビット・フィールドが複数ある場合，同じメモリ単位中に十分な空きが残っていれば続くビット・フィールドは，隣接するビットに詰めて入れられます。幅が0で無名のビット・フィールドを置いた場合は，同じメモリ単位中に次のビット・フィールドは詰め込まれません。無名のビット・フィールドは宣言子を持たずコロンおよび幅のみで宣言します。

ビット・フィールド・オブジェクトに単項＆演算子（アドレス）は適用できません。

注1. 本コンパイラの場合は，char型，unsigned char型，signed char型も指定できます。
ただし，本コンパイラは，signed 型ビット・フィールドをサポートしていないので，すべてunsigned 型とみなします。

2. 本コンパイラでは，コンパイラ・オプション-RBにより，ビット・フィールドの割り付け方向を変更できます（詳細は，第11章　拡張機能を参照してください）。

ビット・フィールドの例を次に示します。

```
struct data {
        unsigned int a:2;
        unsigned int b:3;
        unsigned int c:1;
}no1 ;
```

3. 2. 2　列挙型指定子

列挙型指定子は，順番付けされるオブジェクトを示します。列挙型指定子によって宣言されるオブジェクトはint型を持つ定数として宣言されます。

列挙型指定子は，次のように宣言します。

```
enum 識別子　｛ 列挙子並び ｝
```

オブジェクトは，列挙子並びによって宣言します。オブジェクトは，宣言された順に先頭を0で，それに続くオブジェクトを順に1ずつ加えた値に定義します。また，‘ = ’によって定数値を指定できます。

次の例では‘ hue ’を列挙の識別子（タグ）にし，‘ col ’をこの型を持つオブジェクト，‘ cp ’をこの型を持つオブジェクトへのポインタとしています。この宣言で列挙された値は，“ ｛ 0, 1, 20, 21 ｝ ”になります。

```
enum hue {
        chartreuse,
        burgundy,
        claret=20,
        winedark
};

enum hue col, *cp;
void main (void) {
    col = claret;
    cp = &col;
    /*...*/ (*cp != burgundy) /*...*/
        ⋮
```

3.2.3 タ　グ

タグは，構造体／共用体および列挙型に付ける名前です。タグは宣言された型を持ち，タグにより同じ型のオブジェクトを宣言できます。

次の宣言の識別子がタグ名です。

```
構造体／共用体　識別子　{　メンバ宣言並び　}
または，
enum　識別子　{　列挙子並び　}
```

タグは，メンバ宣言並びによって定義される構造体／共用体，および列挙の内容を持ちます。一度タグを指定し，構造体／共用体および列挙を指定すると中かっこ“{ }”で囲まれた並びを省略できます。次からの宣言は，タグが持つ内容と同じ構造を持ちます。同じ有効範囲中のそれ以後の宣言は中かっこで囲まれる並びを省略しなければなりません

次の型指定子は，内容が未定義なので構造体または共用体は不完全型です。

```
構造体／共用体　識別子
```

この場合のタグは，オブジェクトの大きさが必要ないときにのみ使えます。これは，同じ有効範囲の中で，タグの内容を定義することにより型が完全になります。

次の例でタグ‘ tnode ’は，整数および2つの同じ型のオブジェクトへのポインタを含む構造体を指定しています。

```
struct tnode  {
            int count;
            struct tnode *left, *right;
};
```

次の例は‘ s ’をタグで示される型のオブジェクトとして宣言し，‘ sp ’をタグで示される型のオブジェクトへのポインタとして宣言します。この宣言により，式‘ sp->left ’は‘ sp ’が指すオブジェクトの左の‘ struct tnode ’へのポインタを指すことを示します。

また，式‘ s.right->count ’は，‘ s ’の右の‘ struct tnode ’のメンバである‘ count ’を指すことを示します。

```
typedef struct tnode TNODE;
struct tnode  {
          int count;
          struct tnode *left, *right;
};

TNODE s, *sp;
void main (void)  {
     sp->left = sp->right;
     s.right->count = 2;
}
```

3. 3　型修飾子

型修飾子には，‘ const ’と‘ volatile ’の2つがあります。これらは，左辺値に対してのみ影響します。

const修飾型で定義されたオブジェクトを，非const修飾型を持つ左辺値を使って変更できません。また，volatile修飾型で定義されたオブジェクトを，非volatile修飾型を持つ左辺値を使って参照できません。

volatile修飾型を持つオブジェクトは，コンパイラからは認識されない方法で変更でき，また他の認識されない副作用を持てます。したがって，このオブジェクトを参照する式はC言語で書かれたプログラムがどのように実行されるかを，順序規則に従って厳密に評価しなければなりません。さらに，そのオブジェクトに最後に格納された値は，未知の要因による変更点を除き，すべての副作用完了点で，プログラムによって定められたものと一致する必要があります。

配列型の指定に型修飾子がある場合，型修飾子は配列ではなく配列の要素を修飾します。関数型の指定に型修飾子を含めることはできませんが，2. 2　**キーワード**で示した本コンパイラ独自の型修飾子，callt, _ _callt, callf, _ _callf, noauto, norec, _ _leaf, _ _interrupt, _ _interrupt_brk, _ _rtos_interrupt, _ _pascalを型修飾子として含めることはできます。

なお，sreg, _ _sreg, _ _directmap, _ _tempも型修飾子です。

次の例で‘ real_time_clock ’はハードウエアによって変更できますが，代入や，インクリメント，デクリメントなどの操作はできません。

```
extern const volatile int real_time_clock;
```

型修飾子が集成体型を修飾する場合を次に示します。

```
const struct s {int mem;} cs = {1};
struct s ncs;       /* オブジェクトncsは変更可能である */
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}};   /* const intの配列の配列 */
int *pi;
const int *pci;

ncs = cs;           /* 正しい */
cs = ncs;           /* 代入演算子の変更可能な左辺値の制約に違反している */
pi = &ncs.mem;      /* 正しい */
pi = &cs.mem;       /* 代入演算子の型の制約に違反している */
pci = &cs.mem;      /* 正しい */
pi = a[0];          /* 正しくない：a[0]は“ const int * ”型を持つ */
```

3.4 宣 言 子

宣言子は，1つの識別子を宣言します。ここでは，特にポインタ宣言子，配列宣言子および関数宣言子を説明します。宣言子により，識別子の有効範囲，記憶域期間および型を持つ関数，またはオブジェクトが決まります。

次に各宣言子の記述方法を示します。

3.4.1 ポインタ宣言子

ポインタ宣言子は，宣言する識別子がポインタであることを示します。ポインタは，値が格納されているところをポイントする（指す）ものです。

ポインタ宣言は，次のように行います。

```
        *   型修飾子並び   識別子
```

この宣言により識別子は，型修飾子で修飾されたポインタになります。

次の2つの宣言は，‘ 定数値への可変ポインタ ’および‘ 変数値への不変ポインタ ’を示しています。

```
    const int *ptr_to_constant;
    int *const constant_ptr;
```

1行目の宣言は，ptr_to_constant が指す const intの内容は変更されてはならないが，ptr_to_constant自身は他のconst intを指すように変更されてもよいことを表しています。同様に2行目の宣言では，constant_ptrが指すintの内容は変更されてもよいが，constant_ptr自身は常に同じ位置を指します。

不変ポインタconstant_ptrの宣言は，‘ int型へのポインタ ’型に対する定義を含むことによって明確にできます。

次の例は，constant_ptrを‘ intへのconst修飾ポインタ ’型を持つオブジェクトとして宣言しています。

```
    typedef int *int_ptr;
    const int_ptr constant_ptr;
```

3.4.2　配列宣言子

配列宣言子は，宣言する識別子が配列型を持つオブジェクトであることを宣言します。

配列宣言は，次のように行います。

```
型　識別子 [ 定数式 ]
```

この宣言により識別子は，宣言された型の大きさを持つ配列になります。定数式の値が配列の要素数になります。定数式は，0より大きい値を持つ整数定数式です。配列の宣言において定数式の指定がないと，配列は不完全型になります。

次の例は，11の要素数からなるchar型の配列‘ a[ ] ’と，17の要素数からなるchar型のポインタの配列‘ ap[ ] ’を宣言しています。

```
char  a[11],  *ap[17];
```

次の例で，最初の宣言の x は，int型へのポインタであることを宣言しています。2番目の宣言では，y が他のところで宣言される大きさの指定のないint型の配列であることを宣言しています。

```
extern int *x;
extern int y[ ];
```

3.4.3　関数宣言子（プロトタイプ宣言を含む）

関数宣言子は，関数の返り値および引数の型を宣言します。

関数宣言は，次のように行います。

```
型　識別子（仮引数型並び または 識別子並び）
```

この宣言により，仮引数型並びで指定した型の仮引数を持ち，識別子の前に宣言された型の値を返す関数になります。関数の仮引数は，仮引数識別子並びによって指定します。これらの並びにより，引数を示す識別子とその型が決まります。ヘッダ・ファイル‘ stdarg.h ’で定義されているマクロは，省略記法（, ...）で書かれた並びを仮引数に変換します。また，仮引数がない関数は，仮引数型並びを‘ void ’にします。

3.5 型　　名

型名は，関数またはオブジェクトの大きさを示す型の名前です。文法的にみた型名は，関数またはオブジェクトに対する宣言から識別子を除いたものです。

次に型名の例をあげます。

- int ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ int型を指定します。
- int *・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ int型へのポインタを指定します。
- int *[3]・・・・・・・・・・・・・・・・・・・・・・・・・・・・ int型へのポインタを要素とする配列（要素数3）を指定します。
- int （*）[3] ・・・・・・・・・・・・・・・・・・・・・・・・ int型を要素とする配列（要素数3）へのポインタを指定します。
- int *（） ・・・・・・・・・・・・・・・・・・・・・・・・・・・ 仮引数指定を持たないint型へのポインタを返り値とする関数を指定します。
- int （*）（void）・・・・・・・・・・・・・・・・・・・・ 仮引数を持たず，int型を返り値とする関数へのポインタを指定します。
- int （*const [ ]）（unsigned int, ...） ・・ unsigned int型の仮引数，およびその他の不定個の仮引数を持ち，int型を返り値とする個々の関数への不変ポインタを要素とする配列（要素数不定）を指定します。

3. 6　typedef

typedefは，識別子が指定した型と同義語であることを定義します。定義された識別子がtypedef名となります。typedef名の定義は，次のように行います。

```
typedef　型　識別子；
```

次の例でdistanceはint型であり，metricpはパラメータ指定を持たずint型を返す関数へのポインタです。また，zの型は指定された構造体であり，zpはこの構造体へのポインタです。

```
typedef  int  MILES, KLICKSP () ;
typedef  struct {long re, im;} complex;
/*...*/
MILES  distance;
extern  KLICKSP *metricp;
complex  z, *zp;
```

次に示す例は，typedef名 tをsigned int型で，typedef名 plainをint型でそれぞれ宣言し，3つのビット・フィールド・メンバを持つ構造体を宣言しています。ビット・フィールド・メンバは，次のとおりです。

- 名前が tで，0-15の範囲の値をとるもの
- 名前がなくconst修飾された，（アクセスされたならば）－16 ～ ＋15の範囲の値をとるもの
- 名前が rで，－16 ～ ＋15の範囲の値をとるもの

```
typedef signed int t;
typedef int  plain;
struct tag {
        unsigned  t:4;
        const  t:5;
        plain  r:5;
};
```

この例で，1番目のビット・フィールド宣言はunsignedを型指定子とする（このため tを構造体メンバの名前となる）のに対し，2番目のビット・フィールド宣言はconstを型修飾子とする（typedef名として参照できるtを修飾する）という点で異なっています。この宣言のあとで，内側の有効範囲中で，

```
t f(t(t));
long t;
```

があれば，関数fは“仮引数を1つ持ち，signed intを返す関数”として宣言され，その仮引数は，“仮引数を1つ持ち，signed intを返す関数へのポインタ型”として宣言されています。また，識別子 t はlong型として宣言さ

れます。

typedef名は，プログラムを読みやすくするために使用できます。次に示すsignal関数の3つの宣言はすべて，1番目に定義されたtypedef名を使用しないものと同じ型を指定します。

```
typedef  void  fv(int);
typedef  void  (*pfv)(int);

void  (*signal(int, void(*)(int)))(int);
fv  *signal(int, fv *);
pfv  signal(int, pfv);
```

3.7 初 期 化

初期化は，オブジェクトに前もって値を設定することです。オブジェクトの初期化は，初期化子によって行います。

初期化は，次のように行います。

```
オブジェクト = { 初期化子並び };
```

初期化子並びには，初期化するオブジェクトの数だけ初期化子を指定します。

静的記憶域期間を持つオブジェクトと集成体／共用体型を持つオブジェクトに対する初期化子，または初期化子並び中のすべての式は定数式によって指定します。

識別子の宣言がブロック有効範囲を持ち，外部結合または内部結合を持つ場合，初期化できません。

(1) 静的記憶域期間を持つオブジェクトの初期化

静的記憶域期間を持つ算術型のオブジェクトの初期化を行わなかった場合は，暗黙的に0に初期化されます。同様に，静的記憶域期間を持つポインタ型のオブジェクトは，空ポインタ定数に初期化されます。

例

```
unsigned int  gval1;            /* 0で初期化される */
static int gval2;               /* 0で初期化される */
void func(void){
        static char  aval;      /* 0で初期化される */
}
```

(2) 自動記憶域期間を持つオブジェクトの初期化

自動記憶域期間を持つオブジェクトの初期化を行わなかった場合は，その値は不定となり保証されません。

例

```
void func(void){
        char  aval;             /* この時点では不定 */
          ⋮
        aval = 1;               /* 1に初期化 */
}
```

(3) 文字配列の初期化

文字配列は，文字列リテラル（" "で囲まれた文字列）によって初期化できます。同様に，文字列リテラルの連続した文字は配列の要素を初期化します。

次の例では，‘ 型修飾子なし ’の配列オブジェクト s，t が定義され，各配列の要素は文字列リテラルで初期化されます。

```
char s[ ] = "abc", t[3] = "abc";
```

次の例は，上に示した初期化と同じです。

```
char s[ ] = {'a', 'b', 'c', '¥0'},
     t[ ] = {'a', 'b', 'c'};
```

次の例は，pを“charへのポインタ”型として定義し，要素（メンバ）が文字列リテラルで初期化され，長さが4の“char配列”型オブジェクトを指すように初期化しています。

```
char *p = "abc";
```

(4) 集成体，共用体オブジェクトの初期化

・集成体

集成体型のオブジェクトの初期化は，添字の昇順またはメンバの順に書かれた初期化子の並びで行います。指定する初期化子の並びは中かっこで囲みます。

その並び中の初期化子が集成体のメンバ数より少ない場合，残りのメンバは静的記憶域期間を持つオブジェクトと同じように暗黙的に初期化されます。

大きさのわからない配列は，初期化子の数によって配列の要素数が決まり，不完全型でなくなります。

・共用体

共用体型オブジェクトの初期化は，共用体の最初のメンバに対する中かっこで囲んだ初期化子です。

次の例では，大きさがわからない配列xが初期化によって3つのメンバを持つint型の1次元配列となります。

```
int x [ ] = {1, 3, 5};
```

次の例は，中かっこで囲まれた初期化子を持つ完全な定義です。“{1, 3, 5}”は，配列オブジェクト‘ y[0] ’の第1行目‘ y[0][0] ’，‘ y[0][1] ’，‘ y[0][2] ’を初期化します。同様に，次の2行は‘ y[1] ’，‘ y[2] ’を初期化します。‘ y[3] ’の初期値は，指定されていないので0となります。配列の初期化は，次のように指定できます。

```
char y[4][3] = {
        {1, 3, 5},
        {2, 4, 6},
        {3, 5, 7},
};
```

次の例は，前にあげた例と同じ結果になります。

```
char z[4][3] = {
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

次の例は，zの第1列が指定した値に初期化され，残りの要素は0になります。

```
 char z[4][3] = {
        {1}, {2}, {3}, {4}
};
```

次の例は，3次元配列を初期化しています。

q[0][0][0]は1に，q[1][0][0]は2に，q[1][0][1]は3に初期化され，さらに，4，5および6はそれぞれq[2][0][0]，q[2][0][1]およびq[2][1][0]を初期化します。そして残りはすべて0になります。

```
  short q[4][3][2] = {
     {1},
     {2, 3},
     {4, 5, 6}
  };
```

次の例は，前にあげた3次元配列の初期化と同じ値に初期化されます。

```
  short q[4][3][2] = {
         1, 0, 0, 0, 0, 0,
         2, 3, 0, 0, 0, 0,
         4, 5, 6
  };
```

次の例は，前にあげた初期化を中かっこを使って完全にしたものです。

```
  short q[4][3][2] = {
         {
                  {1},
         },
         {
                  {2, 3},
         },
         {
                  {4, 5, 6},
         }
  };
```

第4章　型の変換

式中で，2つの演算数（オペランド）の型が違う場合，自動的に型変換が行われます。これは，キャスト演算子によって得られる変換と同じようなものです。この自動的な型変換は，暗黙の型変換といわれます。本章では，この暗黙の型変換について説明します。

型変換には，正常に変換されるものと，切り捨てまたは四捨五入されるもの，また符号が変わるものがあります。型変換の一覧表を**表**4－1　**型変換一覧**に示します。

表4－1　型変換一覧

変換前＼変換後		(signed) char	unsigned char	(signed) short int	unsigned short int	(signed) int	unsigned int	(signed) long int	unsigned long int	float	double	long double
(signed) char	＋	＼	○	○	○	○	○	○	○	○	○	○
	－	＼	N	○	N	○	N	○	N	○	○	○
unsigned char		△	＼	○	○	○	○	○	○	○	○	○
(signed) short int	＋			＼	○	＼	○	○	○	○	○	○
	－			＼	N	＼	N	○	N	○	○	○
unsigned short int				△	＼	△	＼	○	○	○	○	○
(signed) int	＋			＼	○	＼	○	○	○	○	○	○
	－			＼	N	＼	N	○	N	○	○	○
unsigned int				△	＼	△	＼	○	○	○	○	○
(signed) long int	＋							＼	○	○	○	○
	－							＼	N	○	○	○
unsigned long int								△	＼	○	○	○
float										＼	○	○
double											＼	＼
long double											＼	＼

備考1. signed**は省略できます。ただし，**char**型の場合にかぎり，コンパイル時の条件（オプション）によって**signed char**または**unsigned char**とみなされます。**

2. ○　**：正しく変換されます。**

＼　**：型変換は，行われません。**

N　**：正しい値になりません（符号なし整数とみなされます）。**

△　**：ビット・イメージ的には変わりませんが，正の数で表現しきれない場合は，正しい値になりません（符号付き整数とみなされます）。**

空欄：変換時にあふれた部分は，切り捨てられます。符号も変換後の型によって変わる場合もあります。

4.1 算術オペランド

(1) 文字型と整数型（汎整数拡張）

char，short int，intのビット・フィールドと，これらの符号付き，または符号なしのもの，または列挙型を持つオブジェクトは，いずれの場合も，int型で表現できる範囲内にあればint型に変換されます。int型で表現できない場合は，unsigned int型に変換されます。これらを“汎整数拡張”と呼びます。その他のすべての算術型は，汎整数拡張によって変わることはありません。汎整数拡張は，符号も含めて値を保持します。

本コンパイラでは，修飾子なしのcharを符号付きとして扱います。なお，オプションにより，unsigned charとして扱うこともできます。

(2) 符号付き整数型と符号なし整数型

整数型を持つ値が他の整数型に変換される場合，値が変換後の整数型で表現できればその値は変わりません。

符号付き整数がそれと等しいか，より大きいサイズを持つ符号なし整数に変換される場合，符号付き整数の値は負でなければ変わりません。符号付き整数の値が負で，符号なし整数が符号付き整数より大きいサイズを持つ場合は，まず，符号付き整数が符号なし整数と同じ大きさの符号付き整数に拡張され，次に，符号なし整数型で表現できる最大数に1を加えた値を足して変換前の符号付き整数の値が符号なしの値に変換されます。

整数型を持つ値がより小さいサイズを持つ符号なし整数に変換される場合，その結果は変換後の符号なし整数型で表現できる最大の符号なし数に1を加えた値で割った非負の余りとなります。整数型を持つ値がより小さいサイズを持つ符号付き整数に変換される場合，または符号なし整数が同じ大きさの符号付き整数に変換される場合，変換後の値を表現できなければ，あふれた部分は無視されます。変換パターンは，**表4－1　型変換一覧**を参照してください。

符号付き整数から符号なし整数への変換には，次の場合があります。

表4－2　符号付き整数から符号なし整数への変換

		unsigned	
		値の範囲小	値の範囲大
signed	＋	／	○
	－	／	＋

○：正しく変換されます。

＋：正の整数に変換されます。

／：変換される型の最大値に1を加えた値で割った余り（剰余）となります。

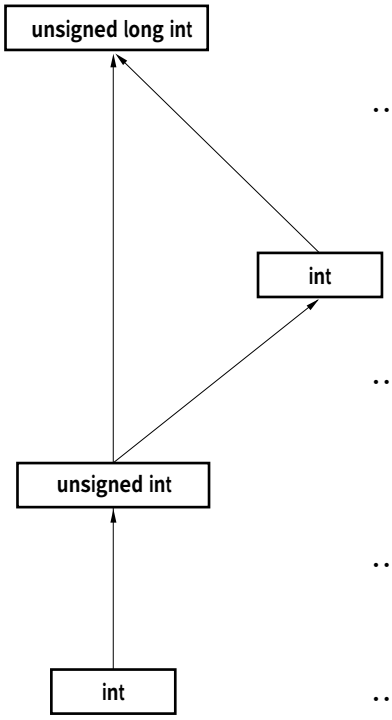
(3) 通常の算術型変換

算術型の演算結果の型は，広い値の範囲を持つ方の型になります。演算結果の型変換は次のとおり行われます。

- 一方のオペランドがlong double型を持てば，もう一方のオペランドはlong double型に変換される。
- 一方のオペランドがdouble型を持てば，もう一方のオペランドはdouble型に変換される。
- 一方のオペランドがfloat型を持てば，もう一方のオペランドはfloat型に変換される。

上記以外は，次の規則に従って汎整数拡張を両方のオペランドに対して行います。図4－1に規則を示します。

図4－1　通常の算術型変換



・・・・・・・・・・・・・一方のオペランドがunsiged long int型である場合と，一方がlong int型でもう一方がunsigned int 型のときlong int型で，unsigned int型の値を表現できない場合は，両方のオペランドはunsigned long int型に変換されます。

・・・・・・・・・・・・・上記以外の場合で，一方がlong int型であり，もう一方のオペランドの値をすべて表現できる場合は，long int型に変換されます。

・・・・・・・・・・・・・上記以外の場合で，一方がunsignded int型であれば，もう一方はunsigned int型になります。

・・・・・・・・・・・・・上記以外の場合は，int型に変換されます。

本コンパイラでは，コンパイル条件（最適化オプション）により，意図的に int型に変換させないようにできます（詳しくは，CC78K0S　C**コンパイラ 操作編**（U14871J）　**第5章　コンパイラ・オプション**を参照してください）。

4.2　他のオペランド

(1) 左辺値と関数指示子

‘左辺値’とは，オブジェクトを指定する（オブジェクト型またはvoidを除く不完全型を持つ）式です。

配列型，不完全型，またはconst修飾型を持たない左辺値およびconst修飾型のメンバを持たない構造体または共用体は“変換可能な左辺値”です。

sizeof演算子，単項＆演算子，＋＋演算子，－－演算子のオペランドまたは，演算子もしくは代入演算子の左オペランドである場合を除いて，配列型を持たない左辺値は指定されるオブジェクトに格納されている値に変換されます。変換されることにより左辺値ではなくなります。

不完全型を持ち，配列型を持たない左辺値は保証されません。

文字配列を除き，‘…の配列’型を持つ左辺値は配列オブジェクトの先頭のメンバを指す‘…へのポインタ’型を持つ式に変換されます。これは左辺値ではありません。

‘関数指示子’は，関数型を持つ式です。sizeof演算子または単項＆演算子のオペランドを除いて，‘…返す関数’型を持つ関数指示子は‘･･･を返す関数へのポインタ’型を持つ式に変換されます。

(2) void

void式（void型を持つ式）の（存在しない）値は，どのような方法でも使えません。そして，この式に対してvoidを除く暗黙的な，または明示的な変換は適用されません。他の型の式がvoid式を必要とする文脈中に現れると，その値，または指示子はないものとされます。

(3) ポインタ

voidポインタは，任意の不完全型またはオブジェクト型へのポインタに変換できます。任意の不完全型またはオブジェクト型へのポインタはvoidポインタに変換できます。結果の値はもとのポインタと等しくなければなりません。

値0を持つ整数定数式が，void * 型にキャストされた式は‘空ポインタ定数’と呼ばれます。空ポインタ定数があるポインタに代入されるか，またはあるポインタと等しいか，または比較されれば空ポインタ定数はそのポインタに変換されます。

〔メ　モ〕

第5章　演算子と式

この章では，C言語で使用される演算子と定数式について説明します。

C言語には，算術演算，論理演算を行うための豊富な演算子が用意されています。また，ビット演算やアドレス演算を行う演算子も用意されています。

式は，値の計算，オブジェクトまたは関数の指示，副作用の生成とこれらの組合せを実行する演算子およびオペランドの列です。

次に演算子の例を示します。

```
#define  TRUE  1
#define  FALSE 0
#define  SIZE  200

void lprintf(char *, int);
void putchar(char c);
char   mark [SIZE+1];  ──── + ·································· 算術演算子

void main(void){
    int   i, prime, k, count;
    count = 0;         ──── = ·································· 代入演算子
    for(i = 0 ; i <= SIZE ; i++) ─┬── ++ ························ 後置演算子
        mark[i]= TRUE;            └── <= ························ 関係演算子

    for(i = 0 ; i <= SIZE ; i++){
        if(mark [i]){
                prime = i + i + 3; ──── + ······················· 算術演算子
                lprintf("%d", prime);
                count++;           ──── ++ ······················ 後置演算子
                if((count%8) == 0) ──── == ······················ 関係演算子
                    putchar('¥n');
                for(k = i + prime ;k<=SIZE;k += prime) ──── += ·· 代入演算子
                    mark[k] = FALSE;
        }
    }
```

```
      lprintf("Total %d¥n", count);
   loop1:;
        goto loop1;
}

lprintf(char *s, int i){
        int  j;
        char  *ss;
        j = i;
        ss = s;
}

void putchar(char c){
        char  d;
        d = c;
}
```

表5－1　演算子の評価順序にC**言語で使用される演算子とその優先順位を示します。**

表5－1　演算子の評価順序

分　類	演 算 子	結　合	優先順位
後　置	[] () . －> ++ －－	→	高　い
単　項	++ －－ & * + － ~ ！ sizeof	←	
キャスト	（型名）	←	
乗　法	* ／ %	→	
加　法	+ －	→	
シ フ ト	<< >>	→	
関　係	< > <= >=	→	
等　値	== ！=	→	
ビットごとのAND	&	→	
ビットごとの排他的OR	^	→	
ビットごとのOR	|	→	
論理AND	& &	→	
論理OR	| |	→	
条　件	？ ：	←	
代　入	= *= ／= %= += －= <<= >>= &= ^= |=	←	
コ ン マ	,	→	低　い

同一行の演算子は，同じ優先順位を持ちます。式中に同じ優先順位を持つ演算子が2つ以上ある場合，矢印で示される方向に評価されていきます。

5.1　一 次 式

一次式には，次のものがあります。

- オブジェクトまたは関数として宣言されている識別子
- 定数
- 文字列リテラル
- かっこで囲まれる式

一次式となる識別子は，オブジェクトの場合左辺値で，関数の場合は関数指示子です。定数は，2.4　**定数**で説明するように指定する値によって型が決まります。

文字列リテラルは，2.5　**文字列リテラル**で説明する型を持つ左辺値です。

5.2　後置演算子

後置演算子は，オブジェクトの後ろに置かれる演算子です。

次に各後置演算子について説明します。

(1) [ ] 添字演算子

後置演算子　　**[ ] 添字演算子**

【機　　能】

添字演算子 ‘ [ ] ’ は，配列オブジェクトのメンバを指定します。配列 ‘ E1[E2] ’ は，‘ (*(E1＋(E2))) ’ と同じであると定義されています。つまりE1の値は配列の先頭メンバへのポインタであり，E2（整数なら）はE1のE2番目（0から数えて）のメンバを示します。多次元配列の場合，配列の次元数の分，添字演算子を連結します。

次の例で，xは3*5のint型配列になります。xは3つのメンバを持ち，各々のメンバは5つのint型メンバを持つ配列です。

```
int x [3][5] ;
```

添字演算子を連続して指定することにより，多次元配列を指定できます。Eがi* j*･･･*kのn次元配列（n≧2）であるとき，Eはn個の添字演算子によって表せます。このとき，Eはj*･･･*kの（n－1）次元配列へのポインタになります。

【構　　文】

```
後置式　［添字式］
```

【注　　意】

“ ･････のオブジェクトへのポインタ ” を持たなければなりません。添字式は，整数型で指定します。結果は，“ ･････ ” 型になります。

(2) ( ) 関数呼び出し

後置演算子　　**( ) 関数呼び出し**

【機　　能】

関数を呼び出します。関数呼び出しは，後置演算子 ‘ ( ) ’ によって行われます。後置式によって，呼び出す関数を指定し，かっこ内に呼び出す関数へ渡す引数を示します。

関数に関する記述は，関数プロトタイプ宣言と，関数定義（関数本体），関数呼び出しがあります。関数プロトタイプ宣言は，関数が返す値と引数の型および記憶クラスを指定します。

関数呼び出しで関数プロトタイプ宣言が参照されなければ各引数は，汎整数拡張されます。これは，“ ディフォールトの実引数拡張 ” と呼ばれます。関数プロトタイプ宣言を行うことにより，ディフォールトの実引数拡張を避け，引数の型や数，返り値の型のミスを検出できます。

“ 識別子( )；” のように記憶クラス指定および型指定がない関数呼び出しは，外部オブジェクトを持ち，引数に関する情報がなくintを返す関数呼び出しであると解釈されます。つまり次の宣言

```
extern int 識別子 ( ) ;
```

が暗黙的に行われます。

【構　　文】

```
後置式 ( 引数式並び ) ;
```

後置演算子　　**( ) 関数呼び出し**

【関数呼び出し例】

```
int  func(char, int);              /* 関数プロトタイプ宣言 */
char a;
int b, ret;
void main(void){
        ret = func(a, b);          /* 関数呼び出し */
}
int  func(char c, int i){          /* 関数定義 */
        ⋮
        return i;
}
```

【注　　意】

呼び出す関数は，配列を除くオブジェクトを返し，後置式はこの関数へのポインタ型です。

プロトタイプを含む関数呼び出しでは，引数の型を対応する仮引数に代入可能な型にします。また，引数の数も合わせなければなりません。

(3) 構造体と共用体のメンバ

後置演算子	．　－＞

①　．ドット

【機　　能】

‘．’は，構造体または共用体のメンバを指定します。後置式は指定する構造体または共用体オブジェクトの名前であり，識別子はそのメンバの名前です。

【構　　文】

```
後置式　．　識別子
```

②　－＞矢印

【機　　能】

‘－＞’は，構造体または共用体のメンバを指定します。後置式は指定する構造体または共用体オブジェクトへのポインタの名前であり，識別子はそのメンバの名前です。

【構　　文】

```
後置式　->　識別子
```

後置演算子　．　－＞

【‘．’，‘－＞’演算子の例】

```
#include <stdlib.h>

union {
     struct {
          int    type ;
     }n ;
     struct {
          int    type ;
          int    intnode ;
     }ni ;
     struct {
          int    type ;
          struct{
               long    longnode ;
          } *nl_p ;
     }nl ;
}u ;

void  func (void) {
    u.nl.type = 1 ;
    u.nl.nl_p -> longnode = -31415L ;
    /*...*/
    if(u.n.type == 1)
       u.nl.nl_p -> longnode = labs(u.nl.nl_p -> longnode) ;
}
```

(4) 後置インクリメントと後置デクリメント演算子

後置演算子	++　−−

① 後置インクリメント演算子

【機　　能】

後置インクリメント演算子はオブジェクトの値を1加算します。この演算は，オブジェクトの型を考慮して行われます。

【構　　文】

```
後置式　++
```

② 後置デクリメント演算子

【機　　能】

後置デクリメント演算子はオブジェクトの値を1減算します。この演算は，オブジェクトの型を考慮して行われます。

【構　　文】

```
後置式　--
```

【注　　意】

後置インクリメントと後置デクリメント演算子のオペランドは，修飾された，または，されていない変換可能な左辺値です。

5.3　単項演算子

単項演算子は，1つのオブジェクトおよび項目に対して演算を行います。単項演算子には，次のものがあります。

- 前置インクリメントと前置デクリメント演算子

  ++　--

- アドレスと間接演算子

  &　*

- 単項算術演算子

  +　-　~　!

- sizeof演算子

  sizeof

単項演算子それぞれについて次に説明します。

(1) 前置インクリメントと前置デクリメント演算子

単項演算子	++ −−

① 前置インクリメント演算子

【機　　能】

前置インクリメント演算子はオブジェクトの値を1加算します。前置インクリメント演算子の式‘＋＋E’は，次の式と同じ結果になります。

```
E = E + 1
```

または

```
E += 1
```

【構　　文】

```
++ 単項式
```

② 前置デクリメント演算子

【機　　能】

前置デクリメント演算子はオブジェクトの値を1減算します。前置デクリメント演算子の式‘−− E’は，次の式と同じ結果になります。

```
E = E - 1
```

または

```
E -= 1
```

【構　　文】

```
-- 単項式
```

(2) アドレスと間接演算子

単項演算子	& *

① 単項 & 演算子

【機　　能】

指定したオブジェクトのアドレスを返します。

【構　　文】

```
& オペランド
```

② 単項 * 演算子

【機　　能】

指定したポインタが指す値を返します。

【構　　文】

```
* オペランド
```

【注　　意】

単項＆演算子のオペランドは，register記憶域クラス指定子で宣言されていないオブジェクトを指す左辺値です。関数指示子またはビット・フィールドは，単項＆演算子のオペランドに使用できません。

単項 * 演算子のオペランドは，ポインタ型です。

(3) 単項算術演算子（＋ － ~ ！）

単項演算子　＋ － ~ ！

【機　　能】

単項 ＋ 演算子は，オペランドに対して正の整数拡張を行います。

単項 － 演算子は，オペランドに対して負の整数拡張を行います。

単項 ~ 演算子は，オペランドのビットごとの補数を返します。

単項！演算子は，論理否定演算子と呼ばれます。論理否定演算子は，オペランドの値が‘0’のとき‘1’を返します。それ以外のときは‘0’を返します。

【構　　文】

```
+ オペランド
- オペランド
~ オペランド
! オペランド
```

（4）sizeof演算子

単項演算子　　sizeof演算子

【機　　能】

指定したオブジェクトの大きさをバイト単位で返します。返り値はオブジェクトの型で決まり，オブジェクトの値自体は評価しません。

sizeof演算したchar型，unsigned char型またはsigned char型（それらの修飾型も含める）のオブジェクトが返す値は1です。配列型のオブジェクトでは，配列の総バイト数になります。また，構造体または共用体型のオブジェクトの場合，結果の値は領域を保持するために入れられた内部的な詰めものを含めたオブジェクトの総バイト数です。

結果は整数定数であり，その型はsize_tです。これはヘッダ‘ stddef.h ’で定義されています。sizeof演算子は，おもに記憶域の割り当ておよび入出力システムとのやり取りに使用します。

【構　　文】

```
sizeof　単項式
sizeof　（　型名　）
```

【使 用 例】

次の例は，配列の総バイト数をメンバの大きさで割ることにより，配列のメンバ数を求めています。numには，5が入ります。

```
int num;
char array[ ]= {0, 1, 2, 3, 4};

void func(void){
     num = sizeof array / sizeof array [0];
}
```

【注　　意】

sizeof演算子のオペランドには，関数型または不完全型を持つ式とビット・フィールド・オブジェクトを指すものを使用できません。

5.4　キャスト演算子

キャスト演算子は，データの型を変更します。おもにポインタ型の変換を行う場合にキャスト演算子を使用します。

キャスト演算子　（型名）

【機　　能】

オブジェクトの型を，かっこ内で示した型に変換します。

【構　　文】

```
( 型名 ) 式
```

【使 用 例】

```
void func (void){
        int val;
        float f;

        f = 3.14F;
        val = (int)f;           /* キャストにより，3がvalに入る */
        val = *(int *)0x10000;  /* 定数をキャスト */
}
```

5.5　算術演算子

算術演算子は，優先順位により乗除演算子と加減演算子に分かれます。乗除演算子は，2つのオペランドの積，商，余りを求め，加減演算子は，2つのオペランドの和と差を求めます。

- 乗除演算子　　*　/　%
- 加減演算子　　+　-

表5－2　除算／剰余算の演算結果の符号

a / b		b	
		+	−
a	+	+	−
	−	−	+

a % b		b	
		+	−
a	+	+	+
	−	−	−

備考　a, bは各オペランドを示します。

除算は符号を取った数値によって行い，小数点以下は切り捨てます。剰余算も同じように符号を取った数値によって行います。除算／剰余算の演算結果は，符号を取って計算された値に表5－2の符号を付けたものです。表5－2は，2つのオペランドの符号のみの計算結果を示しています。

次に，乗除演算子，加減演算子について説明します。なお，構文の説明で使用されるE1，E2は，オペランドあるいは式を示します。

(1) 乗除演算子

乗除演算子	* ／ %

① * 演算子

【機　　能】

* 演算子は，2つのオペランドの積を求めます。

【構　　文】

```
E1  *  E2
```

② ／演算子

【機　　能】

／演算子は，左オペランドを右オペランドで除算した商を求めます。

【構　　文】

```
E1  /  E2
```

③ %演算子

【機　　能】

%演算子は，左オペランドを右オペランドで除算した余りを求めます。

【構　　文】

```
E1  %  E2
```

(2) 加減演算子

加減演算子　＋　－

①　＋演算子

【機　　能】

2つのオペランドの和を求めます。

【構　　文】

```
E1  +  E2
```

②　－演算子

【機　　能】

左オペランドから右オペランドを引いた差を求めます。

【構　　文】

```
E1  -  E2
```

5. 6　ビット単位のシフト演算子

シフト演算子は，演算子のオペランドを記号で示された方向へ移動します。
シフト演算子は次の2つです。

・シフト演算子　<<　　>>

表5－3　シフト演算

a << b		b[注]
a	＋	0
	－	0

a >> b		b[注]
a	＋	0
	－	－1

注　bにaのビット幅以上の数値が指定され，シフト演算によりオーバフローが起こった場合の結果を表に示します。bに負の数が指定された場合は符号なし型とし，正の数として処理します。

備考　a, bは各オペランドを示します。

次に，各シフト演算子について説明します。なお，E1，E2は，オペランドあるいは式を示します。

シフト演算子　<<　>>

①　<< 演算子

【機　　能】

左オペランドを右オペランドが示す値（ビット）分左にシフトし，空いたビットに0を入れます。“E1 << E2”で，‘E1’が符号なし型であれば結果の値は，‘E1’に2の‘E2’乗をかけた値になります。

【構　　文】

```
E1 << E2
```

②　>> 演算子

【機　　能】

左オペランドを右オペランドが示す値（ビット）分右にシフトします。

本コンパイラでは，‘E1’が符号なし型の場合，シフトして空いたビットに0を入れます。
‘E1’が符号付き型の場合，空いたビットに符号ビットと同じものを入れます。

“E1 >> E2”で，‘E1’が符号なし型または符号付き型でかつ非負の値を持つ場合，結果の値は，‘E1’を2の‘E2’乗で割った値です。

【構　　文】

```
E1 >> E2
```

5. 7　関係演算子

関係を示す演算子には，2つのオペランドの大小関係を示す‘ 関係演算子 ’と，等しいか等しくないかを示す‘ 等値演算子 ’があります。

関係演算子と，等値演算子は次のものがあります。

・関係演算子	< > <= >=
・等値演算子	== !=

関係演算子で，2つのポインタを比較した場合の大小関係は，ポインタで指されるオブジェクトのアドレス空間内での相対位置によって決まります。

本コンパイラでは，関係演算子，等値演算子は，指定された関係が真であれば‘ 1 ’を，偽であれば‘ 0 ’を生成し，それらの結果はint型を持ちます。

関係演算子，等値演算子について次に説明します。なお，構文の説明で使用されるE1，E2は，オペランドあるいは式を示します。

(1) 関係演算子

関係演算子	< > <= >=

① < 演算子

【機　　能】

左オペランドが右オペランドより小さいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　　文】

```
E1 < E2
```

② > 演算子

【機　　能】

左オペランドが右オペランドより大きいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　　文】

```
E1 > E2
```

関係演算子　　**< > <= >=**

③ <= 演算子

【機　　能】

左オペランドが右オペランドより小さいか，等しいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　　文】

```
E1  <=  E2
```

④ >= 演算子

【機　　能】

左オペランドが右オペランドより大きいか，等しいときに‘1’を返します。それ以外の場合には，‘0’を返します。

【構　　文】

```
E1  >=  E2
```

(2) 等値演算子

等値演算子	== !=

① == 演算子

【機　　能】

2つのオペランドが等しい場合‘1’を返し，等しくない場合に‘0’を返します。

【構　　文】

```
E1  ==  E2
```

② != 演算子

【機　　能】

2つのオペランドが等しくない場合‘1’を返し，等しい場合に‘0’を返します。

【構　　文】

```
E1  !=  E2
```

5.8　ビット単位の論理演算子

ビット単位の論理演算子は，オブジェクトの値をビット単位で論理演算します。ビット単位の論理演算にはAND，排他OR，ORがあり，それぞれ次の演算子で示します。

- ビット単位のAND演算子　　&
- ビット単位の排他OR演算子　　^
- ビット単位のOR演算子　　|

ビット単位の論理演算子について次に説明します。なお，構文の説明で使用されるE1，E2は，オペランドあるいは式を示します。

(1) ビット単位のAND演算子

ビット単位のAND演算子 &

【機　　能】

‘ & ’ はビットごとの論理積を返す，ビット単位のAND演算子です。ビット単位のAND演算子は，それぞれ対応するビットが‘ 1 ’ のときのみ‘ 1 ’ を返します。

ビット単位のAND演算子は，‘ &演算子 ’ によって指定します。

表5－4　ビット単位のAND演算子

		左オペランドの1ビットの値	
		1	0
右オペランドの1ビットの値	1	1	0
	0	0	0

【構　　文】

```
E1  &  E2
```

(2) ビット単位の排他OR演算子

ビット単位の排他OR演算子　　^

【機　　能】

‘ ^ ’はビットごとの排他的論理和を返す，ビット単位の排他OR演算子です。ビット単位の排他OR演算子は，それぞれ対応するビットが異なるときのみ‘ 1 ’を返します。

表5－5　ビット単位の排他的OR演算子

		左オペランドの1ビットの値	
		1	0
右オペランドの1ビットの値	1	0	1
	0	1	0

【構　　文】

```
E1 ^ E2
```

(3) ビット単位のOR演算子

ビット単位のOR演算子 |

【機　　能】

‘ | ’はビットごとの論理和を返す，ビット単位のOR演算子です。ビットごとのOR演算子は，それぞれ対応するビットが‘ 0 ’のときのみ‘ 0 ’を返します。

表5－6　ビット単位のOR演算子

		左オペランドの1ビットの値	
		1	0
右オペランドの1ビットの値	1	1	1
	0	1	0

【構　　文】

```
E1 | E2
```

5. 9　論理演算子

論理演算子は，2つのオペランドの論理積と論理和を行います。論理積は，論理AND演算子によって，論理和は論理OR演算子によって指定します。各論理演算子は，次のものです。

- 論理AND演算子　　&&
- 論理OR演算子　　||

両論理演算子の各オペランドはともに int型の値‘ 0 ’または‘ 1 ’を返します。次に，各論理演算子について説明します。なお，構文の説明で使用されるE1，E2は，オペランドあるいは式を示します。

(1) 論理AND演算子

論理AND演算子 &&

【機　　能】

&& 演算子は，2つのオペランドの論理AND演算を行います。論理AND演算は，2つのオペランドが‘0’以外のときのみ‘1’を返します。これ以外の場合‘0’を返します。結果の型は，int型です。

表5－7　論理AND演算子

		左オペランドの値	
		0	0以外
右オペランドの値	0	0	0
	0以外	0	1

【構　　文】

```
E1  &&  E2
```

【注　　意】

&& 演算子は，オペランドを左から右に評価します。左オペランドの値が‘0’であれば，右オペランドの評価を行いません。

(2) 論理OR演算子

論理OR演算子 ||

【機　　能】

|| 演算子は，2つのオペランドの論理ORを行います。論理OR演算は2つのオペランドが‘0’のときのみ‘0’を返します。これ以外のときは，‘1’を返します。結果の型は，int型です。

表5－8　論理OR演算子

		左オペランドの値	
		0	0以外
右オペランドの値	0	0	1
	0以外	1	1

【構　　文】

```
E1 || E2
```

【注　　意】

|| 演算子は，オペランドを左から右に評価します。左オペランドの値が‘0’以外であれば，右オペランドの評価を行いません。

5. 10　条件演算子

条件演算子は第1オペランドの値によって次に行う処理を判断します。条件演算子は，‘？’と‘：’によって判断します。条件演算子について次に説明します。

条件演算子　　？　：

【機　　能】

条件演算は第1オペランドを評価して，値が‘0’以外であれば第2オペランドを評価し，‘0’であれば第3オペランドを評価します。条件演算子の結果の値は，第2または第3オペランドの値になります。

【構　　文】

第1オペランド　？　第2オペランド　：　第3オペランド

【使 用 例】

```
#define TRUE   1
#define FALSE  0
char  flag ;
int  ret ;
int  func(){
        ret = flag ? TRUE : FALSE ;
        return  ret ;
}
```

【注　　意】

第2および第3オペランドの型がともに算術型ならば，それらを共通の型にするために通常の算術型変換を行います。結果の型は，その共通の型とします。両オペランドの型がともに構造体型または共用体型なら，結果の型はその型とします。また，両オペランドがvoid型ならば結果の型はvoid型とします。

5. 11　代入演算子

代入演算子は右オペランドそのものを左のオブジェクトに格納する単純代入と，両オペランドの演算結果を左のオブジェクトに格納する複合代入があります。

代入演算子は次のものがあります。

・代入演算子

```
=   *=   /=   %=   +=   -=   <<=   >>=
&=   ^=   |=
```

次に，各代入演算子について説明します。なお，構文の説明で使用されるE1，E2は，オペランドあるいは式を示します。

(1) 単純代入

単純代入 =

【機　　能】

単純代入は，右オペランドを左オペランドの型に変換し，左のオブジェクトに格納します。

次の例では，単純代入の型変換によって関数から返されるint型の値はchar型に変換され，あふれた部分は切り捨てられます。そして‘－1’との比較は，再びint型に変換されてから行われます。修飾子なしで宣言されている変数‘c’がunsigned charとみなされれば変換の結果は負にならず‘－1’との比較は決して等しくなりません。このような場合，移植性を完全にするために変数‘c’はint型で宣言します。

```
int f(void) ;


char c ;
/*...*/  ((c = f()) == -1)  /*...*/
```

【構　　文】

```
E1  =  E2
```

(2) 複合代入

複合代入　　*=　/=　%=　+=　−=　<<=　>>=　&=　^=　|=

【機　　能】

複合代入演算子は，左右のオペランドの演算を行い，結果を左のオブジェクトに格納します。格納される値は，左オペランドの型に変換されます。

“ E1 op= E2 ” の複合代入は，左オペランド‘ E1 ’が1度しか評価されないことを除き，単純代入式“ E1 = E1 op (E2) ” と同じです。

次の複合代入演算の結果は，右の単純代入式の結果と同じになります。

```
a *= b;          a = a * b;
a /= b;          a = a / b;
a %= b;          a = a % b;
a += b;          a = a + b;
a -= b;          a = a - b;
a <<= b;         a = a << b;
a >>= b;         a = a >> b;
a &= b;          a = a & b;
a ^= b;          a = a ^ b;
a |= b;          a = a | b;
```

【構　　文】

```
E1  *=   E2
E1  /=   E2
E1  %=   E2
E1  +=   E2
E1  -=   E2
E1  <<=  E2
E1  >>=  E2
E1  &=   E2
E1  ^=   E2
E1  |=   E2
```

5. 12　コンマ演算子

コンマ演算子　　,

【機　　能】

コンマ演算子は，左のオペランドをvoid型として評価します。それから右のオペランドを評価し，その値を返します。

構文によって示されるように，コンマを区切り子として使用するところ（関数の引数並びおよび初期化子並び中）では本章で示すコンマ演算子は現れません。

次の例では，コンマ演算子によって関数 f()に渡す第2引数の値を求めています。コンマ演算子により，第2引数の値は5になります。

```
int a, c, t;
void main(void) {
    f(a, (t=3, t+2), c) ;
}
```

【構　　文】

```
        E1  ,  E2
```

5. 13　定 数 式

定数式には，汎整数定数式と算術定数式，アドレス定数式および初期化子中の定数式があります。定数式の評価は実行中でなく，ほとんどコンパイル中に行われます。

定数式では，sizeof演算子の中で使用する以外の，次のような演算子は使用できません。

- 代入演算子
- インクリメント演算子
- デクリメント演算子
- 関数呼び出し演算子
- コンマ演算子

(1) 汎整数定数式

汎整数定数式は汎整数型です。汎整数定数式のオペランドには次のものが使用できます。

- 整数定数
- 列挙定数
- 文字定数
- sizeof式
- 浮動小数点定数

(2) 算術定数式

算術定数式は算術型です。算術定数式のオペランドには，次のものが使用できます。

- 整数定数
- 列挙定数
- 文字定数
- sizeof式
- 浮動小数点定数

(3) アドレス定数

アドレス定数は，静的記憶域期間を持つオブジェクトへのポインタまたは関数指示子へのポインタです。アドレス定数は，配列型または関数型の式を用いることにより，暗黙的に指定できます。明示的に指定するときは単項＆演算子を用います。

アドレス定数は，次の演算子を用いて指定できます。しかし，これを利用してもオブジェクトの値は参照できません。

- 配列添字演算子‘［］’
- メンバ・アクセス演算子‘．’
- メンバ・アクセス演算子‘－＞’
- アドレス単項演算子‘＆’
- 間接単項演算子‘＊’
- ポインタへのキャスト

〔メ　モ〕

第6章　C言語の制御構造

この章は，C言語の制御構造とCで実行される文について説明します。

一般的に，どのような複雑な処理でも基本的な3つの制御構造で表せます。この3つの制御構造は，順次，選択および繰り返しです。また強制的にプログラムの流れを変える場合，分岐を使います。

① 順次処理

順次処理は，プログラムに記述された順に上から下へ実行します。順次実行される文は，特に指定する必要はなく順次実行されます。

② 選択処理

選択処理は，実行中のプログラムの状態により次に実行する文が選択され，実行します。選択の条件は，制御文として指示します。制御文により2つまたは多岐にわたる文の1つが選択され実行されます。

③ 繰り返し処理

繰り返し処理は，同じ処理を複数回実行します。制御される文は，制御文で示した状態の間，または指定した回数の間繰り返し実行されます。

④ 分岐処理

分岐処理は，強制的に現在のプログラムの流れから抜け出し，指定したレーベルに制御を移します。分岐により指定したレーベル名の次の文から実行されます。

C言語で実行される文には，次の6つがあります。

- レーベル付き文 ‥‥‥‥‥‥ switch文の取る値とgoto文の分岐先により，分岐を引き起こします。
- 複合文（ブロック） ‥‥‥‥ 複数の文の集まりを1つの構文単位としてまとめます。
- 式文 ‥‥‥‥‥‥‥‥‥‥‥‥ 1つの式とセミコロンからなる文です。
- 選択文 ‥‥‥‥‥‥‥‥‥‥‥ 式の値に応じていくつかの文から一つの文を選択します。
- 繰り返し文 ‥‥‥‥‥‥‥‥ ループ本体と呼ばれる文を制御式が0と比較して等しくなるまでの間，繰り返して実行します。
- 分岐文 ‥‥‥‥‥‥‥‥‥‥‥ 別の場所への無条件分岐を引き起こします。

これらの文の記述例を次に示します。

【記 述 例】

```
#define SIZE  10
#define TRUE  1
#define FALSE 0

extern void putchar(char);
extern void lprintf(char *, int);

char    mark [SIZE+1];
void main(void){
        int  i, prime, k, count;

        count = 0;
        for(i = 0 ; i <= SIZE ; i++)              /* for ・・・・・・・・・・・・・・・・・繰り返し文 */
                mark [i] = TRUE ;
        for(i = 0 ; i <= SIZE ; i++) {            /* for ・・・・・・・・・・・・・・・・・繰り返し文 */
                if(mark[i]){                      /* if ・・・・・・・・・・・・・・・・・・選択文     */
                        prime = i + i + 3;
                        lprintf("%d", prime);
                        if((count%8) == 0)        /* if ・・・・・・・・・・・・・・・・・・選択文     */
                            putchar('¥n');
                        for(k = i + prime ; k <= SIZE ; k += prime)
                            mark [k] = FALSE;
                }
        }
       lprintf("Total %d¥n", count);

loop1:                                    /*  loop1：・・・・・・・・・・・・・・・・レーベル付き文 */
        goto loop1;                       /*  goto ・・・・・・・・・・・・・・・・・分岐文         */
}
```

6. 1　レーベル付き文

レーベル付き文は，switch**文と**goto**文の分岐先を指定します。**switch**文は，複数の選択肢がある文から制御式で指定した文を選択し，実行します。レーベル付き文は，**switch**文で実行される文のレーベルになります。**goto**文は，通常の処理の流れから対応するレーベルへ無条件に分岐します。**

レーベル付き文について次に説明します。

（1）caseレーベル

レーベル付き文　　caseレーベル

【機　　能】

caseは，switch文中にのみ使用します。switch文の制御式の取る値を列挙します。

【構　　文】

```
        case  定数式  :  文
```

【使用例1】

```
int  f(void), i ;
void main (void){
       /* ... */
         switch(f()) {
               case 1:
                 i = i + 4 ;
                 break ;
               case 2:
                 i = i + 3 ;
                 break ;
               case 3:
                 i = i + 2 ;
       }
       /* ... */
}
```

【説　　明】

使用例1では，f()の返り値が1のとき最初のcase文が選択され‘ i = i＋4 ’の式が実行されます。同じように値が2のときは2番目のcaseが，3のとき3番目のcaseが選択されます。使用例のbreak文は途中でswitch文から抜け出すためのものです。

このようにcaseは，複数の選択肢がある場合に使用します。

レーベル付き文　　case**レーベル**

【使用例2】

```
int  i ;
void main (void){
        /* ... */
        i = 2;
        switch(i) {
                case 1:
                  i = i + 4 ;
                case 2:
                  i = i + 3 ;
                case 3:
                  i = i + 2 ;
        }
        /* ... */
}
```

【説　　明】

使用例2では，iに2が入っているので，2番目のcase文から実行されますが，case文の中にbreak文を含まないため，続けて3番目のcase文も実行されます。このように，case文の定数式と制御式が一致した場合，それ以降のプログラムを順次実行します。途中でswitch文から抜けたい場合は，break文を使用します。

(2) defaultレーベル

レーベル付き文　　defaultレーベル

【機　　能】

defaultは，switch文中にのみ使用します。defaultは，switch文中に対応するcaseがない場合の処理を指定します。

【構　　文】

```
default ： 文
```

【使 用 例】

```
int  f (void),  i ;

switch (f()) {
   case 1:
        i = i + 4 ;
        break ;
   case 2:
        i = i + 3 ;
        break ;
   case 3:
        i = i + 2 ;
   default:
        i = 1;
}
```

【説　　明】

使用例では，f()の返り値が1-3のときは対応するcaseが選択され，それに続く文が実行されます。使用例のbreak文は途中でswitch文から抜け出すためのものです。f()の返り値が1-3以外の場合，defaultに続く文が実行されｉの値は1になります。

6.2　複合文（ブロック）

複合文は，複数の文を1つの構文単位とします。複数の文は，中かっこ“ { } ”で囲まれることにより複合文となります。たとえば複合文は，ある状態のときに行わせる処理が複数ある場合，その文を中かっこ“ { } ”で囲み処理させます。

6.3　式文と空文

1つの文とセミコロンからなる文を式文といいます。また，セミコロンのみからなる文を空文といいます。空文は，空のループ本体やレーベルを置くために使用します。

式文と空文の記述例を次に示します。

次の例のように，式文として副作用を得るためだけに呼ばれる関数は，キャスト式を用いて明示的に返り値の値を捨てることができます。

```
int p(int) ;
void main(void){
    /* ... */
    (void)p(0) ;
}
```

空文は，繰り返し文のループ本体として使用できます。

```
char *s ;
void main(void){
    /*...*/
    while (*s++ != '0') ;
    /*...*/
}
```

また複合文を閉じる‘｝’の前にレーベルを置くためにも使用できます。

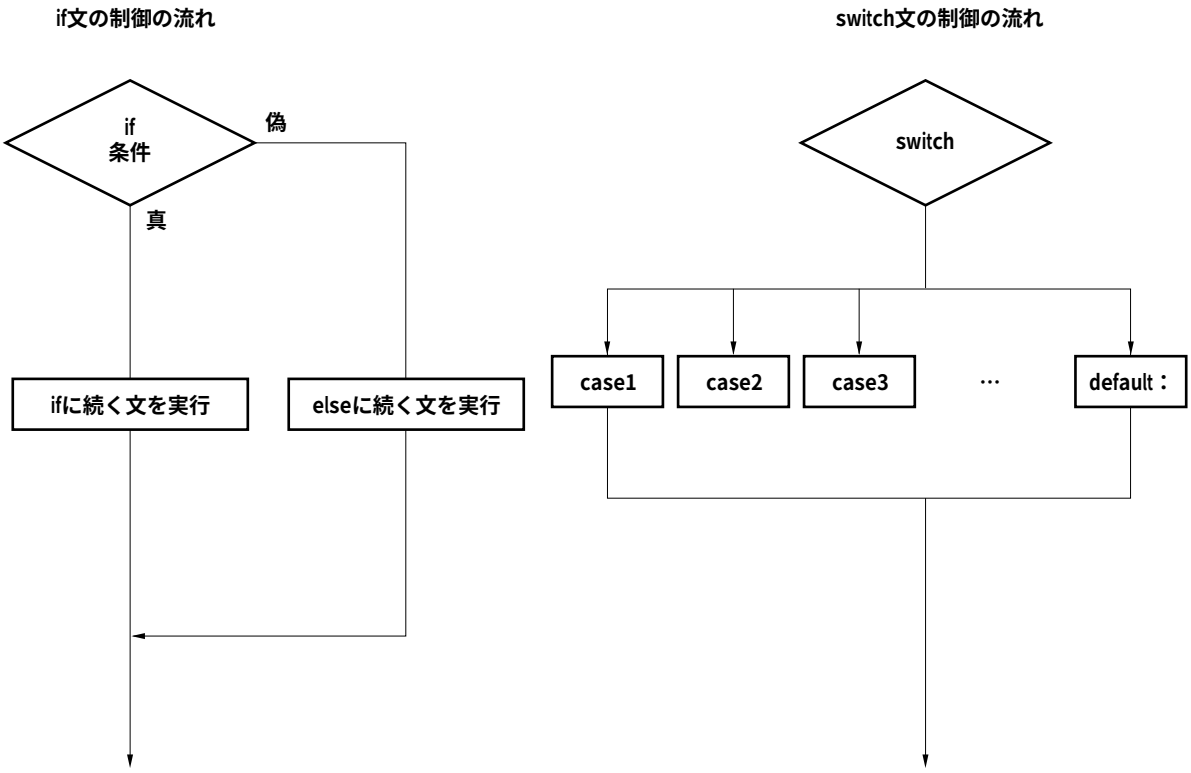
```
 void func(void) {
          /*...*/
      while(loop1) {
          /*...*/
          while(loop2) {
              /*...*/
              if(want_out)
                  goto end_loop1 ;
              /*...*/
          }
      end_loop1: ;
      }
}
```

6.4 選 択 文

選択文には，if文，switch文があります。選択文は“（ ）”で囲まれた制御式の値によって，文の集まりの中から行う処理を選びます。

if文，switch文の制御の流れを次に示します。

図6－1　選択文の制御の流れ



(1) if文，if～else文

選択文 if文，if～else文

【機　　能】

if文，if～else文は，“ ( ) ”で囲まれた制御式の値が0でなければ次に続く文を実行します。if～else文の場合，式の値が0になるとelse文の文を実行します。

【構　　文】

```
if （ 式 ） 文
if （ 式 ） 文 else 文
```

【使 用 例】

```
unsigned char   uc ;
void func (void){
     if( uc < 10 ){
           /* 111 */
     }
     else{
           /* 222 */
     }
}
```

【説　　明】

この例では，if文中の制御式によりucの値が10より小さい場合は“{ /* 111 */ }”のブロックが実行され，10以上の場合は“{ /* 222 */ }”のブロックが実行されます。

【注　　意】

if文，if～else文のあとに，“ { } ”で処理が囲まれていない場合は，if文／else文の次の1行の処理のみを本体とみなし実行します。

（2）switch文

選択文 switch**文**

【機　　能】

switch**文は，“（ ）”で囲った制御式に対応するcaseのスイッチ本体に制御を移します。制御式に対応するcaseがないときは，**default**に続く文が実行され，またdefaultがないときはどの文も実行されません。**

【構　　文】

```
switch　（ 式 ）　文
```

【使 用 例】

```
extern void func(void);
unsigned char   mode ;
void main(void){
     switch(mode){
          case 2:
               mode = 8 ;
               break ;
          case 4:
               mode = 2 ;
               break ;
          case 8:
               func( );
     }
}
```

【注　　意】

1**つの**switch**文の各**case**は，同じ値を設定できません。また，**default**は**1**つの**switch**文中で**1**度しか使用できません。**

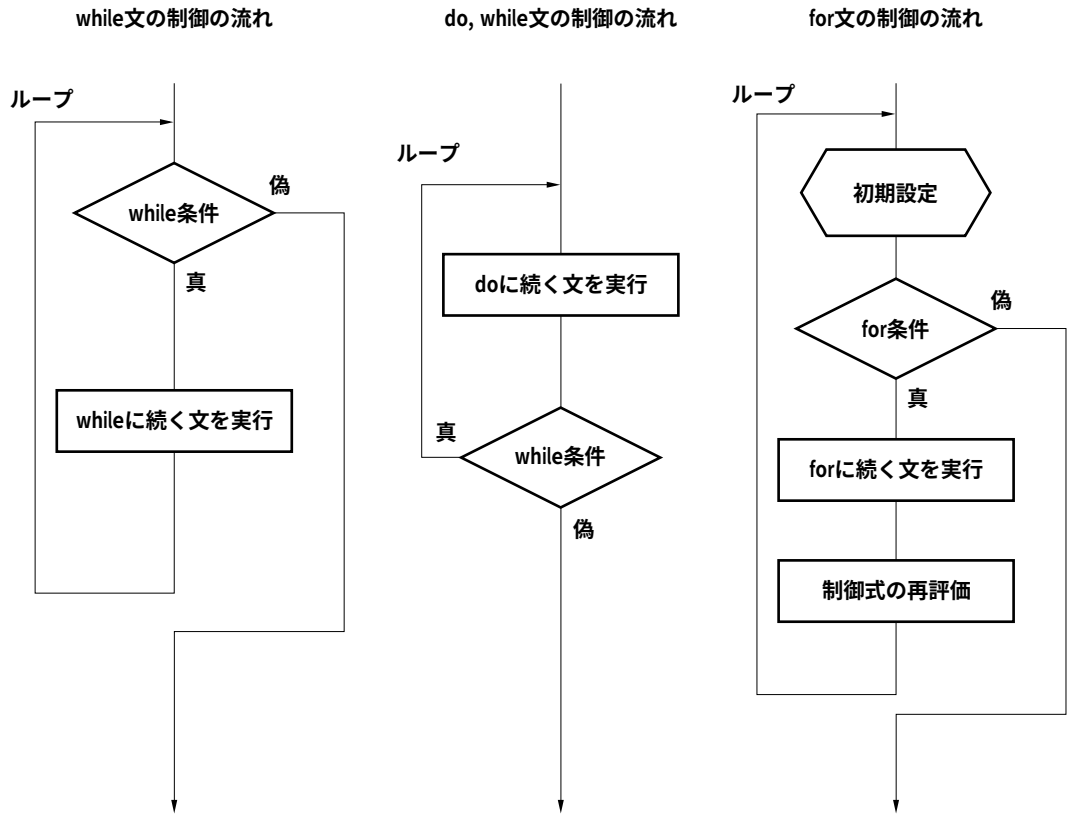
6.5　繰り返し文

繰り返し文は，“（ ）”で囲まれた制御式が正しい間（‘0’以外のとき）ループ本体を繰り返し実行します。繰り返し文には次の3つがあります。

- while文
- do文
- for文

繰り返し文の制御の流れを次に示します。

図6－2　繰り返し文の制御の流れ



（1）while文

繰り返し文　while文

【機　　能】

while文は，“（ ）”で囲まれた制御式が正しい間（0以外のとき）ループ本体を繰り返し実行します。while文は，制御式をループ本体の実行前に評価します。

【構　　文】

```
while （ 式 ） 文
```

【使 用 例】

```
int   i, x ;
void main (void){
     i=1,  x=0 ;

     while( i < 11 ){
                x += i ;
                i++ ;
     }
}
```

【説　　明】

使用例は，xに1から10までの整数の総和を求めるものです。このwhile文のループ本体は，中かっこで囲まれた部分です。制御式“i < 11”は，iが11になると0を返します。このため，iが1から10になる間ループ本体が繰り返し実行されます。

“while(1) {文}”は，永久にループ文を実行するために使用します。

(2) do文

繰り返し文 do文

【機　　能】

do文は，“（ ）”で囲まれた制御式が正しい間（0以外のとき）ループ本体を繰り返し実行します。do文は，制御式をループ本体の実行後に評価します。

【構　　文】

```
do　文　while（ 式 ）；
```

【使 用 例】

```
int  i, x ;
void main (void) {
      i=1,  x=0 ;

      do {
        x += i ;
        i++ ;
      } while( i < 11 );
}
```

【説　　明】

使用例は，xに1から10までの整数の総和を求めるものです。このdo文のループ本体は，中かっこで囲まれた部分です。制御式“i < 11”は，iが11になると0を返します。このため，iが1から10になる間ループ本体が繰り返し実行されます。do文の制御式は実行後に評価されるので，ループ本体は必ず1回以上実行されます。

（3）for文

繰り返し文 for文

【機　　能】

for文は，“（ ）”で囲まれた中の第2の式が正しい間（0以外のとき）ループ本体を繰り返し実行します。第1の式は，カウンタとして使用する変数の初期化を行い，ループの最初に1回だけ実行します。第2の式でカウンタの判断を行います。第3の式は，ループごとに最後に実行する式で，この式の実行後，変数の再評価を行います。

【構　　文】

```
for （ 第1の式 ； 第2の式 ； 第3の式 ） 文
```

【使 用 例】

```
int  i, x=0 ;

for(i=1 ; i<11 ; ++i)
  x += i ;
```

【説　　明】

使用例は，xに1から10までの整数の総和を求めるものです。このforループの本体は，“x ＋= i”です。制御式“ i<11 ”は，iが11になると0を返します。このため，iが1から10になる間ループ本体が繰り返し実行されます。

【注　　意】

for文のあとに，“ { } ”で処理が囲まれていない場合は，for文の次の1行の処理のみをfor文のループ本体とみなします。

for文の第1の式と第3の式は，省略できます。第2の式を省略した場合は，0でない定数によって置き換えます。“for(； ；) 文”の記述は，永久にループ本体を実行する場合に用います。

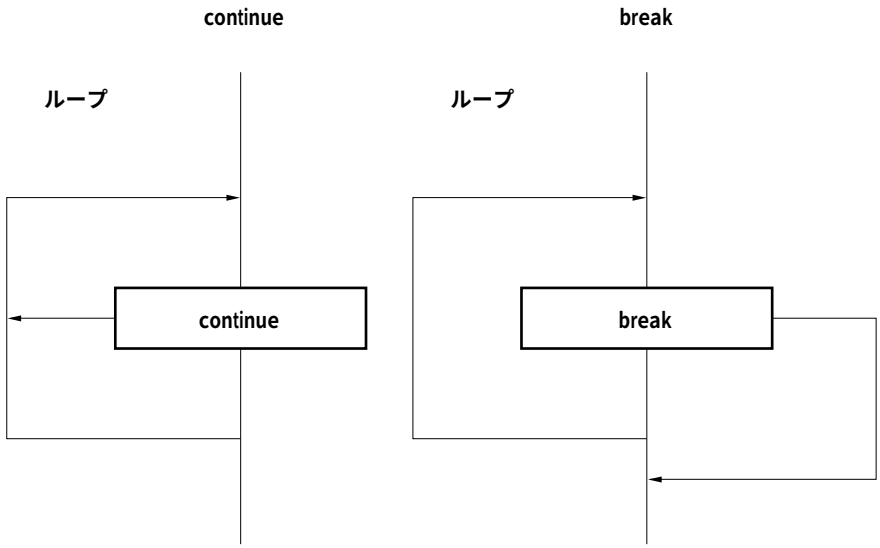
6.6　分 岐 文

分岐文は，現在の制御の流れから抜け出し，任意の場所へ無条件に制御を移すものです。分岐文には，次の4つがあります。

- goto文
- continue文
- break文
- return文

分岐文の制御の流れを次に示します。

図6－3　分岐文の制御の流れ



（1）goto文

分岐文 goto**文**

【機　　能】

goto**文は，現在の関数中に指定したレーベル名へ無条件にジャンプします。**

【構　　文】

```
goto 識別子 ;
```

【使 用 例】

```
do{
        /*...*/
         goto point ;
        /*...*/
}while(/*...*/) ;
        /*...*/
point: ;
```

【説　　明】

この例で，goto**文に制御が移るとループ処理から無条件に抜け出し，**point**の次の文に制御が移ります。**

【注　　意】

goto**文で示される分岐先（レーベル名）は，その**goto**文を含む関数中のどこかに必ず示します。**

（2）continue文

分岐文 continue**文**

【機　　能】

continue**文は，繰り返し文のループ本体中で使用します。**continue**文により制御の流れは，ループ本体の最後に無条件に分岐します。**continue**文は，これを囲む最も内側の繰り返し文に作用します。**

【構　　文】

```
continue ;
```

【使 用 例】

```
while(/*...*/){
        /*...*/
        continue ;
        /*....*/
        contin: ;
}
```

【説　　明】

この例で，ループ本体中の処理がcontinue**文にくると制御は，レーベル‘** contin **’に無条件に分岐します。レーベル‘** contin **’は，分岐先を示したもので特につける必要はありません。この例は，**goto**文を使い**continue**文を‘** goto contin **；’に替えても同じ動作をします。**

【注　　意】

continue**文は，ループ本体またはループ本体中にのみ使用できます。**

(3) break文

分岐文 break文

【機　　能】

break文は，繰り返し文またはswitch文中から抜け出し，繰り返し文またはswitch文の次の文へ制御を移します。

【構　　文】

```
        break  ;
```

【使 用 例】

```
int   i;
unsigned char count, flag;

void main(void){
    /*...*/
    for(i = 0;i < 20;i++){
        switch(count){
              case 10:
                   flag = 1;
                   break;                /* switch文を抜ける */
              default:
                   func();
        }
        if (flag)
              break;                     /* forループから抜ける */
    }
}
```

分岐文 break**文**

【説　　明】

この例でbreak**文は，**switch**文中で必要以上の評価を行わないように使用されています。**switch**文の評価で，適合する**case**レーベルがあると，続く**break**文により**switch**文から抜け出します。**

【注　　意】

break**文は，スイッチ本体として使用するか，ループ本体としてのみ使用できます。**

（4）return文

分岐文 return**文**

【機　　能】

return**文は，**return**を含む関数から抜け出し，これを呼び出した関数に制御を移します。また，**return**文の式の値を，関数呼び出し式の値として呼び出し元に返します。**

1**つの関数中に複数の**return**文を使用できます。**

関数の最後を‘ } ’で閉じることは，式を持たないreturn**文を実行することと同じです。**

【構　　文】

```
return  式  ;
```

【使 用 例】

```
int  f (int);

void main(void){
          /*...*/
          int  i =0,  y = 0 ;
          y = f(i) ;
          /*...*/
}

int  f(int  i){
          int x = 0 ;
          /*...*/
          return(x) ;
}
```

分岐文　　return**文**

【説　　明】

この例で関数“ f() **”は，**return**文に制御が移ると**main**関数へリターンします。**return**文では，返り値として変数‘** x **’の値を返しているので，代入演算子により変数‘** y **’に変数‘** x **’の値が代入されます。**

【注　　意】

void**型の関数では，返り値を示す式を**return**文に使用できません。**

第7章　構造体と共用体

構造体，共用体は，1つの名前でまとまった，異なった型を持つメンバ・オブジェクトの集まりです。構造体は，メンバ・オブジェクトが連続的に領域に割り付けられ，共用体は重なり合う領域を割り付けられます。

7.1 構 造 体

構造体は，連続的に割り付けられるメンバ・オブジェクトの集まりです。

(1) 構造体と構造体変数の宣言

構造体は，‘ struct ’のキーワードによって，構造体宣言リストおよび構造体変数を宣言します。構造体宣言リストにはタグ名と呼ばれる任意の名前を付けられ，以降このタグ名によって同一構造の構造体変数を宣言できます。

【構造体の宣言】

```
struct  タグ名 {  構造体宣言リスト  } 変数名;
```

次の例では，最初のstructでdataというタグ名を持つint型のcode，char型のname，addr，tel配列を宣言し，no1をその変数として宣言しています。次のstructではタグ名によりno1と同じ構造の構造体変数no2，no3，no4，no5を宣言しています。

【使 用 例】

```
struct data {
          int code;
          char name [12];
          char addr [50];
          char tel [12];
} no1;
struct data  no2, no3, no4, no5;
```

(2) 構造体宣言リスト

構造体宣言リストは，宣言する構造体型の構造を示します。構造体宣言リスト中の個々の要素をメンバといい，宣言されたメンバの順に領域が確保されます。次の“構造体宣言リストの例”では，変数a，配列b，二次元配列cの順に領域が確保されます。

メンバの型は，不完全型（大きさがわからない配列），関数型であってはなりません。したがって，構造体宣言リスト中に自分自身を含んではいけません。以上の型を除き，メンバはどんなオブジェクト型でも持てます。さらに，メンバをビット数で指定するビット・フィールドも指定できます。

ビット・フィールドは，変数のとる値が0か1の2値である場合，必要最小限のビット数の指定である1ビットを指定します。ビット・フィールドにより，必要最小限のビット数の指定で，複数のメンバを1個の整数領域に格納できます。

【構造体宣言リストの例】

```
int  a;
char b [7];
char c [5] [10];
```

【ビット・フィールド宣言の例】

```
struct  bf_tag {
          unsigned int a:2;
          unsigned int b:3;
          unsigned int c:1;
} bit_field;
```

ビット・フィールド

(3) 配列，ポインタ

構造体変数も他のオブジェクトと同様に配列にしたり，ポインタをとれます。構造体の配列では，配列の要素も構造体となります。

【構造体の配列】

構造体の配列宣言は他のオブジェクトと同じように行います。

```
struct data{
        char name [12];
        char addr [50];
        char tel [12];
};
struct data no [5];
```

【構造体のポインタ】

構造体のポインタは，ポインタが示す構造体の特徴を持ちます。つまり，構造体のポインタがインクリメントされるとポインタは構造体の大きさの分加算され，次の構造体を指すようになります。

次の例で‘ dt_p ’は，‘ struct data ’型の値に対するポインタであることを示しています。

ここで‘ dt_p ’をインクリメントすると‘ &no[1] ’と同じ値になります。

```
struct data no [5];
struct data *dt_p = no;
```

(4) 構造体メンバの参照方法

構造体メンバを参照するには構造体変数と変数へのポインタを使う2通りの方法があります。構造体変数による参照には‘．’演算子を，ポインタによる参照には‘－＞’演算子を使います。

【構造体変数による参照】

構造体変数によるメンバの参照には，‘．’演算子を使います。

```
struct data{
            char name [12];
            char addr [50];
            char tel [12];
}no[5] = {"NAME", "ADDR", "TEL"}, *data_ptr = no;

void main(){
            char  c;
            c = no[0]. name[1];
}
```

【ポインタによる参照】

ポインタ変数によるメンバの参照には，‘－＞’演算子を使います。

```
struct data{
            char name [12];
            char addr [50];
            char tel [12];
}no[5] = {"NAME", "ADDR", "TEL"}, *data_ptr = no;

void main(){
            char  c;
            data_ptr -> tel [3] = ‘ E ‘;
}
```

7.2 共 用 体

共用体は，同じ領域に割り付けられるメンバの集まりです。

(1) 共用体と共用体変数の宣言

共用体は，‘ union ’のキーワードによって，共用体宣言リストおよび共用体変数を宣言します。共用体宣言リストにはタグ名と呼ばれる任意の名前を付けられ，以降このタグ名によって同一構造の共用体変数を宣言できます。

【共用体の宣言】

```
union  タグ名 { 共用体宣言リスト } 変数名;
```

次の例では，最初のunionでdataというタグ名を持つchar型のname，addr，tel配列を宣言し，no1をその変数として宣言しています。次のunionではタグ名によりno1と同じ構造の共用体変数no2，no3，no4，no5を宣言しています。

```
union data {
        char name [12];
        char addr [50];
        char tel [12];
} no1;
union data  no2, no3, no4, no5;
```

(2) 共用体宣言リスト

共用体宣言リストは，宣言する共用体型の構造を示します。共用体宣言リスト中の個々の要素をメンバといい，宣言されたメンバは同じ領域に確保されていきます。次の“共用体宣言リストの例”では，メンバの中で一番大きな領域となる‘ c ’について領域が確保され，他のメンバは，新たに領域を採ることはせず，同じ領域を使用します。

メンバの型は，構造体宣言リストと同様，不完全型（大きさがわからない配列），関数型であってはなりません。以上の型を除き，メンバはどんなオブジェクト型でも持てます。

【共用体宣言リストの例】

```
int  a;
char b [7];
char c [5] [10];
```

(3) 配列，ポインタ

共用体変数も他のオブジェクトと同様，配列やポインタを採れます。

【共用体の配列】

共用体の配列宣言は他のオブジェクトと同じように行います。

```
union data{
      char  name[12];
      char  addr[50];
      char  tel[12];
};
union data  no[5];
```

【共用体のポインタ】

共用体のポインタは，ポインタが示す共用体の特徴を持ちます。つまり，共用体のポインタがインクリメントされるとポインタは共用体の大きさ分加算され，次の共用体を指すようになります。

次の例で‘ dt_p ’は，union data型の値に対するポインタです。

```
union data  no[5];
union data  *dt_p = no;
```

(4) 共用体メンバの参照方法

共用体メンバを参照するには，共用体変数と変数へのポインタを使う二通りの方法があります。共用体変数による参照には‘．’演算子を，ポインタによる参照には‘－＞’演算子を使います。

【共用体変数による参照】

共用体変数によるメンバの参照には，‘．’演算子を使います。

```
union data{
      char name[12];
      char addr[50];
      char tel[12];
}no[5] = { “NAME”, “ADDR”, “TEL” };

void main (void) {
      no[0].addr[10] = ‘B’ ;
           ⋮
}
```

【ポインタによる参照】

ポインタによるメンバの参照には，‘－＞’演算子を使います。

```
union data{
      char name[12];
      char addr[50];
      char tel[12];
} *data_ptr;

void main(void) {
      data_ptr -> name[1] = 'N' ;
           ⋮
}
```

第8章　外部定義

プログラムの中で，前処理のあとには外部宣言の並びがあります。これらは，関数の外で現れファイル有効範囲を持つので，‘ 外部 ’と呼ばれます。

外部オブジェクトに対し，識別子で名前付けを行う宣言，または関数のために記憶域の確保を行う宣言を，外部定義と呼びます。外部結合を持って宣言される識別子が式中（sizeof演算子の演算数の部分を除く）で使われる場合，プログラム全体のどこかに，その識別子に対する外部定義が1つ必要です。

```
#define  TRUE    1
#define  FALSE   0
#define  SIZE    200
void printf (char *, int) ;
void putchar (char c) ;

char    mark [SIZE+1];        ◄――― 外部オブジェクト定義

main()
{
    int  i, prime, k, count;

    count = 0;

    for ( i = 0 ; i <= SIZE ; i++)
         mark[i] = TRUE;
    for ( i = 0 ; i <= SIZE ; i++) {
         if (mark[i]) {
                prime = i + i + 3;
                printf("%d ", prime);
                count++;
                if((count%8) == 0) putchar('¥n');
                for ( k = i + prime ; k <= SIZE ; k += prime )
                   mark[k] = FALSE;
         }
    }
    printf("Total %d¥n", count);

loop1:
    goto loop1;
}
```

8.1　関数定義

関数の定義は，外部定義です。関数定義は，記憶域クラス指定子を省略した場合でも‘ extern ’で定義されたとみなされます。外部関数定義は，定義された関数が他のファイルから参照できることを示しています。たとえば，複数のファイルからなるプログラムにおいて，他のファイルにある関数を使用する場合，この関数は外部定義にします。

関数の記憶域クラス指定子は，externまたはstaticです。externと定義した場合，他の関数から参照できますが，staticで定義すると他のファイルから参照できません。

次の例でexternは記憶域クラス指定子，intは型指定子です。これらは，ディフォールトの値なので省略できます。‘ max(int a, int b) ’は，関数宣言子です。そして，‘ {return a >b ? a : b ; } ’が関数本体になります。

【関数定義の例】

```
extern int max(int a, int b)
{
     return a > b ? a : b;
}
```

この関数定義は，関数宣言中で仮引数の型を指定しているので，強制的に引数の型変換が行われます。仮引数に対して識別子並びの形を用いても記述できます。次にこの例を示します。

```
extern int max(a, b)
int a, b;
{
   return a > b ? a : b;
}
```

関数呼び出しの引数として，関数のアドレスを渡せます。関数名を式中に使うことによって，その関数のポインタが生成されます。

```
int f(void);
void main(){
     ⋮
    g(f);
}
```

この例では，関数fを指すポインタにより関数gに関数fを渡しています。関数gでは，たとえば次のように定義します。

```
void g (int(*funcp)(void))
{
     (*funcp)(); /*または funcp();*/
}
```

あるいは

```
void g (int func(void))
{
     func();    /*または (*func)();*/
}
```

8.2　外部オブジェクト定義

外部オブジェクト定義は，オブジェクトに対する識別子の宣言がファイル有効範囲および初期化子を持つものです。また，ファイル有効範囲を持つオブジェクトに対する識別子の宣言で，記憶域クラス指定子がなく初期化子を持たないか，記憶域クラス指定子がstaticである場合は仮の定義です。この場合，初期化子0のファイル有効範囲を持つ宣言とみなされます。

外部オブジェクト定義の例を次に示します。

【外部オブジェクト定義の例】

```
・int  i1 = 1 ; ・・・・・・・・・・・・・・・・ 外部結合を持つ定義
・static int  i2 = 2 ; ・・・・・・・・・ 内部結合を持つ定義
・extern int  i3 = 3 ; ・・・・・・・・・ 外部結合を持つ定義
・int  i4 ; ・・・・・・・・・・・・・・・・・・・・ 外部結合を持つ仮の定義
・static int  i5 ; ・・・・・・・・・・・・・ 内部結合を持つ仮の定義
・int  i1 ; ・・・・・・・・・・・・・・・・・・・・ 前のものを参照する正しい仮の定義
・int  i2 ; ・・・・・・・・・・・・・・・・・・・・ 結合規則違反
・int  i3 ; ・・・・・・・・・・・・・・・・・・・・ 前のものを参照する正しい仮の定義
・int  i4 ; ・・・・・・・・・・・・・・・・・・・・ 前のものを参照する正しい仮の定義
・int  i5 ; ・・・・・・・・・・・・・・・・・・・・ 結合規則違反
・extern int  i1 ; ・・・・・・・・・・・・・ 外部結合された前のオブジェクトの参照
・extern int  i2 ; ・・・・・・・・・・・・・ 内部結合された前のオブジェクトの参照
・extern int  i3 ; ・・・・・・・・・・・・・ 外部結合された前のオブジェクトの参照
・extern int  i4 ; ・・・・・・・・・・・・・ 外部結合された前のオブジェクトの参照
・extern int  i5 ; ・・・・・・・・・・・・・ 内部結合された前のオブジェクトの参照
```

〔メ　モ〕

第9章　前処理指令（コンパイラに対する指令）

前処理指令は，‘ # ’前処理字句から改行文字までの前処理字句の列です。

前処理字句列の間で使用できる空白文字は，スペースおよび水平タブだけです。

前処理指令は，ソース・ファイルのコンパイル前に行う処理を指定します。前処理には，ソース・ファイルの一部を条件によって処理またはスキップさせる指令や，他のソース・ファイルを取り込む指令，マクロに置き換える指令などがあります。次に，それぞれの前処理指令について説明します。

9. 1　条件付きコンパイル

条件付きコンパイルは，定数式の値によりソース・ファイルの一部分のコンパイルをスキップします。条件付きコンパイル指令で指定された定数式の値が偽（0）のとき，続く文はコンパイルされません。定数式には，sizeof演算子，キャスト，列挙定数を使用できません。

条件付きコンパイルの指令には‘ #if ’，‘ #elif ’，‘ #ifdef ’，‘ #ifndef ’，‘ #else ’，‘ #endif ’があります。

条件付きコンパイルでは，次の単項式を指定できます。

```
defined 識別子
または
defined （識別子）
```

この単項式は，識別子が前処理指令＃defineで定義されていれば1を返します。定義されていないか，定義を取り消してある場合は0を返します。

【使 用 例】

この例では，SYMが定義されているので，1を返し，#if～#endifの間をコンパイルします（#if～#endifの説明は，次頁以降の説明を参照してください）。

```
#define  SYM  0

#if  defined SYM
     ⋮
#endif
```

(1) #if指令

条件付きコンパイル #if

【機　　能】

定数式の値が偽であればソース・ファイルの一部分のコンパイルをスキップします。

【構　　文】

```
#if　定数式　改行　「グループ」
```

【使 用 例】

```
#if  FLAG == 0
     ⋮
#endif
```

【説　　明】

使用例では，‘ FLAG == 0 ’によって，後ろに続く文をコンパイルするかどうか判断しています。‘ FLAG ’の値が0以外であれば，#if指令と#endif指令間のプログラムはコンパイルされず，0の場合コンパイルされます。

（2）＃elif指令

条件付きコンパイル #elif

【機　　能】

この指令は，通常 #if指令の後ろにきます。#if指令の定数式が偽のとき，後ろに続く#elifの定数式が評価され，偽であれば #elifの後ろのプログラムはコンパイルをスキップされます。

【構　　成】

```
#elif　定数式　改行　「グループ」
```

【使 用 例】

```
#if   FLAG == 0
      ⋮
#elif  FLAG != 0
      ⋮
#endif
```

【説　　明】

使用例では，‘ FLAG ’の値によって，後ろに続く文をコンパイルするかどうか判断しています。‘ FLAG ’の値が0の場合，#if指令と#elif指令間のプログラムがコンパイルされます。そして，0以外の場合 #elif指令と#endif指令間のプログラムがコンパイルされます。

（3）＃ifdef指令

条件付きコンパイル #ifdef

【機　　能】

#ifdef指令は，#if指令の定数式がdefined識別子になったものです。

識別子が #define指令で定義されていれば，後ろに続くプログラムをコンパイルし，定義されていないか，定義を取り消してある場合にはコンパイルをスキップします。

【構　　文】

```
#ifdef　識別子　改行　「グループ」
```

【使 用 例】

```
#define ON
#ifdef ON
   ⋮
#endif
```

【説　　明】

使用例では，#define指令によって‘ ON ’が定義されているので，#ifdefと#endifの間のプログラムはコンパイルされます。‘ ON ’が定義されていなければ，#ifdefと#endifの間のプログラムはコンパイルされません。

（4）＃ifndef指令

条件付きコンパイル #ifndef

【機　　能】

#ifndef指令は，#if指令の定数式が !defined識別子となったものと同じです。この指令は識別子が前に定義されていれば，後ろに続くプログラムをコンパイルしません。

【構　　文】

```
#ifndef　識別子　改行　「グループ」
```

【使 用 例】

```
#define ON
#ifndef ON
        ⋮
#endif
```

【説　　明】

使用例では，＃define指令によって‘ ON ’が定義されているので，＃ifndefと＃endifの間のプログラムはコンパイルされません。‘ ON ’が定義されていなければ，＃ifndefと＃endifの間のプログラムはコンパイルされます。

(5) #else指令

条件付きコンパイル #else

【機　　能】

#else指令は，前にある条件付きコンパイル指令の識別子が偽の場合にのみ，後ろに続くプログラムをコンパイルします。#else指令の前にくる指令は，#if，#elif，#ifdef，#ifndef指令があります。

【構　　文】

```
#else　　改行　「グループ」
```

【使 用 例】

```
#define ON
#ifdef ON
  ⋮
#else
  ⋮
#endif
```

【説　　明】

使用例では，#define指令によって‘ ON ’が定義されているので，#ifdefと #elseの間のプログラムがコンパイルされます。‘ ON ’が定義されていなければ，#elseと #endifの間のプログラムがコンパイルされます。

（6）＃endif指令

条件付きコンパイル #endif

【機　　能】

#endif指令は，前にある条件付きコンパイル指令の有効範囲が終わったことを示します。

【構　　文】

```
#endif　改行
```

【使 用 例】

```
#define ON
#ifdef ON
  ⋮
#endif
```

【説　　明】

使用例で‘ ＃endif ’は，条件付きコンパイルifdef指令の有効範囲の終わりを示しています。

9. 2　ソース・ファイルの取り込み

前処理指令 #include**は指定したヘッダの検索を行い，**#include**指令とヘッダの内容全部を置き換えます。**#include**によるソース・ファイルの取り込みには**3**つの方法があります。**

- # include　**＜ファイル名＞**
- # include　**”ファイル名”**
- # include　**前処理字句の列**

#include**により取り込まれるソースの中で，**#include**指令が現れても良いですが，本コンパイラでは，**#include**指令のネストの制限があります。制限については，表**1－1　**本**C**コンパイラの最大性能**を参照してください。**

備考　**前処理字句列：**#define**指令で定義された文字列**

(1) #include　< > 指令

ソース・ファイルの取り込み　　#include <>

【機　　能】

指定されたヘッダを -iコンパイラ・オプションで指定したディレクトリ，INC78K環境変数で指定されているディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sから順に検索し，#include指令をヘッダの内容すべてに置き換えます。

【構　　文】

```
#include　＜　ファイル名　＞　改行
```

【使 用 例】

```
#include <stdio.h>
```

【説　　明】

INC78K環境変数により指定されたディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sの中から‘ stdio.h ’を検索し，前処理指令‘ #include <stdio.h> ’を‘ stdio.h ’の内容に置き換えます。

注意　上記のディレクトリは，インストール方法によって異なります。

(2) #include " " 指令

ソース・ファイルの取り込み #include ” ”

【機　　能】

この前処理指令によって取り込まれるソース・ファイルは，はじめにカレント・ディレクトリの中から検索します。そして，目的のファイルがないと次に -iコンパイラ・オプションで指定されたディレクトリ，INC78K環境変数で指定されているディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sから順に検索します。このようにして検索されたファイルは＃include指令と置き換えられます。

【構　　文】

```
#include  ”ファイル名 ”  改行
```

【使 用 例】

```
#include  "myprog.h"
```

【説　　明】

カレント・ディレクトリ，INC78K環境変数により指定されたディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sの中から‘ myprog.h ’を検索し，前処理指令‘ #include "myprog.h" ’を‘ myprog.h ’の内容に置き換えます。

注意　上記のディレクトリは，インストール方法によって異なります。

(3) #include　前処理字句列　指令

ソース・ファイルの取り込み　　#include　**前処理字句列**

【機　　能】

前処理字句列の置き換えによりヘッダ・ファイルが示されます。そして，ヘッダ・ファイルが検索され #include 指令と置き換わります。

【構　　文】

```
#include　前処理字句列　改行
```

【使 用 例】

```
#define   INCFILE  〝 myprog.h 〝
#include  INCFILE
```

【説　　明】

“ #include　前処理字句列　改行 ”によるソース・ファイルの取り込みでは，指定された前処理字句列がマクロ置換により＜ファイル名＞または　”ファイル名　”に置き換わらなければなりません。＜ファイル名＞に置き換わった場合，ソース・ファイルは -iコンパイラ・オプションにより指定されたディレクトリ，INC78K環境変数で指定されているディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sから順に検索します。”ファイル名　”の場合はカレント・ディレクトリから検索し，なければ -iコンパイラ・オプションにより指定されたディレクトリ，INC78K環境変数で指定されているディレクトリ，レジストリに登録されているディレクトリ¥NECTools32¥INC78K0Sから順に検索します。

注意　上記のディレクトリは，インストール方法によって異なります。

9.3　マクロ置換

マクロ置換は，識別子で指定した文字列（マクロ名）を“置換要素並び”に置き換えます。
マクロ置換には，オブジェクト形式と関数形式の2つがあります。

・オブジェクト形式

```
#define 識別子　置換要素並び　改行
```

・関数形式

```
#define 識別子（「識別子リスト」）　置換要素並び　改行
```

(1) 実引数置換

実引数の置き換えは，関数形式マクロの呼び出しの引数が識別されたあとに行われます。置換要素並びの仮引数に＃または＃＃前処理字句を前に付けずに，＃＃前処理字句が後ろに続かなければ，並び中に含まれるマクロがすべて展開されたあとに対応する引数に置き換えられます。

(2) ＃演算子

＃前処理字句は，対応する引数をchar文字列処理字句に置き換えます。置換要素並び中の仮引数の前にこれを付けると，対応する引数は文字または文字列になります。

(3) ＃＃演算子

＃＃前処理字句は，前後にある字句を結合します。結合は，次のマクロ展開が行われる前に実行され，＃＃前処理字句は削除されます。この結果，生成される字句にマクロ名があれば，さらにマクロ展開されます。

【＃＃演算子の例】

この例では，次のようにマクロ展開されます。

```
printf("x""1""=%d, x""2""=%s", x1, x2)；
```

さらに，char文字列が結合され次のようになります。

```
printf("x1=%d, x2=%s", x1, x2)；
```

```
#include <stdio.h>
#define  debug(s, t)  printf("x"#s"=%d, x"#t"=%s", x##s, x##t);

void main(){
      int  x1,  x2;
      debug (1, 2);
}
```

(4) 再走査とそれ以上の置き換え

マクロ置換によって置き換えられた結果の前処理字句，およびソース・ファイルの残りの前処理字句の中にマクロ名がある場合，マクロ置換を行います。現在置き換え中のマクロ名（ソース・ファイルの残りの前処理字句は含まない）が置換要素並びの走査中に見つけられても，置き換えられません。

(5) マクロ定義の有効範囲

マクロ定義は，対応する #undef指令が現れるまで置き換え続けます。

（6）＃define指令

マクロ置換 #define

【機　　能】

#define指令は，指定した識別子を置換要素並びに置き換えます。この指令以降の同じ識別子は置換要素並びに置き換えられます。

【構　　文】

```
#define  識別子  置換要素並び  改行
```

【使 用 例】

```
#define  PAI  3.1415
```

【説　　明】

使用例では，ソース・リスト中‘ PAI ’が現れると，すべて‘ 3.1415 ’に置き換えられます。

（7）#define（）指令

マクロ置換 #define（）

【機　　能】

関数形式のマクロ指令は，関数形式で指定した識別子を置換要素並びに置き換えます。この指令以降の同じ識別子は置換要素並びに置き換えられます。また，関数形式のマクロ置換では引数を含む置き換えができます。

【構　　文】

```
#define　識別子　（　「識別子リスト」　）　置換要素並び　改行
```

【使 用 例】

```
#define  F(n)  (n*n)
void main() {
    int i;
    i=F(2);
}
```

【説　　明】

使用例のF(2)は，#define指令により‘ (2*2) ’に置き換えられます。したがって，iの値は4となります。

関数形式のマクロは，関数定義と違い単なる文字の置き換えです。したがって，安全のために#define指令の置換要素並びは（ ）で囲んでおきます。

（8）＃undef指令

マクロ置換 #undef

【機　　能】

対応するマクロ置換指令を終わらせます。

【構　　文】

```
#undef　識別子　改行
```

【使 用 例】

```
#define F(n) (n*n)
   ⋮
#undef F
```

【説　　明】

使用例で‘ #undef ’は，前に指定されていた‘ ＃define F(n) (n*n) ’を無効にします。

9.4 行 制 御

行制御は，コンパイラがコンパイル時に使用する行番号を‘ #line ’によって指定された番号に置き換えます。また，文字列を指定した場合，コンパイラが持つソース・ファイル名を指定した文字列に置き換えます。

(1) 行番号を変更する場合

行番号を変更する場合，次のように指定します。数字列には，0および32767より大きい数は指定できません。

```
#line　数字列　改行
```

【使 用 例】

```
#line　10
```

(2) 行番号とファイル名を変更する場合

行番号とファイル名を変更する場合，次のように指定します。

```
#line　数字列　“文字列”　改行
```

【使 用 例】

```
#line　10　“file1.c”
```

(3) 前処理字句列を使用して変更する場合

上記の指定の他に，次のように指定できます。この場合には，指定した前処理字句列は，すべての置き換えのあとに，前記の2つの例のいずれかになるようにします。

```
#line　前処理字句列　改行
```

【使 用 例】

```
#define  LINE_NUM  100
#line   LINE_NUM
```

9. 5　#error前処理指令

#error前処理指令は，指定した前処理字句を含むメッセージを出力し，コンパイルを不成功に終わらせる指定です。この前処理により，コンパイルを終了させたい場合に使用します。

次のように指定します。

```
#error　“前処理字句列”　改行
```

【使 用 例】

この使用例では，本コンパイラが持つ，デバイスのシリーズを示すマクロ名“_ _K0S_ _”を使用しています。デバイスが78K/0Sシリーズであれば，#if～#else間のプログラムをコンパイルします。そうでない場合は，#else～#endif間のプログラムをコンパイルしますが，#error指令により，“not for 78K0S”というメッセージをエラーとして出力しコンパイルを終了します。

```
#if _ _K0S_ _
    ⋮
#else
#error “not for 78K0S”
    ⋮
#endif
```

9. 6　#pragma（プラグマ）指令

#pragma指令は，コンパイラに対し，コンパイラ定義の方法で動作することを指示する指令です。本コンパイラでは，78K/0Sシリーズ用のコードを生成するために#pragma指令が何種類か用意されています（#pragma指令の詳細は第11章　拡張機能を参照してください）。

【使 用 例】

この例では，#pragma NOP指令により，CソースでNOP命令を直接出力するように記述できます。

```
        #pragma NOP
```

9. 7　空指令（Null指令）

空指令は，コンパイラに対して何の影響も与えません。

#　改行

9.8　コンパイラ定義のマクロ名

コンパイラには，次のマクロ名があらかじめ定義されています。

_ _LINE_ _	カレント・ソース行の行番号（10進定数）
_ _FILE_ _	ソース・ファイル名（文字列リテラル）
_ _DATE_ _	ソース・ファイルのコンパイル日付 （‘ Mmm dd yyyy ’の形をした文字列リテラル）
_ _TIME_ _	ソース・ファイルのコンパイル時刻（‘ hh:mm:ss ’の形をした文字列リテラル）
_ _STDC_ _	ANSI[注]の規格に合致していることを意味する10進定数‘ 1 ’

注　ANSIとは，American National Standards Instituteの略称です。

これらのマクロ名およびdefined識別子は，#defineまたは #undef前処理指令の適用を受けてはなりません。また，すべてのコンパイラ定義のマクロ名は，アンダスコアではじめます。その後ろには，英大文字または2番目のアンダスコアが続きます。

本コンパイラでは，上記の他に応用製品の開発対象となるデバイスにより，デバイスのシリーズ名を示すマクロ名と，デバイス名を示すマクロ名を持ちます。これらは，ターゲット・デバイス用のオブジェクト・コードを出力するためにコンパイル時のオプション，またはCソース中のデバイス種別によって指定します。

・デバイスのシリーズ名を示すマクロ名

```
‘_ _K0S_ _’
```

・デバイス名を示すマクロ名

デバイス種別名の前に‘ _ _ ’，後ろに‘ _ ’を付与したもの

英字は大文字で記述してください。

例　_ _9026_　_ _9216_

備考　デバイス種別名は，-Cオプションで指定するものと同じです。デバイス種別名については，デバイス・ファイルに関する資料を参照してください。

また本Cコンパイラは，メモリ・モデルを示すマクロ名を持ちます。

・スタティック・モデル指定時に，

```
#define _ _STATIC_MODEL_ _   1
```

と定義します。

コンパイル時のデバイス種別の指定は，次のものをコマンド・ラインに追加することにより行います。

‘ -C**デバイス種別名** ’

```
例  cc78k0s  -c9216Y   prime.c
```

次のように，C**ソース・プログラムの先頭にデバイス種別を指定することにより，コンパイル時に指定する必要がなくなります。**

‘ #pragma PC (**種別**) ’

```
例  #pragma  PC (9216Y)
          ⋮
```

ただし，次のものは ‘ #pragma PC (**種別**) ’ の前に記述できます。

- コメント文
- 変数の定義または参照，および関数の定義または参照を生成しない前処理指令

〔メ　モ〕

第10章　ライブラリ関数

C言語には，外部（周辺）装置，機器との入出力を行う命令がありません。これは，C言語の設計者が，C言語の機能を最小限度に抑えるように設計したためです。しかし，実際にシステムを開発するには入出力操作が必要となります。このため，C言語には入出力操作を行うためのライブラリ関数が用意されています。

本Cコンパイラには，入出力，文字／メモリ操作，プログラム制御，数学関数等のライブラリ関数があります。本章では，本コンパイラが持つライブラリ関数について説明します。

10. 1　関数間のインタフェース

ライブラリ関数は，関数呼び出しで利用します。関数の呼び出しは，call命令により行います。引数はスタック，返り値はレジスタにより受け渡しが行われます。ただし，ノーマル・モデルで旧関数インタフェース対応オプション（-ZO）が指定されていない場合，可能であれば，第1引数もレジスタにより受け渡します。またスタティック・モデルは，すべての引数をレジスタにより受け渡します。

なお，-ZOオプションについては，CC78K0S Cコンパイラ　ユーザーズ・マニュアル　操作編（U14871J）　第5章　コンパイラ・オプションを参照してください。

10. 1. 1　引　　数

引数をスタックへ積むことと取り去ることは，呼び出す側が行います。呼び出される側はその値の参照だけを行います。ただし，引数をレジスタにより受け渡した場合は，呼び出された側が直接レジスタを参照し，必要に応じ，別のレジスタに引数の値のコピーを行います。また，関数呼び出しインタフェース自動パスカル関数化オプション-ZR指定時，引数をスタックにより受け渡す場合，引数をスタックから取り去ることは，呼び出された側が行います。

引数をスタック渡しする場合は，引数は最後から先頭に向かう順番でスタックに積まれます。

スタックに積まれる最小単位は16ビットであり，16ビットより大きい型は上位から順番に16ビット単位で積まれます。8ビットの型は，16ビットに拡張されます。

スタティック・モデルの場合，引数をすべてレジスタで受け渡します。

渡せる引数は，最大3引数，6バイトまでです。また，float, double, 構造体引数の受け渡しはサポートしません。

次に，引数の受け渡し一覧を示します。ノーマル・モデルで第2引数以降は，スタックにより渡されます。

標準ライブラリの関数インタフェース（引数の受け渡し，返り値の格納）は，通常関数と同じです。

表10－1　第1引数受け渡し一覧（ノーマル・モデル）

第1引数の型	受け渡し方法
1バイト，2バイト整数	AX
3バイト整数	AX, BC
4バイト整数	AX, BC
浮動小数点数（float型）	AX, BC
浮動小数点数（double型）	AX, BC
その他	スタック渡し

備考　上記の型で，1-4バイト整数には，構造体，共用体を含みます。

表10－2　引数受け渡し一覧（スタティック・モデル）

引数の型	第1引数	第2引数	第3引数
1バイト整数	A	B	H
2バイト整数	AX	BC	HL

備考　引数が4バイトの場合，AX, BCに割り当て，残りの引数をHLまたはHに割り当てます。

1-4バイト整数には，構造体と共用体は含まれません。

10. 1. 2　返 り 値

返り値は，最小単位を16ビットとしてレジスタBCからDEまで下位から16ビット単位で格納します。構造体を返す場合は，構造体の先頭アドレスをBCに格納します。ポインタを返す場合は，BCに格納します。次に，返り値格納一覧を示します。返り値の格納方法は，通常関数の場合と同じです。

表10－3　返り値格納一覧

(1) ノーマル・モデルの場合

返り値の型	格納方法
1ビット	CY
1バイト，2バイト整数	BC
4バイト整数	BC（下位），DE（上位）
浮動小数点数（float型）	BC（下位），DE（上位）
浮動小数点数（double型）	BC（下位），DE（上位）
構造体	返却する構造体を関数固有の領域にコピーし，アドレスをBCに格納します。
ポインタ	BC

(2) スタティック・モデルの場合

返り値の型	格納方法
1ビット	CY
1バイト整数	A
2バイト整数	AX
4バイト整数	AX（下位），BC（上位）
ポインタ	AX

10. 1. 3　個々のライブラリによる使用レジスタの保存

HL（ノーマル・モデルの場合），DE（スタティック・モデルの場合）を使用するライブラリは，それらの使用するレジスタをスタックに保存します。

saddr領域を使用するライブラリは，使用するsaddr領域をスタックに保存します。

また，ライブラリが使用するワーク・エリアは，スタック領域を使用します。

(1) -ZRオプションを指定しない場合

引数と返り値の受け渡し手順の例を次に示します。

呼び出す関数 “`long func (int a, long b, char *c);`”

① 引数をスタックに積む（呼び出す側）

c, bの上位16ビット，bの下位16ビットの順にスタックに積まれます。aはAXレジスタ渡しとなります。

② callにより命令funcを呼び出す（呼び出す側）

bの下位16ビットの次に戻り番地が積まれ，関数funcに制御が移ります。

③　関数内で使用するレジスタを保存する（呼び出される側）

HLを使用する場合HLがスタックに積まれます。

④　レジスタで渡された第1引数をスタックに積む（呼び出される側）

⑤　関数funcの処理を行い，返り値をレジスタに格納する（呼び出される側）

返り値‘ long ’の下位16ビットが BCに，上位16ビットがDEに格納されます。

⑥　格納した第1引数を復帰する（呼び出される側）

⑦　退避したレジスタを復帰する（呼び出される側）

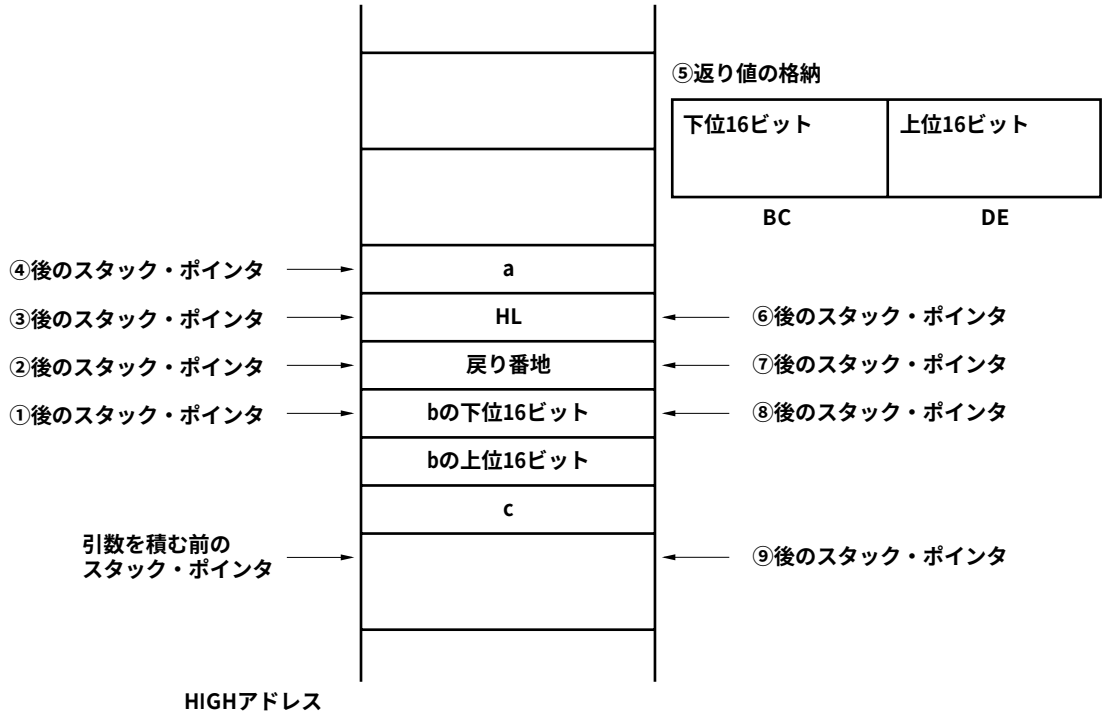
⑧　ret命令で呼び出した関数に制御を戻す（呼び出される側）

⑨　引数をスタックから取り除く（呼び出す側）

引数のバイト数（2バイト単位）がスタック・ポインタに加えられます。

図10－1の場合6が加えられます。

図10－1　関数呼び出し時のスタック領域（-ZR未指定時）



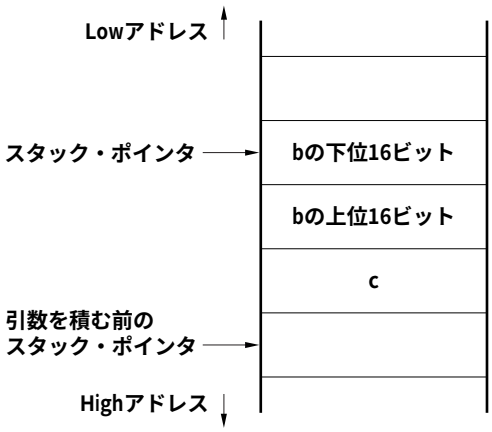
（2）-ZRオプションを指定する場合

-ZRオプションを指定した場合の引数と返り値の受け渡し手順の例を次に示します。

呼び出す関数 “long func (int a, long b, char *c);”

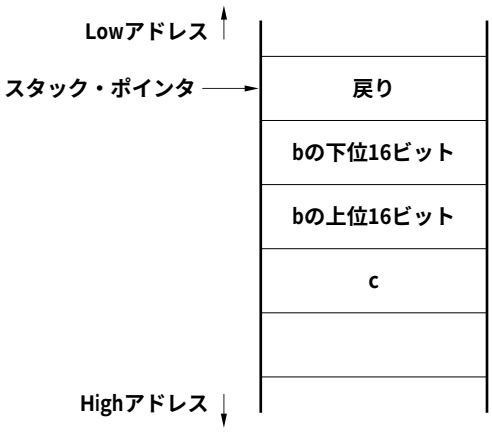
①　引数をスタックに積む（呼び出す側）

c, bの上位16ビット，bの下位16ビットの順にスタックに積まれます，aはAXレジスタ渡しとなります。

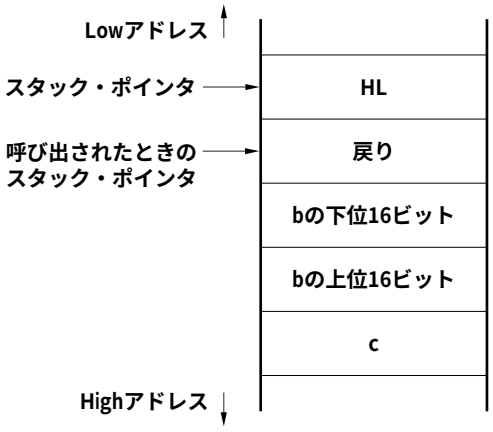


②　call命令によりfuncを呼び出す（呼び出す側）

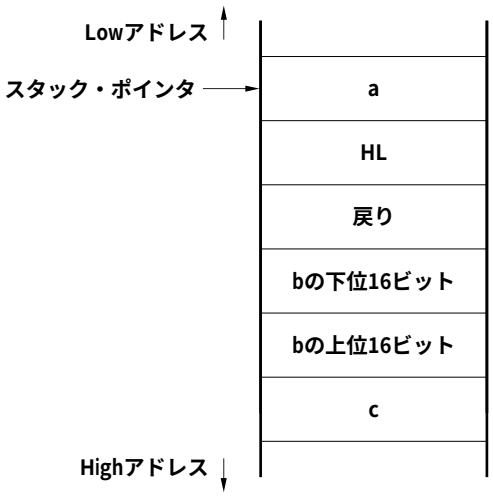
次の図に示すスタックの状態で関数funcに制御を渡します。



③　使用するレジスタを保存する（呼び出される側）



④　レジスタで呼び出された第1引数をスタックに積む

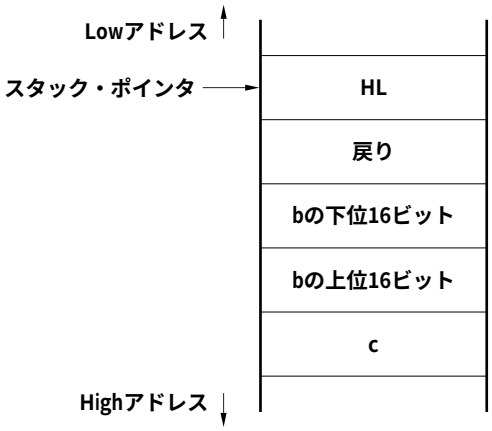


⑤　関数funcの処理を行い，返り値をレジスタに格納する（呼び出される側）

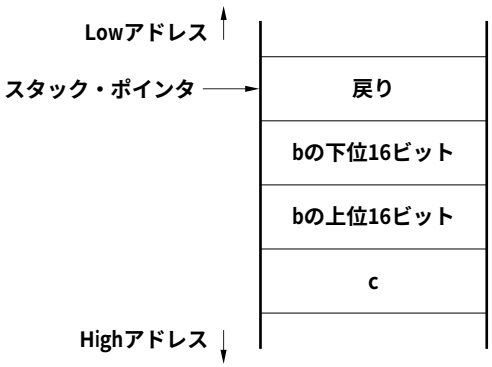
返り値（long）の下位16ビットをBC，上位16ビットをDEに格納します。

BC	DE
返り値の下位16ビット	返り値の上位16ビット

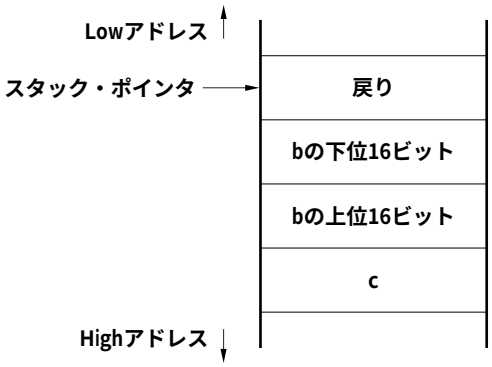
⑥　格納した第1引数を復帰する（呼び出される側）



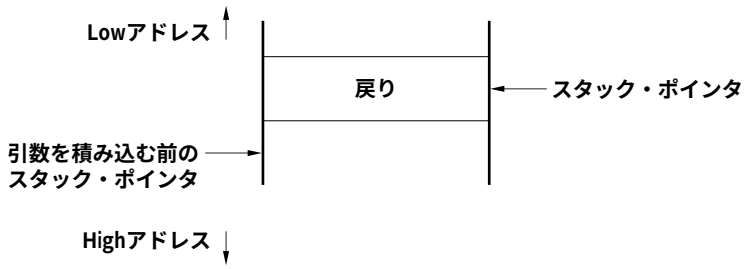
⑦　退避したレジスタを復帰する（呼び出される側）



⑧　戻り番地をレジスタに格納し，引数を積み込む前の位置までスタック・ポインタを移動して，引数をスタックから取り除きます（呼び出される側）



⑨　レジスタに格納しておいた戻り番地をスタックに積み直す（呼び出される側）



⑩　ret命令で呼び出す側の関数に制御を戻す（呼び出される側）

10. 2　ヘッダ・ファイル

本Cコンパイラには，13個のヘッダ・ファイルがあり，標準ライブラリ関数，型名，マクロ名を定義または宣言しています。

本Cコンパイラのヘッダ・ファイルを次に示します。

ctype.h	setjmp.h	stdarg.h	stdio.h	
stdlib.h	string.h	error.h	errno.h	
limits.h	stddef.h	math.h	float.h	assert.h

注意　メモリ・モデル（ノーマル・モデル／スタティック・モデル）により，サポートする関数が異なります。また，-ZI, -ZL**オプションにより正常動作する関数が異なります。**-ZI, -ZL**オプションの有無により正常動作しない関数については，「プロトタイプ宣言が行われていません」というワーニングが出力されます。**

(1) ctype.h

ctype.hは，文字・文字列関数を定義します。ctype.hでは，次の関数が定義されています。ただし，コンパイラ・オプション -ZA（ANSI規定外の機能を無効とし，ANSI規定の一部の機能を有効とするオプション）を指定した場合は，_toupper，_tolower の定義を行わず，代わりにtolow，toupの定義を行います。-ZAを指定しない場合は，tolow，toupの定義は行われません。また，オプションおよび指定モデルにより，宣言する関数が異なります。

表10ー4　ctype.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル				スタティック・モデル			
	なし	ZI	ZL	ZI ZL	なし	ZI	ZL	ZI ZL
isalnum	○	○	○	○	○	ー	○	ー
isalpha	○	○	○	○	○	ー	○	ー
iscntrl	○	○	○	○	○	ー	○	ー
isdigit	○	○	○	○	○	ー	○	ー
isgraph	○	○	○	○	○	ー	○	ー
islower	○	○	○	○	○	ー	○	ー
isprint	○	○	○	○	○	ー	○	ー
ispunct	○	○	○	○	○	ー	○	ー
isspace	○	○	○	○	○	ー	○	ー
isupper	○	○	○	○	○	ー	○	ー
isxdigit	○	○	○	○	○	ー	○	ー
tolower	○	○	○	○	○	ー	○	ー
toupper	○	○	○	○	○	ー	○	ー
isascii	○	○	○	○	○	ー	○	ー
toascii	○	○	○	○	○	ー	○	ー
_tolower	○	○	○	○	○	ー	○	ー
_toupper	○	○	○	○	○	ー	○	ー
tolow	○	○	○	○	○	ー	○	ー
toup	○	○	○	○	○	ー	○	ー

○：サポートする

ー：サポートしない

(2) setjmp.h

setjmp.hは，プログラム制御関数を定義します。setjmp.hでは次の関数が定義されています。なお，オプションおよび指定モデルにより，宣言する関数が異なります。

表10－5　setjmp.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル				スタティック・モデル			
	なし	ZI	ZL	ZI ZL	なし	ZI	ZL	ZI ZL
setjmp	○	○	○	○	○	－	○	－
longjmp	○	○	○	○	○	－	○	－

○：サポートする

－：サポートしない

setjmp.hでは，次のオブジェクトが宣言されています。

【int型配列の型‘ jmp_buf ’の宣言】

・ノーマル・モデルの場合

```
typedef int jmp_buf［11］;
```

・スタティック・モデルの場合

```
typedef int jmp_buf［3］;
```

(3) stdarg.h（ノーマル・モデルのみ）

stdarg.hは，特殊関数を定義します。stdarg.hでは，次の関数が定義されています。

表10－6　stdarg.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル			
	なし	ZI	ZL	ZI ZL
va_arg	○	○	○	○
va_start	△	△	△	△
va_end	○	○	○	○

○：サポートする

△：サポートするが動作に制限がある

stdarg.hでは，次のオブジェクトが定義されています。

【charへのポインタ型‘ va_list ’の宣言】

```
typedef char  *va_list;
```

（4）stdio.h

stdio.hは，入出力関数を定義します。stdio.hでは，次の関数が定義されています。

ただし，オプションおよび指定モデルにより，宣言する関数が異なります。

表10－7　stdio.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル				スタティック・モデル			
	なし	ZI	ZL	ZI ZL	なし	ZI	ZL	ZI ZL
sprintf	○	×	○	×	－	－	－	－
sscanf	○	×	○	×	－	－	－	－
printf	○	×	○	×	－	－	－	－
scanf	○	×	○	×	－	－	－	－
vprintf	○	×	○	×	－	－	－	－
vsprintf	○	×	○	×	－	－	－	－
getchar	○	○	○	○	○	－	○	－
gets	○	○	○	○	○	○	○	○
putchar	○	○	○	○	○	－	○	－
puts	○	○	○	○	○	－	○	－

○：サポートする

×：動作を保証しない

－：サポートしない

次のマクロ名を宣言しています。

```
#define   EOF    (-1)
#define   NULL   (void *)0
```

(5) stdlib.h

stdlib.hは，文字・文字列関数，メモリ関数，プログラム制御および数学関数，特殊関数を定義します。stdlib.hでは，次の関数が定義されています。

ただし，コンパイラ・オプション -ZA（ANSI規定外の機能を無効とし，ANSI規定の機能を有効とするオプション）を指定した場合は，brk, sbrk, itoa, ltoa, ultoaの定義は行わず，代わりにstrbrk, strsbrk, stritoa, strltoa, strultoaの定義を行います。-ZAを指定しない場合は，これらの関数の定義は行われません。

表10－8　stdlib.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル				スタティック・モデル			
	なし	ZI	ZL	ZI ZL	なし	ZI	ZL	ZI ZL
atoi	○	×	○	×	○	－	○	－
atol	○	○	×	×	－	－	－	－
strtol	○	○	×	×	－	－	－	－
strtoul	○	○	×	×	－	－	－	－
calloc	○	○	○	○	○	－	○	－
free	○	○	○	○	○	－	○	－
malloc	○	○	○	○	○	－	○	－
realloc	○	○	○	○	○	－	○	－
abort	○	○	○	○	○	○	○	○
atexit	○	○	○	○	○	－	○	－
exit	○	○	○	○	○	－	○	－
abs	○	○	○	○	○	－	○	－
div	○	－	○	－	－	－	－	－
labs	○	○	×	×	－	－	－	－
ldiv	○	○	－	－	－	－	－	－
brk	○	○	○	○	○	－	○	－
sbrk	○	○	○	○	○	－	○	－
atof	○	○	○	○	－	－	－	－
strtod	○	○	○	○	－	－	－	－
itoa	○	○	○	○	○	－	○	－
ltoa	○	○	－	－	－	－	－	－
ultoa	○	○	－	－	－	－	－	－
rand	○	×	○	×	－	－	－	－
srand	○	○	○	○	－	－	－	－
bsearch	○	○	○	○	－	－	－	－
qsort	○	○	○	○	－	－	－	－
strbrk	○	○	○	○	○	－	○	－
strsbrk	○	○	○	○	○	－	○	－
stritoa	○	○	○	○	○	－	○	－
strltoa	○	○	－	－	－	－	－	－
strultoa	○	○	－	－	－	－	－	－

○：サポートする

×：動作を保証しない

－：サポートしない

stdlib.hでは，次のオブジェクトが宣言されています。

【int型のメンバ‘ quot ’，‘ rem ’を持つ構造体型‘ div_t ’の宣言（スタティック・モデルを除く）】

```
typedef struct {
    int quot ;
    int rem ;
} div_t ;
```

【long int型のメンバ‘ quot ’，‘ rem ’を持つ構造体型‘ ldiv_t ’の宣言（スタティック・モデルおよびノーマル・モデルで-ZL指定時を除く）】

```
typedef struct {
    long int quot ;
    long int rem ;
} ldiv_t ;
```

【マクロ名‘ RAND_MAX ’の定義】

```
#define  RAND_MAX  32767
```

【マクロ名の宣言】

```
#define   EXIT_SUCCESS   0
#define   EXIT_FAILURE   1
```

（6）string.h

string.hは，文字・文字列関数，メモリ関数および特殊関数を定義します。

string.hでは，次の関数が定義されています。

ただし，オプションおよび指定モデルにより，宣言する関数が異なります。

表10－9　string.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル				スタティック・モデル			
	なし	ZI	ZL	ZI ZL	なし	ZI	ZL	ZI ZL
memcpy	○	○	○	○	○	－	○	－
memmove	○	○	○	○	○	－	○	－
strcpy	○	○	○	○	○	○	○	○
strncpy	○	○	○	○	○	－	○	－
strcat	○	○	○	○	○	○	○	○
strncat	○	○	○	○	○	－	○	－
memcmp	○	×	○	×	○	－	○	－
strcmp	○	×	○	×	○	－	○	－
strncmp	○	×	○	×	○	－	○	－
memchr	○	○	○	○	○	－	○	－
strchr	○	○	○	○	○	－	○	－
strcspn	○	×	○	×	○	－	○	－
strpbrk	○	○	○	○	○	○	○	○
strrchr	○	○	○	○	○	－	○	－
strspn	○	×	○	×	○	－	○	－
strstr	○	○	○	○	○	○	○	○
strtok	○	○	○	○	○	○	○	○
memset	○	○	○	○	○	－	○	－
strerror	○	○	○	○	○	－	○	－
strlen	○	×	○	×	○	－	○	－
strcoll	○	×	○	×	○	－	○	－
strxfrm	○	×	○	×	○	－	○	－

○：サポートする

×：動作を保証しない

－：サポートしない

（7）error.h

error.hは，errno.hをインクルードしています。

(8) errno.h

次のオブジェクトが定義されています。

【マクロ名‘EDOM’，‘ERANGE’，‘ENOMEM’の定義】

```
#define EDOM            1
#define ERANGE          2
#define ENOMEM          3
```

【volatile int型の外部変数‘errno’の宣言】

```
extern volatile int errno ;
```

(9) limits.h

limits.hでは，次のマクロ名が定義されています。

```
#define  CHAR_BIT               8
#define  CHAR_MAX               +127
#define  CHAR_MIN               -128
#define  INT_MAX                +32767
#define  INT_MIN                -32768
#define  LONG_MAX               +2147483647
#define  LONG_MIN               -2147483648

#define  SCHAR_MAX              +127
#define  SCHAR_MIN              -128
#define  SHRT_MAX               +32767
#define  SHRT_MIN               -32768
#define  UCHAR_MAX              255U
#define  UINT_MAX               65535U
#define  ULONG_MAX              4294967295U
#define  USHRT_MAX              65535U

#define  SINT_MAX               +32767
#define  SINT_MIN               -32768
#define  SSHRT_MAX              +32767
#define  SSHRT_MIN              -32768
```

ただし，修飾子なしcharをunsigned charとみなす -QUオプションを指定した場合は，コンパイラが宣言するマクロ _ _CHAR_UNSIGNED_ _により，CHAR_MAX，CHAR_MINを次のように宣言します。

```
#define  CHAR_MAX       (255U)
#define  CHAR_MIN       (0)
```

コンパイラ・オプションに-ZI（int型/short型をchar型，unsigned int型/unsigned short型をunsigned char型とみなす）オプションを指定した場合，コンパイラが宣言するマクロ _ _FROM_INT_TO_CHAR_ _により，INT_MAX, INT_MIN, SHRT_MAX, SHRT_MIN, SINT_MAX, SINT_MIN, SSHRT_MAX, SSHRT_MIN, UINT_MAX, USHRT_MAXを次のように宣言します。

```
#define  INT_MAX     CHAR_MAX
#define  INT_MIN     CHAR_MIN
#define  SHRT_MAX    CHAR_MAX
#define  SHRT_MIN    CHAR_MIN
#define  SINT_MAX    SCHAR_MAX
#define  SINT_MIN    SCHAR_MIN
#define  SSHRT_MAX   SCHAR_MAX
#define  SSHRT_MIN   SCHAR_MIN
#define  UINT_MAX    UCHAR_MAX
#define  USHRT_MAX   UCHAR_MIN
```

コンパイラ・オプションに-ZL（long型をint型，unsigned long型をunsigned int型とみなす）オプションを指定した場合，コンパイラが宣言するマクロ _ _FROM_LONG_TO_INT_ _ により，LONG_MAX, LONG_MIN, ULONG_MAXを次のように宣言します。

```
#define  LONG_MAX    (+32767)
#define  LONG_MIN    (-32768)
#define  ULONG_MAX   (65535U)
```

（10）stddef.h

stddef.hでは，次のオブジェクトが宣言，定義されています。

【int型の型‘ ptrdiff_t ’の宣言】

```
typedef int  ptrdiff_t;
```

【unsigned int型の型‘ size_t ’の宣言】

```
typedef unsigned int  size_t;
```

【マクロ名‘ NULL **’の定義】**

```
#define NULL  (void *)0;
```

【マクロ名‘ offsetof **’の定義】**

```
#define offsetof (type, member)  ((size_t)&(((type*)0) ー> member))
```

※ offsetof (**型，メンバ指示子**)

型size_t**を持つ汎整数定数式に展開し，その値は，（型が指示する）構造体の先頭から（メンバ指示子が指示する）構造体メンバまでのバイト単位でのオフセット値とします。**

メンバ指示子は，static　**型**　t**；という宣言があった場合，式＆（**t. **メンバ指示子）を評価した結果がアドレス定数になるものでなければなりません。指定されたメンバがビット・フィールドの場合，その動作は保証しません。**

（11）math.h（ノーマル・モデルのみ）

math.hでは，次の関数が定義されています。

表10－10　math.hの内容（1/2）

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル			
	なし	ZI	ZL	ZI ZL
acos	○	○	○	○
asin	○	○	○	○
atan	○	○	○	○
atan2	○	○	○	○
cos	○	○	○	○
sin	○	○	○	○
tan	○	○	○	○
cosh	○	○	○	○
sinh	○	○	○	○
tanh	○	○	○	○
exp	○	○	○	○
frexp	○	○	○	○
ldexp	○	○	○	○
log	○	○	○	○
log10	○	○	○	○
modf	○	○	○	○
pow	○	○	○	○
sqrt	○	○	○	○
ceil	○	○	○	○
fabs	○	○	○	○
floor	○	○	○	○
fmod	○	○	○	○
matherr	○	－	○	－
acosf	○	○	○	○
asinf	○	○	○	○
atanf	○	○	○	○
atan2f	○	○	○	○
cosf	○	○	○	○
sinf	○	○	○	○
tanf	○	○	○	○
coshf	○	○	○	○
sinhf	○	○	○	○
tanhf	○	○	○	○
expf	○	○	○	○
frexpf	○	○	○	○
ldexpf	○	○	○	○

○：サポートする

－：サポートしない

表10－10　math.hの内容（2/2）

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル			
	なし	ZI	ZL	ZI ZL
logf	○	○	○	○
log10f	○	○	○	○
modff	○	○	○	○
powf	○	○	○	○
sqrtf	○	○	○	○
ceilf	○	○	○	○
fabsf	○	○	○	○
floorf	○	○	○	○
fmodf	○	○	○	○

○：サポートする

次のオブジェクトが，定義されています。

【マクロ名‘HUGE_VAL’の定義】

```
#define  HUGE_VAL  DBL_MAX
```

（12）float.h

float.hでは，次のオブジェクトが定義されています。

double型の大きさが32ビットのときコンパイラが宣言するマクロ，_ _DOUBLE_IS_32BITS_ _により，定義するマクロを切り分けます。

```
#ifndef  _FLOAT_H

#define FLT_ROUNDS                  1
#define FLT_RADIX                   2

#ifdef _ _DOUBLE_IS_32BITS_ _
#define FLT_MANT_DIG                24
#define DBL_MANT_DIG                24
#define LDBL_MANT_DIG               24

#define FLT_DIG                     6
#define DBL_DIG                     6
#define LDBL_DIG                    6

#define FLT_MIN_EXP                 -125
#define DBL_MIN_EXP                 -125
#define LDBL_MIN_EXP                -125
```

```
#define FLT_MIN_10_EXP              -37
#define DBL_MIN_10_EXP              -37
#define LDBL_MIN_10_EXP             -37

#define FLT_MAX_EXP                 +128
#define DBL_MAX_EXP                 +128
#define LDBL_MAX_EXP                +128

#define FLT_MAX_10_EXP              +38
#define DBL_MAX_10_EXP              +38
#define LDBL_MAX_10_EXP             +38

#define FLT_MAX                     3.40282347E+38F
#define DBL_MAX                     3.40282347E+38F
#define LDBL_MAX                    3.40282347E+38F

#define FLT_EPSILON                 1.19209290E-07F
#define DBL_EPSILON                 1.19209290E-07F
#define LDBL_EPSILON                1.19209290E-07F

#define FLT_MIN                     1.17549435E-38F
#define DBL_MIN                     1.17549435E-38F
#define LDBL_MIN                    1.17549435E-38F

#else   /* _ _DOUBLE_IS_32BITS_ _ */
#define FLT_MANT_DIG                24
#define DBL_MANT_DIG                53
#define LDBL_MANT_DIG               53

#define FLT_DIG                     6
#define DBL_DIG                     15
#define LDBL_DIG                    15

#define FLT_MIN_EXP                 -125
#define DBL_MIN_EXP                 -1021
#define LDBL_MIN_EXP                -1021

#define FLT_MIN_10_EXP              -37
#define DBL_MIN_10_EXP              -307
#define LDBL_MIN_10_EXP             -307
```

```
#define FLT_MAX_EXP                 +128
#define DBL_MAX_EXP                 +1024
#define LDBL_MAX_EXP                +1024

#define FLT_MAX_10_EXP              +38
#define DBL_MAX_10_EXP              +308
#define LDBL_MAX_10_EXP             +308

#define FLT_MAX                     3.40282347E+38F
#define DBL_MAX                     1.7976931348623157E+308
#define LDBL_MAX                    1.7976931348623157E+308

#define FLT_EPSILON                 1.19209290E-07F
#define DBL_EPSILON                 2.2204460492503131E-016
#define LDBL_EPSILON                2.2204460492503131E-016

#define FLT_MIN                     1.17549435E-38F
#define DBL_MIN                     2.225073858507201E-308
#define LDBL_MIN                    2.225073858507201E-308
#endif   /* _ _DOUBLE_IS_32BITS_ _ */

#define   _FLOAT_H
#endif   /* !_FLOAT_H */
```

（13）assert.h（ノーマル・モデルのみ）

表10－11　assert.hの内容

-ZI, -ZL指定の有無 / 関数名	ノーマル・モデル			
	なし	ZI	ZL	ZI ZL
_ _assertfail	○	○	○	○

○：サポートする

assert.hでは，次のオブジェクトが定義されています。

```
#ifdef  NDEBUG
#define assert (p)      ((void)0)
#else
extern int  _ _assertfail (char *_ _msg, char *_ _cond, char *_ _file, int _ _line);
#define assert(p)   ((p) ? (void) 0 : (void)_ _assertfail ( ¥
        "Assertion failed: %s, file %s, line %d¥n", #p, _ _FILE_ _, _ _LINE_ _ ))
#endif  /* NDEBUG */
```

ただし，assert.hヘッダ・ファイルは，assert.hヘッダ・ファイルでは定義しないもう一つのマクロNDEBUGを参照し，ソース・ファイル中にassert.hを取り込む時点で，NDEBUGがマクロとして定義されている場合，assertマクロを単に，

```
#define assert(p)  ((void)0)
```

と宣言し，_ _assertfailの定義も行いません。

10. 3　リエントラント性（ノーマル・モデルのみ）

リエントラントとは，あるプログラムから呼び出されている関数が，続けて他のプログラムによって呼び出し可能である状態です。

本コンパイラの標準ライブラリは，リエントラント性を考慮し，静的領域を使用していません。したがって，関数が使用する記憶域のデータが，他プログラムからの呼び出しによって破壊されることはありません。

ただし，次の（1）～（3）の関数は，リエントラントではありませんので注意してください。

（1）リエントラント化できない関数

setjmp, longjmp, atexit, exit

（2）スタートアップ・ルーチンで確保している領域を使用する関数

div, ldiv, brk, sbrk, rand, srand, strtok

（3）浮動小数点を扱う関数

sprintf, sscanf, printf, scanf, vprintf, vsprintf [注]

atof, strtod, **数学関数すべて**

注　sprintf, sscanf, printf, scanf, vprintf, vsprintf**のうち，浮動小数点未対応のものは，リエントラントです。**

10. 4　標準ライブラリ関数

本Cコンパイラの標準ライブラリ関数を機能別に分けて説明します。標準ライブラリは，すべて-ZFオプション指定時もサポートしています。

- 項番（1－x）····· 文字・文字列関数
- 項番（2－x）····· プログラム制御関数
- 項番（3－x）····· 特殊関数
- 項番（4－x）····· 入出力関数
- 項番（5－x）····· ユーティリティ関数
- 項番（6－x）····· 文字列／メモリ関数
- 項番（7－x）····· 数学関数
- 項番（8－x）····· 診断関数

1ー1　is〜　　文字・文字列関数

【機　　能】

・is〜は文字種の判定を行います。

【ヘッダ・ファイル】

・すべてctype.h

【関数プロトタイプ】

・`int  is〜(int c);`

関 数 名	引　　数	返 り 値
is〜	c･･･判定する文字	文字cが目的の文字である場合･･･ 1 文字cが目的の文字でない場合･･･ 0

【説　　明】

関 数 名	範　　囲
isalpha	cが英文字（A-Z，a-z）であるかを判定します。
isupper	cが英大文字（A-Z）であるかを判定します。
islower	cが英小文字（a-z）であるかを判定します。
isdigit	cが数字（0-9）であるかを判定します。
isalnum	cが英数字（0-9，A-Z，a-z）であるかを判定します。
isxdigit	cが16進数字（0-9，A-F，a-f）であるかを判定します。
isspace	cが空白文字（空白，タブ，復帰，改行，垂直，タブ，改ページ）であるかを判定します。
ispunct	cが空白文字と英数字以外の表示可能文字であるかを判定します。
isprint	cが表示可能文字であるかを判定します。
isgraph	cが空白以外の表示可能文字であるかを判定します。
iscntrl	cがコントロール文字であるかを判定します。
isascii	cがASCII文字であるかを判定します。

1ー2　toupper
　　tolower　　文字・文字列関数

【機　　能】

・文字種の変換を行います。
・toupperは，英小文字を英大文字に変換します。
・tolowerは，英大文字を英小文字に変換します。

【ヘッダ・ファイル】

・ctype.h

【関数プロトタイプ】

・`int  to〜(int c);`

関 数 名	引　　数	返 り 値
toupper tolower	c･･･変換される文字	cが変換可能な場合･･･ 文字cに対応した変換後の文字 cが変換不可能な場合･･･c

【説　　明】

toupper

・toupperは，引数が英小文字であることを確認したうえで英大文字に変換します。

tolower

・tolowerは，引数が英大文字であることを確認したうえで英小文字に変換します。

1ー3　toascii　　文字・文字列関数

【機　　能】

・toasciiは，ASCIIコードへの変換を行います。

【ヘッダ・ファイル】

・ctype.h

【関数プロトタイプ】

・`int  toascii (int c);`

関 数 名	引　　数	返 り 値
toascii	c･･･変換される文字	cのASCIIコードの範囲以外のビットを0にした値

【説　　明】

・cのASCIIコードに変換します。ASCIIコードの範囲（ビット0-6）以外のビット（ビット7-15）は0にします。

1ー4　_toupper／toup
　　_tolower／tolow　　**文字・文字列関数**

【機　　能】

- _toupper／toup**は，c**から‘ a ’を引き‘ A ’を加えます。　　a：英小文字
- _tolower／tolow**は，c**から‘ A ’を引き‘ a ’を加えます。　　A：英大文字

（ _toupper**と**toup，_tolower**と**tolow**はまったく同じです。）**

備考　a：英小文字，A：英大文字

【ヘッダ・ファイル】

- ctype.h

【関数プロトタイプ】

- `int _to〜(int c);`

関 数 名	引　　数	返 り 値
_toupper toup	c･･･変換される文字	c**から‘ a ’を引き‘ A ’を加えた値**
_tolower tolow		c**から‘ A ’を引き‘ a ’を加えた値**

備考　a：英小文字，A：英大文字

【説　　明】

_toupper

- _toupper**は，**toupper**と似ていますが，引数が英小文字であることを確認しません。**

_tolower

- _tolower**は，**tolower**と似ていますが，引数が英大文字であることを確認しません。**

2ー1　setjmp
　　longjmp　　プログラム制御関数

【機　　能】

- setjmpは，呼び出し時の環境をセーブします。
- longjmpは，setjmpでセーブされた環境を復帰します。

【ヘッダ・ファイル】

- setjmp.h

【関数プロトタイプ】

- `int setjmp (jmp_buf env);`
- `void longjmp (jmp_buf env, int val);`

関 数 名	引　　数	返 り 値
setjmp	env･･･環境をセーブする配列	直接呼び出された場合･･･ 0 対応するlongjmpの呼び出しから返る場合･･･ 対応するlongjmpの呼び出し時のvalの値，ただしvalが0の場合は1
longjmp	env･･･setjmpでセーブした環境の配列 val･･･setjmpに返す値	envに環境をセーブしたsetjmpの次に実行を移すのでlongjmpには戻りません。

【説　　明】

setjmp

- setjmpは，直接呼び出された場合，HLレジスタ，レジスタ変数として使用するsaddr領域，SPおよび関数のリターン・アドレスを env にセーブし，0を返します。

longjmp

- longjmpは，envに保存された環境（HLレジスタおよびレジスタ変数として使用するsaddr領域，SP）を復帰し，対応するsetjmpがval（ただしvalが0の場合は1）を返したかのようにプログラムの実行が続きます。

3ー1　va_start（ノーマル・モデルのみ）
va_arg（ノーマル・モデルのみ）
va_end（ノーマル・モデルのみ）　特殊関数

【機　　能】

- va_startは，可変個の引数の処理のための設定を行います（マクロ）。
- va_argは，可変個の引数の処理を行います（マクロ）。
- va_endは，可変個の引数の処理の終了を知らせます（マクロ）。

【ヘッダ・ファイル】

- stdarg.h

【関数プロトタイプ】

- `void  va_start (va_list ap, parmN);`
- `type  va_arg (va_list ap, type);`
- `void  va_end (va_list ap);`

関 数 名	引　　数	返 り 値
va_start	ap･･･va_arg，va_endで使えるように初期化される変数 parmN･･･可変引数の1個前の引数	なし
va_arg	ap･･･引数リストの処理のための変数 type･･･可変引数の該当箇所をポイントするための型（typeは可変長の型で，たとえば va_arg(va_list ap, int)と記述すればint型，va_arg(va_list ap，long)と記述すればlong型となる）	正常の場合･･･可変引数の該当箇所の値 apが空ポインタの場合･･･ 0
va_end	ap･･･可変個の引数の処理のための変数	なし

3ー1　va_start
　　va_arg
　　va_end　　特殊関数

【説　　明】

va_start

・va_start**で引数**ap**は，**va_list**型（**char * **型）のオブジェクトです。**

・ap**に**parmN**の次の引数を指すポインタを格納します。**

・parmN**は，関数定義上での右端のパラメータの名前です。**

・parmN**がレジスタ記憶クラスで宣言されている場合は，正常動作は保証しません。**

va_arg

・va_arg**で引数**ap**は，**va_start**で初期化された**va_list**型の**ap**と同じでなければなりません（それ以外の正常動作は保証しません）。**

・**可変引数の該当箇所（**va_start**の直後は可変引数の先頭，その後は**va_arg**ごとに進めます）の値を**type**型で返します。**

・ap**が空ポインタの場合は，**type**型の**0**を返します。**

va_end

・va_end**は，すべての可変引数を処理し終わったことをマクロ系に知らせるために，**ap**に空ポインタをセットします。**

4ー1　sprintf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・sprintfは，フォーマットに従ってデータを文字列に書きます。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int sprintf (char * s, const char * format,  …);`

関 数 名	引　　数	返 り 値
sprintf	s･･･出力する文字列へのポインタ format･･･出力変換仕様を示す文字列へのポインタ ･･･ ･･･変換される0個以上の引数	sに書かれた文字数（終端のナル文字は数えません）

【説　　明】

・書式に対して実引数が不足しているときの動作は保証しません。実引数が残っているにもかかわらず書式が尽きてしまう場合，余分の実引数は評価するだけで無視します。

・formatで指定された出力変換仕様に従い，formatの後ろに続く（0個以上の）引数を変換してsで示された文字列に書き出します。

・出力変換仕様は，0個以上の指令です。通常の文字（%で始まる変換仕様以外）はそのまま文字列sに出力します。変換仕様は（0個以上の）後続の引数を取り出し，変換して文字列sに出力します。

・各変換仕様は%で始まり，次のものが順に続きます（変換指定が不正な場合には，その文字を出力します。この際，フラグと最小フィールド幅は有効です）。

　・0個以上のフラグ（後述）は変換仕様の意味を修飾します。

　・最小フィールド幅を指定するオプションの10進整数

もし変換後の幅が，このフィールド幅よりも小さい場合，左にパッドを入れます（左寄せのフラグ（－）が指定されていれば右にパッドが入ります）。パッドは，フィールド幅整数が0で始まり右寄せの場合は0，その他はスペース文字です。変換後の幅がフィールド幅より多くても切り捨てません。

4ー1　sprintf　　入出力関数

・オプションの精度指定（．整数）

d, i, o, u, x, X変換の場合は，最小の桁数を指定します。s変換では最大文字数を指定します。e, Eおよびf変換については小数点文字の後ろに出力すべき桁数を，gおよびG変換については最大の有効桁数を指定します。

この精度指定は，（．整数）の形をしています。整数部が省略されたときは0とみなします。

この精度指定から生ずるパッドの量は，フィールド幅指定のパッドに優先します。

・オプションのh，lまたはL

hは引き続くd, i, o, u, x, X変換をshort intまたはunsigned short intに対して行うように指定します。また，hは引き続くn変換をshort intへのポインタに対して行うように指定します。

l は引き続くd, i, o, u, x, X変換をlong intまたはunsigned long intに対して行うように指定します。また，lは引き続くn変換をlong intへのポインタに対して行うように指定します。

その他の変換に対してはh, lまたはLは無視します。

・変換を指定する文字（後述の変換指定）

フィールド幅または精度指定は，整数文字列の代わりに * を指定できます。このとき，int引数が整数値を与えます（変換される引数の前）。この結果生じる負のフィールド幅は，－フラグのあとに正のフィールドが続いたものと解釈します。負の精度は無視されます。

4ー1　sprintf　　入出力関数

◇フラグは次のとおりです。

- ー　　・・・変換した結果をフィールド内で左寄せします。
- ＋　　・・・符号付き変換の結果に＋またはーの符号を付けます。
- スペース ・・・符号付き変換の結果に符号がない場合，スペースを頭に付けます。スペースと＋フラグを同時に指定するとスペース・フラグは無視されます。
- ＃　　・・・結果を‘代替形式’に変換します。
  o変換では，最初の桁が0になるように精度を上げます。x, X変換では，非ゼロの結果には0x（または0X）が頭に付きます。
  e, E, f変換では，すべての場合に出力値に強制的に小数点が入ります（＃なしのディフォールトでは，後続の数値がある場合にのみ小数点が表示されます）。
  g, G変換ではすべての場合に出力値に強制的に小数点が入り，後続する0の切り捨てを許しません（＃なしのディフォールトでは，後続の数値がある場合にのみ小数点が表示されます。後続の0は切り捨てられます）。
  その他の変換では，＃フラグを無視します。

◇変換指定は次のとおりです。

- d, i, o, u, x, X・・・・・・・・ int引数を符号付き10進（dまたはi），無符号8進（o），無符号10進（u），無符号16進（xまたはX）表記に変換します。x変換はa〜f，X変換はA〜Fの文字を16進文字として使います。

精度指定は結果の最小桁数を指定し，結果が足りないときには頭の不足分の0を付けます。精度指定の省略時は1とします。0を精度指定0で変換すると何も現れません。

- f・・・・・double引数を［ー］dddd.ddddの形式を持つ符号付きの値として変換します。
  ddddは，1個または複数の10進数です。小数点の前の桁数はその数の絶対値によって決定され，小数点のあとの桁数は要求された精度によって決定されます。精度が省略された場合は，精度を6として解釈します。
- e ・・・・double引数を［ー］d.dddd　e　［sign］　ddd の形式を持つ符号対の値として変換します。dは1個の10進数，ddddは1個または複数の10進数です。dddは正確に3桁の10進数で，signは＋またはーです。精度が省略された場合は，精度を6として解釈します。
- E ・・・・指数の前に付くのがeではなくEである点を除いて，eのフォーマットと同様です。
- g ・・・・double引数をfまたはeのフォーマットのうち，指定された精度に基づいて変換したときに，より短くなる方式を用います。eフォーマットは，値の指数部が，ー4より小さいか精度で指定された数よりも大きい場合にのみ用います。
  あとに続く0は切り捨てられ，小数点は1個または複数の数字が続く場合にのみ表示されます。
- G・・・・指数の前にあるのがeではなくEである点を除いて，gのフォーマットと同様です。
- c ・・・・int引数をunsigned charに変換し，結果の文字が書かれます。

4ー1　sprintf　　入出力関数

・s ‥‥引数は文字列へのポインタで，その文字列からの各文字は終端のナル文字（出力には含みません）まで書かれます。
精度指定があれば，それより多くの文字は書きません。
精度が指定されない場合または精度が配列の大きさよりも大きい場合，配列はナル文字を含まなければなりません。

・p ‥‥引数はvoidへのポインタの値を無符号16進4桁で表記（4桁未満は頭に0を付けます）します。ラージ・モデルは，無符号16進8桁で表記（上位2桁0でパディングし，6桁未満は頭に0を付ける）します。精度指定は無視します。

・n ‥‥引数は整数へのポインタで，そこに対してこれまでに文字列sに書き出した文字数を入れます。変換は行いません。

・% ‥‥%が書かれます。引数は変換しません（フラグと最小フィールド幅は有効です）。

・無効な変換指定子に対する動作は，保証しません。

・実引数が共用体または集成体であるか，またはそれを指すポインタである場合（%ｓ変換のときの文字型配列または%p変換のときのポインタを除きます），動作は保証しません。

・フィールド幅が存在しないとき，または小さいときでも，変換結果を切り捨てることはありません。すなわち，変換結果の文字数がフィールド幅より大きい場合，その変換結果を含む幅までフィールドを拡張します。

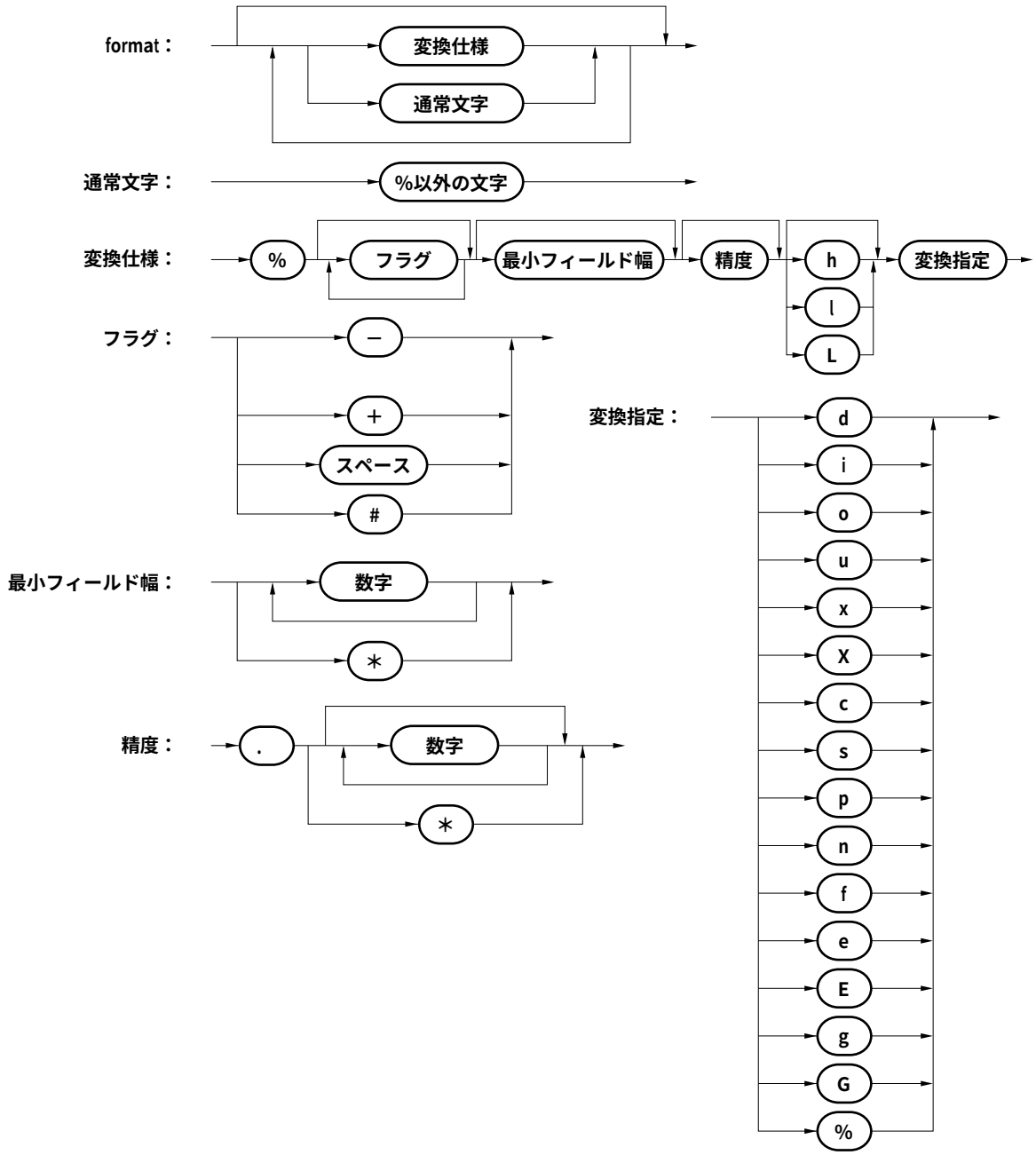
・%f，%e，%E，%g，%G変換時の特別の出力文字列の形式を次に示します。

非数　→　“（NaN）”
＋∞　→　“（＋INF）”
－∞　→　“（－INF）”

文字列sの末尾にナル文字（返り値のカウントには含まない）を書きます。
formatの構文図を図10ー2に示します。

4ー1　sprintf　　**入出力関数**

図10ー2　出力formatの構文図



4ー2　sscanf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・入力文字列からフォーマットに従ってデータを読みます。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

```
・int  sscanf(const char *s, const char *format, ...);
```

関 数 名	引　　数	返 り 値
sscanf	s ･･･ 入力文字列へのポインタ format ･･･ 入力変換仕様を示す文字列へのポインタ ･･･ ･･･ 変換された値を入れるオブジェクトへのポインタ（0個以上の）引数	文字列sが空の場合 ･･･ －1 文字列sが空でない場合･･･代入された入力項目の数

【説　　明】

・sが指す文字列から入力します。formatが指す文字列により許される入力列を指定します。
formatが以降の引数をオブジェクトへのポインタとして用います。formatは入力列から，どのように変換するかを指定します。

・formatに対して引数が足りない場合の正常動作は保証しません。過剰な引数の場合，式の評価は行いますが入力はされません。

・formatは0以上の指令からなります。指令は次のとおりです。

（1）1個以上の空白文字（isspaceが真となる文字）

（2）通常文字（%以外）

（3）変換指示

4ー2　sscanf　　入出力関数

・変換指示は%から始まり，%の後ろに次のものが順に続きます。
　・オプションの代入禁止文字 *（引数へは代入しないことを示します）
　・オプションの最大フィールド幅を指定する10進整数（0の場合，指定のないものとします）
　・オプションのh，lまたはL（受信する側のオブジェクトのサイズを示します）
　変換指示子d, i, n, o, xにhが先行すれば，引数はintでなくshort intへのポインタです。lがこれらに先行した場合はlong intへのポインタです。
　同様に変換指示子uにhが先行すれば，引数はunsigned short intへのポインタです。lが先行した場合は，unsigned long intへのポインタです。
　変換指示子e, E, f, g, Gに l が先行すれば，引数はdoubleへのポインタです（ l なしのディフォールトでは引数はfloatへのポインタ）。またLが先行した場合，無視します。

備考　変換指示子：対応する変換の種類を示す文字（後述）

sscanfはformat中の指令を順に実行します。指令が失敗すればsscanfは戻ります。

(1) 空白文字からなる指令は，最初の非空白文字（これは読み込みません）までか，読む文字がなくなるまで入力を読むことで実行されます。空白文字指令は非空白文字が発見できなければ失敗します。

(2) 通常文字の指令は，次の文字を読むことで実行されます。その文字と指令文字が異なるとき，指令は失敗します。

(3) 変換指示の指令は，各変換指示子（後述）ごとに一致する入力列の集合を定義します。変換指示は次のステップ順に実行されます。

　・入力空白文字（isspaceで指定される）はスキップされます。ただし，変換指示子が［，c，nの場合を除きます。

・入力項目が文字列sから読まれます。ただしn変換指示子のときは除きます。入力項目とは，変換指示子で指示される文字列の最初の部分列のうち，最長の入力列（ただし，最大フィールド幅が指定されている場合は，その長さで打ち切ります）と定義します。入力項目の次の文字は，まだ読まれていないとみなします。入力項目の長さが0のとき，指令の実行は失敗します。
・%変換指示子を除いて，入力項目（%n指令の場合は，入力文字数）が変換指示子により定まる型に変換されます。入力項目が指示する形式と合わない場合は指令の実行は失敗します。
　* によって入力禁止が指定されないかぎり，変換の結果はformatに続く変換結果を受け取っていない最初の引数に指されるオブジェクトにストアされます。

4ー2　sscanf　　入出力関数

変換指示子は次のとおりです。

- d ・・・・・・・・・・・・・・・10進整数（符号が付いてもよい）に変換します。対応する引数は整数へのポインタです。
- I・・・・・・・・・・・・・・・整数（符号が付いてもよい）に変換します。数値部の先頭が0xまたは0Xの場合16進整数，0の場合は8進整数その他は10進整数とみなします。対応する引数は整数へのポインタです。
- o ・・・・・・・・・・・・・・・8進整数（符号が付いてもよい）に変換します。対応する引数は整数へのポインタです。
- u ・・・・・・・・・・・・・・・無符号の10進整数に変換します。対応する引数は無符号整数へのポインタです。
- x ・・・・・・・・・・・・・・・16進整数（符号が付いてもよい）に変換します。
- e, E, f, g, G ・・・・・・オプションの符号（＋または－），小数点を含む1個または複数個の連続する10進数，およびオプションの指数（“ｅ”または“Ｅ”）とそれに続くオプションの符号付き整数値から構成される浮動小数点値。変換の結果オーバフローとなった場合，±∞を変換結果としアンダフローとなった場合，非正規化数または，±0を変換結果とします。対応する引数はfloatへのポインタです。
- s ・・・・・・・・・・・・・・・非空白文字列からなる文字列を入力します。対応する引数は整数へのポインタです。16進整数の先頭には0xまたは0Xを付けることができます。対応する引数は，この文字列と終端のナル文字を収容するのに十分な大きさを持つ配列へのポインタです。終端のナル文字は自動的に付加されます。
- [・・・・・・・・・・・・・・・期待する文字群（scansetという）からなる文字列を入力します。対応する引数は，この文字列と終端のナル文字を収容するのに十分な大きさを持つ配列へのポインタです。終端のナル文字は自動的に付加されます。変換指示はこの文字以降から右角かっこ（]）まで続きます。角かっこにはさまれた文字列（scanlistという）は，左角かっこの直後の文字がサーカムフレックス（＾）の場合を除きscansetを構成します。＾の場合は，このサーカムフレックスから右角かっこの間のscanlist以外のすべての文字がscansetを構成します。ただし，［］または［＾］で始まる場合はこの右角かっこはscanlistに入り，次の右角かっこが，scanlistの終端になります。
  scanlistの左端，右端以外のハイフン（－）は範囲指定です。－の左の文字が右の文字よりASCIIコードが小さくない場合はハイフンは－そのものの文字とします。
- c ・・・・・・・・・・・・・・・フィールド幅（指定のないときは1）で指定された個数の文字からなる文字列を入力します。対応する引数は，この文字列を収容するのに十分な大きさを持つ配列へのポインタです。終端のナル文字は追加しません。
- p ・・・・・・・・・・・・・・・無符号の16進整数として変換します。対応する引数はvoidへのポインタのポインタです。
- n ・・・・・・・・・・・・・・・文字列sからは入力しません。対応する引数は整数へのポインタであり，これまでこの関数で文字列sから読み出された文字数がそのポインタの指すオブジェクトに格納されます。%n指令は返り値の代入カウントには含めません。
- % ・・・・・・・・・・・・・・%を読みます。いかなる変換も代入も起こりません。

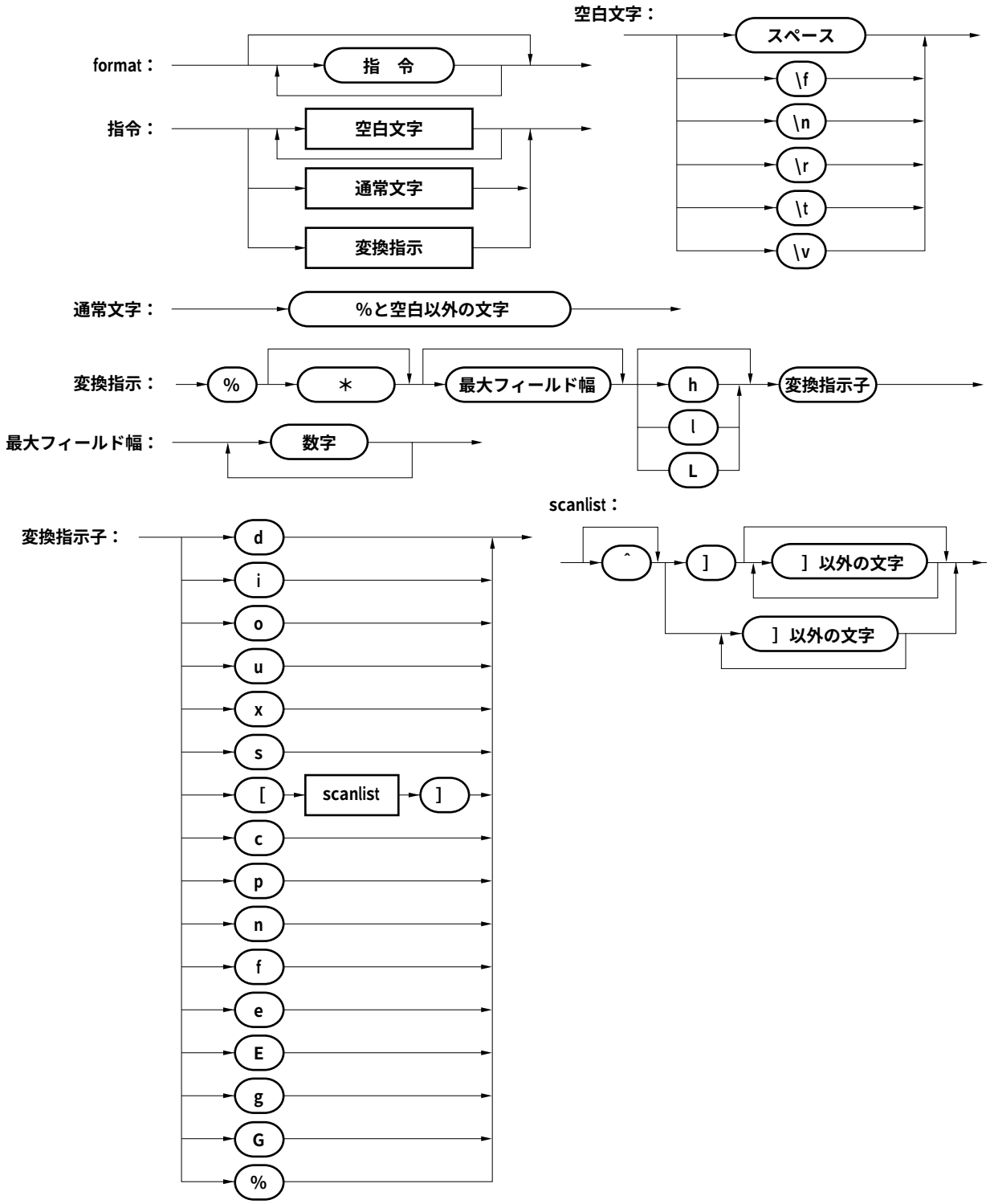
変換指示が不正な場合，指令は失敗します。

入力文字列に終端のナル文字が出現したらsscanfは戻ります。

整数変換の場合（d，i，o，u，x，p）はオーバフローした場合，変換後の型のビット数より上位は切り捨てます。

formatの構文図を次に示します。

図10ー3　入力formatの構文図



4－3　printf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・printfは，フォーマットに従ってデータをSFRに出力します。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int printf (const char * format, ...);`

関 数 名	引　　数	返 り 値
printf	format ･･･ 出力変換仕様を示す文字列へのポインタ ･･･ ･･･変換される0個以上の引数	sに出力された文字数（終端のナル文字は数えません）

【説　　明】

・formatで指定された出力変換仕様に従い，formatのあとに続く（0個以上の）引数を変換してputchar関数を使用して出力します。

・出力変換仕様は，0個以上の指令です。通常の文字（%で始まる変換仕様以外）はそのままputchar関数を使用して出力します。変換仕様は（0個以上の）後続の引数を取り出し変換してputchar関数を使用して出力します。

・各変換仕様は，sprintf関数と同じです。

4ー4　scanf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・SFRからフォーマットに従ってデータを読みます。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int  scanf (const char  * format, ...);`

関 数 名	引　　数	返 り 値
scanf	format ･･･ 入力変換仕様を示す文字列へのポインタ ･･･ ･･･変換された値を入れるオブジェクトへのポインタ（0個以上の）引数	文字列sが空でない場合 ･･･ 代入された入力項目の数

【説　　明】

・getchar関数を使用し，入力を行います。 formatが指す文字列により許される入力列を指定します。format以降の引数をオブジェクトへのポインタとして使用します。formatは入力列からどのように変換するかを指定します。

・formatに対して引数が足りない場合の正常動作は保証しません。過剰な引数の場合，式の評価は行いますが入力はされません。

・formatは0以上の指令からなります。指令は次のとおりです。

（1）1個以上の空白文字（isspaceが真となる文字）
（2）通常文字（%以外）
（3）変換指示

・指令と矛盾する入力文字によって変換が終了した場合，その矛盾した入力文字は切り捨てます。変換指示は，sscanf関数と同じです。

4ー5　vprintf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・vprintfは，フォーマットに従ってデータをSFRに出力します。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int vprintf (const char *format, va_list  p);`

関 数 名	引　　数	返 り 値
vprintf	format ･･･ 出力変換仕様を示す文字列へのポインタ p ･･･ 引数並びへのポインタ	出力された文字数（終端のナル文字は数えません）

【説　　明】

・formatで指定された出力変換仕様に従い，引数並びのポインタが指す引数を変換してputchar関数を使用し出力します。

・各変換仕様は，sprintf関数と同じです。

4－6　vsprintf（ノーマル・モデルのみ）　入出力関数

【機　　能】

・vsprintfは，フォーマットに従ってデータを文字列に書きます。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int vsprintf (char *s, const char * format, va_list  p);`

関 数 名	引　　数	返 り 値
vsprintf	s ･･･ 出力を書く文字列へのポインタ format ･･･ 出力変換仕様を示す文字列へのポインタ p ･･･ 引数並びへのポインタ	sに出力された文字数（終端のナル文字は数えません）

【説　　明】

・formatで指定された出力変換仕様に従い，引数並びのポインタが指す引数をsが指す文字列に書き出します。

・出力変換仕様は，sprintf関数と同じです。

4ー7　getchar　　入出力関数

【機　　能】

- SFRから，1文字読み込みます。

【ヘッダ・ファイル】

- stdio.h

【関数プロトタイプ】

- `int  getchar (void);`

関 数 名	引　　数	返 り 値
getchar	なし	SFRから読み込んだ1文字

【説　　明】

- SFRシンボルP0（ポート0）から読み込んだ値を返します。
- 読み込みに関して，エラー・チェックは行いません。
- 読み込むSFRの変更を行う場合は，ソースを変更しライブラリに登録し直すか，ユーザが新たにgetchar 関数を作成する必要があります。

4ー8　gets　　入出力関数

【機　　能】

・文字列を読み取ります。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`char  *gets (char *s);`

関 数 名	引　　数	返 り 値
gets	s ･･･ 入力文字列へのポインタ	正常な場合 ･･･ s 1文字も読み取らずファイルの終わりを検出した場合 ･･･ 空ポインタ

【説　　明】

・getchar関数を使用して文字列を読み取り，sが指す配列に格納します。

・ファイルの終わりを検出したとき（getchar関数が －1を返したとき），または改行文字を読み取ったときに，文字列の読み取りは終了します。そして読み取った改行文字を捨て，最後に配列に格納した文字の最後にナル文字を書きます。

・正常の場合は，sを返します。

・ファイルの終わりを検出し，かつ配列に1文字も読み取っていなかった場合は，配列の内容は変化せずに残し，空ポインタを返します。

4ー9　putchar　　入出力関数

【機　　能】

・SFRに1文字出力します。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

・`int  putchar (int  c);`

関 数 名	引　　数	返 り 値
putchar	c ･･･ 出力する文字	出力した文字

【説　　明】

・SFRシンボルP0（ポート0）にcで指定された文字を（unsigned char型に変換して）書き込みます。

・書き込みに関して，エラー・チェックは行いません。

・書き込むSFRの変更を行う場合は，ソースを変更しライブラリに登録し直すか，ユーザが新たにputchar関数を作成する必要があります。

4ー10　puts　　入出力関数

【機　　能】

・文字列を出力します。

【ヘッダ・ファイル】

・stdio.h

【関数プロトタイプ】

```
・int  puts (const char  *s);
```

関 数 名	引　　数	返 り 値
puts	s ･･･ 出力文字列へのポインタ	正常な場合 ･･･ 0 putchar関数が －1を返したとき ･･･ －1

【説　　明】

・putchar関数を使用し，sが指す文字列を書き込みます。そして出力の最後に改行文字を追加します。

・文字列の終端のナル文字の書き込みは行いません。

・正常の場合0を返し，putchar関数が －1を返したとき，－1を返します。

5ー1　atoi
　　atol　　ユーティリティ関数

【機　　能】

- atoiは，10進整数文字列をintに変換します。
- atolは，10進整数文字列をlongに変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `int atoi (const char *nptr);`
- `long int atol (const char *nptr);`

関 数 名	引　　数	返 り 値
atoi	nptr ･･･ 変換する文字列	正常の場合 ･･･ 変換された値 正のオーバフローの場合･･･ INT_MAX (32767) 負のオーバフローの場合 ･･･ INT_MIN (ー32768) 不正文字列の場合 ･･･ 0
atol		正常の場合 ･･･ 変換された値 正のオーバフローの場合 ･･･ LONG_MAX (2147483647) 負のオーバフローの場合 ･･･ LONG_MIN (ー2147483648) 不正文字列の場合 ･･･ 0

5ー1　atoi
　　atol　　ユーティリティ関数

【説　　明】

atoi

・nptrが指す文字列の最初の部分をintに変換します。

・つまり先頭から0個以上の空白文字（isspaceが真となる文字）の列をスキップし，次の文字からの省略可能な符号と引き続く10進数字の列（10進数字以外か終端のナル文字が現れるまで）を整数に変換します。10進数字がない場合は0を返します。オーバフローが起こった場合は，正のときは INT_MAX（32767）負のときは INT_MIN（ー32768）を返します。

atol

・nptrが指す文字列の最初の部分をlongに変換します。

・つまり先頭から0個以上の空白文字（isspaceが真となる文字）の列をスキップし，次の文字からの省略可能な符号と引き続く10進数字の列（10進数字以外か終端のナル文字が現れるまで）を整数に変換します。10進数字がない場合は0を返します。オーバフローが起こった場合は，正のときは LONG_MAX（2147483647）負のときは LONG_MIN（ー2147483648）を返します。

5ー2　strtol
　　strtoul　　ユーティリティ関数

【機　　能】

- strtol**は，文字列を**long**に変換します。**
- strtoul**は，文字列を**unsigned long**に変換します。**

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `long int strtol (const char *nptr, char **endptr, int base);`
- `unsigned long int strtoul (const char *nptr, char **endptr, int base);`

関 数 名	引　　数	返 り 値
strtol	nptr ･･･ **変換する文字列** endptr ･･･ **認識不可能部へのポインタを格納するポインタ** base ･･･ **指定する基数**	**正常の場合 ･･･ 変換した値** **正のオーバフローの場合** ･･･ LONG_MAX (2147483647) **負のオーバフローの場合** ･･･ LONG_MIN (ー2147483648) **変換が行われない場合 ･･･** 0
strtoul		**正常の場合 ･･･ 変換した値** **オーバフローの場合** ･･･ ULONG_MAX (4294967295U) **変換が行われない場合 ･･･** 0

5ー2　strtol
　　strtoul　　ユーティリティ関数

【説　　明】

strtol

・nptrが指す文字列を次の3部分に分解します。

（1）空であってもよい空白文字列（isspaceで指定される）

（2）baseの値により決定される基数による整数表現

（3）1文字以上の認識できない文字（終端のナル文字を含む）の列

（2）の文字列を整数に変換し，その結果を返します。

・baseが0ならばcの数値表現と解釈します（数値は，0x～または0X～（16進数），0～（8進数），0以外の数字～（10進数）で符号が前にあってもよい）。

・baseが2-36のときは，それを基数とします（符号が前にあってもよい）。a（A）からz（Z）は10から35までを表します。baseが16のときは，（あれば）符号の次に0xまたは0Xがついてもかまいません。

・（endptrがナル・ポインタでなければ）（3）の文字列へのポインタをendptrが指すオブジェクトへ格納します。

・オーバフローの場合，正は LONG_MAX（2147483647），負は LONG_MIN（－2147483648）を返し，errnoにERANGE（2）を入れます。

・（2）の文字列が空あるいは期待する型式に反する場合，変換は行わず（endptrが空ポインタでなければ）endptrが指すオブジェクトに文字列へのポインタを格納し，0を返します。baseが0，2-36以外の場合も同様です。

strtoul

・nptrが指す文字列を次の3部分に分解します。

（1）空であってもよい空白文字列（isspaceで指定される）

（2）baseの値により決定される基数による整数表現

（3）1文字以上の認識できない文字（終端のナル文字を含む）の列

（2）の文字列を無符号整数に変換し，その結果を返します。

・baseが0ならばCの数値表現（0x～または0X～（16進数），0～（8進数），0以外の数字～（10進数））と解釈します。

・baseが2-36のときは，それを基数とします。a（A）からz（Z）は10から35までを表します。baseが16のときは，0xまたは0Xがついてもかまいません。

・（endptrがナル・ポインタでなければ）（3）の文字列へのポインタをendptrが指すオブジェクトへ格納します。

・オーバフローの場合，ULONG_MAX（4294967295U）を返し，errnoにERANGE（2）を入れます。

・（2）の文字列が空あるいは期待する型式に反する場合，変換は行わず（endptrが空ポインタでなければ）endptrが指すオブジェクトに文字列へのポインタを格納し，0を返します。baseが0，2-36以外の場合も同様です。

5ー3　calloc　　ユーティリティ関数

【機　　能】

- callocは，配列の領域を割り付けて0で初期化します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `void *calloc (size_t nmemb, size_t size);`

関 数 名	引　　数	返 り 値
calloc	nmemb ･･･ 配列の個数 size ･･･ 配列のサイズ	割り付けられる場合 ･･･ 割り付けられた領域の先頭へのポインタ 割り付けられない場合 ･･･ 空ポインタ

【説　　明】

- sizeバイトの配列nmemb個分の領域を割り付け，その領域を0で初期化します。
- 割り付けられた領域の先頭へのポインタを返します。
- 割り付けられない場合には，空ポインタを返します。
- 割り付けは，ブレーク値から割り付け，割り付けられた領域の次のアドレスを新たなブレーク値とします。ブレーク値は，brkで設定します。brkについては5ー11　brkを参照してください。

5ー4　free　　ユーティリティ関数

【機　　能】

・割り付けられているブロックを解放します。

【ヘッダ・ファイル】

・stdlib.h

【関数プロトタイプ】

・`void free (void * ptr);`

関 数 名	引　　数	返 り 値
free	ptr ･･･ 解放するブロックの先頭へのポインタ	なし

【説　　明】

・ptrが指す領域からの割り付け済みの領域(ブレーク値の前まで)を解放します(freeの後で呼ばれるmalloc,calloc,reallocは，ptrからの領域を割り付けます）。

・ptrが割り付け済みの領域を指していなければ何もしません（解放は，ptrを新たなブレーク値とすることで行います）。

5ー5　malloc　　ユーティリティ関数

【機　　能】

- mallocは，ブロックを割り付けます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `void *malloc (size_t size);`

関 数 名	引　　数	返 り 値
malloc	size ･･･ 割り付けるブロックの大きさ	割り付けられる場合 ･･･ 割り付けられた領域の先頭へのポインタ 割り付けられない場合 ･･･ 空ポインタ

【説　　明】

- sizeバイト分の領域を割り付け，割り付けられた領域の先頭へのポインタを返します。
- 割り付けられない場合は，空ポインタを返します。
- 割り付けは，ブレーク値から割り付け，割り付けられた領域の次のアドレスを新たなブレーク値とします。ブレーク値は，brkで設定します。brkについては5ー11　brkを参照してください。

5ー6　realloc　　ユーティリティ関数

【機　　能】

- reallocは，ブロックの再割り付けを行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `void *realloc (void *ptr, size_t size);`

関 数 名	引　　数	返 り 値
realloc	ptr ･･･ 再割り付けされるブロックの先頭へのポインタ size ･･･ 再割り付けするブロックの大きさ	再割り付けされる場合 ･･･ 再割り付けした領域の先頭へのポインタ ptrが空ポインタで割り付けられる場合 ･･･ 割り付けられた領域の先頭へのポインタ 再割り付け，割り付けできない場合 ･･･ 空ポインタ

【説　　明】

- ptrが指す領域からの割り付け済みの領域（ブレーク値の前まで）の大きさをsizeに変更します。再割り付けする領域と再割り付けされる割り付け済みの領域の小さい方の大きさまでの内容は変化しません。大きさが増加する場合は増加分の割り付けを行い，減少する場合は減少分を解放します。
- ptrが空ポインタの場合は，size分の領域を新たに割り付けます（mallocと同じ）。
- ptrが割り付け済みの領域を指していない場合，または割り付けられない場合は，何もせずに空ポインタを返します。
- 再割り付けは，ptrにsizeバイトを加えたアドレスを新たなブレーク値として行います。

5ー7　abort　　ユーティリティ関数

【機　　能】

・abortは，プログラムを異常終了させます。

【ヘッダ・ファイル】

・stdlib.h

【関数プロトタイプ】

・`void abort (void);`

関 数 名	引　　数	返 り 値
abort	なし	戻りません。

【説　　明】

・ループして戻りません。

・ユーザはabortの処理を作成します。

5ー8　atexit
　　exit　　ユーティリティ関数

【機　　能】

- atexitは，正常終了時に呼び出される関数を登録します。
- exitは，プログラムを終了させます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `int atexit (void(*func)(void));`
- `void exit (int status);`

関 数 名	引　　数	返 り 値
atexit	func ･･･ 登録する関数へのポインタ	関数の登録が成功した場合 ･･･ 0 関数が登録できない場合 ･･･ 1
exit	status ･･･ 終了状態を示す値	戻りません。

【説　　明】

atexit

- atexitは，プログラムの正常終了時にfuncの指す関数が引数なしで呼ばれるように登録します。
- 関数は32個まで登録できます。登録できた場合は，0を返します。登録されている関数が32個あり，これ以上登録できない場合は，登録せずに1を返します。

exit

- exitは，プログラムを正常終了させます。
- 最初にatexitで登録した関数を登録と逆の順に呼びます。
- 内容はループになっており，exit関数からは戻りません。
- ユーザはexitの処理を作成します。

5ー9　abs
　　labs　　ユーティリティ関数

【機　　能】

- absは，int型の値の絶対値を求めます。
- labsは，long型の値の絶対値を求めます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `int abs (int j);`
- `long int labs (long int j);`

関 数 名	引　　数	返 り 値
abs	j ･･･ 絶対値を求める値	ー32767≦j≦32767の場合 ･･･ jの絶対値 jが ー32768の場合 ･･･ ー32768 （0x8000）
labs		ー2147483647≦j≦2147483647の場合 ･･･ jの絶対値 jが ー2147483648の場合 ･･･ ー2147483648（0x80000000）

【説　　明】

abs

- absは，jの値（int型）の絶対値を求めます。
- jが ー32768の場合は，ー32768を返します。

labs

- labsは，jの値（long型）の絶対値を求めます。
- jが ー2147483648の場合は，ー2147483648を返します。

5－10　div（ノーマル・モデルのみ）
ldiv（ノーマル・モデルのみ）　ユーティリティ関数

【機　　能】

- divは，int型の除算を行い，商と剰余を求めます。
- ldivは，long型の除算を行い，商と剰余を求めます。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `div_t div (int numer, int denom);`
- `ldiv_t ldiv (long int numer, long int denom);`

関 数 名	引　　数	返 り 値
div	numer ･･･ 被除数 denom ･･･ 除数	div_t型のメンバquotに商，remに剰余を返します。
ldiv		ldiv_t型のメンバquotに商，remに剰余を返します。

【説　　明】

div

- divは，numerをdenomで割った商と剰余を求めます。
- 商の絶対値は，numerの絶対値をdenomの絶対値で割った値以下の最大の整数です。符号は数学と同じです（numerとdenomが同符号の場合は正，異符号の場合は負）。
- 剰余は，numer － denom * 商の値です。
- denomが0の場合，商は0，剰余はnumerです。
- numerが －32768，denomが －1の場合，商は －32768，剰余は0です。

ldiv

- ldivは，numerをdenomで割った商と剰余を求めます。
- 商の絶対値は，numerの絶対値をdenomの絶対値で割った値以下の最大の（long int型）整数です。符号は数学と同じです（numerとdenomが同符号の場合は正，異符号の場合は負）。
- 剰余は，numer － denom * 商の値です。
- denomが0の場合，商は0，剰余はnumerです。
- numerが －2147483648，denomが －1の場合は，商は －2147483648，剰余は0です。

5ー11　brk
　　sbrk　　ユーティリティ関数

【機　　能】

- brkは，ブレーク値をセットします。
- sbrkは，ブレーク値を増減します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `int brk (char *endds);`
- `char *sbrk (int incr);`

関 数 名	引　　数	返 り 値
brk	endds ･･･ 設定するブレーク値	正常の場合 ･･･ 0 ブレーク値を変更できない場合 ･･･ −1
sbrk	incr ･･･ ブレーク値を増減する量	正常の場合･･･旧ブレーク値 ブレーク値が増減できない場合 ･･･ −1

【説　　明】

brk

- brkは，enddsで与えられた値をブレーク値に設定します。
- enddsが許容範囲外の場合は，ブレーク値を変更せず，errnoにENOMEM（3）をセットし，−1を返します。

sbrk

- sbrkは，ブレーク値をincrバイト増減（incrの符号による）します。
- 増減したあとのブレーク値が許容範囲外になる場合は，ブレーク値を変更せず，errnoにENOMEM(3)をセットし，−1を返します。

5ー12　atof
　　strtod　　ユーティリティ関数

【機　　能】

- atofは，10進整数文字列をdoubleに変換します。
- strtodは，文字列をdoubleに変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `double atof (const char *nptr);`
- `double strtod (const char *nptr, char **endptr);`

関 数 名	引　　数	返 り 値
atof	nptr ･･･ 変換する文字列	正常の場合 ･･･ 変換された値 正のオーバフローの場合･･･ HUGE_VAL（オーバフローした値の符号を持つ） 負のオーバフローの場合 ･･･ 0 不正文字列の場合 ･･･ 0
strtod	nptr ･･･ 変換する文字列 endptr ･･･ 認識不可能部へのポインタを格納するポインタ	正常の場合 ･･･ 変換された値 正のオーバフローの場合･･･ HUGE_VAL（オーバフローした値の符号を持つ） 負のオーバフローの場合 ･･･ 0 不正文字列の場合 ･･･ 0

5ー12　atof
　　strtod　　ユーティリティ関数

【説　　明】

atof

- nptrが指す文字列をdoubleに変換します。
- つまり先頭から0個以上の空白文字（isspaceが真となる文字）の列をスキップし，次の文字からの文字列（10進数字以外か終端のナル文字が現れるまで）を浮動小数点数に変換します。
- 変換が正常に行われると，浮動小数点数を返します。
- 変換でオーバフローが生じた場合には，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- アンダフローまたはオーバフローによる有効桁数の消滅が生じた場合には，それぞれ非正規化数，±0を返し，errnoにERANGEをセットします。
- 変換が行えない場合には，0を返します。

strtod

- nptrが指す文字列をdoubleに変換します。
- つまり先頭から0個以上の空白文字（isspaceが真となる文字）の列をスキップし，次の文字からの文字列（10進数字以外か終端のナル文字が現れるまで）を浮動小数点数に変換します。
- 変換が正常に行われると，浮動小数点数を返します。
- 変換でオーバフローが生じた場合には，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- アンダフローまたはオーバフローによる有効桁数の消滅が生じた場合には，それぞれ非正規化数，±0を返し，errnoにERANGEをセットします。またこのときendptrは，次の文字列へのポインタを格納します。
- 変換が行えない場合には，0を返します。

5ー13　itoa

ltoa（ノーマル・モデルのみ）

ultoa（ノーマル・モデルのみ）　　ユーティリティ関数

【機　　能】

- itoaは，intを文字列に変換します。
- ltoaは，longを文字列に変換します。
- ultoaは，unsigned longを文字列に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `char * itoa (int value, char *string, int radix);`
- `char * ltoa (long value, char *string, int radix);`
- `char * ultoa (unsigned long value, char *string, int radix);`

関 数 名	引　　数	返 り 値
itoa	value ･･･ 変換する数値	正常な場合 ･･･ 変換した文字列へのポインタ
ltoa	string ･･･ 変換結果へのポインタ	
ultoa	radix ･･･ 指定する基数	それ以外の場合 ･･･ 空ポインタ

【説　　明】

itoa， ltoa， ultoa

- 指定した数値valueをナル文字で終了する文字列に変換し，結果をstringで指される領域に格納します。変換は，指定された基数radixで行い，変換した文字列へのポインタを返します。
- radixは2-36の範囲でなければなりません。それ以外の場合には，変換を行わず，空ポインタを返します。

5ー14　rand
　　srand　　ユーティリティ関数

【機　　能】

- randは，疑似乱数を発生させます。
- srandは，疑似乱数の発生状態の初期化を行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `int rand (void);`
- `void srand (unsigned int seed);`

関 数 名	引　　数	返 り 値
rand	なし	0からRAND_MAXの範囲の疑似乱数
srand	seed ･･･ 疑似乱数の発生状態の初期値	なし

【説　　明】

rand

- randは，0から RAND_MAX の範囲の疑似乱数を発生させます。

srand

- srandは，疑似乱数の発生状態の初期化を行います。rand関数が呼ばれたときの返り値である疑似乱数列の基となる値としてseedを使います。seedの値が同じであれば，再びsrand関数が呼ばれても，疑似乱数の列は変わりません。
- srand関数をコールせずにrand関数をコールすることは，seed = 1でsrand関数をコールしたあとにrand関数をコールするのと同じです。

5ー15　bsearch（ノーマル・モデルのみ）　ユーティリティ関数

【機　　能】

・bsearchは，バイナリ・サーチを行います。

【ヘッダ・ファイル】

・stdlib.h

【関数プロトタイプ】

・
```
void *bsearch (const void *key, const void *base, size_t nmemb,
               size_t size, int (*compare)(const void *, const void *));
```

関 数 名	引　　数	返 り 値
bsearch	key ･･･ サーチする値へのポインタ base ･･･ サーチする配列へのポインタ nmemb ･･･ 配列要素の数 size ･･･ 配列の1要素のサイズ compare ･･･ keyと配列の要素を比較し，その関係を返す関数	マッチする配列要素がある場合 ･･･ 最初にマッチした配列要素へのポインタ マッチする配列要素がない場合 ･･･ 空ポインタ

【説　　明】

・ポインタbaseの指す配列からkeyの指すものをバイナリ・サーチします。ポインタbaseの指す配列はsizeの大きさのnmemb個の昇順にソートされた配列です。

・compare関数はkeyによって指されるものと配列要素を比較し，その関係を次の値により返します。compare関数の第1引数はkey，第2引数は配列要素です。

0より小さい ･･･ keyによって指されるものの方が小さい
0 ･･･ 両者は等しい
0より大きい ･･･ keyによって指されるものの方が大きい

・-ZRオプション指定時，bsearch関数の引数に渡す関数は，パスカル関数でなくてはなりません。

5－16　qsort（ノーマル・モデルのみ）　ユーティリティ関数

【機　　能】

- qsortは，クイック・ソートを行います。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- ```
  void qsort (void *base, size_t nmemb, size_t size,
                  int (*compare)(const void *, const void *));
  ```

関 数 名	引　　数	返 り 値
qsort	base ･･･ ソートする配列へのポインタ nmemb ･･･ 配列要素の数 size ･･･ 配列の1要素のサイズ compare ･･･ 配列の2つの要素を比較し，その関係を返す関数	なし

【説　　明】

- ポインタbaseの指す配列を昇順になるようにクイック・ソートします。ポインタbaseの指す配列はsizeの大きさのnmemb個の配列です。
- compare関数は2つの配列要素（配列要素1と2）を比較し，その関係を次の値により返します。
- compare関数の第1引数は配列要素1，第2引数は配列要素2です。

  0より小さい ･･･ 配列要素1の方が小さい
  0 ･･･ 両者は等しい
  0より大きい ･･･ 配列要素1の方が大きい

- 等しい配列要素であった場合には，配列の先頭に近い方にあったものが先になります。
- -ZRオプション指定時，qsort関数の引数に渡す関数は，パスカル関数でなくてはなりません。
  ```

5ー17　strbrk　　ユーティリティ関数

【機　　能】

・ブレーク値をセットします。

【ヘッダ・ファイル】

・stdlib.h

【関数プロトタイプ】

・`int strbrk ( char *endds );`

関 数 名	引　　数	返 り 値
strbrk	endds ･･･ 設定するブレーク値	正常な場合 ･･･ 0 ブレーク値を変更できない場合 ･･･ −1

【説　　明】

・enddsで与える値をブレーク値（割り当てられる領域の終わりのアドレスの次のアドレス）に設定します。

・enddsが許容範囲外の場合はブレーク値を変更せず，errnoにENOMEM（3）をセットし−1を返します。

5ー18　strsbrk　　ユーティリティ関数

【機　　能】

・ブレーク値を増減します。

【ヘッダ・ファイル】

・stdlib.h

【関数プロトタイプ】

・`char *strsbrk (int incr);`

関 数 名	引　　数	返 り 値
strsbrk	incr ･･･ ブレーク値を増減する量	正常な場合 ･･･ 旧ブレーク値 ブレーク値が増減できない場合 ･･･ −1

【説　　明】

・ブレーク値をincrバイト増減（incrの符号によります）します。

・増減した後のブレーク値が許容範囲外になる場合は，ブレーク値を変更せずerrnoにENOMEM（3）をセットし−1を返します。

5ー19　stritoa
　　strltoa（ノーマル・モデルのみ）
　　strultoa（ノーマル・モデルのみ）　　ユーティリティ関数

【機　　能】

- stritoaは，intを文字列に変換します。
- strltoaは，longを文字列に変換します。
- strultoaは，unsigned longを文字列に変換します。

【ヘッダ・ファイル】

- stdlib.h

【関数プロトタイプ】

- `char *stritoa (int value, char *string, int radix);`
- `char *strltoa (long value, char *string, int radix);`
- `char *strultoa (unsigned long value, char *string, int radix);`

関 数 名	引　　数	返 り 値
stritoa strltoa strultoa	value･･･変換する文字列 string･･･変換結果へのポインタ radix･･･指定する基数	正常な場合 ･･･ 変換した文字列へのポインタ それ以外の場合 ･･･ 空ポインタ

【説　　明】

stritoa，strltoa，strultoa

- 指定した数値valueをナル文字で終了する文字列に変換し，結果をstringで指される領域に格納します。変換は，指定された基数radixで行い，変換した文字列へのポインタを返します。
- radixは2-36の範囲でなければなりません。それ以外の場合には，変換を行わず，空ポインタを返します。

6ー1　memcpy
　　memmove　　文字列／メモリ関数

【機　　能】

・memcpyは，バッファを指定文字数分コピーします。

・memmoveは，バッファを指定文字数分コピーします（バッファが重なっても正常に動作します）。

【ヘッダ・ファイル】

・string.h

【関数プロトタイプ】

・`void *memcpy (void *s1, const void *s2, size_t n);`

・`void *memmove (void *s1, const void *s2, size_t n);`

関 数 名	引　　数	返 り 値
memcpy memmove	s1 ･･･ コピー先のオブジェクトへのポインタ s2 ･･･ コピー元のオブジェクトへのポインタ n ･･･ 指定文字数	s1の値

【説　　明】

memcpy

・memcpyは，s2が指すオブジェクトのn文字をs1が指すオブジェクトへコピーします。

・s2＜s1＜s2＋nの場合，正常動作は保証しません（先頭から順にコピーするため）。

memmove

・memmoveは，s2が指すオブジェクトのn文字をs1が指すオブジェクトへコピーします。

・s1とs2の指すオブジェクトが重なった場合も正常に動作します。

6ー2　strcpy
　　strncpy　　**文字列／メモリ関数**

【機　　能】

- strcpy**は，文字列をコピーします。**
- strncpy**は，文字列の先頭から指定の文字数分コピーします。**

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strcpy (char *s1, const char *s2);`
- `char *strncpy (char *s1, const char *s2, size_t n);`

関 数 名	引　　数	返 り 値
strcpy strncpy	s1 ･･･ コピー先文字列へのポインタ s2 ･･･ コピー元文字列へのポインタ n ･･･ コピーする文字数	s1の値

【説　　明】

strcpy

- strcpy**は，**s2**が指す文字列（終端のナル文字を含みます）を**s1**が指す文字列へコピーします。**
- s2＜s1≦**（**s2＋コピーする文字列の長さ**）**の場合，正常動作は保証しません（先頭から順にコピーするため）。**

strncpy

- strncpy**は，**s2**が指す文字列の**n**文字以内を**s1**が指す配列へコピーします。**
- s2＜s1≦**（**s2＋コピーする文字列の長さ，または**s2＋n－1**の最小値）の場合，正常動作は保証しません（先頭から順にコピーするため）。**
- s2**が指す文字列が**n**文字未満の場合には，終端のナル文字までをコピーします。**n**文字以上の場合には，先頭から**n**文字分をコピーし終端のナル文字はコピーしません。**

6ー3　strcat

　　strncat　　**文字列／メモリ関数**

【機　　能】

- **strcatは，文字列に文字列を追加します。**
- **strncatは，文字列に指定文字数分の文字列を追加します。**

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strcat (char *s1, const char *s2);`
- `char *strncat (char *s1, const char *s2, size_t n);`

関 数 名	引　　数	返 り 値
strcat	s1 ･･･ 追加される文字列へのポインタ	s1の値
	s2 ･･･ 追加する文字列へのポインタ	
strncat	n ･･･ 追加する文字数	

【説　　明】

strcat

- **strcatは，s1が指す文字列の終わりにs2が指す文字列（終端のナル文字を含みます）をコピーして追加します。s2の初めの文字をs1の終端のナル文字に上書きします。**
- **重なり合うオブジェクト間で複写を行う場合，その動作は保証しません。**

strncat

- **strncatは，s1が指す文字列の終わりにs2が指す文字列（終端のナル文字を含みません）のうちn文字分を追加します。s2の初めの文字をs1の終端の文字に上書きします。**
- **s2が指す文字列がn文字未満の場合には，終端のナル文字までを追加します。n文字以上の場合には，先頭からn文字分追加します。**
- **終端のナル文字は必ず追加します。**
- **重なり合うオブジェクト間で複写を行う場合，その動作は保証しません。**

6ー4　memcmp　　文字列／メモリ関数

【機　　能】

- memcmpは，2つのバッファの指定文字数分を比較します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `int memcmp (const void *s1, const void *s2, size_t n);`

関 数 名	引　　数	返 り 値
memcmp	s1 ･･･ 比較するオブジェクトへのポインタ s2 ･･･ 比較するオブジェクトへのポインタ n ･･･ 比較する文字数	s1とs2がn文字分等しい場合 ･･･ 0 s1とs2がn文字以内で異なる場合 ･･･ 最初の異なる文字をintに変換した値の差 （s1の文字 － s2の文字）

【説　　明】

- s1の指すオブジェクトとs2の指すオブジェクトをn文字分比較します。
- s1とs2がn文字分等しい場合，0を返します。
- s1とs2がn文字以内で異なる場合，最初の異なる文字をintに変換した値の差（s1の文字ーs2の文字）を返します。

6ー5　strcmp
strncmp　　文字列／メモリ関数

【機　　能】

- strcmpは，2つの文字列を比較します。
- strncmpは，2つの文字列の指定文字数分を比較します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `int strcmp (const char *s1, const char *s2);`
- `int strncmp (const char *s1, const char *s2, size_t n);`

関 数 名	引　　数	返 り 値
strcmp	s1 ･･･ 比較文字列へのポインタ s2 ･･･ 比較文字列へのポインタ	文字列s1と文字列s2が等しい場合 ･･･ 0 文字列s1と文字列s2が異なる場合 ･･･ 最初の異なる文字をintに変換した値の差 （s1の文字 － s2の文字）
strncmp	s1 ･･･ 比較文字列へのポインタ s2 ･･･ 比較文字列へのポインタ n ･･･ 比較する文字数	文字列s1と文字列s2がn文字分等しい場合 ･･･ 0 文字列s1と文字列s2がn文字分異なる場合 ･･･ 最初の異なる文字をintに変換した値の差 （s1の文字 － s2の文字）

6ー5　strcmp
　　strncmp　　**文字列／メモリ関数**

【説　　明】

strcmp

- strcmp**は，**s1**の指す文字列と**s2**の指す文字列を比較します。**
- **文字列**s1**と**s2**が等しい場合，**0**を返します。文字列**s1**と**s2**が異なる場合には，最初の異なる文字を**int**に変換した値の差（**s1**の文字ー**s2**の文字）を返します。**

strncmp

- strncmp**は，**s1**の指す文字列と**s2**の指す文字列の**n**文字分を比較します。**
- **文字列**s1**と**s2**が**n**文字以内で等しい場合，**0**を返します。文字列**s1**と**s2**が**n**文字以内で異なる場合には，最初の異なる文字を**int**に変換した値の差（**s1**の文字ー**s2**の文字）を返します。**

6ー6　memchr　　文字列／メモリ関数

【機　　能】

- memchrは，指定文字数分のバッファから指定文字を探します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `void *memchr (const void *s, int c, size_t n);`

関 数 名	引　　数	返 り 値
memchr	s ･･･ 検索されるオブジェクトへのポインタ c ･･･ 指定文字 n ･･･ 検索するオブジェクトの文字数	文字 c がある場合 ･･･ 最初に出現した文字 c へのポインタ 文字 c がない場合 ･･･ 空ポインタ

【説　　明】

- sが指すオブジェクトの先頭からn文字以内で最初に出現する（unsigned charに変換した）cの位置へのポインタを返します。
- 出現しない場合は，空ポインタを返します。

6ー7　strchr

strrchr　　**文字列／メモリ関数**

【機　　能】

- strchr**は，文字列中から指定された文字を探し，最初の出現位置を返します。**
- strrchr**は，文字列中から指定された文字を探し，最後の出現位置を返します。**

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strchr (const char *s, int c);`
- `char *strrchr (const char *s, int c);`

関 数 名	引　　数	返 り 値
strchr strrchr	s ･･･ 検索される文字列へのポインタ c ･･･ 指定文字	文字列s中に文字cがある場合 ･･･ 文字列s中に最初／最後に出現した文字cを指すポインタ 文字列s中に文字cがない場合･･･ 空ポインタ

【説　　明】

trchr

- strchr**は，sが指す文字列中の（**char**型へ変換した）**c**の最初の出現位置を求め，そのポインタを返します。**
- **終端のナル文字は，文字列の一部とみなします。**
- **文字列**s**中に文字**c**がない場合は，空ポインタを返します。**

strrchr

- strrchr**は，sが指す文字列中の（**char**型へ変換した）**c**の最後の出現位置を求め，そのポインタを返します。**
- **終端のナル文字は，文字列の一部とみなします。**
- **文字列**s**中に文字**c**がない場合は，空ポインタを返します。**

6ー8　strspn

strcspn　　**文字列／メモリ関数**

【機　　能】

- strspn**は，検索される文字列の中で指定文字列中に含まれる文字だけで構成されている部分の先頭からの長さを求めます。**
- strcspn**は，検索される文字列の中で指定文字列中に含まれる文字以外で構成されている部分の先頭からの長さを求めます。**

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `size_t strspn (const char *s1, const char *s2);`
- `size_t strcspn (const char *s1, const char *s2);`

関 数 名	引　　数	返 り 値
strspn	s1 ･･･ **検索される文字列へのポインタ**	**文字列**s1**中の**s2**で指定される文字で構成される部分の長さ**
strcspn	s2 ･･･ **指定文字列を示す文字列へのポインタ**	**文字列**s1**中の**s2**で指定される文字以外で構成される部分の長さ**

【説　　明】

strspn

- strspn**は，**s1**が指す文字列中で**s2**が指す文字列中に含まれる，文字だけで構成されている部分の長さを返します。**
- s2**の終端のナル文字は**s2**の一部とはみなしません。**

strcspn

- strcspn**は，**s1**が指す文字列中で**s2**が指す文字列中に含まれる，文字以外で構成されている部分の長さを返します。**
- s2**の終端のナル文字は**s2**の一部とはみなしません。**

6ー9　strpbrk　　文字列／メモリ関数

【機　　能】

・strpbrkは，指定された文字列のどれかの文字が，検索される文字列中で最初に現れる位置を求めます。

【ヘッダ・ファイル】

・string.h

【関数プロトタイプ】

・`char *strpbrk (const char *s1, const char *s2);`

関 数 名	引　　数	返 り 値
strpbrk	s1 ･･･ 検索される文字列へのポインタ s2 ･･･ 指定文字を示す文字列へのポインタ	文字列s1中に文字列s2内のどれかの文字がある場合 ･･･ 文字列s2内のどれかの文字が文字列s1中で最初に現れる文字へのポインタ 文字列s1中に文字列s2内の文字がない場合 ･･･ 空ポインタ

【説　　明】

・s2が指す文字列内のどれかの文字がs1が指す文字列中で最初に現れる位置を求め，そのポインタを返します。

・文字列s1中に文字列s2内の文字がない場合，空ポインタを返します。

6ー10　strstr　　**文字列／メモリ関数**

【機　　能】

- strstrは，指定文字列が，検索される文字列中に最初に現れる位置を求めます。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strstr (const char *s1, const char *s2);`

関 数 名	引　　数	返 り 値
strstr	s1 ･･･ 検索される文字列へのポインタ s2 ･･･ 指定文字列へのポインタ	文字列s1中に文字列s2がある場合 ･･･ 文字列s2が文字列s1中で最初に現れる位置の先頭へのポインタ 文字列s1中に文字列s2がない場合　･･･ 空ポインタ s2が空文字列の場合 ･･･ s1の値

【説　　明】

- s1が指す文字列中でs2が指す文字列（終端のナル文字を除く）と全文字が一致する最初の位置の先頭へのポインタを返します。
- 文字列s1中に文字列s2がない場合，空ポインタを返します。
- s2が空文字列を指す場合，s1の値を返します。

6ー11　strtok　　文字列／メモリ関数

【機　　能】

- 文字列を区切り文字以外からなる文字列に分解する。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strtok (char *s1, const char *s2);`

関 数 名	引　　数	返 り 値
strtok	s1 ･･･ 分解される文字列へのポインタまたは，空ポインタ	字句がある場合 ･･･ 字句の第1文字へのポインタ
	s2 ･･･ 字句の区切り文字を示す文字列へのポインタ	字句がない場合 ･･･ 空ポインタ

【説　　明】

- 字句とは，指定される文字列中の区切り文字以外の文字からなる文字列です。
- s1が空ポインタの場合は，前回のstrtokの呼び出しでの保存ポインタが指す文字列を分解される文字列とします。ただし，保存ポインタが空ポインタの場合は何もせずに空ポインタを返します。
- s1が空ポインタでない場合は，s1が指す文字列を分解される文字列とします。
- s2が指す文字列に含まれない文字を分解される文字列から探し，見つからなければ保存ポインタを空ポインタにして，空ポインタを返します。見つかれば，その文字を字句の第1文字とします。
- 字句の第1文字が見つかった場合，文字列s2に含まれる文字を字句の第1文字以降から探します。見つからなければ，保存ポインタを空ポインタにします。見つかれば，その文字の位置にナル文字を上書きし，その次の文字へのポインタを保存ポインタにします。
- 字句の第1文字へのポインタを返します。

6ー12　memset　　文字列／メモリ関数

【機　　能】

- memsetは，バッファの指定文字数分を指定文字で初期化します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `void *memset (void *s, int c, size_t n);`

関 数 名	引　　数	返 り 値
memset	s ･･･ 初期化するオブジェクトへのポインタ c ･･･ 指定文字 n ･･･ 指定文字数	sの値

【説　　明】

- sが指すオブジェクトの先頭からn文字分に（unsigned char型に変換された）cの値をコピーします。

6ー13　strerror　　文字列／メモリ関数

【機　　能】

- strerrorは，指定されたエラー番号に対応するエラー・メッセージの文字列を格納する領域へのポインタを返します。

【ヘッダ・ファイル】

- string.h

【関数プロトタイプ】

- `char *strerror (int errnum);`

関 数 名	引　　数	返 り 値
strerror	errnum ･･･ エラー番号	エラー番号に対応するエラーがある場合 ･･･ エラー・メッセージの文字列へのポインタ エラー番号に対応するエラーがない場合 ･･･ 空ポインタ

【説　　明】

- errnumの値に対応して，次の文字列へのポインタを返します。

0	･･･“ Error 0 ”
1（EDOM）	･･･“ Argument too large ”
2（ERANGE）	･･･“ Result too large ”
3（ENOMEM）	･･･“ Not enough memory ”

その他は空ポインタを返します。

6ー14　strlen　　文字列／メモリ関数

【機　　能】

・文字列の長さを求めます。

【ヘッダ・ファイル】

• string.h

【関数プロトタイプ】

• `size_t strlen (const char *s);`

関 数 名	引　　数	返 り 値
strlen	s ･･･ 文字列へのポインタ	文字列sの長さ

【説　　明】

・sが指す文字列の文字数を返します。文字数は，文字列の先頭から終端を示すナル文字の前までの文字数です。

6ー15　strcoll　　文字列／メモリ関数

【機　　能】

・地域特有の情報に基づいて2つの文字列を比較します。

【ヘッダ・ファイル】

• string.h

【関数プロトタイプ】

• `int  strcoll (const char *s1, const char *s2);`

関 数 名	引　　数	返 り 値
strcoll	s1 ･･･ 比較文字列へのポインタ s2 ･･･ 比較文字列へのポインタ	文字列s1と文字列s2が等しい場合 ･･･ 0 文字列s1と文字列s2が異なる場合 ･･･ 最初の異なる文字をintに変換した値の差 （s1の文字 － s2の文字）

【説　　明】

・本コンパイラは，文化圏固有操作はサポートしていません。strcmpと同じ動作をします。

6ー16　strxfrm　　文字列／メモリ関数

【機　　能】

・地域特有の情報に基づいて文字列を変換します

【ヘッダ・ファイル】

• string.h

【関数プロトタイプ】

```
• size_t  strxfrm (char  *s1, const char  *s2, size_t  n);
```

関 数 名	引　　数	返 り 値
strxfrm	s1 ･･･ 比較文字列へのポインタ s2 ･･･ 比較文字列へのポインタ n ･･･ s1に入る最大文字数	変換した結果の文字列（終端を示す文字列を含みません）の長さを返します。 返却された値がn以上の場合，s1で示される配列の内容は不定とします。

【説　　明】

・本コンパイラは，文化圏固有操作はサポートしていません。次の関数と同じ動作をします。

strncpy (s1, s2, c)；

return (strlen (s2))；

7－1　acos（ノーマル・モデルのみ）　数学関数

【機　　能】

- acosを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double　acos（double　x）;`

関 数 名	引　　数	返 り 値
acos	x ･･･ 演算を行う数値	－1≦x≦1のとき ･･･ xのacos x＜－1，1＜x，x = NaNのとき ･･･ NaN

【説　　明】

- xのacos（0からπの範囲内）を計算します。
- xが非数の場合は，NaNを返します。
- x＜－1，1＜xの定義域エラーの場合は，NaNを返しerrnoにEDOMをセットします。

7－2　asin（ノーマル・モデルのみ）　数学関数

【機　　能】

- asinを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double　asin（double　x）;`

関 数 名	引　　数	返 り 値
asin	x ･･･ 演算を行う数値	－1≦x≦1のとき ･･･ xのasin x＜－1，1＜x，x = NaNのとき ･･･ NaN x = －0のとき ･･･ －0 アンダフロー時　･･･ 非正規化数

【説　　明】

- xのasin（－π/2から＋π/2の範囲内）を計算します。
- x＜－1，1＜xの領域エラーの場合は，NaNを返しerrnoにEDOMをセットします。
- xが非数の場合は，NaNを返します。
- xが　－0の場合は，－0を返します。
- 演算の結果アンダフローが生じた場合は，非正規化数を返します。

7ー3　atan（ノーマル・モデルのみ）　数学関数

【機　　能】

・atanを求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　atan（double　x）;`

関 数 名	引　　数	返 り 値
atan	x ･･･演算を行う数値	正常時 ･･･ xのatan x = NaNのとき ･･･ NaN x = －0のとき ･･･ －0

【説　　明】

・xのatan（－π/2から＋π/2の範囲内）を計算します。

・xが非数の場合は，NaNを返します。

・xが－0の場合は，－0を返します。

・演算の結果アンダフローが生じた場合は，非正規化数を返します。

7ー4　atan2（ノーマル・モデルのみ）　数学関数

【機　　能】

- y/xのatanを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  atan2 (double  y, double x) ;`

関 数 名	引　　数	返 り 値
atan2	x ･･･ 演算を行う数値 y ･･･ 演算を行う数値	正常時　･･･ y/xのatan yがともに0か，y/xが表現できない値の場合。あるいはx，yのどちらかがNaN，x，yがともに±∞の場合　･･･ NaN 非正規化数　･･･ アンダフロー時

【説　　明】

- y/xのatan（－πから＋πの範囲内）を計算します。xとyが共に0か，y/xが表現できない値の場合，あるいは，x，yがともに無限大の場合には，NaNを返しerrnoにEDOMをセットします。
- x，yのどちらかが非数の場合は，NaNを返します。
- 演算の結果アンダフローが生じた場合は，非正規化数を返します。

7ー5　cos（ノーマル・モデルのみ）　数学関数

【機　　能】

- cosを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double　cos（double　x）;`

関 数 名	引　　数	返 り 値
cos	x ･･･ 演算を行う数値	正常時 ･･･ xのcos x = NaN，x = ±∞の場合 ･･･ NaN

【説　　明】

- xのcosを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー6　sin（ノーマル・モデルのみ）　数学関数

【機　　能】

・sinを求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　sin（double　x）;`

関 数 名	引　　数	返 り 値
sin	x ･･･ 演算を行う数値	正常時 ･･･ xのsin x = NaN，x = ±∞の場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数

【説　　明】

・xのsinを計算します。

・xが非数の場合は，NaNを返します。

・xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。

・演算の結果アンダフローが生じた場合は，非正規化数を返します。

・xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー7　tan（ノーマル・モデルのみ）　数学関数

【機　　能】

- tanを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  tan（double  x）;`

関 数 名	引　　数	返 り 値
tan	x ･･･ 演算を行う数値	正常時 ･･･ xのtan x = NaN，x = ±∞の場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数

【説　　明】

- xのtanを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- 演算の結果アンダフローが生じた場合は，非正規化数を返します。
- xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー8　cosh（ノーマル・モデルのみ）　数学関数

【機　　能】

- coshを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  cosh（double  x）;`

関 数 名	引　　数	返 り 値
cosh	x ･･･ 演算を行う数値	正常時 ･･･ xのcosh オーバフロー時，x = ±∞の場合 ･･･ HUGE_VAL（正の符号を持ちます） x = NaN ･･･ NaN

【説　　明】

- xのcoshを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，正の無限大の値を返します 。
- 演算の結果オーバフローが生じた場合は，正の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー9　sinh（ノーマル・モデルのみ）　数学関数

【機　　能】

- sinhを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  sinh（double  x）;`

関 数 名	引　　数	返 り 値
sinh	x ･･･ 演算を行う数値	正常時 ･･･ xのsinh x = NaNの場合 ･･･ NaN x = ±∞の場合 ･･･ ±∞ オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます） アンダフロー時 ･･･ ±0

【説　　明】

- xのsinhを計算します。
- xが非数の場合は，NaNを返します。
- xが±∞の場合は，±∞を返します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- 演算の結果，アンダフローが生じた場合は，±0を返します。

7ー10　tanh（ノーマル・モデルのみ）　数学関数

【機　　能】

- tanhを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  tanh（double  x）;`

関 数 名	引　　数	返 り 値
tanh	x ･･･ 演算を行う数値	正常時 ･･･ xのtanh x = NaNの場合 ･･･ NaN x = ±∞の場合 ･･･ ±1 アンダフロー時 ･･･ ±0

【説　　明】

- xのtanhを計算します。
- xが非数の場合は，NaNを返します。
- xが±∞の場合は，±1を返します。
- 演算の結果，アンダフローが生じた場合は，±0を返します。

7ー11　exp（ノーマル・モデルのみ）　数学関数

【機　　能】

・指数関数を求めます。

【ヘッダ・ファイル】

• math.h

【関数プロトタイプ】

• `double  exp（double  x）;`

関 数 名	引　　数	返 り 値
exp	x ･･･ 演算を行う数値	正常時 ･･･ xの指数関数 x = NaNの場合 ･･･ NaN x = ±∞の場合 ･･･ ±∞ オーバフロー時 ･･･ HUGE_VAL （正の符号を持ちます） アンダフロー時 ･･･ 非正規化数 アンダフローによる有効桁数の消滅時 ･･･ +0

【説　　明】

・xの指数関数を計算します。

・xが非数の場合は，NaNを返します。

・xが±∞の場合は，±∞を返します。

・演算の結果，アンダフローが生じた場合は，非正規化数を返します。

・演算の結果，アンダフローによる有効桁数の消滅が生じた場合は，+0を返します。

・演算の結果，オーバフローが生じた場合は，正の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー12　frexp（ノーマル・モデルのみ）　数学関数

【機　　能】

・仮数部と指数部を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double  frexp（double  x, int  *exp）;`

関 数 名	引　　数	返 り 値
frexp	x ･･･ 演算を行う数値	正常時 ･･･ xの仮数
	exp ･･･ 指数部を格納するポインタ	x = NaN，x = ±∞の場合 ･･･ NaN
		x = ±0のとき ･･･ ±0

【説　　明】

・浮動小数点数xをx = m＊2＾nのような仮数mと指数nに分け，仮数mを返します。

・指数nはポインタexpの指し示すところに格納します。ただし，mの絶対値は0.5以上1.0未満です。

・xが非数の場合，NaNを返し，*expの値は0とします。

・xが無限大の場合は，NaNを返し，*expの値を0とし，errnoにEDOMをセットします。

・xが±0の場合，±0を返し，*expの値は0とします。

7ー13　ldexp（ノーマル・モデルのみ）　数学関数

【機　　能】

- x∗2 ^ expを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  ldexp (double x,  int exp) ;`

関 数 名	引　　数	返 り 値
ldexp	x ･･･ 演算を行う数値 exp ･･･ べき乗数	正常時 ･･･ x∗2 ^ exp x = NaNの場合 ･･･ NaN x = ±∞ の場合 ･･･ ±∞ x = ±0の場合 ･･･ ±0 オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます） アンダフロー時 ･･･ 非正規化数 アンダフローによる有効桁数の消滅時 ･･･ ±0

【説　　明】

- x∗2 ^ expを計算します。
- xが非数の場合は，NaNを返します。
- xが±∞の場合は，±∞を返します。
- xが±0の場合，±0を返します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値を持つHUGE_VALを返し，errnoにERANGEをセットします。
- 演算の結果，アンダフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダフローによる有効桁数の消滅が生じた場合は，±0を返します。

7ー14　log（ノーマル・モデルのみ）　数学関数

【機　　能】

・自然対数を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double log(double x);`

関 数 名	引　　数	返 り 値
log	x ･･･ 演算を行う数値	正常時 ･･･ xの自然対数 x≦0のとき ･･･ HUGE_VAL（負の符号を持ちます） xが非数の場合 ･･･ NaN xが無限大のとき ･･･ ＋∞

【説　　明】

・xの自然対数を求めます。

・xが非数の場合は，NaNを返します。

・xが ＋∞の場合は，＋∞を返します。

・x＜0の領域エラーの場合は，負の符号を持つHUGE_VALを返し，errnoにEDOMをセットします。

・x = 0の場合は，負の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー15　log10（ノーマル・モデルのみ）　数学関数

【機　　能】

- 10を底とした対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  log10 (double  x) ;`

関 数 名	引　　数	返 り 値
log10	x ･･･ 演算を行う数値	正常時 ･･･ xの10を底とした対数 x≦0のとき ･･･ HUGE_VAL（負の符号を持ちます） xが非数の場合 ･･･ NaN xが無限大のとき ･･･ ＋∞

【説　　明】

- xの10を底とした対数を求めます。
- xが非数の場合は，NaNを返します。
- xが＋∞の場合は，＋∞を返します。
- x＜0の領域エラーの場合は，負の符号を持つHUGE_VALを返しerrnoにEDOMをセットします。
- x = 0の場合は，負の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー16　modf（ノーマル・モデルのみ）　数学関数

【機　　能】

・小数部と整数部を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double  modf (double x, double *iptr) ;`

関 数 名	引　　数	返 り 値
modf	x ･･･ 演算を行う数値 iptr ･･･ 整数部へのポインタ	正常時 ･･･ xの小数部 xが非数またはxが無限大の場合 ･･･ NaN xが±0のとき ･･･ ±0

【説　　明】

・浮動小数点数xを小数部と整数部に分けます。

・xと同じ符号を持つ小数部を返し，整数部はポインタiptrの指し示すところに格納します。

・xが非数の場合は，NaNを返し，ポインタiptrの指し示すところにNaNを格納します。

・xが無限大の場合は，NaNを返し，ポインタiptrの指し示すところにNaNを格納し，errnoにEDOMをセットします。

・x = ±0の場合は，ポインタiptrの指し示すところに±0を格納します。

7ー17　pow（ノーマル・モデルのみ）　数学関数

【機　　能】

- xのy乗を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  pow（double x, double y）;`

関 数 名	引　　数	返 り 値
pow	x ･･･ 演算を行う数値 y ･･･ 乗数	正常時 ･･･ x ^ y x = NaN または y = NaN， x = ＋∞ かつ y = 0， x＜0 かつ y≠整数， x＜0かつy = ±∞， x = 0かつ y≦0， 　のいずれかの場合　･･･ NaN オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます。） アンダフロー時 ･･･ 非正規化数 アンダフローによる有効桁数の消滅時 ･･･ ±0

【説　　明】

- x ^ yを計算します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- x = NaNまたはy = NaNの場合は，NaNを返します。
- x = ＋∞かつy = 0，x＜0かつy≠整数，x＜0かつy = ±∞，x = 0かつy≦0のいずれかの場合は，NaNを返し，errnoにEDOMをセットします。
- アンダフローが生じた場合は，非正規化数を返します。
- アンダフローによる有効桁数の消滅が生じた場合は，±0を返します。

7ー18　sqrt（ノーマル・モデルのみ）　数学関数

【機　　能】

・平方根を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　sqrt（double　x）;`

関 数 名	引　　数	返 り 値
sqrt	x ･･･ 演算を行う数値	x≧0の場合 ･･･ xの平方根 x = ±0の場合 ･･･ ±0 x＜0の場合 ･･･ NaN

【説　　明】

・xの平方根を計算します。

・x＜0の領域エラーの場合は，0を返し，errnoにEDOMをセットします。

・xが非数の場合は，NaNを返します。

・xが±0の場合は，±0を返します。

7ー19　ceil（ノーマル・モデルのみ）　数学関数

【機　　能】

・xより小さくない最小の整数を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　ceil（double　x）;`

関 数 名	引　　数	返 り 値
ceil	x ･･･ 演算を行う数値	正常時 ･･･ xより小さくない最小の整数 xが非数のとき，またはx = ±∞の場合 ･･･ NaN x = －0の場合 ･･･ ＋0 xより小さくない最小の整数を表現できない場合 ･･･ x

【説　　明】

・xより小さくない最小の整数を求めます。

・xが非数の場合は，NaNを返します。

・xが －0の場合は，＋0を返します。

・xが無限大の場合は，NaNを返しerrnoにEDOMをセットします。

・xより小さくない最小の整数を表現できない場合は，xを返します。

7－20　fabs（ノーマル・モデルのみ）　数学関数

【機　　能】

・浮動小数点数xの絶対値を返します。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　fabs（double　x）;`

関 数 名	引　　数	返 り 値
fabs	x ･･･絶対値を求める値	正常時 ･･･ xの絶対値 xが非数の場合 ･･･ NaN x = －0の場合 ･･･ ＋0

【説　　明】

・xの絶対値を求めます。

・xが非数の場合は，NaNを返します。

・xが －0の場合は，＋0を返します。

7ー21　floor（ノーマル・モデルのみ）　数学関数

【機　　能】

・xより大きくない最大の整数を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`double　floor（double　x）；`

関 数 名	引　　数	返 り 値
floor	x ･･･演算を行う数値	正常時 ･･･ xより大きくない最大の整数 xが非数のとき，または無限大の場合 ･･･ NaN x = －0の場合 ･･･ ＋0 xより大きくない最大の整数を表現できない場合 ･･･ x

【説　　明】

・xより大きくない最大の整数を求めます。

・xが非数の場合は，NaNを返します。

・xが －0の場合は，＋0を返します。

・xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。

・xより大きくない最大の整数を表現できない場合は，xを返します。

7ー22　fmod（ノーマル・モデルのみ）　数学関数

【機　　能】

- x/yの余りを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `double  fmod (double x, double y) ;`

関 数 名	引　　数	返 り 値
fmod	x ･･･ 演算を行う数値	正常時 ･･･ x/yの余り
	y ･･･ 演算を行う数値	xが非数またはyが非数， yが±0の場合，xが±∞の場合 ･･･ NaN
		x≠∞かつy = ±∞の場合 ･･･ x

【説　　明】

- xーi＊yで表されるx/yの余りを計算します。iは整数です。
- y≠0の場合は，返り値はxと同じ符号を持ち，その絶対値はyの絶対値より小さくなります。
- yが±0あるいはx = ±∞の場合は，NaNを返し，errnoにEDOMをセットします。
- xが非数またはyが非数の場合は，NaNを返します。
- yが無限大の場合は，xが無限大でなければxを返します。

7ー23　matherr（ノーマル・モデルのみ）　数学関数

【機　　能】

・浮動小数点数を扱うライブラリの例外処理を行います。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`void matherr(struct exception *x);`

関 数 名	引　　数	返 り 値

matherr	`struct exception {`
	`  int type;`
	`  char *name;`
	`}`
	
	type ・・・ 演算例外を示す値
	name ・・・ 関数名
		なし

【説　　明】

・浮動小数点数を扱う，標準ライブラリ，ランタイム・ライブラリにおいて，例外発生時に呼び出されます。

・標準ライブラリから呼び出された場合は，errnoにEDOM，ERANGEを設定します。

次に演算例外typeとerrnoの関係を示します。

type	演算例外	errnoに設定する値
1	アンダフロー	ERANGE
2	消滅	ERANGE
3	オーバフロー	ERANGE
4	ゼロ除算	EDOM
5	演算不能	EDOM

matherrを変更あるいは作成することで，独自のエラー処理ができます。

7－24　acosf（ノーマル・モデルのみ）　数学関数

【機　　能】

- acosを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  acosf（float  x）;`

関 数 名	引　　数	返 り 値
acosf	x ･･･ 演算を行う数値	－1≦x≦1の場合 ･･･ xのacos x＜－1，1＜x，x＝　の場合 ･･･ NaN

【説　　明】

- xのacos（0からπの範囲内）を計算します。
- xが非数の場合は，NaNを返します。
- x＜－1，1＜xの定義域エラーの場合は，NaNを返し，errnoにEDOMをセットします。

7ー25　asinf（ノーマル・モデルのみ）　数学関数

【機　　能】

- asinを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  asinf（float  x）;`

関 数 名	引　　数	返 り 値
asinf	x ･･･ 演算を行う数値	ー1≦x≦1の場合 ･･･ xのasin x＜ー1，1＜x，x＝ NaNの場合 ･･･ NaN x＝ ー0 ･･･ ー0 アンダフロー時 ･･･ 非正規化数

【説　　明】

- xのasin（ーπ/2から ＋π/2の範囲内）を計算します。
- xが非数の場合は，NaNを返します。
- x＜ー1，1＜xの定義域エラーの場合は，NaNを返し，errnoにEDOMをセットします。
- x＝ ー0の場合は，ー0を返します。
- 演算の結果，アンダフローが生じた場合は，非正規化数を返します。

7ー26　atanf（ノーマル・モデルのみ）　数学関数

【機　　能】

- atanを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  atanf（float  x）;`

関 数 名	引　　数	返 り 値
atanf	x ･･･ 演算を行う数値	正常時 ･･･ xのatan x　=　NaNのとき ･･･ NaN x　=　ー0のとき ･･･ ー0

【説　　明】

- xのatan（ーπ/2から　＋π/2の範囲内）を計算します。
- xが非数の場合は，NaNを返します。
- x =　ー0の場合は，ー0を返します。
- 演算の結果，アンダフローが生じた場合は，非正規化数を返します。

7ー27　atan2f（ノーマル・モデルのみ）　数学関数

【機　　能】

- y/xのatanを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  atan2f (float  y,  float  x) ;`

関 数 名	引　　数	返 り 値
atan2f	x ･･･ 演算を行う数値 y ･･･ 演算を行う数値	正常時 ･･･ y/xのatan xとyがともに0か，y/xが表現できない値の場合，あるいはx，yのどちらかがNaN，x，yがともに±∞の場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数

【説　　明】

- y/xのatan（−πから＋πの範囲内）を計算します。xとyがともに0か，y/xが表現できない値の場合，あるいはx，yがともに無限大の場合には，NaNを返し，errnoにEDOMをセットします。
- x，yのどちらかが非数の場合は，NaNを返します。
- 演算の結果アンダフローが生じた場合は，非正規化数を返します。

7ー28　cosf（ノーマル・モデルのみ）　数学関数

【機　　能】

- cosを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  cosf（float  x）;`

関 数 名	引　　数	返 り 値
cosf	x ･･･ 演算を行う数値	正常時 ･･･ xのcos x = NaN，x = ±∞の場合 ･･･ NaN

【説　　明】

- xのcosを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー29　sinf（ノーマル・モデルのみ）　数学関数

【機　　能】

- sinを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  sinf（float  x）;`

関 数 名	引　　数	返 り 値
sinf	x ･･･ 演算を行う数値	正常時 ･･･ xのsin x = NaN，x = ±∞の場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数

【説　　明】

- xのsinを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- 演算の結果，アンダフローが生じた場合は，非正規化数を返します。
- xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー30　tanf（ノーマル・モデルのみ）　数学関数

【機　　能】

・tanを求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  tanf（float  x）;`

関 数 名	引　　数	返 り 値
tanf	x ･･･ 演算を行う数値	正常時 ･･･ xのtan x = NaN，x = ±∞の場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数

【説　　明】

・xのtanを計算します。

・xが非数の場合は，NaNを返します。

・xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。

・演算の結果，アンダフローが生じた場合は，非正規化数を返します。

・xの絶対値が非常に大きい場合，演算結果はほとんど意味のない値となります。

7ー31　coshf（ノーマル・モデルのみ）　数学関数

【機　　能】

- coshを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  coshf（float  x）;`

関 数 名	引　　数	返 り 値
coshf	x ･･･ 演算を行う数値	正常時 ･･･ xのcosh オーバフロー時，x = ±∞の場合 ･･･ HUGE_VAL （正の符号を持ちます） x = NaN　･･･ NaN

【説　　明】

- xのcoshを計算します。
- xが非数の場合は，NaNを返します。
- xが無限大の場合は，正の無限大の値を返します。
- 演算の結果，オーバフローが生じた場合は，正の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー32　sinhf（ノーマル・モデルのみ）　数学関数

【機　　能】

- sinhを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  sinhf（float  x）;`

関 数 名	引　　数	返 り 値
sinhf	x ･･･ 演算を行う数値	正常時 ･･･ xのsinh オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます） x = NaN ･･･ NaN x = ±∞の場合 ･･･ ±∞ アンダフロー時 ･･･ ±0

【説　　明】

- xのsinhを計算します。
- xが非数の場合は，NaNを返します。
- xが±∞の場合は，±∞を返します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- 演算の結果，アンダフローが生じた場合は，±0を返します。

7ー33　tanhf（ノーマル・モデルのみ）　数学関数

【機　　能】

- tanhを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  tanhf（float  x）;`

関 数 名	引　　数	返 り 値
tanhf	x ･･･ 演算を行う数値	正常時 ･･･ xのtanh x = NaN ･･･ NaN x = ±∞の場合 ･･･ ±1 アンダフロー時 ･･･ ±0

【説　　明】

- xのtanhを計算します。
- xが非数の場合は，NaNを返します。
- xが±∞の場合は，±1を返します。
- 演算の結果，アンダフローが生じた場合は，±0を返します。

7ー34　expf（ノーマル・モデルのみ）　数学関数

【機　　能】

・指数関数を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  expf（float  x）;`

関 数 名	引　　数	返 り 値
expf	x ･･･ 演算を行う数値	正常時 ･･･ xの指数関数 オーバフロー時 ･･･ HUGE_VAL （正の符号を持ちます） x = NaN ･･･ NaN x = ±∞の場合 ･･･ ±∞ アンダフロー時 ･･･ 非正規化数 アンダフローによる有効桁数の消滅時 ･･･ ＋0

【説　　明】

・xの指数関数を計算します。

・xが非数の場合は，NaNを返します。

・xが±∞の場合は，±∞を返します。

・演算の結果，オーバフローが生じた場合は，正の符号を持つHUGE_VALを返し， errnoにERANGEをセットします。

・演算の結果，アンダフローが生じた場合は，非正規化数を返します。

・演算の結果，アンダフローによる有効桁数の消滅が生じた場合は，＋0を返します。

7ー35　frexpf（ノーマル・モデルのみ）　数学関数

【機　　能】

・仮数部と指数部を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  frexpf（float  x,  int  *exp）;`

関 数 名	引　　数	返 り 値
frexpf	x ･･･ 演算を行う数値 exp ･･･ 指数部を格納するポインタ	正常時 ･･･ xの仮数 x = NaN，x = ±∞の場合 ･･･ NaN x = ±0の場合 ･･･ ±0

【説　　明】

・浮動小数点数xをx = m＊2 ^ nのような仮数mと指数nに分け，仮数mを返します。

・指数nはポインタexpの指し示すところに格納します。ただし，mの絶対値は0.5以上1.0未満です。

・xが非数の場合は，NaNを返し，*expの値は0とします。

・xが±∞の場合は，NaNを返し，*expの値は0とし，errnoにEDOMをセットします。

・xが±0の場合は，±0を返し，*expの値は0とします。

7ー36　ldexpf（ノーマル・モデルのみ）　数学関数

【機　　能】

- x∗2 ^ expを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  ldexpf (float  x, int  exp) ;`

関 数 名	引　　数	返 り 値
ldexpf	x ･･･ 演算を行う数値 exp ･･･ べき乗数	正常時 ･･･ x∗2 ^ exp x = NaNの場合 ･･･ NaN x = ±∞の場合 ･･･ ±∞ x = ±0の場合 ･･･ ±0 オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます） アンダフロー時 ･･･ 非正規化数 アンダフローによる有効桁数の消滅時 ･･･ ±0

【説　　明】

- x∗2 ^ expを計算します。
- xが非数の場合はNaN， ±∞のときは±∞，±0のときは，±0を返します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- 演算の結果，アンダフローが生じた場合は，非正規化数を返します。
- 演算の結果，アンダフローによる有効桁数の消滅が生じた場合は，±0を返します。

7ー37　logf（ノーマル・モデルのみ）　数学関数

【機　　能】

・自然対数を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  logf（float  x）;`

関 数 名	引　　数	返 り 値
logf	x ･･･ 演算を行う数値	正常時 ･･･ xの自然対数 xが非数の場合 ･･･ NaN xが無限大の場合 ･･･ ＋∞ x≦0の場合 ･･･ HUGE_VAL（負の符号を持ちます）

【説　　明】

・xの自然対数を求めます。

・xが非数の場合は，NaNを返します。

・xが＋∞の場合は，＋∞を返します。

・x＜0の領域エラーの場合は，負の符号を持つHUGE_VALを返しerrnoにEDOMをセットします。

・x = 0の場合は，負の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー38　log10f（ノーマル・モデルのみ）　数学関数

【機　　能】

- 10を底とした対数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  log10f（float  x）;`

関 数 名	引　　数	返 り 値
log10f	x ･･･ 演算を行う数値	正常時 ･･･ xの10を底とした対数 xが非数の場合 ･･･ NaN x = ＋∞の場合 ･･･ ＋∞ x≦0の場合 ･･･ HUGE_VAL（負の符号を持ちます）

【説　　明】

- xの10を底とした対数を求めます。
- xが非数の場合は，NaNを返します。
- xが＋∞の場合は，＋∞を返します。
- x＜0の領域エラーの場合は，負の符号を持つHUGE_VALを返し，errnoにEDOMをセットします。
- x = 0の場合は，負の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。

7ー39　modff（ノーマル・モデルのみ）　数学関数

【機　　能】

・小数部と整数部を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  modff (float  x,  float  *iptr) ;`

関 数 名	引　　数	返 り 値
modff	x ･･･ 演算を行う数値 iptr ･･･ 整数部へのポインタ	正常時 ･･･ xの小数部 xが非数または無限大の場合 ･･･ NaN x = ±0の場合 ･･･ ±0

【説　　明】

・浮動小数点数xを小数部と整数部に分けます。

・xと同じ符号を持つ小数部を返し，整数部はポインタiptrの指し示すところに格納します。

・xが非数の場合は，NaNを返し，ポインタiptrの指し示すところにNaNを格納します。

・xが無限大の場合は， NaNを返し，ポインタiptrの指し示すところにNaNを格納し，errnoにEDOMをセットします。

・x = ±0の場合は，±0を返し，ポインタiptrの指し示すところに±0を格納します。

7ー40　powf（ノーマル・モデルのみ）　数学関数

【機　　能】

- xのy乗を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  powf（float  x, float  y）;`

関 数 名	引　　数	返 り 値
powf	x ･･･ 演算を行う数値 y ･･･ 乗数	正常時 ･･･ x ^ y x = NaN または y = NaN， x = ＋∞ かつ y = 0， x＜0 かつ y≠整数， x＜0 かつ y = ±∞， x = 0 かつ y≦0， 　のいずれかの場合 ･･･ NaN アンダフロー時 ･･･ 非正規化数 オーバフロー時 ･･･ HUGE_VAL （オーバフローした値の符号を持ちます。） アンダフローによる有効桁数の消滅時 ･･･ ±0

【説　　明】

- x ^ yを計算します。
- 演算の結果，オーバフローが生じた場合は，オーバフローした値の符号を持つHUGE_VALを返し，errnoにERANGEをセットします。
- x = NaN または y = NaNの場合は，NaNを返します。
- x = ＋∞ かつ y = 0，x＜0 かつ y≠整数， x＜0 かつ y = ±∞，x = 0 かつ y≦0，のいずれかの場合は，NaNを返し，errnoにEDOMをセットします。
- アンダフローが生じた場合は，非正規化数を返します。
- アンダフローによる有効桁数の消滅が生じた場合は，±0を返します。

7ー41　sqrtf（ノーマル・モデルのみ）　数学関数

【機　　能】

・平方根を求めます。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float  sqrtf（float  x）;`

関 数 名	引　　数	返 り 値
sqrtf	x ･･･ 演算を行う数値	x≧0の場合 ･･･ xの平方根 x = ±0の場合 ･･･ ±0 x＜0の場合 ･･･ NaN

【説　　明】

・xの平方根を計算します。

・x＜0の領域エラーの場合は，0を返し，errnoにEDOMをセットします。

・xが非数の場合は，NaNを返します。

・xが±0の場合は，±0を返します。

7ー42　ceilf（ノーマル・モデルのみ）　数学関数

【機　　能】

- xより小さくない最小の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  ceilf（float  x）;`

関 数 名	引　　数	返 り 値
ceilf	x ･･･ 演算を行う数値	正常時 ･･･ xより小さくない最小の整数 xが非数の場合または，x = ±∞の場合 ･･･ NaN x = －0の場合 ･･･ ＋0 xより小さくない最小の整数を表現できない場合 ･･･ x

【説　　明】

- xより小さくない最小の整数を求めます。
- xが非数の場合は，NaNを返します。
- xが－0の場合は，＋0を返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- xより小さくない最小の整数を表現できない場合は，xを返します。

7－43　fabsf（ノーマル・モデルのみ）　数学関数

【機　　能】

・浮動小数点数xの絶対値を返します。

【ヘッダ・ファイル】

・math.h

【関数プロトタイプ】

・`float　fabsf（float　x）；`

関 数 名	引　　数	返 り 値
fabsf	x ･･･ 絶対値を求める値	正常時 ･･･ xの絶対値 xが非数の場合 ･･･ NaN x = －0の場合　･･･ ＋0

【説　　明】

・xの絶対値を求めます。

・xが非数の場合は，NaNを返します。

・xが－0の場合は，＋0を返します。

7ー44　floorf（ノーマル・モデルのみ）　数学関数

【機　　能】

- xより大きくない最大の整数を求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  floorf（float  x）;`

関 数 名	引　　数	返 り 値
floorf	x ･･･ 演算を行う数値	正常時 ･･･ xより大きくない最大の整数 xが非数の場合，または無限大の場合 ･･･ NaN x = －0の場合　･･･ ＋0 xより大きくない最大の整数を表現できない場合 ･･･ x

【説　　明】

- xより大きくない最大の整数を求めます。
- xが非数の場合は，NaNを返します。
- xが－0の場合は，＋0を返します。
- xが無限大の場合は，NaNを返し，errnoにEDOMをセットします。
- xより大きくない最大の整数を表現できない場合は，xを返します。

7ー45　fmodf（ノーマル・モデルのみ）　数学関数

【機　　能】

- x/yの余りを求めます。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `float  fmodf (float  x, float  y) ;`

関 数 名	引　　数	返 り 値
fmodf	x ･･･ 演算を行う数値	正常時 ･･･ x/yの余り
	y ･･･ 演算を行う数値	xが非数またはyが非数， yが±0の場合，xが±∞の場合 ･･･ NaN
		x≠∞ かつ y = ±∞の場合 ･･･ x

【説　　明】

- xーi＊yで表されるx/yの余りを計算します。iは整数です。
- y≠0の場合は，返り値はxと同じ符号を持ち，その絶対値はyの絶対値より小さくなります。
- yが±0あるいはx = ±∞の場合は，NaNを返し，errnoにEDOMをセットします。
- xが非数またはyが非数の場合は，NaNを返します。
- yが無限大の場合は，xが無限大でなければxを返します。

8ー1　_ _assertfail（ノーマル・モデルのみ）　診断関数

【機　　能】

- assertマクロのサポートをします。

【ヘッダ・ファイル】

- math.h

【関数プロトタイプ】

- `int _ _assertfail (char *_ _msg, char *_ _cond, char *_ _file, int _ _line) ;`

関 数 名	引　　数	返 り 値
_ _assertfail	_ _msg ･･･ printf関数に渡す出力変換仕様を示す文字列へのポインタ _ _cond ･･･ assertマクロの実引数 _ _file ･･･ ソース・ファイル名 _ _line ･･･ ソース行番号	不定

【説　　明】

- _ _assertfail関数は，assertマクロ（10.2　**ヘッダ・ファイル** (13) assert.hを参照してください）から情報を受け取り，printf関数を呼び，情報の出力を行い，さらにabort関数の呼び出しを行います。
- assertマクロはプログラム中に診断機能を付け加えます。assertマクロを実行するときpが偽（0と等しい）の場合，assertマクロは，偽の値をもたらした特定の呼び出しに関する情報（情報の中には，実引数のテキスト，ソース・ファイル名およびソース行番号を含みます。あとの2つはそれぞれマクロ _ _FILE_ _，および _ _LINE_ _の値とします）を _ _assertfail関数に渡します。

10. 5　スタートアップ・ルーチン，ライブラリ関数更新用バッチ・ファイル

本コンパイラは，一部の標準ライブラリ関数およびスタートアップ・ルーチンを更新するためのバッチ・ファイルを提供しています。BATディレクトリ下にあるバッチ・ファイルについて，表10－12に示します。

注意　BATディレクトリにあるd9026.78kは，ライブラリなどの更新用バッチ・ファイル起動時に使用するもので，開発用ではありません。開発時には，デバイス・ファイル（別売）が必要です。

表10－12　ライブラリ関数更新用バッチ・ファイル

バッチ・ファイル	用　途
mkstup.bat	スタートアップ・ルーチン（cstart〔n〕.asm）を更新します。 スタートアップ・ルーチンを変更した場合は，本バッチ・ファイルを使用してアセンブルを行ってください。
reprom.bat	ROM化終端ルーチン（rom.asm）を更新します。 rom.asmを更新した場合は，本バッチ・ファイルを使用してライブラリを更新してください。
repgetc.bat	getchar関数を更新します。 ディフォールトでは，SFRのP0が入力ポートに設定されています。入力ポートを変更したい場合は，getchar.asm中のPORTのEQU定義値を変更し，本バッチ・ファイルを使用してライブラリを更新してください。
repputc.bat	putchar関数を更新します。 ディフォールトでは，SFRのP0が出力ポートに設定されています。 出力ポートを変更したい場合は，putchar.asm中のPORTのEQU定義値を変更し，本バッチ・ファイルを使用してライブラリを更新してください。
repputcs.bat	putchar関数をSM78K0S対応に更新します。 SM78K0Sでputchar関数の出力を確認したい場合は，本バッチ・ファイルを使用してライブラリを更新してください。
repselo.bat	setjmp/longjmp関数の退避／復帰処理において，コンパイラの予約領域（_@KREGxx）の退避／復帰を行うようにします（ディフォールトは退避／復帰を行いません）。-QRオプションを指定する場合は，本バッチ・ファイルを使用してライブラリを更新してください。
repselon.bat	setjmp/longjmp関数の退避／復帰処理において，コンパイラの予約領域（_@KREGxx）の退避／復帰を行わないようにします（ディフォールトは退避／復帰を行いません）。-QRオプションを指定しない場合は，本バッチ・ファイルを使用してライブラリを更新してください。

10. 5. 1　バッチ・ファイルの使用法

サブディレクトリBATの下に置かれたバッチ・ファイルを使用します。アセンブラ，ライブラリアンの起動を行うバッチ・ファイルとなっているため，アセンブラ・パッケージRA78K0S Ver.1.30以上が動作する環境が必要です。バッチ・ファイルを使用する前に，RA78K0Sの実行形式ファイルがあるディレクトリを環境変数PATHで設定してください。

バッチ・ファイルは，BATと同レベルのサブディレクトリ（LIB）を作成し，その下にアセンブル後のファイルを置きます。Cスタートアップ・ルーチンおよびライブラリが，BATと同レベルのサブディレクトリLIBにインストールされている場合は，それらのファイルを上書きします。

バッチ・ファイルの使用方法は，カレント・ディレクトリをサブディレクトリBATに移動し，各バッチ・ファイルを実行します。その際，次のパラメータが必要です。

品種 = chiptype（**ターゲット・チップの種別**）
9026…μPD789026**など**

次に各バッチ・ファイルの使用方法を示します。

（1）スタートアップ・ルーチン用

- PC-9800**シリーズ，**IBM PC/AT**互換機の場合**

  mkstup　**品種**

例　`mkstup  9026`

- HP9000**シリーズ**700™, SPARCstation™**ファミリの場合**

  /bin/sh mkstup.sh　**品種**

例　`/bin/sh  mkstup.sh  9026`

（2）ROM**化ルーチン更新用**

- PC-9800**シリーズ，**IBM PC/AT**互換機の場合**

  reprom　**品種**

例　`reprom  9026`

- HP9000**シリーズ**700, SPARCstation**ファミリの場合**

  /bin/sh reprom.sh　**品種**

例　`/bin/sh  reprom.sh  9026`

（3）getchar関数更新用

・PC-9800シリーズ，IBM PC/AT互換機の場合

regetc　**品種**

例　repgetc　9026

・HP9000シリーズ700, SPARCstationファミリの場合

/bin/sh repgetc.sh　**品種**

例　/bin/sh　repgetc.sh　9026

（4）putchar関数更新用

・PC-9800シリーズ，IBM PC/AT互換機の場合

repputc　**品種**

例　repputc　9026

・HP9000シリーズ700, SPARCstationファミリの場合

/bin/sh repputc.sh　**品種**

例　/bin/sh　repputc.sh　9026

（5）putchar関数（SM78K0S対応）更新用

・PC-9800シリーズ，IBM PC/AT互換機の場合

repputcs　**品種**

例　repputcs　9026

・HP9000シリーズ700, SPARCstationファミリの場合

/bin/sh repputcs.sh　**品種**

例　/bin/sh　repputcs.sh　9026

（6）setjmp/longjmp関数更新用（復帰／退避処理あり）

・PC-9800シリーズ，IBM PC/AT互換機の場合

repselo　**品種**

例　repselo　9026

・HP9000シリーズ700, SPARCstationファミリの場合

/bin/sh repselo.sh　品種

```
例　/bin/sh  repselo.sh  9026
```

（7）setjmp/longjmp関数更新用（復帰／退避処理なし）

・PC-9800シリーズ，IBM PC/AT互換機の場合

repselon　品種

```
例　repselon  9026
```

・HP9000シリーズ700, SPARCstationファミリの場合

/bin/sh repselon.sh　品種

```
例　/bin/sh  repselon.sh  9026
```

第11章　拡張機能

この章では，ANSI（American National Standards Institute）**規格に規定されていない，本**Cコンパイラ特有の拡張機能について説明します。

本Cコンパイラの拡張機能は，ターゲット・デバイスである78K/0Sシリーズを有効的に利用するためのコードを生成します。この拡張機能すべてが常に有効とはかぎりませんので，目的にあわせて有効なもののみ使用することをお勧めします。拡張機能の効率的な使用法がこのマニュアルの**第**13**章　効率の良いコンパイラの活用法**で説明されておりますので，この章とあわせて参照してください。

本Cコンパイラの拡張機能を使ったCソース・プログラムは，マイクロコンピュータに依存した機能を利用しますが，他のマイクロコンピュータへの移植に関してはC言語レベルで互換性を持っています。このため，拡張機能を使って作成されたCソース・プログラムにおいても，簡単な修正により他のマイクロコンピュータへ移植できます。

備考　この章の説明において，“RTOS**”は，**78K/0**シリーズ リアルタイム**OS**の意味です。**

11. 1　マクロ名

本Cコンパイラは，ターゲット・デバイスのシリーズ名を示すマクロ名と，デバイス名を示すマクロ名の2種類の名前を持ちます。これらは，ターゲット・デバイス用のオブジェクト・コードを出力するためにコンパイル時のオプション，またはCソース中のデバイス種別によって指定されます。例では，_ _K0S_ _ と，_ _9026_ が指定されたことになります。

マクロ名の詳細については，9. 8　**コンパイラ定義のマクロ名**を参照してください。

【例】

コンパイル時のオプション：

```
> CC78K0S  -C9026  prime.c ...
```

デバイス種別指定：

```
#pragma pc (9026)
```

11. 2　キーワード

本Cコンパイラでは，拡張機能を実現するために次の字句をキーワードとして追加しています。これらの字句もANSI-Cのキーワードと同様，レーベルや変数名として使用できません。

キーワードは，すべて英小文字で記述します。このため，英大文字が含まれているとキーワードと判断されません。

次に本コンパイラで追加されているキーワード一覧を示します。これらのキーワードのうち，“_ _”で始まらないキーワードは，ANSI-C言語仕様のみを許可するオプション（-ZA）指定により，無効にできます（ANSI-Cキーワードについては，2. 2　**キーワード**を参照してください）。

表11－1　追加キーワード一覧

キーワード		用　途
_ _callt	callt	callt / _ _callt関数
_ _callf[注]	callf	callf / _ _callf関数
_ _sreg	sreg	sreg / _ _sreg変数
	noauto	noauto関数
_ _leaf	norec	norec / _ _leaf関数
_ _boolean	boolean	boolean型 / _ _boolean型変数
	bit	bit型変数
_ _interrupt		ハードウエア割り込み
_ _interrupt_brk[注]		ソフトウエア割り込み
_ _banked1～15[注]		バンク関数
_ _asm		ASM文
_ _rtos_interrupt[注]		RTOS用割り込みハンドラ
_ _pascal		パスカル関数
_ _directmap		絶対番地配置指定
_ _temp		テンポラリ変数

注　callf, _ _callf, _ _interrupt_brk, _ _banked1～15, _ _rtos_interruptの記述に対しては，ワーニングを出力し無視します。

(1) 関　　数

callt, _ _callt, noauto, norec, _ _leaf, _ _interrupt, _ _pascalは，修飾属性子です。これは，関数の宣言時に先頭に記述します。修飾宣言子の記述形式を次に示します。

修飾属性子　通常の宣言子　関数名　（仮引数型並び／識別子並び）

【例】

```
_ _callt  int  func  (int);
```

修飾属性子の指定は，次のものに限ります（noautoと，norec / _ _leafは，同時に指定できません）。

なお，calltと_ _callt, callfと_ _callf, norecと_ _leafは，同じ指定とみなされます。ただし，‘ _ _ ’が付加されている修飾属性子は，-ZAオプション指定時でも有効となります。

- callt
- noauto
- norec
- callt noauto
- callt norec
- noauto callt
- norec callt
- _ _interrupt
- _ _pascal
- _ _pascal noauto
- _ _pascal callt
- noauto _ _pascal
- callt _ _pascal
- callt noauto _ _pascal

(2) 変　　数

- sreg，_ _sregの指定は，C言語のregisterと同じ規定です（sregの詳細については，11. 5（3）saddr領域利用を参照してください）。
- bit，boolean，_ _boolean型の指定は，C言語のcharまたはint型指定子と同じ規定です。
  ただし，これらの型は，関数の外で定義された変数（外部変数）にのみ指定できます。
- _ _directmapの指定は，C言語の型修飾子と同じ規定です（詳細については，11. 5（32）絶対番地配置指定を参照してください）。
- _ _tempの指定は，C言語の型修飾子と同じ規定です（詳細については，11. 5（34）テンポラリ変数を参照してください）。

11.3　メ モ リ

メモリ・モデルは，ターゲット・デバイスのメモリ空間により決定します。

(1) メモリ・モデル

メモリ空間は最大64 Kバイトなので，コード部・データ部合わせて64 Kのモデルとします。

(2) レジスタ・バンク

レジスタ・バンクはありません。

(3) メモリ空間

本Cコンパイラは，次のようにメモリ空間を利用します。

表11－2　メモリ空間の利用（1/2）

(a) ノーマル・モデル（ディフォールト）の場合

アドレス		用　途	サイズ（バイト）
00	40-7FH	CALLTテーブル	64
FE	20-D7H	sreg変数，boolean型変数	184
FE	D8-E7H	レジスタ変数[注1]	16
FE	E8-EFH	norec関数の引数[注2]	8
FE	F0-F7H	norec関数のオートマティック変数[注3]	8
FE	F8-FFH	ランタイム・ライブラリの引数[注4]	8
FF	00-FFH	sfr変数	256

表11－2　メモリ空間の利用（2/2）

（b）スタティック・モデル（-SM16指定時）の場合

アドレス		用　途	サイズ（バイト）
00	40-7FH	CALLTテーブル	64
FE	20-EFH	sreg変数，boolean型変数	208
FE	F0-FFH	共有領域[注5]	16
FE	20-FFHの連続した領域	引数，オートマティック変数，ワーク用[注6]	8
FF	00-FFH	sfr変数	256

注1. レジスタ変数用として使用しなかった領域は，sreg変数，boolean型変数用領域として使用します。

2. レジスタ変数を一切使用しなかった場合，norec関数の引数用として使用しなかった領域は，sreg変数，boolean型変数用領域として使用します。

3. レジスタ変数とnorec関数の引数を一切使用しなかった場合，norec関数のオートマティック変数用として使用しなかった領域は，sreg変数，boolean型変数用領域として使用します。

4. レジスタ変数と，norec関数の引数およびオートマティック変数を一切使用しなかった場合，ランタイム・ライブラリ用引数として使用しなかった領域は，sreg変数，boolean型変数用領域として使用します。

5. -SMオプションのパラメータによってコンパイラが使用する領域は変化します。共有領域として使用しない領域は，sreg変数，boolean型変数として利用できます。

6. スタティック・モデル拡張仕様オプション（-ZM）指定時のみ有効です。

備考　レジスタ変数最適化オプション（-QR）未指定時は，注1～3の領域は常にsreg変数，boolean型変数として使用します。

11. 4　#pragma指令

#pragma指令は，ANSIでサポートされている前処理指令の一つです。#pragmaに続く文字列により，コンパイラで決められた方法で翻訳するようにコンパイラに指示するものです。#pragma指令がコンパイラによってサポートされていなければ，#pragma指令は無視されコンパイルが続けられます。指令によりキーワードの追加がある場合は，そのキーワードがCソース中にある場合にエラーが出力されます。これを避けるためには，Cソース中のキーワードを削除するか，#ifdefで切り分けます。

本Cコンパイラでは，拡張機能を実現するために次の #pragma指令をサポートしています。

なお，#pragmaの後ろに指定するキーワードは，大文字でも小文字でも記述できます。

この指令を使用した拡張機能については，11. 5　**拡張機能の使用方法**を参照してください。

表11－3　#pragma指令リスト

#pragma指令	用　　途
#pragma sfr	SFR名をcで記述する → 11. 5（4）sfr領域利用
#pragma asm	Cソース中にASM文を入れる → 11. 5（8）ASM文
#pragma vect #pragma interrupt	割り込み処理をCで記述する → 11. 5（10）割り込み関数
#pragma di #pragma ei	DI / EI命令をCで記述する → 11. 5（12）割り込み機能
#pragma halt #pragma stop #pragma nop	CPU制御命令をCで記述する → 11. 5（13）CPU制御命令
#pragma access	絶対番地アクセス関数を使用する → 11. 5（14）絶対番地アクセス関数
#pragma section	コンパイラ出力セクション名を変更し，セクション配置を指定する → 11. 5（16）コンパイラ出力セクション名の変更
#pragma name	モジュール名を変更する → 11. 5（18）モジュール名変更機能
#pragma rot	ローテート関数を使用する → 11. 5（19）ローテート関数
#pragma mul	乗算関数を使用する → 11. 5（20）乗算関数
#pragma div	除算関数を使用する → 11. 5（21）除算関数
#pragma bcd	BCD演算関数を使用する → 11. 5（22）BCD演算関数
#pragma opc	データ挿入関数を使用する → 11. 5（23）データ挿入関数
#pragma realregister	レジスタ直接参照関数を使用する → 11. 5（30）レジスタ直接参照関数
#pragma inline	標準ライブラリ関数memcpy, memsetをインライン展開する →11. 5（31）メモリ操作関数

11.5　拡張機能の使用方法

個々の拡張機能について，次の内容を説明します。

機　能：拡張機能により実現される機能を説明します。
効　果：拡張機能により得られる効果を説明します。
方　法：拡張機能の利用法を説明します。
使用例：拡張機能の使用例を示します。
制　限：拡張機能を利用する場合の制限を説明します。
説　明：使用例の説明をします。
互換性：他のCコンパイラによって開発されたCソース・プログラムを本Cコンパイラによってコンパイルする場合のCソース・プログラムの互換性を説明します。

（1）callt関数

callt関数	callt / _ _callt

【機　　能】

- callt命令は，calltテーブルと呼ばれる領域 [40H-7FH] に，呼ぶ関数のアドレスを格納し，直接関数を呼ぶよりも短いコードで関数を呼ぶことを可能にします。
- callt宣言（あるいは _ _callt宣言）された関数（callt関数と呼ぶ）の呼び出しには，関数名の先頭に？を付加した名前を使用します。呼び出しには，callt命令を使用します。
- 呼ばれる関数は，通常の関数と変わりません。

【効　　果】

- オブジェクト・コードを短縮できます。

【方　　法】

- 呼び出す関数にcallt / _ _callt属性を追加します（先頭に記述します）。

```
callt  extern  型名　関数名
_ _callt  extern  型名　関数名
```

【使 用 例】

```
_ _callt  void  func1 (void);

_ _callt  void  func1 (void) {
            ⋮
         /* 関数本体 */
            ⋮
}
```

callt関数　　callt / _ _callt

【制　　限】

- callt / _ _callt宣言された関数のアドレスは，calltテーブルに配置されます。しかし，calltテーブルへの配置はリンク時に行われるので，アセンブラ・ソース・モジュール中でcalltテーブルを利用する場合，作成するルーチンはシンボルを使いリロケータブルにします。
- callt関数の数に関するチェックはリンク時に行います。
- -ZAオプション指定時は，_ _calltが有効となり，calltは無効となります。
- calltテーブルは40H-7FHの領域です。
- 許されるcallt属性の関数の数を越えてcalltテーブルを使用した場合は，コンパイル・エラーとなります。
- -QLオプションの指定により，calltテーブルを使用します。そのため，1ロード・モジュール当たり，およびリンクするモジュールのトータルで許されるcallt属性の数は表11ー4に示すとおりとなります。
- プロローグ／エピローグ対応ライブラリ使用オプション（-ZD）指定時は，-QL4オプションは使用できません。また，プロローグ／エピローグ対応ライブラリでcalltエントリをノーマル・モデル時に2個，スタティック・モデル時に最大10個使用するため，最大数はノーマル・モデル時に2個，スタティック・モデル時に最大10個減ります。

表11ー4　-QLオプション指定時に使用できるcallt属性の関数の数

- -QQオプションを同時指定しない場合

オプション	-QL1	-QL2	-QL3	-QL4
ノーマル・モデル	30	27	13	0
スタティック・モデル	30	29	15	2

- -QQオプションを同時指定する場合

オプション	-QL1	-QL2	-QL3	-QL4
ノーマル・モデル	30	27	18	11
スタティック・モデル	30	29	20	13

- -QLオプション未使用時およびディフォールトは次のようになります。

表11ー5　callt関数の使用制限

callt関数	制 限 値
1ロード・モジュール当たりの個数	最大30
リンクするモジュールでトータルの個数	最大30

callt関数　　callt / _ _callt

【使 用 例】

（Cソース）

```
============ ca1.c ============  ============ ca2.c ============
 _ _callt extern int tsub();

 void main()                     _ _callt int tsub()
 {                              {
        int ret_val;                    int val;
        ret_val = tsub();               return val;
 }                               }
```

（コンパイラの出力オブジェクト）

ca1のモジュール

```
      EXTRN   ?tsub             ;宣言
      callt   [?tsub]           ;呼び出し
```

ca2のモジュール

```
      PUBLIC  _tsub             ;宣言
      PUBLIC  ?tsub             ;
@@CALT CSEG   CALLT0            ;セグメントへの割り付け
?tsub: DW     _tsub
@@CODE CSEG
_tsub:                          ;関数定義
```

⋮

関数本体

⋮

【説　　明】

・呼ばれる関数“ tsub() ”はcalltテーブルにアドレスを格納するためにcallt属性を加えてあります。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・キーワードcallt / _ _calltを使用していなければ修正する必要はありません。

・callt関数に変更する場合，前記の方法に従って修正します。

＜本Cコンパイラから他のCコンパイラ＞

・＃defineによって行います。詳しくは，11. 6　Cソースの修正を参照してください。

(2) レジスタ変数

レジスタ変数 register

【機　　能】

- 宣言した変数（関数引数を含む）をレジスタ（HL），saddr領域（_@KREG00～_@KREG15）に割り当てます。レジスタ宣言をしたモジュールの前処理・後処理中にレジスタあるいはsaddr領域の退避・復帰を行います。
- 参照回数に基づき割り当てを行いますので，どのレジスタ，saddr領域に割り当てるかは不定となります。
- レジスタ変数の割り当て方法の詳細については，11. 7　**関数呼び出しインタフェース**を参照してください。
- レジスタ変数は，コンパイル条件により，次のように割り当てられる領域が変わります（各オプションについては，CC78K0S C**コンパイラ ユーザーズ・マニュアル 操作編**（U14871J）を参照してください）。

1. ノーマル・モデルの場合，レジスタ変数は参照回数に基づきレジスタHL，saddr領域［FED0H-FEDFH］に割り当てます。ただし，レジスタHLには，スタック・フレームがない場合のみレジスタ変数を割り当てます。saddr領域には，-QRオプションを指定した場合のみ割り当てます。
2. スタティック・モデルの場合，レジスタ変数は参照回数に基づきレジスタDE, -SM指定で確保した_@KREGxxに割り当てます。_@KREGxxには，-ZM2オプションを指定した場合のみ割り当てます。-ZM2オプションについては，11. 5（33）**スタティック・モデル拡張仕様**を参照してください。

【効　　果】

- レジスタ，saddr領域に対する命令は，通常メモリに対する命令より短く，オブジェクト・コードの短縮，実行速度の向上が図れます。

レジスタ変数　register

【方　法】

・記憶域クラス指定子registerで，registerクラスであることを宣言します。

```
register  型名  変数名
```

【使 用 例】

```
void main(void){
        register  unsigned char  c;
                ⋮
}
```

【制　限】

・レジスタ変数の使用回数が少ない場合は，逆にオブジェクト・コードが増加することもあります（ソースの規模，内容に依存します）。

・レジスタ変数宣言は，char / int / short / long / float / double / long doubleおよびポインタに対して使用できます。

（ノーマル・モデルの場合）

・charは他の型に対して1/2の領域を，long / float / double / long doubleは2倍の領域を使用します。char同士はバイト境界を持ちますが，それ以外の場合はワード境界を持ちます。

・int/short，ポインタの場合で，1関数当たり最大8変数まで使用可能とします。9変数目からは通常のメモリに割り当てます。

・スタック・フレームがない関数の場合は，int/short，ポインタの場合で1関数あたり最大9変数まで使用可能とし，10変数目からは通常のメモリに割り当てます。

（スタティック・モデルの場合）

・charは他の領域に対して1/2の領域を使用します。

・int/short/ポインタの場合で1関数当たり最大1変数まで使用可能とします。

・2変数目からは通常のメモリに割り当てます。

・long / float / double / long doubleに対しては無効とします。

表11－6　レジスタ変数の使用制限

データ型	使用可能な数（1関数当たり）	
	ノーマル・モデル	スタティック・モデル
int/short	最大8変数	最大1変数
ポインタ	最大8変数 （ただし，スタック・フレームがない関数の場合は，最大9変数）	最大1変数

レジスタ変数　register

【使 用 例】

（Cソース）

```
void func();
void main()
{
      register int i, j;
      i = 0;    j = 1;
      i += j;
      func();
}
```

（コンパイラの出力オブジェクト）

・-SMオプション未指定時（レジスタ変数がレジスタHL**とsaddr領域に割り当てられた例）**

次のレーベルはスタートアップ・ルーチンで宣言されます（付録A　saddr**領域のレーベル一覧を参照してください）。**

```
    EXTRN       _@KREG00            ;使用するsaddr領域の参照を行う
_main:
    push        hl                  ;関数の先頭でレジスタの内容を退避する
    movw        ax, _@KREG14        ;関数の先頭でsaddrの内容を退避する
    push        ax                  ;

    movw        hl, #00H            ;関数中では次のようなコードを出力する
    movw        ax, hl              ;
    incw        ax                  ;
    movw        _@KREG14, ax        ;
    xch         a, x                ;
    add         a, l                ;
    xch         a, x                ;
    addc        a, l                ;
    movw        hl, ax              ;
    call        !_func              ;

    pop         ax                  ;関数の終わりでsaddrの内容を復帰する
    movw        _@KREG00, ax        ;
    pop         hl                  ;関数の終わりでレジスタの内容を復帰する
    ret
```

レジスタ変数　register

・-SMオプション指定時（レジスタ変数がレジスタDEに割り当てられた例）

```
_main:
  push      de                ；関数の先頭でレジスタの内容を退避する

  movw      de,#00H ; 0       ;
  movw      de,ax             ;
  incw      ax                ;
  movw      !?L0003+1,a       ;
  xch       a,x               ;
  mov       !?L0003, a        ;
  add       a,e               ;
  xch       a,x               ;
  addc      a,d               ;
  mov       de,ax             ;
  call      !_func            ;
  pop       de                ；関数の終わりでレジスタの内容を復帰する
  ret
```

【説　　明】

・レジスタ変数を使用するには，変数の記憶域クラスをregisterクラスにするだけです。

・レーベル _@KREG00等は，本Cコンパイラに添付されているライブラリ内にPUBLIC宣言されたモジュールが含まれています。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・register宣言をサポートしているコンパイラであれば修正する必要はありません。

・レジスタ変数にしたい場合は，register宣言を追加します。

＜本Cコンパイラから他のCコンパイラ＞

・register宣言をサポートしているコンパイラであれば修正する必要はありません。

・レジスタ変数がいくつまで，また，どのような領域に割り当てられるかは使用するコンパイラに依存します。

（3）saddr領域利用

saddr領域利用	sreg / _ _sreg

（1）sreg宣言による利用

【機　　能】

- sreg宣言，あるいは_ _sreg宣言された外部変数および関数内static変数（sreg変数と呼ぶ）は，自動的にsaddr［FE20H-FED7H］（ノーマル・モデル），［FE20H-FEEFH］（スタティック・モデル）領域にリロケータブルに割り当てられます。前記領域を超える場合は，コンパイル・エラーとなります。
- Cソース中におけるsreg変数は通常の変数と同様に扱います。
- char / short / int / long型のsreg変数の各ビットは，自動的に boolean型変数になります。
- 初期値なしで宣言されたsreg変数は初期値0を持ちます。
- アセンブラ・ソース中で宣言した sreg変数のうち参照できる領域は，saddr領域 [FE20H-FEFFH] です。ただし，［FED8H-FEFFH］（ノーマル・モデル），［FEF0H-FEFFH］（スタティック・モデル）はコンパイラが使用するので，注意が必要です（表11－2　メモリ空間の利用を参照してください）。

【効　　果】

- saddr領域に対する命令は，通常メモリに対する命令よりも短く，オブジェクト・コードが短縮し実行速度が向上します。

【方　　法】

- 変数を定義するモジュール中および関数の中で，sreg宣言あるいは _ _sreg宣言を行います。関数の中では，static記憶域クラス指定子が付いている変数のみsreg変数にできます。

```
sreg  型名  変数名  /  sreg  static  型名  変数名
_ _sreg  型名  変数名  /  _ _sreg  static  型名  変数名
```

- sreg外部変数を参照するモジュール中では，次の宣言を行います。関数内でも記述できます。

```
extern  sreg  型名  変数名  /  extern  _ _sreg  型名  変数名
```

saddr領域利用　　sreg / _ _sreg

【制　　限】

- const型，または関数にsreg / _ _sregを指定した場合は，ワーニング・メッセージを出力し，sreg宣言を無視します。
- char型は，他の型の半分の領域，long / float / double / long double型は2倍の領域を使用します。
- char同士はバイト境界を持ちますが，それ以外の場合はワード境界を持ちます。
- -ZA指定時は，_ _sregのみ有効となり，sregが無効となります。
- int/short，ポインタの場合で1ロード・モジュールあたり92変数まで使用可能とします（saddr領域［FE20H-FED7H］を使用した場合）。
  ただしbit, boolean型変数，レジスタ変数，norec, noauto関数を使用した場合，使用できる数は減ります（ノーマル・モデル）。
- int/short，ポインタの場合で1ロード・モジュールあたり104変数まで使用可能とします（saddr領域［FE20H-FEEFH］を使用した場合）。ただしbit, boolean型変数，共有領域を使用した場合，使用できる数は減ります（スタティック・モデル）。

次に，1ロード・モジュール中に使用可能なsreg変数の最大数を示します。

表11－7　sreg変数の使用制限

データ型	使用可能な数（1ロード・モジュールあたり）	
	saddr領域［FE20H-FED7H］使用時	saddr領域［FE20H-FEEFH］使用時
Int/short，ポインタ	最大92変数[注]	最大104変数[注]

注　bit, boolean型変数を使用した場合，使用できる数は減ります。

【使 用 例】

（Cソース）

```
extern sreg int hsmm0;
extern sreg int hsmm1;
extern sreg int *hsptr;

void main( ) {
        hsmm0 -= hsmm1;
}
```

saddr領域利用　　sreg / _ _sreg

（アセンブラ・ソース）

sreg変数の定義コードをユーザが作成する場合の例です。ただし，Cソースにextern宣言をつけない場合は，本Cコンパイラが次のコードを出力します。この場合ORG疑似命令は出力しません。

```
                PUBLIC      _hsmm0  ;宣言
                PUBLIC      _hsmm1  ;
                PUBLIC      _hsptr  ;

@@DATS          DSEG        SADDRP  ;セグメントに割り付けます。
                ORG         0FE20H  ;
_hsmm0:         DS          (2)     ;
_hsmm1:         DS          (2)     ;
_hsptr:         DS          (2)     ;
```

（コンパイラの出力オブジェクト）

関数中では次のようなコードを出力します。

```
        movw    ax, _hsmm0
        xch     a, x
        sub     a, _hsmm1
        xch     a, x
        subc    a, _hsmm1+1
        movw    _hsmm0, ax
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・キーワードsreg / _ _sregを使用していなければ修正する必要はありません。
sreg変数に変更する場合，前記の方法に従って修正します。

＜本Cコンパイラから他のCコンパイラ＞

・＃defineによって行います。詳しくは11. 6　Cソースの修正を参照してください。これによりsreg変数は，通常の変数として扱われます。

saddr領域利用　-RD

（2）外部変数 / 外部static変数のsaddr自動割り当てオプションによる利用

【機　　能】

- 外部変数／外部static変数（const型を除く）をsreg宣言あり／なしにかかわらず，自動的にsaddr領域に割り当てます。
- nの値により，割り当てる外部変数，外部static変数を次のように指定できます。

表11－8　-RDオプションによりsaddr領域に割り当てられる変数

nの値	saddr領域に割り当てる変数
1の場合	char，unsigned char型の変数
2の場合	1の場合の変数とshort，unsigned short，int，unsigned int，enum，ポインタ型の変数
4の場合	2の場合の変数とlong，unsigned long，float，double，long double型の変数
省略した場合	すべての変数（この場合のみ構造体，共用体，配列も含む）

- sreg宣言された変数は上記の指定にかかわらず，saddr領域に割り当てます。
- extern宣言により参照する変数についても上記に従い，saddr領域に割り当てられているものとして処理します。
- このオプションによって，saddr領域に割り当てられた変数は，sreg変数と同じ扱いとなり，機能，制限は（1）で記述したとおりとなります。

【指定方法】

-RD [n]（nは1，2，または4）オプションを指定します。

【制　　限】

- -RD [n] オプションで，異なるnを指定したモジュール同士はリンクできません。

saddr領域利用　-RS

（3）内部static変数のsaddr自動割り当てオプションによる利用

【機　　能】

- 内部static変数（const型を除く）をsreg宣言あり／なしにかかわらず，自動的にsaddr領域に割り当てます。
- nの値により，割り当てる内部static変数を次のように指定できます。

表11－9　-RSオプションによりsaddr領域に割り当てられる変数

nの値	saddr領域に割り当てる変数
1の場合	char, unsigned char型の変数
2の場合	1の場合の変数とshort, unsigned short, int, unsigned int, enum, ポインタ型の変数
4の場合	2の場合の変数とlong, unsigned long, float, double, long double型の変数
省略した場合	すべての変数（この場合のみ構造体，共用体，配列も含む）

- sreg宣言された変数は上記の指定にかかわらず，saddr領域に割り当てます。
- このオプションによって，saddr領域に割り当てられた変数は，sreg変数と同じ扱いとなり，機能，制限は（1）で記述したとおりとなります。

【指定方法】

-RS [n]（nは1，2，または4）オプションを指定します。

備考　-RS [n] オプションで，異なるnを指定したモジュール同士もリンクできます。

saddr領域利用　-RK

（4）引数／オートマティック変数に対するsaddr自動割り当てオプションによる利用

【機　　能】

- 引数およびオートマティック変数（const型を除く）をsreg宣言あり／なしにかかわらず，自動的にsaddr領域に割り当てます。
- nの値により，割り当てる引数とオートマティック変数を次のように指定できます。

表11－10　-RKオプションによりsaddr領域に割り当てる変数

nの値	saddr領域に割り当てる変数
1の場合	char, unsigned char型の変数
2の場合	1の場合の変数とshort, unsigned short, int, unsigned int, enum, ポインタ型の変数
4の場合	2の場合の変数とlong, unsigned long, float, double, long double型の変数
省略した場合	すべての変数（この場合のみ構造体，共用体，配列も含む）

- sreg宣言された変数は上記の指定にかかわらず，saddr領域に割り当てます。
- このオプションによって，saddr領域に割り当てられた変数は，sreg変数と同じ扱いとなります。
- -RK［n］オプションで異なるnを指定したモジュール同士もリンクできます。

【方　　法】

- -RK［n］（n = 1, 2, 4）オプションを指定します。

【制　　限】

- スタティック・モデルのみサポートします。-SMオプション未指定時はワーニング・メッセージを出力し，自動割り当てを行いません。
- レジスタ変数宣言した引数／変数は，saddr領域には割り当たりません。
- -QVオプションが同時に指定されている場合は，レジスタDEに対する割り当てが優先されます。

saddr領域利用　-RK

【使 用 例】

（Cソース）

```
sub(int hsmarg)
{
     int hsmauto;
     hsmauto = hsmarg;
}
```

（コンパイラの出力オブジェクト）

```
@@DATS   DSEG     SADDRP
?L0003:  DS       (2)
?L0004:  DS       (2)
@@CODE   CSEG
_sub:
         movw     ?L0003, ax
         movw     ?L0004, ax       ;hsmauto
         ret
```

（4）sfr領域利用

sfr領域利用	sfr

【機　　能】

- sfr領域は，78K/0Sシリーズの各種周辺ハードウエアに対するモード・レジスタや制御レジスタなどの特別な機能が割り付けられたレジスタ群の領域です。
- sfr名の使用を宣言することにより，sfr領域に関する操作がCソース・レベルで記述できます。
- sfr変数は，初期値なし（不定）の外部変数です。
- 読み出し専用sfr変数の書き込みチェックを行います。
- 書き込み専用sfr変数の読み出しチェックを行います。
- sfr変数に不正な定数データを代入した場合，コンパイル・エラーとします。
- 使用できるsfr名は，［FF00H-FFFFH］中に割り付けてあるものです。

【効　　果】

- sfr領域に関する操作を，Cソース・レベルで記述できます。
- sfrに対する命令は，メモリに対する命令よりも短く，オブジェクト・コードの短縮，実行速度の向上を図れます。

【方　　法】

- #pragma指令により，Cソース中にsfr名を使用することを宣言します（キーワードの sfr は，大文字でも小文字でも記述できます）。

```
#pragma sfr
```

- #pragma sfrは，Cソースの先頭に記述します。ただし，#pragma PC（種別）を指定する場合は，それよりも後ろに#pragma sfrを記述します。

  次のものは#pragma sfrの前に記述できます。

  - コメント
  - 前処理指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- Cソース中では，デバイスが持つsfr名をそのまま記述します。このとき，sfr名を宣言する必要はありません。

sfr領域利用　sfr

【制　　限】

・sfr名は，大文字で記述します。小文字は通常の変数扱いとなります。

【使 用 例】

（Cソース）

```
#ifdef _ _K0S_ _
      #pragma sfr
#endif

void main()
{
      P0 -= RXB00;
      /* RXB00 = 10;      ==> error */
}
```

（コンパイラの出力オブジェクト）

宣言に関するコードは何も出力されず，関数中で次のようなコードを出力します。

```
mov    a, P0
sub    a, RXB00
mov    P0, a
```

sfr**領域利用** sfr

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・デバイスやコンパイラに依存しない部分であれば，修正する必要はありません。

＜本Cコンパイラから他のCコンパイラ＞

・“ #pragma sfr **”文を削除するか，または‘** #ifdef **’により切り分け，**sfr**変数であった変数の宣言を追加します。次に例を示します。**

```
#ifdef  _ _K0S_ _
        #pragma sfr
#else
/* 変数の宣言 */
unsigned char P0;
#endif

void main(void) {
        P0 = 0;
}
```

・sfr**またはそれに代わる機能を持つデバイスの場合，その領域をアクセスするためには専用のライブラリを作成しなければなりません。**

(5) noauto関数

noauto関数	noauto

【機　　能】

- noauto関数は，オートマティック変数に制限を設けて，前後処理（スタック・フレームの形成）のコードを出力しないようにします。
- 引数はすべてレジスタ，またはレジスタ変数用saddr領域（FEE4H-FEE7H）に割り当てます。レジスタに割り当てることができない引数があれば，コンパイル・エラーとします。
- 引数割り当てで余ったレジスタおよびレジスタ変数用saddr領域に，すべてのオートマティック変数が割り当たるときのみ，オートマティック変数を使用できます。
- レジスタ変数用saddr領域には，コンパイル時に-QRオプションを指定した場合のみ割り当てます。
- レジスタに割り当てた以外の引数は，レジスタ変数用saddr領域に格納します。
  引数の記述順に昇順に格納します（**付録**A　saddr**領域のレーベル一覧**を参照してください）。
- noauto関数をコールする際のコードは通常関数をコールする場合と同じコードを出力します。
- -SMオプション指定時は，最初にnoautoを記述した行にのみワーニング・メッセージを出力し，noauto関数をすべて通常の関数として扱います。

【効　　果】

- オブジェクト・コードの短縮と，実行速度の向上が図れます。

【方　　法】

- 関数宣言時にnoauto属性を宣言します。

```
noauto 型名　関数名
```

noauto関数　noauto

【制　　限】

- -ZA指定時は，noautoは無効となります。
- noauto関数の引数およびオートマティック変数は，型や数に制限があります。noauto関数で使用できる引数の型を次に示します。ただし，レジスタHLには，long / signed long / unsigned long, float / double / long double は割り当てず，その他の引数を割り当てます。

- ポインタ
- char / signed char / unsigned char
- int / signed int / unsigned int
- short / signed short / unsigned short
- long / signed long / unsigned long
- float / double / long double

- 使用できる引数とオートマティック変数の数は，合計サイズが最大6バイトです。
- これらの制限は，コンパイル時にチェックされます。
- 引数にregister宣言した場合，register宣言は無視します。

【使 用 例】

（Cソース）

-QRオプション指定時

```
noauto short nfunc（short a, short b, short c）;
short l, m;
void main（）
{
        static short ii, jj, kk;
        l = nfunc（ii, jj, kk）;
}
noauto short nfunc（short a, short b, short c）
{
        m = a+b+c;
        return（m）;
}
```

noauto関数　　noauto

（コンパイラの出力オブジェクト）

```
@@CODE CSEG
_main:
;line    5:  static short ii,jj,kk;
;line    6:  l = nfunc(ii,jj,kk);
  mov     a,!?L0005               ;kk
  xch     a,x
  mov     a!?L0005+1              ;kk
  push    ax
  mov     a,!?L0004               ;jj
  xch     a,x
  mov     a,!?L0004+1             ;jj
  push    ax
  mov     a,!?L0003               ;ii
  xch     a,x
  mov     a,!?L0003+1             ;ii
  call    !_nfunc                 ;関数nfunc（a, b, c）呼び出し
  pop     ax
  pop     ax
  movw    ax,bc
  mov     !_1+1,a                 ;返り値を外部変数lに代入
  xch     a,x
  mov     !_l, a
;line    7:  }
  ret
;line    8:  noauto short nfunc(short a,short b,short c)
;line    9: {
_nfunc:
  push    hl                      ;HLを退避
  xch     a,x                     ;
  xch     a, _@KREG12             ;_@KREG12に引数aをセットし，
  xch     a,x                     ;
  xch     a,_@KREG13              ;
  push    ax                      ;_@KREG12を退避
  movw    ax_@KREG14              ;
  push    ax                      ;_@KREG14を退避
  movw    ax,sp                   ;
  movw    hl,ax                   ;
  mov     a,[hl+10]               ;
  xch     a,x                     ;
  mov     a,[hl+11]               ;
  movw    _@KREG14,ax             ;_@KREG14に引数cをセット
  mov     a,[hl+8]                ;
  xch     a,x                     ;
```

noauto関数　　noauto

（コンパイラの出力オブジェクト～続き～）

```
        mov     a,[hl+9]                ;
        movw    hl,ax                   ;HLに引数bをセット
      ;line    10:  m=a+b+c;
        movw    ax,hl                   ;
        xch     a,x                     ;
        add     a,_@KREG12              ;
        xch     a,x                     ;
        addc    a,_@KREG13              ;
        xch     a,x                     ;
        add     a,_@KREG14              ;
        xch     a,x                     ;
        addc    a,_@KREG15              ;a(_@KREG12)にb(HL)とc(_@KREG14)を加算
        mov     !_m+1,a                 ;演算結果を外部変数mに代入
        xch     a,x
        mov     !_m,a
      ;line    11:  return(m);
        xch     a,x
        movw    bc,ax                   ;外部変数mの内容を返す
      ;line    12:  }
        pop     ax                      ;
        movw    _@KREG14,ax             ;_@KREG14を復帰
        pop     ax                      ;
        movw    _@KREG12,ax             ;_@KREG12を復帰
        pop     hl                      ;HLを復帰
        ret
```

【説　　明】

・この例では，ヘッダ部分でnoauto属性を追加しています。

noautoを宣言してスタック・フレームの生成を行わないようにしています。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・キーワードnoautoを使用していなければ修正する必要はありません。

・noauto関数に変更する場合，前記の方法に従って修正します。

＜本Cコンパイラから他のCコンパイラ＞

・＃defineによって行います。詳しくは，11. 6　Cソースの修正を参照してください。

（6）norec関数

norec関数	norec

【機　　能】

- 関数自身から他の関数を呼び出さない関数は，norec関数にすることができます。
- norec関数では，関数の前後処理（スタック・フレームの形成）のコードを出力しません。
- norec関数の引数は，レジスタ，norec関数の引数用saddr領域（FEE8H-FEEFH）に割り当てます。
- レジスタ，saddr領域に割り当てられない場合は，コンパイル・エラーとなります。
- 引数はレジスタあるいはsaddr領域（FEE8H-FEEFH）に格納し，norec関数を呼び出します。
- オートマティック変数は，saddr領域（FEF0H-FEF7H）に割り当てます。レジスタ変数も同様です。
- saddr領域にはコンパイル時に-QRオプションを指定した場合のみ割り当てられます。
- 引数がlong / float / double / long double型以外の場合，第1引数をレジスタAX，第2引数をレジスタDE，第3引数以降をsaddr領域に昇順に格納します。引数がlong / float / double / long double型の場合，第1引数からsaddr領域へ昇順に格納します。ただし，レジスタAX, DEに格納されるのは，引数の型によらず1引数ずつのみです。
- AXに格納された引数は，norec関数の先頭で，DEに格納された引数がなければDEにコピーされ，DEに格納された引数があれば_@RTARG6, 7にコピーされます。
- オートマティック変数がlong / float / double / long double型以外の場合，引数の割り当て後，余っていれば宣言された順にDE, _@RTARG6, 7, _@NRARG0,1, …の順に格納していきます。
  オートマティック変数がlong / float / double / long double型の場合，引数の割り当て後，余っていれば宣言された順に，_@NRARG0,1, …の順に格納していきます。
  残りの変数は宣言された順にsaddr領域に格納されます（付録A　saddr領域のレーベル一覧を参照してください）。

【効　　果】

- オブジェクト・コードが短縮でき，プログラムの実行速度が向上します。

【方　　法】

- 関数の宣言時に，norec属性を宣言します。

```
norec　型名　関数名
```

- norec の代わりに _ _leaf の記述も可能です。

norec関数　norec

【制　限】

・norec関数中から他の関数は，呼び出せません。
・norec関数の引数およびオートマティック変数には，サイズや数の制限があります。
・-ZA指定時は，norecは無効となり，__leafのみ有効となります。
・-SMオプション指定時は，最初にnorecを記述した行にのみワーニング・メッセージを出力し，norec関数をすべて通常の関数として扱います。
・引数，オートマティック変数に関する制限は，コンパイル時にチェックし，エラーとします。
・引数およびオートマティック変数にレジスタ宣言した場合は，レジスタ宣言を無視します。
・norec関数で使用できる引数およびオートマティック変数の型を次に示します。
なお，char / signed char / unsigned char同士であれば，連続してsaddr領域に割り当てますが，それ以外の型と連続する場合は，2バイト・アラインで割り当てます。

・ポインタ
・char / signed char / unsigned char
・int / signed int / unsigned int
・short / signed short / unsigned short
・long / signed long / unsigned long
・float / double / long double

（-QRオプション指定なしの場合）

・使用できる引数の数は，long / float / double / long double型以外の場合は2変数で，long / float / double / long double型は使用できません。
・norec関数内で使用できるオートマティック変数は，long / float / double / long double型以外の場合は引数で使用せず余ったバイト数分で，最大4バイトです。long / float / double / long double型は使用できません。

（-QRオプション指定ありの場合）

・使用できる引数の数は，long / float / double / long double型以外の場合は6変数で，long / float / double / long double型の場合は2変数です。
・norec関数内で使用できるオートマティック変数は，引数で使用せず余ったバイト数分とsaddrのサイズ分で，long / float / double / long double型以外の場合は最大20バイト，long / float / double / long double型の場合は最大16バイトです。
・これらの制限はコンパイル時にチェックしエラーとします。

norec関数　　norec

【使 用 例】

（Cソース）

```
norec int  rout (int  a, int  b, int  c);

int  i,  j;
void main ( ) {
        int  k, l, m;
        i = l + rout (k, l, m) + ++k ;
}

norec  int rout (int a, int b, int c)
{
        int x, y;
        return (x + (a<<2));
}
```

norec関数　　norec

（コンパイラの出力オブジェクト）

-QRオプション指定ありの場合

```
                EXTRN       _@NRARG0            ;使用するsaddr領域の参照を行う。
                EXTRN       _@NRARG1            ;
                EXTRN       _@NRARG6            ;
                     ⋮
                _@NRARG0    ←m                  ;引数をsaddr領域に格納する
                     ⋮
                de          ←1                  ;引数をDEに格納する
                     ⋮
                ax          ←k                  ;引数をAXに格納する
                call        !_rout              ;norec関数を呼び出す


    _rout:
                movw         _@RTARG6, ax
                                                 ;saddr領域から引数を受け取る
                mov          c, #02H
                xch          a, x
                add          a, a
                xch          a, x
                rolc         a, 1
                dbnz         c, $$-5
                xch          a, x
                add          a, _@NRARG1         ;saddr領域のオートマティック変数を使用する
                xch          a, x                ;
                addc         a, _@NRARG1+1       ;saddr領域のオートマティック変数を使用する
                movw         bc, ax              ;
                ret
```

【説　　明】

rout関数の定義に，norec関数であることを示すためのnorec属性を付けます。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・キーワードnorecを使用していなければ修正する必要はありません。

・norec関数に変更する場合，前記の方法に従って修正します。

＜本Cコンパイラから他のCコンパイラ＞

・＃defineによって行います。詳しくは，11. 6　Cソースの修正を参照してください。

（7）bit型変数

bit型変数	bit
boolean型変数	boolean
	_ _boolean

【機　　能】

- bit，boolean型変数は，1ビットのデータとして扱われ，saddr領域に配置されます。
- bit，boolean型変数は初期値なし（不定）の外部変数と同様に扱います。
- このビット変数に対してコンパイラは，次のビット操作命令を出力します。

```
SET1, CLR1, NOT1, BT, BF命令
```

【効　　果】

- C記述でアセンブラ・ソース・レベルのプログラミング，saddr，sfr領域へのビット・アクセスが可能になります。

【方　　法】

- bit，boolean型変数を使用するモジュール中でbit，boolean型宣言を行います。
- bit の代わりに _ _boolean の記述もできます。

```
bit 変数名
boolean 変数名
_ _boolean 変数名
```

- bit，boolean型変数を参照するモジュール中でextern bit (boolean) 宣言を行います。

```
extern bit 変数名
extern boolean 変数名
extern _ _boolean 変数名
```

- char / int / short / long型のsreg変数（配列の要素，構造体のメンバを除く），および8ビットのsfr変数は自動的にbit型変数としても使用可能になります。

```
変数名.n （nは0-31）
```

bit型変数	bit
boolean型変数	boolean
	_ _boolean

【制　　限】

- bit，boolean型変数同士の演算は，キャリー・フラグを使用して行われます。このため，各ステートメント間でのキャリー・フラグの内容は保証できません。
- 配列の定義／参照はできません。
- 構造体，共用体のメンバとして使用できません。
- 関数の引数の型として使用できません。
- オートマティック変数（スタティック・モデル以外）の型として使用できません。
- bit型変数のみで，1ロード・モジュール当たり最大1472変数まで使用できます（saddr領域［FE20H-FED7H］を使用した場合）（ノーマル・モデル）。
- bit型変数のみで，1ロード・モジュール当たり最大1664変数まで使用できます（saddr領域［FE20H-FEEFH］を使用した場合）（スタティック・モデル）。
- 初期値ありで宣言することはできません。
- const宣言とともに記述された場合は，const宣言を無視します。
- 表11－11に示した演算子による定数との演算は，0，1のみ可能となります。
- ＊，&（ポインタ参照，アドレス参照），sizeof演算を行うことはできません。
- -ZAオプション指定時は，_ _booleanのみ有効となります。

表11－11　定数0か1のみ使用する演算子（ビット型変数使用時）

分　　類	演 算 子	分　　類	演 算 子
代入	=		
ビットごとのAND	&，&=	ビットごとのOR	|，|=
ビットごとのXOR	^，^=		
論理AND	&&	論理OR	||
等しい	==	等しくない	!=

備考　sreg変数を使用した場合と，-RD, -RS, -RK（saddr自動割り当てオプション）指定時には，使用できる数は減ります。

bit型変数	bit
boolean型変数	boolean
	_ _boolean

【使 用 例】

（Cソース）

```
#define ON 1
#define OFF 0

extern bit data1;
extern bit data2;

void main()
{
        data1 = ON;
        data2 = OFF;
        while (data1)    {
                data1 = data2;
                testb();
        }

        if (data1 && data2) {
                chgb();
        }
}
```

（アセンブラ・ソース）

bit型変数の定義コードをユーザが作成する場合を示します。ただし，extern宣言を付けない場合はコンパイラが次のコードを出力します。この時にはORG疑似命令は出力しません。

```
PUBLIC  _data1                  ;宣言
PUBLIC  _data2

@@BITS  BSEG                    ;セグメントへの割り付け
        ORG     0FE20H
_data1  DBIT
_data2  DBIT
```

bit**型変数**	bit
boolean**型変数**	boolean
	_ _boolean

（コンパイラの出力オブジェクト）

関数中では次のようなコードを出力します。

```
    set1    _data1                  （初期化）
    clr1    _data2                  （初期化）
    bf      _data1, $?L0001         （判断）
    bf      _data1, $?L0005         （論理AND式）
    bf      _data2, $?L0005         （論理AND式）
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **キーワード**bit，boolean，_ _boolean**を使用していなければ修正する必要はありません。**
- bit，boolean**型変数に変更する場合，前記の方法に従って修正します。**

＜本Cコンパイラから他のCコンパイラ＞

- ＃define**によって行います。詳しくは，**11. 6　C**ソースの修正を参照してください（この変更により，**bit, boolean**型変数は通常の変数として扱われます）。**

(8) ASM文

ASM文	#asm　#endasm _ _asm

【機　　能】

(a) #asm～#endasm

- 本Cコンパイラが出力するアセンブラ・ソース・ファイル中に，ユーザが記述したアセンブラソースを埋め込みます。
- #asmの行と#endasmの行は出力しません。

(b) _ _asm

- 文字列リテラルにアセンブリ・コードを記述することで，アセンブリ命令を出力し，アセンブラ・ソース中に挿入します。

【効　　果】

- Cソースのグローバル変数をアセンブラ・ソースで操作できます。
- Cソースには記述できない機能を実現できます。
- Cコンパイラが出力したアセンブラ・ソースをハンド・オプティマイズし，Cソース中に埋め込むことにより，効率の良いオブジェクトが得られます。

【方　　法】

(a) #asm～#endasm

- #asmでアセンブラ・ソースの開始を示し，#endasmでアセンブラ・ソースの終了を示します。アセンブラ・ソースは #asm，#endasmの間に記述します。

```
#asm
  ⋮            /* アセンブラ・ソース */
#endasm
```

ASM文	#asm　#endasm _ _asm

(b) _ _asm

- ASM文を記述するモジュールの先頭で，#pragma asm 指定により_ _asm の使用を宣言します（#pragma以降のキーワードは，大文字でも小文字でも記述できます）。
- 次の項目は，#pragma asmの前に記述できます。
  - コメント
  - 他の #pragma 指令
  - 前処理指令のうち，変数の定義／参照，関数の定義／参照を生成しないもの
- Cソース中に次の形式で記述します。

```
_ _asm （文字列リテラル）；
```

- 文字列リテラルの記述方法はANSIに準拠し，エスケープ文字列（¥n：改行，¥t：タブなど）や ¥ による行の継続，文字列の連結などの記述ができます。

【制　　限】

- ＃asmのネストは許されません。
- ASM文を使用した場合，オブジェクト・モジュール・ファイルは生成されず，アセンブラ・ソース・ファイルが生成されます。
- _ _asmは，小文字の記述のみ許します。大文字や大文字小文字混在で記述された場合，ユーザ関数とみなします。
- -ZAオプション指定時は，_ _asmのみ有効となります。
- “ #asm～#endasm ”および _ _asmは，Cソースの関数中にしか記述できません。したがって，アセンブラ・ソースはセグメント名 @@CODEのCSEGに出力されます。

ASM文	#asm　#endasm _ _asm

【使 用 例】

（a）#asm～#endasm

（Cソース）

```
void main ( ) {
        #asm
                callt   [init]
        #endasm
}
```

（コンパイラの出力オブジェクト）

ユーザの記述したアセンブラ・ソースをアセンブラ・ソース・ファイルへ出力します。

```
@@CODE  CSEG
_main:
        callt   [init]
        ret
        END
```

【説　　明】

・#asmと#endasmの間をアセンブル・ソースとしてアセンブラ・ソース・ファイルへ出力します。

ASM文	#asm　#endasm _ _asm

(b) _ _asm

(Cソース)

```
#pragma asm

int a, b;

void main( ) {
        _ _asm("¥tmovw ax, _a ¥t;ax <- a");
        _ _asm("¥tmovw _b, ax ¥t;b <- ax");
}
```

(アセンブラ・ソース)

```
@@CODE   CSEG
_main:
        movw ax, _a     ;ax <- a
        movw _b, ax     ;b <- ax
        ret
        END
```

【互 換 性】

- **#asmをサポートしているCコンパイラには，そのCコンパイラで指定されるフォーマットに従って修正してください。**
- **ターゲット・デバイスが異なる場合，アセンブラ・ソース部分を修正してください。**

(9) 漢　　字

漢　　字	/* 漢字 */ // 漢字

【機　　能】

- Cソースのコメント文中に漢字を記述できます。
- コメント中の漢字はコメントとして扱われコンパイルの対象とはしません。
- コメント中で使用される漢字のコードを，オプションまたは環境変数により選択できます。オプションの指定がない場合，環境変数 LANG78Kに設定されたものが設定されます。
- オプションと環境変数 LANG78K の両方が指定されている場合は，オプションで指定したものが有効になります。
- 環境変数LANG78KにEUCと設定された場合は，コメント中の漢字種別をEUCコードと解釈します。
- 環境変数LANG78KにSJISと設定された場合は，コメント中の漢字種別をシフトJISコードと解釈します。
- 環境変数LANG78KにNONEと設定された場合は，コメント中に漢字コードがないと解釈します。

なお，ディフォールトは，次のとおりになっています。

〈Windows™ベースの場合〉

- 日本語Windows環境下ではSJIS，それ以外ではNONEを設定したものとします。

〈EWSベースの場合〉

- EUCを設定したものとします。

【効　　果】

- 理解しやすいコメントを書くことができ，Cソースの管理が容易になります。

漢　字	/* 漢字 */ // 漢字

【方　法】

・コンパイラ・オプションまたは環境変数のいずれかにより，漢字コードを設定します（ディフォールトの設定で良い場合は，設定の必要はありません）。

(a) コンパイラ・オプションによる設定

表11－12のオプションのうち，いずれかを指定します。

表11－12　漢字オプション

オプション	説　明
-ZS	SJIS（シフトJISコード）
-ZE	EUC（EUCコード）
-ZN	NONE（漢字コードなし）

(b) 環境変数 LANG78Kによる設定

・EUC，SJIS，またはNONEのいずれかを設定します（autoexec.bat，.cshrc等のファイルに必要に応じ記述します）。

・EUC，SJIS，NONEは，大文字でも小文字でも記述できます。

・Cソースのコメント文中に漢字（環境変数 LANG78Kに EUCを設定した場合は EUCコード，SJISを設定した場合はシフトJISコード）を記述します。

```
SET LANG78K = EUC        （EUCコードの場合）
SET LANG78K = SJIS       （シフトJISコードの場合）
SET LANG78K = NONE       （漢字コードがない場合）
```

【制　限】

・OSで漢字をサポートしていない場合は，漢字を使用できません。

・漢字が記述できるのは，コメント文中のみです。

漢　字	/* 漢字 */ // 漢字

【使 用 例】

（Cソース）

```
void main()     /*  main関数  */
{
                /*  コメント文  */
}
```

（コンパイラの出力オブジェクト）

アセンブラ・ソース中に漢字種別情報を出力します。

```
$KANJICODE SJIS
```

アセンブラ・ソース中にCソースを出力する場合，コメント中の漢字も出力します。

```
;line 1         void main()     /*  main関数  */
;line 2         {
;line 3                         /*  コメント文  */
```

【説　明】

・Cソースのコメント文中にのみ漢字を使えます。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・コメント文を書ける以外の場所（“ /*･･･*/ ”または“ //･･･改行 ”の外）に漢字がある場合，修正しなければなりません。

・漢字コードが違う場合は，漢字コードの変換が必要です。

＜本Cコンパイラから他のCコンパイラ＞

・コメント中に漢字を書けるCコンパイラに対しては，Cソースの修正はありません。

・コメント中に漢字を書けないCコンパイラの場合は，Cソースの漢字を削除しなければなりません。

（10）割り込み関数

割り込み関数　　#pragma vect
#pragma interrupt

【機　　能】

- 記述された関数名のアドレスを，指定された割り込み要求名に対応する割り込みベクタ・テーブルに登録します。
- 割り込み関数では，次のもののうち，使用しているもの（ASM文中で使用されているものは除く）をスタックに退避／復帰を行うためのコードを，割り込み関数の先頭と終わりに出力します。

① レジスタ
② レジスタ変数用saddr領域
③ norec関数の引数／auto変数用saddr領域（使用の有無を問わない）
④ ランタイム・ライブラリ用saddr領域（ノーマル・モデルのみ）

ただし，割り込み関数の指定や状況によっては，次のとおり，退避／復帰領域が異なります。

- 無変更指定時は，レジスタの退避／復帰，およびsaddr領域の退避／復帰を行うためのコードを使用の有無にかかわらず出力しません。
- 無変更指定がない場合で，割り込み関数内に関数呼び出しがある場合は，レジスタに関しては，使用／未使用にかかわらず，全領域を退避／復帰します。

（ノーマル・モデルの場合）

- コンパイル時に-QRオプションを指定しない場合は，レジスタ変数用saddr領域，norec関数の引数／auto変数用のsaddr領域は未使用のため，退避／復帰コードを出力しません。
  なお，全退避コードの方がサイズが小さい場合は，全退避コードを出力します。
- 以上をまとめると，退避／復帰領域は表11－13のようになります。

割り込み関数　　#pragma vect
#pragma interrupt

表11－13　割り込み関数使用時の退避／復帰領域

復帰／退避領域	NO BANK	関数コールあり		関数コールなし	
		-QRなし	-QRあり	-QRなし	-QRあり
使用レジスタ	×	×	×	○	○
全レジスタ	×	○	○	×	×
使用ランタイム・ライブラリ用saddr領域	×	×	×	○	○
全ランタイム・ライブラリ用saddr領域	×	○	○	×	×
使用レジスタ変数用saddr領域	×	×	○	×	○
norec関数の引数／auto変数用全saddr領域	×	×	○	×	×

○：退避する
×：退避しない

（スタティック・モデルの場合）

・コンパイル時に-SMオプションを指定した場合は，レジスタ変数用saddr領域，norec関数の引数/auto変数用のsaddr領域，およびランタイム・ライブラリ用のsaddr領域は存在しないため，レジスタ退避／復帰コードのみを出力し，saddr領域の退避／復帰領域は出力しません。
ただし，leafwork1-16の指定があった場合は，共有領域の上位アドレスから，バイト数をスタックに退避／復帰を行うコードを，割り込み関数の先頭と終わりに出力します（-ZMオプション未指定時は，11. 5（24）**スタティック・モデル**を参照してください。-ZMオプション指定時は，11. 5（33）**スタティック・モデル拡張仕様**を参照してください）。

注意　割り込み関数中にASM**文があり，その中でレジスタやコンパイラの予約領域（上に示した表にある領域）を用いる場合は，その領域の退避はユーザの責任となります。**

割り込み関数　　#pragma vect
#pragma interrupt

【効　　果】

- **Cソース・レベルで割り込み関数の記述が可能となります。**
- **割り込み要求名を認識するため，ベクタ・テーブルのアドレスを意識する必要がありません。**

【方　　法】

- **#pragma指令により割り込み要求名，関数名，スタック切り替え，レジスタおよび使用するsaddr領域の退避／復帰を指定します。なお，#pragma 指令はCソースの先頭に記述します（割り込み要求名に関しては，ご使用のターゲット用デバイス・ユーザーズ・マニュアルを参照してください）。**
- **#pragma　PC(種別)を記述する場合は，それよりも後ろにこの#pragma指令を記述します。次の項目は本#pragma指令の前に記述できます。**

  ・コメント
  ・プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

#pragma △ vect（**または**interrupt）△ **割り込み要求名** △ **関数名** △

［スタック切り替え指定］△ [**スタック使用指定** / **無変更指定** / **共有領域退避/復帰指定** / **退避／復帰対象**]

割り込み要求名　：　**大文字で記述します。ご使用のターゲット・デバイスのユーザーズ・マニュアルを参照してください（例**　NMI, INTP0**など）。**

関数名　：　**割り込み処理を記述した関数名**

スタック切り替え指定　：　SP = **配列名［＋オフセット位置］（例**　SP = buff+10）
配列は，unsigned char**で定義してください（例**　unsigned char buff [10]；）。

割り込み関数　　#pragma vect
#pragma interrupt

スタック使用指定　：　STACK（ディフォールト）

無変更指定　：　NOBANK

共有領域退避／復帰指定：　leafwork1-16（-SMオプション指定時のみ）

退避／復帰対象　：　SAVE_R　退避／復帰対象をレジスタに限定
SAVE_RN　退避／復帰対象をレジスタ，_@NRATxxに限定
（-SM, -ZMオプション指定時のみ）

△　：　スペース

【制　限】

- レジスタ・バンク指定は，サポートしません。
- 割り込み要求名は大文字で記述します。
- 1モジュール単位でのみ割り込み要求名の重複チェックを行います。
- 優先順位指定フラグ・レジスタ，割り込みマスク・フラグ・レジスタ等の内容によりベクタ割り込み処理中に重複して割り込み（同じ，または他の割り込み）が発生した場合や無変更指定の場合，レジスタの内容を書き換えてしまい，不具合となる可能性がありますが，コンパイラはこれをチェックできません。
- 割り込み関数は，callt / noauto / norec / _ _callt / _ _leaf / _ _pascalを指定できません。
- 割り込み関数は，引数，返り値を持てないので，void型で指定します（例　void func (void)；）。
- 割り込み関数中にASM文が存在しても，全退避のコードは出力しません。したがって，割り込み関数中のASM文中でコンパイラ予約領域等を使用する場合，またはASM文中で関数コールを行う場合の退避はユーザが行う必要があります。
- #pragma vect / #pragma interrupt指定で退避先として無変更指定，レジスタ・バンク指定，およびスタック切り替え指定をした関数が同一モジュール内で定義されなかった場合，ワーニングを出力し退避先指定，スタック切り替えを無視します。この場合，ディフォールトのスタックが使用されます。

割り込み関数　#pragma vect
#pragma interrupt

- スタック切り替えを指定した場合，配列名シンボルにオフセットを加算した位置にスタック・ポインタを切り替えます。配列名の領域の確保は #pragma指令では行いませんので，別途グローバルのunsigned char型配列として定義する必要があります。
- 関数の先頭にスタック・ポインタを切り替えるコードを，関数の最後にスタック・ポインタを元に戻すコードを生成します。
- スタック切り替え用の配列にsreg / _ _sregキーワードを付加した場合，属性が違う同名の変数が複数定義されたとみなし，コンパイル・エラーとなります。なお，-RDオプションによりsaddr領域に配列を配置させることは可能ですが，スタックとして使用されるため，コードおよびスピードに関し，効率が良くなることはありません。スタック以外の用途でsaddr領域を使用することをお勧めします。
- スタック切り替え指定は，無変更指定とは同時に指定できません。指定した場合はエラーとなります。
- スタック切り替え指定は，スタック使用指定より先に記述しなければなりません。スタック切り替え指定を後に記述した場合はエラーとなります。
- -SMオプション無指定時にleafwork1-16を指定した場合は，ワーニングを出力し，共有領域退避／復帰指定を無視します。

割り込み関数　　#pragma vect
#pragma interrupt

【使 用 例】

（Cソース）

共有領域がある場合（スタティック・モデルのみ）

```
#pragma interrupt INTP0 inter leafwork4
void func();
void inter()
{
      func();
}
```

（コンパイラの出力オブジェクト）

```
    EXTRN      _@KREG12
    EXTRN      _@KREG14

@@CODE CSEG
_inter:
    push       ax                  ;レジスタの退避
    push       bc                  ;      〃
    push       hl                  ;      〃
    movw       ax,_@KREG12         ;共有領域の退避
    push       ax                  ;      〃
    movw       ax,_@KREG14         ;      〃
    push       ax                  ;      〃
    call       !_func
    pop        ax                  ;共有領域の復帰
    movw       _@KREG14,ax         ;      〃
    pop        ax                  ;      〃
    movw       _@KREG12,ax         ;      〃
    pop        hl                  ;レジスタの復帰
    pop        bc                  ;      〃
    pop        ax                  ;      〃
    reti

@@VECT06      CSEG   AT       0006H
_@vect06:
    DW         _inter
```

割り込み関数　　#pragma vect
#pragma interrupt

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **割り込み関数を使用していなければ修正する必要はありません。**
- **割り込み関数に変更する場合は，前記の方法に従って修正します。**

＜本Cコンパイラから他のCコンパイラ＞

- #pragma vect／#pragma interrupt **指定を削除すれば，通常の関数として扱われます。**
- **割り込み関数として使用する場合は，各コンパイラの仕様により変更が必要です。**

（11）割り込み関数修飾子（_ _interrupt）

割り込み関数修飾子 _ _interrupt

【機　　能】

- 関数を _ _interrupt 修飾子で宣言することにより，その関数はハードウエア割り込み関数とみなされ，ノンマスカブル／マスカブル割り込み関数のためのリターン命令 RETI により復帰します。
- この修飾子で宣言された関数は，（ノンマスカブル／マスカブル）割り込み関数とみなされ，次の①～④の内コンパイラの作業領域として使用しているものをスタックに退避／復帰します。
  ただし，この関数中に関数コールの記述がある場合は，全領域をスタックに退避します。

① レジスタ
② レジスタ変数用saddr領域
③ norec関数の引数／auto変数用saddr領域（使用の有無を問わない）
④ ランタイム・ライブラリ用saddr領域

備考　コンパイル時に -QRオプションを指定しない場合（ディフォールト）は，②，③の領域は未使用のため，退避／復帰コードを出力しません。また，コンパイル時に-SMオプションを指定した場合は，②，③，④の領域は未使用のため，退避／復帰コードを出力しません。

【効　　果】

- この修飾子で宣言することにより，ベクタ・テーブルの設定と割り込み関数定義を別のファイルに記述できます。

割り込み関数修飾子　　_ _interrupt

【方　　法】

・割り込み関数の修飾子に _ _interruptを付加します。

＜ノンマスカブル／マスカブル割り込み関数の場合＞

```
_ _interrupt  void  func( ) {処理}
```

【制　　限】

・ソフトウエア割り込みがないので，_ _interrupt_brkはサポートしません。
_ _interrupt_brkキーワードが最初に出現した箇所に対しワーニング・メッセージを出力してキーワードを無視し，通常の関数として処理します。

・割り込み関数は，callt / noauto / norec / _ _callt / _ _leaf / _ _pascalを指定できません。

【注　　意】

・この修飾子を宣言するだけでは，ベクタ・アドレスの設定を行いません。ベクタ・アドレスの設定は #pragma vect / interrupt指令あるいはアセンブラ記述などにより，別途行う必要があります。

・saddr領域，レジスタの退避先はスタックとなります。

・#pragma vect (またはinterrupt)…によりベクタ・アドレスの設定，退避先の変更を行った場合でも，同一ファイル中に関数定義がない場合は，退避先の変更は無視され，ディフォールトであるスタックになります。

・#pragma vect (またはinterrupt) …の指定と同一ファイルに割り込み関数を定義する場合は，この修飾子を記述しなくても #pragma vect (またはinterrupt)…で指定された関数名を割り込み関数と判断します（#pragma vect / interrupt指令の詳細については，11. 5（10）割り込み関数の方法を参照してください）。

割り込み関数修飾子　　_ _interrupt

【使 用 例】

次のように割り込み関数宣言，定義をします。ベクタ・アドレスの設定コードは，#pragma interrupt**により生成されます。**

```
#pragma  interrupt  INTP0  inter

_ _interrupt void inter();                    /* プロトタイプ宣言 */
 _ _interrupt void inter(){  処理  };         /* 関数本体 */
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **割り込み関数をサポートしていなければ修正は必要ありません。**
- **割り込み関数に変更したい場合は上記の方法に従って変更します。**

＜本Cコンパイラから他のCコンパイラ＞

- #define **により可能です。通常の関数として扱えます。**
- **割り込み関数として使用する場合は，各コンパイラの仕様により変更が必要です。**

（12）割り込み機能

割り込み機能	#pragma　DI #pragma　EI

【機　　能】

- オブジェクトにDI，EIのコードを出力し，オブジェクト・ファイルを作成します。
- #pragma指令がない場合，DI ( )，EI ( )は通常の関数とみなされます。
- 関数中の先頭（オートマティック変数の宣言，コメント，プリプロセス指令を除く）に“ DI ()；” が記述された場合は，関数の前処理より前（関数名のレーベルの直後）に DI のコードを出力します。
- 関数の前処理のあとに DI のコードを出力する場合は，“ DI ()；”を記述する前で新たなブロックを開きます（‘ { ’ で区切ります）。
- 関数中の最後（コメント，プリプロセス指令を除く）に“ EI ()；”が記述された場合は，関数の後処理より後ろ（RETのコードの直前）に EI のコードを出力します。
- 関数の後処理の前に EI のコードを出力する場合は，“ EI()；”を記述したあとで新たなブロックを閉じます（‘ } ’ で区切ります）。

【効　　果】

- 割り込み禁止の関数を作成できます。

【方　　法】

- #pragma DI，#pragma EI指令をCソースの先頭に記述します。
  次の項目は#pragma DI，#pragma EIの前に記述できます。

  - コメント
  - 他の#pragma指令
  - 前処理指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- 関数呼び出しと同様の形式でソース中にDI ( )；，EI ( )；と記述します。
- #pragma 以降に記述する DI，EI は大文字でも小文字でも記述できます。

割り込み機能　　#pragma　DI
#pragma　EI

【制　　限】

- **この機能を使用する場合は，関数名として** DI，EI**を使用できません。**
- DI，EI**は大文字で記述します。小文字は通常の関数として扱われます。**

【使 用 例】

```
#ifdef  _ _K0S_ _
        #pragma DI
        #pragma EI
#endif
```

（Cソース1**）**

```
#pragma DI
#pragma EI
void main ()
{
        DI ()  ;
        関数本体
        EI ();
}
```

（コンパイラの出力オブジェクト）

```
_main:
        di
        前処理
        関数本体
        後処理
        ei
        ret
```

割り込み機能　　#pragma　DI
#pragma　EI

＜DI，EIを前／後処理の後と前に出力する場合＞

（Cソース2）

```
#pragma DI
#pragma EI
void main()
{
  {
        DI();
        関数本体
        EI();
  }
}
```

（コンパイラの出力オブジェクト）

```
_main:
        前処理
        di
        関数本体
        ei
        後処理
        ret
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **割り込み機能を使用していなければ修正する必要はありません。**
- **割り込み機能を使用している場合は，前記の方法に従って修正します。**

＜本Cコンパイラから他のCコンパイラ＞

- **#pragma DI，#pragma EI指令を削除するか，あるいは#ifdefで切り分けます。関数名として DI，EIを使用できます（例**. #ifdef _ _K0S_ _ **～** #endif**）。**
- **割り込み機能として使用する場合は，各コンパイラの仕様により変更が必要です。**

（13）CPU制御命令

CPU制御命令	#pragma HALT / STOP / NOP

【機　　能】

・オブジェクトに次のコードを出力し，オブジェクト・ファイルを作成します。

① HALT動作用の命令を出力します（HALT）。
② STOP動作用の命令を出力します（STOP）。
③ NOP命令を出力します。

【効　　果】

・マイクロコンピュータのスタンバイ機能をCプログラムで使用できます。
・CPUを動作させずにクロックを進められます。

【方　　法】

・#pragma HALT，#pragma STOP，#pragma NOP命令をCソースの先頭に記述します。
・次の項目は #pragma 指令の前に記述できます。

  ・コメント
  ・他の#pragma指令
  ・プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

・#pragma以降のキーワードは大文字でも小文字でも記述できます。
・関数呼び出しと同様の形式でCソース中に次のように大文字で記述します。

```
① HALT();
② STOP();
③ NOP();
```

CPU制御命令　　#pragma HALT / STOP / NOP

【制　　限】

・この機能を使用する場合は，関数名として HALT()，STOP()，NOP()を使用できません。

・HALT，STOP，NOPは大文字で記述します。小文字は通常の関数扱いとなります。

【使 用 例】

（Cソース）

```
#pragma HALT
#pragma STOP
#pragma NOP
main()
{
     HALT();
     STOP();
     NOP();
}
```

（コンパイラの出力オブジェクト）

```
     @@CODE  CSEG
     _main:
             halt
             stop
             nop
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・CPU制御命令を使用していなければ修正する必要はありません。

・CPU制御命令を使用したい場合は，前記の方法に従って修正します。

＜本Cコンパイラから他のCコンパイラ＞

・"#pragma HALT", "#pragma STOP", "#pragma NOP"文を削除あるいは#ifdefで切り分けると，関数名として HALT, STOP, NOPを使用できます。

・CPU制御命令として使用する場合は，各コンパイラの仕様により変更が必要です（＃asm, ＃endasmあるいはasm（）；など）。

(14) 絶対番地アクセス関数

絶対番地アクセス関数 #pragma access

【機　　能】

- **オブジェクトに通常の**RAM**空間をアクセスするコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。**
- #pragma**指令がない場合は，絶対番地アクセス用の関数は通常の関数とみなされます。**

【効　　果】

- C**記述により，通常のメモリ空間の特定番地のアクセスが簡単にできます。**

【方　　法】

- #pragma access**指令をC言語ソースの先頭に記述します。**
- **関数呼び出しと同じ形式でソース中に記述します。**
- **次の項目は，** #pragma access **の前に記述できます。**
  - **コメント**
  - **他の**#pragma**指令**
  - **プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの**
- #pragma **以降のキーワードは，大文字でも小文字でも記述できます。**

絶対番地アクセス用の関数名は次の4**つです。**

```
peekb, peekw, pokeb, pokew
```

絶対番地アクセス関数　　#pragma access

［絶対番地アクセス用の関数一覧］

(a) `unsigned char peekb (addr);`
`unsigned int addr;`

アドレス addr の内容1バイトを返します。

(b) `unsigned int peekw (addr);`
`unsigned int addr;`

アドレス addr の内容2バイトを返します。

(c) `void pokeb (addr, data);`
`unsigned int addr;`
`unsigned char data;`

アドレス addr が示す位置に，data の内容1バイトを書き込みます。

(d) `void pokew (addr, data);`
`unsigned int addr;`
`unsigned int data;`

アドレス addr が示す位置に，data の内容2バイトを書き込みます。

【制　　限】

・関数名として絶対番地アクセス用の関数名が使用できません。
・絶対番地アクセス用の関数は小文字で記述します。大文字は通常の関数として扱われます。

絶対番地アクセス関数 #pragma access

【使 用 例】

（Cソース）

```
#pragma access

char a;
int b;

void main()
{
        a = peekb(0x1234);
        a = peekb(0xfe23);
        b = peekw(0x1256);
        b = peekw(0xfe68);

        pokeb(0x1234, 5);
        pokeb(0xfe23, 5);
        pokew(0x1256, 7);
        pokew(0xfe68, 7);
}
```

絶対番地アクセス関数　　#pragma access

（出力アセンブラ・ソース）

```
                ⋮    ⋮
        mov     a,!01234H
        mov     !_a,a
        mov     a,0FE23H
        mov     !_a,a
        mov     a,!01256H
        xch     a,x
        mov     a,!01257H
        movw    de,#_b
        callt   [@@deist]
        movw    ax,0FE68H
        callt   [@@deist]

        mov     a,#05H
        mov     !01234H,a
        mov     0FE23H,#05H
        movw    ax,#07H
        mov     !01257H,a
        xch     a,x
        mov     !01256H,a
        movw    ax,#07H
        movw    0FE68H,ax
```

絶対番地アクセス関数 #pragma access

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **絶対番地アクセス用の関数を使用していなければ，修正は必要ありません。**
- **絶対番地アクセス用の関数に変更したい場合は，前記の方法に従って変更してください。**

＜本Cコンパイラから他のCコンパイラ＞

- **“** #pragma access **”文を削除または** #ifdef **で切り分けます。関数名として絶対番地アクセス用の関数名を使用することができます。**
- **絶対番地アクセス用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（**#asm, #endasm**あるいは**asm( )**；など）。**

(15) ビット・フィールド宣言

ビット・フィールド宣言　　**ビット・フィールド宣言**

(1) 型指定子の拡張

【機　　能】

- unsigned char 型のビット・フィールドは，バイト境界をまたがって割り付けられることはありません。
- unsigned int 型のビット・フィールドは，ワード境界をまたがって割り付けられることはありません。バイト境界をまたがって割り付けることはできます。
- 同じ型のビット・フィールドは，同じバイト単位（またはワード単位）に割り付けられます。違う型の場合は，違うバイト単位（またはワード単位）に割り付けられます。

【効　　果】

- メモリの節約，オブジェクト・コードの短縮，実行速度の向上を図れます。

【方　　法】

- ビット・フィールドの型指定子として unsigned int型に加え，unsigned char型の指定ができます。次のように宣言します。

```
struct タグ名 {
        unsigned char  フィールド名:ビット幅;
        unsigned char  フィールド名:ビット幅;
                ⋮
        unsigned int   フィールド名:ビット幅;
};
```

【使 用 例】

```
struct tagname {
        unsigned char  A:1;
        unsigned char  B:1;
                ⋮
        unsigned int   C:2;
        unsigned int   D:1;
                ⋮
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **ソースの修正は必要ありません。**
- **型指定子に** unsigned char**を使用したい場合は型指定子を変更します。**

＜本Cコンパイラから他のCコンパイラ＞

- **型指定子に** unsigned char**を使用していなければ修正は必要ありません。**
- **型指定子に** unsigned char**を使用している場合は，**unsigned int**に変更します。**

ビット・フィールド　　ビット・フィールド宣言

(2) ビット・フィールドの割り付け方向

【機　　能】

- ビット・フィールドの割り付け方向を -RB オプション指定により MSB側 からに変更します。
- -RBオプション指定がない場合は，LSB側から割り付けられます。

【方　　法】

- ビット・フィールドを MSB 側から割り付ける場合，コンパイル時に -RB オプションを指定します。
- ビット・フィールドを LSB 側から割り付ける場合，オプションは指定しません。

【使用例1】

（ビット・フィールドの宣言）

```
struct t {
    unsigned char A:1;
    unsigned char B:1;
    unsigned char C:1;
    unsigned char D:1;
    unsigned char E:1;
    unsigned char F:1;
    unsigned char G:1;
    unsigned char H:1;
};
```

【説　　明】

a ～ h は8ビット以下なので，1バイト単位中に割り付けます。

図11－1　ビット・フィールド宣言によるビット配置（使用例1）

-RBオプション指定時のMSBから割り付けたビット配置

-RBオプション無指定時のLSBから割り付けたビット配置

【使用例2】

（ビット・フィールドの宣言）

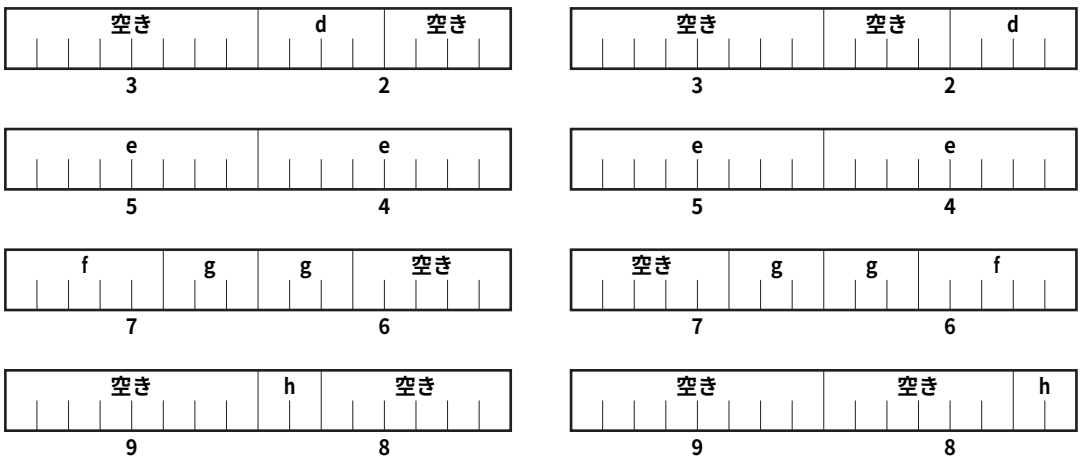
```
struct t {
    char            a;
    unsigned char   b:2;
    unsigned char   c:3;
    unsigned char   d:4;
    int             e;
    unsigned char   f:5;
    unsigned char   g:6;
    unsigned char   h:2;
    unsigned int    i:2;
};
```

【説　　明】

図11－2　ビット・フィールド宣言によるビット配置（使用例2）

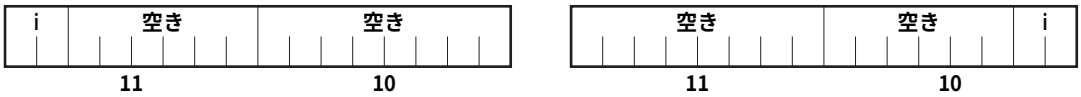
-RBオプション指定時の MSBから割り付けたビット配置	-RBオプション無指定時の LSBから割り付けたビット配置

char型のメンバaを最初のバイト単位に割り付けます。b, cは次のバイト単位から割り付けます。十分な空きがなくなれば，次のバイト単位に割り付けます。ここでは，空きが3ビットで，dが4ビットなので，dは次のバイト単位に割り付けます。



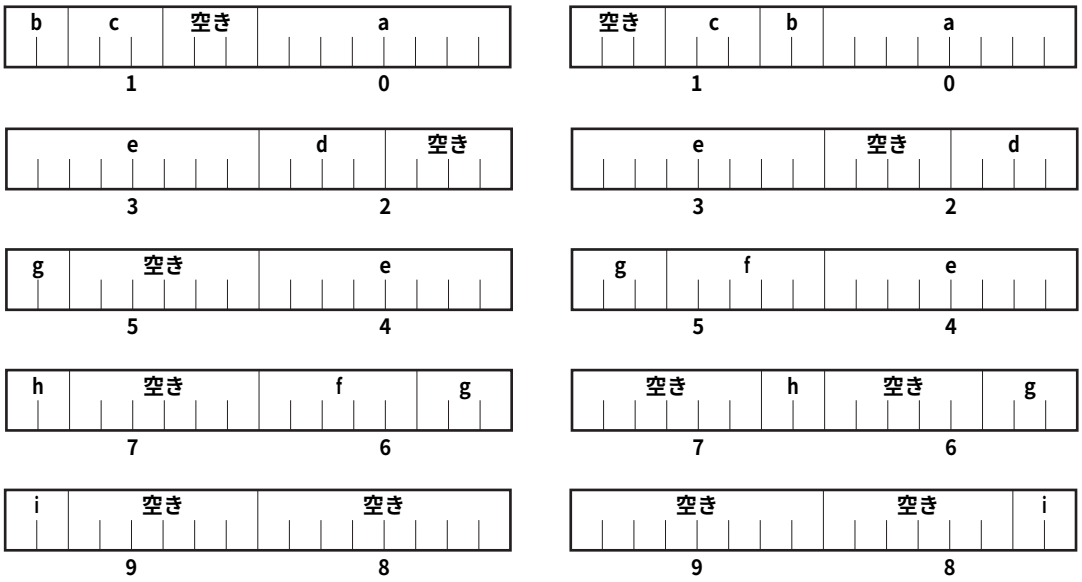
gはunsigned int型のビット・フィールドなのでバイト境界をまたがっても割り付けます。

hはunsigned char型のビット・フィールドなので，unsigned int型のビット・フィールドのgと同じバイト単位ではなく，次のバイト単位に割り付けます。



iはunsigned int型のビット・フィールドなので，次のワード単位に割り付けます。

-RCオプション指定時（構造体メンバをパッキングする）には，前記ビット・フィールドの配置は，次のとおりとなります。



備考　ビット配置図の下の数字は，構造体の先頭からのバイト・オフセット値を示します。

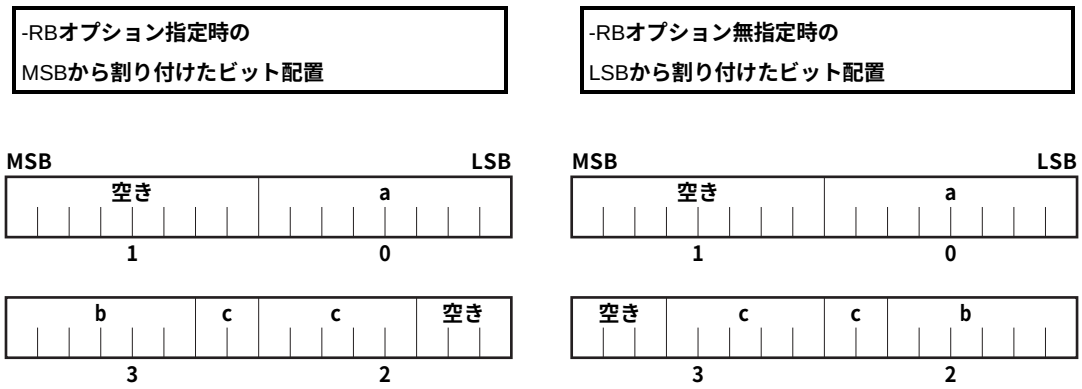
【使用例3】

（ビット・フィールドの宣言）

```
struct t {
    char          a;
    unsigned int  b:6;
    unsigned int  c:7;
    unsigned int  d:4;
    unsigned char e:3;
    unsigned int  f:10;
    unsigned int  g:2;
    unsigned int  h:5;
    unsigned int  i:6;
};
```

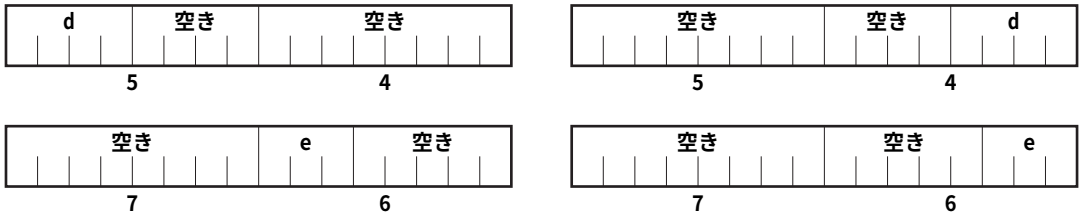
【説　　明】

図11－3　ビット・フィールド宣言によるビット配置（使用例3）



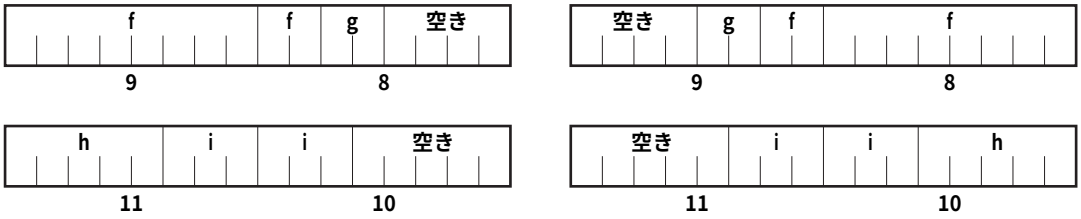
b, cはunsigned int型のビット・フィールドなので，次のワード単位から割り付けます。

dもunsigned int型のビット・フィールドなので，次のワード単位から割り付けます。



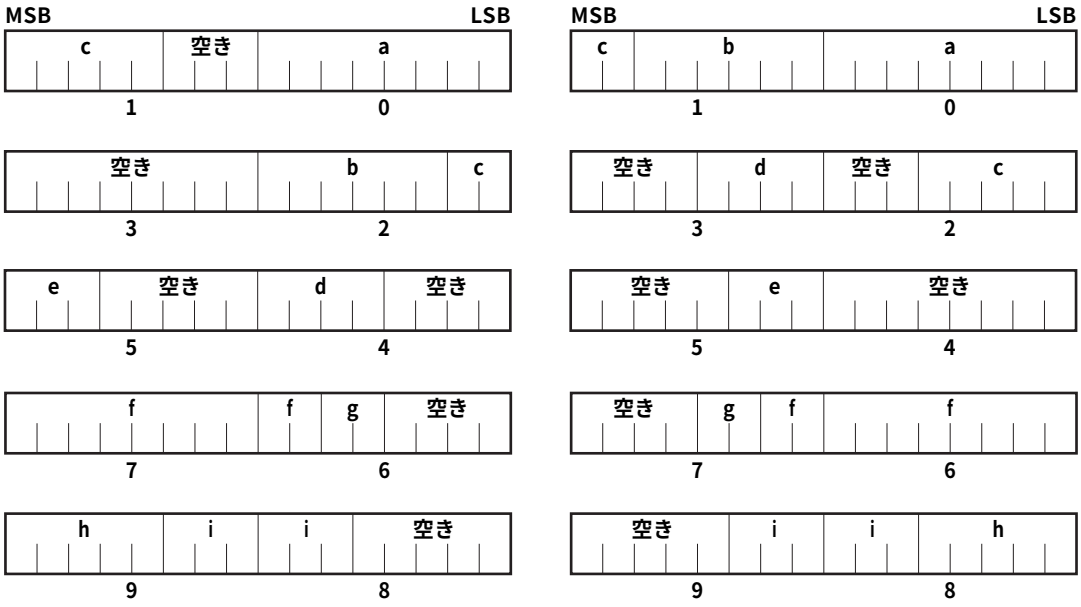
eはunsigned char型のビット・フィールドなので次のバイト単位に割り付けます。

ビット・フィールド宣言　　**ビット・フィールド宣言**



f, gとh, iはそれぞれワード単位ごとに割り付けます。

-RCオプション指定時（構造体メンバをパッキングする）には，前記ビット・フィールドの配置は，次のとおりとなります。



備考　ビット配置図の下の数字は，構造体の先頭からのバイト・オフセット値を示します。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・修正は必要ありません。

＜本Cコンパイラから他のCコンパイラ＞

・-RBオプションを指定し，ビット・フィールドが割り付けられる順序を考慮したコーディングをしている場合は，変更が必要です。

（16）コンパイラ出力セクション名の変更

#pragma section …	#pragma section …

【機　　能】

- コンパイラ出力セクション名の変更と，開始アドレスの指定を行います。開始アドレスを省略した場合は，ディフォールトの配置となります。コンパイラ出力セクション名とディフォールトの配置については，**付録**B **セグメント名一覧**を参照してください。また，開始アドレスを省略し，リンク時にリンク・ディレクティブ・ファイルを使用してセクション配置を指定できます。リンク・ディレクティブについては，RA78K0S**アセンブラ・パッケージ ユーザーズ・マニュアル 操作編**（U14876J）を参照してください。
- @@CALTセクション名を AT 開始アドレス指定付きで変更する場合は，callt関数はソース・ファイル中で他の関数より前，または後ろにまとめて記述しなければなりません。
- #pragma指令が記述された以降にデータを記述した場合，そのデータを変更セクションに配置します。再変更指令も可能であり，再変更指令以降にデータを記述した場合，そのデータを再変更セクションに配置します。変更前に定義したデータを，変更後に再定義した場合，再変更されたセクションに配置します。なお，（関数内）static変数に対しても同様に有効です。

【効　　果】

- コンパイラ出力セクションを1ファイル中に何度も変更することにより，各セクションをそれぞれ独立に配置できるようになるため，独立に配置したいデータの単位で，データを配置できます。

【方　　法】

- 次の #pragma 指令により変更するセクション名と変更後のセクション名およびセクションの開始アドレスを指定します。

  なお，この#pragma 指令は Cソースの先頭に記述します。

  #pragma PC(種別) を記述する場合は，それよりも後ろにこの#pragma指令を記述します。

  次の項目は，この#pragma指令の前に記述できます。

  - コメント
  - 前処理指令のうち，変数の定義／参照，関数の定義／参照を生成しないもの

  ただし，BSEGのすべてのセクション，DSEGのすべてのセクション，およびCSEGのうちの@@CNSTセクションは，Cソース中のどこに記述してもよく，また何度でも再変更指令ができます。元のセクション名に戻す場合は，変更セクションにコンパイラ出力セクション名を記述します。

#pragma section …　　#pragma section …

ファイルの先頭に次のような宣言をします。

```
#pragma  section  コンパイラ出力セクション名　変更セクション名　[AT 開始アドレス]
```

- #pragma以降に記述するキーワードのうち，コンパイラ出力セクション名は必ず大文字で記述してください。section， AT は大文字でも小文字でも大小文字混在でも記述できます。
- 変更セクション名の書式は，アセンブラの仕様に準拠します（セグメント名は8文字までです）。
- 開始アドレスには，C言語の16進数及び，アセンブラの16進数のみ記述できます。

【C言語の16進数】

```
0xn / 0xn…n
0Xn / 0Xn…n
 (n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

【アセンブラの16進数】

```
nH / n…nH
nh / n…nh
 (n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

- 16進数の先頭文字は数字でなければなりません。

  例　値が255の数値を16進数で表現する場合は，F の前にゼロを指定し，0FFH とする必要があります。

- CSEGのうち，@@CNSTセクション以外のセクション，つまり関数を配置するセクションは，Cソースの先頭以外（Cの本文記述後）にこの#pragma指令は記述できません。エラーになります。
- Cの本文記述後にこの#pragma指令を行った場合，オブジェクト・モジュール・ファイルは作成されず，アセンブラ・ソース・ファイルが作成されます。
- Cの本文記述後にこの#pragma指令がある場合，この#pragma指令があり，Cの本文（変数や関数の外部参照宣言を含む）の一切ないファイルはインクルードできません。エラーになります（後述のエラー記述例1を参照）。

#pragma section …　　#pragma section …

- Cの本文記述後にこの#pragma指令を行ったファイルでは，この記述以降，#include文を記述できません。エラーになります（後述のエラー記述例2を参照）。
- Cの本文のあとに#include文があった場合，この記述以降，この#pragma指令を記述できません。エラーになります（後述のエラー記述例3を参照）。

【使用例1】

セクション名@@CODEをCC1に変更し，開始アドレスを2400H番地に指定します。

（Cソース）

```
#pragma  section   @@CODE     CC1 AT 2400H

 void    main()
{
        関数本体
}
```

（出力オブジェクト）

```
 CC1     CSEG   AT  2400H
_main:
        前処理
        関数本体
        後処理
        ret
```

#pragma section …　　#pragma section …

【使用例2】

Cの本文があり，その後にこの#pragma**指令を記述する場合の記述例を示します。**

なお，//以降に配置するセクションを示します。

```
#pragma section @@DATA ??DATA
  int a1;                          // ??DATA
  sreg int b1;                     // @@DATS
  int c1 = 1;                      // @@INITと@@R_INIT
  const int d1 = 1;                // @@CNST
#pragma section @@DATS ??DATS
  int a2;                          // ??DATA
  sreg int b2;                     // ??DATS
  int c2 = 1;                      // @@INITと@@R_INIT
  const int d2 = 1;                // @@CNST
#pragma section @@DATA ??DATA2
  // ??DATAが自動的に閉じられ，??DATA2が有効となる。
  int a3;                          // ??DATA2
  sreg int b3;                     // ??DATS
  int c3 = 3;                      // @@INITと@@R_INIT
  const int d3 = 3;                // @@CNST
#pragma section @@DATA @@DATA
  // ??DATA2が閉じられ，ディフォールト@@DATAに戻る
#pragma section @@INIT ??INIT
#pragma section @@R_INIT ??R_INIT
  // @@INIT，@@R_INITの両方の名前を変えないとROM化が破綻するが，
  //それはユーザ責任。
  int a4;                          // @@DATA
  sreg int b4;                     // ??DATS
  int c4 = 1;                      // ??INITと??R_INIT
  const int d4 = 1;                // @@CNST
#pragma section @@INIT @@INIT
#pragma section @@R_INIT @@R_INIT
  // ??INIT, ??R_INITが閉じられ，ディフォールトに戻る。
#pragma section @@BITS ??BITS
  _ _boolean e4;                   // ??BITS
#pragma section @@CNST ??CNST
  char*const p = "Hello";          // pも"Hello"も??CNST
```

#pragma section …　　#pragma section …

【使用例3】

```
#pragma section @@INIT ??INIT1
#pragma section @@R_INIT ??R_INIT1
#pragma section @@DATA ??DATA1
  char c1;
  int i2;
#pragma section @@INIT ??INIT2
#pragma section @@R_INIT ??R_INIT2
#pragma section @@DATA ??DATA2
  char c1;
  int i2 = 1;
#pragma section @@DATA ??DATA3
#pragma section @@INIT ??INIT3
#pragma section @@R_INIT ??R_INIT3
  extern char c1;                    // ??DATA3
  int i2;                            // ??INIT3と??R_INIT3
#pragma section @@DATA ??DATA4
#pragma section @@INIT ??INIT4
#pragma section @@R_INIT ??R_INIT4
```

Cの本文があり，そのあとにこの#pragma指令を記述する場合の制限を，次のエラー記述例で説明します。

#pragma section …　　#pragma section …

【エラー記述例1】

```
a1.h
  #pragma section @@DATA ??DATA1     // #pragma sectionのみのファイル

a2.h
  extern int func1（void）;
  #pragma section @@DATA ??DATA2     // Cの本文があり，そのあとにこの#pragma指令が
                                     // あるファイル

a3.h
  #pragma section @@DATA ??DATA3     // #pragma sectionのみのファイル

a4.h
  #pragma section @@DATA ??DATA3
  extern int func2（void）;          // Cの本文を含むファイル

a.c
  #include "a1.h"
  #include "a2.h"
  #include "a3.h"                    // ←エラーとなる
                                     // a2.hでCの本文があり，そのあとにこの#pragma
                                     // 指令があるので，このpragma指令のみの
                                     // ファイルであるa3.hをインクルードできない。
  #include"a4.h"
```

#pragma section …	#pragma section …

【エラー記述例2】

```
b1.h
  const int i;

b2.h
  const int j;
  #include "b1.h"                    // Cの本文があり，そのあとにこの#pragma指令
                                     // を行ったファイル（b.c）ではないので，
                                     // エラーではない。

b.c
  const int k;
  #pragma section @@DATA ??DATA1
  #include "b2.h"                    // ←エラーとなる。
                                     // Cの本文があり，そのあとにこの#pragma指令
                                     // を行ったファイル（b.c）においては，include文
                                     // を記述できない。
```

#pragma section …	#pragma section …

【エラー記述例3】

```
c1.h
  extern int j;
  #pragma section @@DATA ??DATA1    // c3.h処理前にインクルードされ，処理されるた
                                    // め，エラーではない。

c2.h
  extern int k;
  #pragma section @@DATA ??DATA2    // ←エラーとなる。
                                    // c3.hでcの本文があり，そのあとに#include文
                                    // があるので，それ以降この#pragma指令は
                                    // できない。

c3.h
  #include "c1.h"
  extern int i;
  #include "c2.h"
  #pragma section @@DATA ??DATA3    // ←エラーとなる
                                    // Cの本文があり，そのあとに#include文がある
                                    // ので，それ以降この#pragma指令はできない。

c.c
  #include "c3.h"
  #pragma section @@DATA ??DATA4    //←エラーとなる。
                                    // c3.hでCの本文があり，そのあとに#include文が
                                    // あるので，それ以降この#pragma指令はでき
                                    // ない。
  int i;
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **セクション名変更機能をサポートしていなければ修正は必要ありません。**
- **セクション名を変更をしたい場合は，上記の方法に従って変更します。**

＜本Cコンパイラから他のCコンパイラ＞

- #pragma section **…を削除または** #ifdef **で切り分けます。**
- **セクション名を変更する場合は，各コンパイラの仕様により変更が必要です。**

#pragma section …　　#pragma section …

【制　　限】

- **ベクタ・テーブル用セグメントを示すセクション名（たとえば** @@VECT02 **等）を変更できません。**
- AT **開始アドレス指定の同名セクションは，（他ファイルも含めて）複数あるとリンク・エラーとなります。**
- **コンパイラ出力セクション名** @@DATS，@@BITS，@@INISを変更する場合は，指定アドレス範囲を 0FE20H-0FED7H**にしてください。**

【注　　意】

- **セクションは，アセンブラにおけるセグメントに相当します。**
- **コンパイラは，変更セクション名と他のシンボルとの重複チェックをしません。したがって，ユーザは出力アセンブル・リストをアセンブルするなどして，重複していないか確認してください。**
- #pragma section **の使用により** ROM**化関連のセクション名（＊）を変更した場合，スタートアップ・ルーチンの変更はユーザ責任となります。**

（＊）ROM**化関連セクション名**

```
@@R_INIT, @@R_INIS, @@INIT, @@INIS
```

ROM**化関連セクション変更にともなうスタートアップ・ルーチン，終端モジュールの変更例について次に示します。**

#pragma section …　　#pragma section …

【ROM化関連セクション名変更に伴うスタートアップ・ルーチン等の変更例】

ROM化関連セクション名変更に伴うスタートアップ・ルーチン（cstart.asmまたはcstartn.asm），終端モジュール（rom.asm）の変更例を示します。

（Cソース）

```
#pragma section  @@R_INIT  RTT1
#pragma section  @@INIT    TT1
```

上に示した #pragma section の記述により，初期値あり外部変数を格納するセクション名を変更した場合，ユーザは変更したセクションに格納する外部変数の初期化処理を，スタートアップ・ルーチンに追加する必要があります。

つまりスタートアップ・ルーチンには，変更したセクションの先頭レーベルの宣言と，初期値のコピーを行う部分を追加し，終端モジュールには終端レーベルの宣言を行う部分を追加します。次にその方法を示します。

RTT1_S， RTT1_Eは，セクション RTT1 の先頭と終端のレーベルの名前であり，TT1_S， TT1_Eは，セクション TT1 の先頭と終端のレーベルの名前です。

（スタートアップ・ルーチンcstartx.asm の変更例）

① 名前を変更したセクションの終端レーベルの宣言を追加します。

```
                ⋮
EXTRN     _main,_exit,_@STBEG
EXTRN     _?R_INIT,_?R_INIS,_?DATA,_?DATS

EXTRN     RTT1_E,TT1_E    ←    RTT1_E,TT1_EのEXTRN宣言を追加する
                ⋮
```

#pragma section …　　#pragma section …

② 名前を変更したRTT1セクションからTT1セクションへの初期値のコピーを行う部分を追加します。

```
                ⋮
LDATS1:
            MOVW        AX, HL
            CMPW        AX, #_?DATS
            BZ          $LDATS2
            MOV         A, #0
            MOV         [HL], A
            INCW        HL
            BR          $LDATS1
LDATS2:
            MOVW        DE,#TT1_S
            MOVW        HL,#RTT1_S
LTT1:
            MOVW        AX, HL
            CMPW        AX, #RTT1_E
            BZ          $LTT2
            MOV         A, [HL]
            MOV         [DE], A
            INCW        HL
            INCW        DE
            BR          $LTT1
LTT2:
;
            CALL        !_main          ;main();
            MOVW        AX, #0
            CALL        !_exit          ;exit(0);
            BR          $$
;
```

RTT1セクションからTT1セクションへ初期値をコピーする部分を追加

#pragma section …　　#pragma section …

③　名前を変更したセクションの先頭のレーベルを設定します。

```
      ⋮
@@R_INIT     CSEG
_@R_INIT:
@@R_INIS     CSEG        UNITP
_@R_INIS:
@@INIT       DSEG
_@INIT:
@@DATA       DSEG
_@DATA:
@@INIS       DSEG        SADDRP
_@INIS:
@@DATS       DSEG        SADDRP
_@DATS:

RTT1         CSEG                    ;セクションRTT1の先頭を示す
RTT1_S:                              ;レーベルの設定を追加
TT1          DSEG                    ;セクションTT1の先頭を示す
TT1_S:                               ;レーベルの設定を追加

@@CALT       CSEG        CALLT0
@@CNST       CSEG
@@BITS       BSEG
;
             END
```

#pragma section …　　#pragma section …

（終端モジュール rom.asm**の変更例）**

①　名前を変更したセクションの終端を示すレーベルの宣言

```
NAME        @rom
;
PUBLIC      _?R_INIT,_?R_INIS
PUBLIC      _?INIT,_?DATA,_?INIS,_?DATS

PUBLIC      RTT1_E,TT1_E      ←    RTT1_E, TT1_Eを追加


;
@@R_INIT    CSEG
_?R_INIT:
@@R_INIS    CSEG        UNITP
_?R_INIS:
@@INIT      DSEG
_?INIT:
@@DATA      DSEG
_?DATA:
@@INIS      DSEG        SADDRP
_?INIS:
@@DATS      DSEG        SADDRP
_?DATS
   ⋮
```

②　終端を示すレーベルの設定

```
   ⋮
RTT1        CSEG                    ;セクションRTT1の終端を示す
RTT1_E:                             ;レーベルの設定を追加

TT1         DSEG                    ;セクションTT1の終端を示す
TT1_E:                              ;レーベルの設定を追加


;
            END
```

（17）2進定数

2進定数	2進定数　0bxxx

【機　　能】

・整数定数が記述可能な位置に2進定数が記述できます。

【効　　果】

・ビット列で定数を記述したい場合，8進数や16進数などに置き換えずに直接記述でき，可読性も良くなります。

【方　　法】

・Cソース中で，2進定数を記述します。2進定数の記述方法は次のとおりです。

```
0b  2進数字
0B  2進数字
```

備考　2進数字：‘0’か‘1’のいずれか1つです。

・2進定数は先頭に0bまたは0Bがあり，0または1の数字の並びが後ろに続きます。

・2進定数の値は2を基数として計算されます。

・2進定数の型は次のリスト中でその値を表現できる最初のものです。

- 添字なし2進数　：　int，
  unsigned int，
  long int
  unsigned long int
- u または U の添字付き　：　unsigned int，
  unsigned long int
- l または L の添字付き　：　long int
  unsigned long int
- u または U の添字および l または L の添字付き
  ：　unsigned long int

2進定数	2進定数　0bxxx

【使 用 例】

（Cソース）

```
unsigned    i;
i = 0b11100101;
```

コンパイラの出力オブジェクトは以下の場合と同じです。

```
unsigned    i;
i = 0xE5;
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・修正の必要はありません。

＜本Cコンパイラから他のCコンパイラ＞

・2進定数をサポートしている場合コンパイラの場合は，そのコンパイラの仕様にあうように修正する必要があります。

・2進定数をサポートしていないコンパイラの場合は，8進，10進，16進などの他の整定数形式に修正する必要があります。

（18）モジュール名変更機能

モジュール名変更機能 #pragma name

【機　　能】

- オブジェクト・モジュール・ファイルのシンボル情報テーブルに，指定されたモジュール名の先頭から8文字を出力します。
- アセンブル・リスト・ファイルに-G2指定時はシンボル情報 (MOD_NAM)として，-NG指定時はNAME疑似命令として，指定されたモジュール名の先頭から8文字を出力します。
- 9文字以上のモジュール名が指定された場合は，ワーニング・メッセージを出力します。
- 許されない文字が記述された場合は，エラーとし，アボートします。
- この#pragma指令が1ソース・ファイル中に複数存在する場合は，ワーニング・メッセージを出力し，後ろに記述した方を有効とします。

【効　　果】

- オブジェクトのモジュール名を任意の名前に変更できます。

【方　　法】

- 記述方法は次のとおりです。

```
#pragma  name  モジュール名
```

モジュール名はOSでファイル名として許す文字から‘(’‘)’と漢字を除いたものとします。大文字／小文字は区別します。

【使 用 例】

```
#pragma  name  module1
          ⋮
```

【制　　限】

- 入力ファイル名の8文字目までに漢字が含まれる場合，この#pragma指令により，モジュール名を変更しないかぎりエラーとし，アボートします（UNIXベースの場合のみ）。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- モジュール名変更機能をサポートしていなければ修正の必要はありません。
- モジュール名を変更したい場合は，上記の方法に従い変更を行います。

＜本Cコンパイラから他のCコンパイラ＞

- #pragma name … を削除または #ifdefで切り分けます。
- モジュール名を変更する場合は，各コンパイラの仕様により変更が必要です。

(19) ローテート関数

ローテート関数 #pragma rot

【機　　能】

- **オブジェクトに式の値をローテートするコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。**
- **#pragmaの指令がない場合は，ローテート用の関数は通常の関数とみなされます。**

【効　　果】

- **CソースまたはASM記述によりローテートを行う処理を記述しなくてもローテート機能を実現できます。**

【方　　法】

- **関数呼び出しと同様の形式でソース中に記述します。ローテート用の関数名は，次の4つです。**

```
rorb, rolb, rorw, rolw
```

［ローテート用の関数一覧］

(a)
```
unsigned char rorb (x, y);
unsigned char x;
unsigned char y;
```

xをy回右ローテートします。

(b)
```
unsigned char rolb (x, y);
unsigned char x;
unsigned char y;
```

xをy回左ローテートします。

(c)
```
unsigned int rorw (x, y);
unsigned int  x;
unsigned char y;
```

xをy回右ローテートします。

ローテート関数　#pragma rot

(d) `unsigned int rolw(x, y);`
`unsigned int  x;`
`unsigned char y;`

xをy回左ローテートします。

注意　上記の関数宣言は-ZIオプションの影響は受けません。

・モジュールの #pragma rot指令によりローテート用の関数の使用を宣言します。
ただし，次の項目は #pragma rotの前に記述できます。

- コメント
- 他の #pragma 指令
- プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

・#pragma以降に記述するキーワードは大文字でも小文字でも可能です。

【使 用 例】

（Cソース）

```
#pragma rot
unsigned char a = 0x11;
unsigned char b = 2;
unsigned char c;
void main ( ) {
    c = rorb(a, b);
}
```

（出力アセンブラ・ソース）

```
 mov     a, !_b
 mov     c, a
 mov     a, !_a
 ror     a, 1
 dbnz    c, $$-1
 mov     !_c, a
```

ローテート関数　#pragma rot

【制　　限】

・関数名としてローテート用の関数名が使用できません。

・ローテート用の関数は小文字で記述します。大文字は通常の関数扱いとなります。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・ローテート用の関数を使用していなければ，修正は必要ありません。

・ローテート用の関数に変更したい場合は，上記の方法に従い変更を行います。

＜本Cコンパイラから他のCコンパイラ＞

・“ #pragma rot ”文を削除または #ifdef で切り分けます。

・ローテート用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（#asm，#endasm あるいは asm()；など）。

(20) 乗算関数

乗算関数 #pragma mul

【機　　能】

- オブジェクトに式の値を乗算するコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma の指令がない場合は，乗算用の関数は通常の関数とみなされます。

【効　　果】

- CC78K0と互換性があり，乗算命令の入出力のデータ・サイズを生かしたコードが生成されるため，通常の乗算式の記述よりサイズが小さいコードが生成できます。

【方　　法】

- 関数呼び出しと同様の形式でソース中に記述します。

```
mulu
```

［乗算関数一覧］

```
unsigned int mulu (x, y);
unsigned char  x;
unsigned char  y;
```

x と yを符号なし乗算します。

- モジュールの #pragma　mul指令により乗算用の関数の使用を宣言します。
  ただし，次の項目は #pragma　mul の前に記述できます。

  - コメント
  - 他の #pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma以降に記述するキーワードは大文字でも小文字でも可能です。

乗算関数 #pragma mul

【制　　限】

・インライン展開をせず，ライブラリ呼び出しとなります。

【使 用 例】

（Cソース）

```
#pragma mul
unsigned char a = 0x11;
unsigned char b = 2;
unsigned int i;
void main()
{
       i = mulu(a, b);
}
```

（コンパイラの出力オブジェクト）

```
mov     a, !_b
mov     x, a
mov     a, !_a
callt   [@@mulu]
movw    de, #_i
callt   [@@deist]
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・乗算用の関数を使用していなければ修正は必要ありません。

・乗算用の関数に変更したい場合は，前記の方法に従い変更を行います。

＜本Cコンパイラから他のCコンパイラ＞

・“ #pragma mul **”文を削除または** #ifdef **で切り分けます。関数名として乗算用の関数名を使用できます。**

・乗算用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（#asm**，**#endasm **あるいは** asm()**；など)。**

(21) 除算関数

除算関数 #pragma div

【機　　能】

- オブジェクトに式の値を除算するコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragmaの指令がない場合は，除算用の関数は通常の関数とみなされます。

【効　　果】

- CC78K0と互換性があり，除算命令の入出力のデータ・サイズを生かしたコードが生成されるため，通常の除算式の記述より，実行スピードが速く，かつサイズが小さいコードが生成できます。

【方　　法】

- 関数呼び出しと同様の形式でソース中に記述します。除算用の関数名は，次の2つです。

```
divuw, moduw
```

[除算関数一覧]

(a)
```
unsigned int  divuw(x, y);
unsigned int  x;
unsigned char  y;
```

x と yを符号なし除算し，商を返します。

(b)
```
unsigned char moduw(x, y);
unsigned int  x;
unsigned char  y;
```

x と yを符号なし除算し，余りを返します。

注意　上記の関数宣言は-ZIオプションの影響を受けません。

- モジュールの #pragma div 指令により除算用の関数の使用を宣言します。
  ただし，次の項目は #pragma div の前に記述できます。
  - コメント
  - 他の #pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの
- #pragma以降に記述するキーワードは大文字でも小文字でも可能です。

除算関数 #pragma div

【制　　限】

・インライン展開をせず，ライブラリ呼び出しとなります。

【使 用 例】

（Cソース）

```
#pragma div
unsigned int a = 0x1234;
unsigned char b = 0x12;
unsigned char c;
unsigned int i;
void main () {
       i = divuw(a, b);
       c = moduw(a, b);
}
```

（コンパイラの出力オブジェクト）

```
 mov     a,!_b
 mov     c,a
 movw    de,#_a
 callt   [@@deilo]
 callt   [@@divuw]
 movw    de,#_i
 callt   [@@deist]
 mov     a,!_b
 mov     c,a
 movw    de,#_a
 callt   [@@deilo]
 callt   [@@divuw]
 mov     a,c
 mov     !_c,a
```

除算関数 #pragma div

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **除算用の関数を使用していなければ修正は必要はありません。**
- **除算用の関数に変更したい場合は，前記の方法に従い変更を行います。**

＜本Cコンパイラから他のCコンパイラ＞

- **“** #pragma div **”文を削除または** #ifdef **で切り分けます。関数名として除算用の関数名を使用できます。**
- **除算用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（**#asm**，**#endasm **あるいは** asm()**；など）。**

（22）BCD演算関数

BCD演算関数	#pragma bcd

【機　　能】

- オブジェクトに式の値をBCD演算するコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragmaの指令がない場合，BCD演算用の関数は通常の関数と見なされます。

【効　　果】

- CソースまたはASM記述によりBCD演算を行う処理を記述しなくても，BCD演算機能を実現できます。

【方　　法】

- 関数呼び出しと同様の形式でソース中に記述します。BCD演算用の関数名は，次の13種類です。詳細は，後述のBCD演算用の関数一覧を参照してください。

```
adbcdb, sbbcdb, adbcdbe, sbbcdbe, adbcdw, sbbcdw, adbcdwe,
sbbcdwe, bcdtob, btobcde, bcdtow, wtobcd, btobcd
```

- モジュールの#pragma bcd指令により除算用の関数の使用を宣言します。ただし，次の項目は#pragma bcdの前に記述できます。

  - コメント
  - 他の#pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma以降に記述するキーワードは，大文字でも小文字でも可能です。

【制　　限】

- BCD演算用の関数名は，関数名として使用できません。
- BCD演算用の関数は小文字で記述します。大文字は通常の関数扱いとなります。
- スタティック・モデル時は，adbcdweとsbbcdweはサポートしません。

BCD演算関数　　#pragma bcd

【使 用 例】

（Cソース）

```
#pragma  bcd
unsigned char a = 0x12;
unsigned char b = 0x34;
unsigned char c;
void main()
{
    c = adbcdb(a, b);
    c = sbbcdb(b, a);
}
```

（出力アセンブラソース）

```
mov      a, !_a
add      a, !_b
adjba
mov      !_c, a
mov      a, !_b
sub      a, !_a
adjbs
mov      !_c, a
```

[BCD演算用の関数一覧]

(1) `unsigned char adbcdb(x, y);`

`unsigned char x;`

`unsigned char y;`

BCD補正命令により，10進法による加算を行います。

(2) `unsigned char sbbcdb(x, y);`

`unsigned char x;`

`unsigned char y;`

BCD補正命令により，10進法による減算を行います。

(3) `unsigned int adbcdbe(x, y);`

`unsigned char x;`

`unsigned char y;`

BCD補正命令により，10進法による加算を行います（結果拡張付き）。

BCD演算関数　#pragma bcd

(4) `unsigned int sbbcdbe(x, y);`

`unsigned char x;`

`unsigned char y;`

BCD補正命令により，10進法による減算を行います（結果拡張付き）。ボローが発生した場合は，上位桁を0x99に設定します。

(5) `unsigned int adbcdw(x, y);`

`unsigned int x;`

`unsigned int y;`

BCD補正命令により，10進法による加算を行います。

(6) `unsigned int sbbcdw(x, y);`

`unsigned int x;`

`unsigned int y;`

BCD補正命令により，10進法による減算を行います。

(7) `unsigned long adbcdwe(x, y);`

`unsigned int x;`

`unsigned int y;`

BCD補正命令により，10進法による加算を行います（結果拡張付き）。

(8) `unsigned long sbbcdwe(x, y);`

`unsigned int x;`

`unsigned int y;`

BCD補正命令により，10進法による減算を行います（結果拡張付き）。ボローが発生した場合は，上位桁を0x9999に設定します。

(9) `unsigned char bcdtob(x);`

`unsigned char x;`

10進法による値を2進法による値に変換します。

(10) `unsigned int btobcde(x);`

`unsigned char x;`

2進法による値を10進法による値に変換します。

(11) `unsigned int bcdtow(x);`

`unsigned int x;`

10進法による値を2進法による値に変換します。

BCD演算関数　#pragma bcd

(12) `unsigned int wtobcd(x);`

`unsigned int x;`

2進法による値を10進法による値に変換します。ただし，xが10000以上の値の場合は，0xffffを返します。

(13) `unsigned char btobcd(x);`

`unsigned char x;`

2進法による値を10進法による値に変換します。ただし，桁あふれは切り捨てます。

注意　上記の関数宣言は，-ZIおよび-ZLオプションの影響を受けません。

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

- BCD演算用の関数を使用していなければ，修正は必要ありません。
- BCD演算用の関数に変更したい場合は，上記の方法に従い変更を行います。

〈本Cコンパイラから他のCコンパイラ〉

- “#pragma bcd”文を削除するか#ifdefで切り分けます。BCD演算用の関数名を，関数名として使用できます。
- BCD演算用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（#asm, #endasmあるいはasm（）；など）。

（23）データ挿入関数

データ挿入関数 #pragma opc

【機　　能】

- カレント・アドレスに定数データを挿入します。
- #pragmaの指令がない場合は，データ挿入用の関数は通常の関数とみなされます。

【効　　果】

- ASM文を使わなくても，特定のデータや命令をコード領域に埋め込めます。
  ASM文を使った場合，アセンブラを通さないとオブジェクトを得られませんが，データ挿入関数を使用した場合，アセンブラを通さなくてもオブジェクトを得られます。

【方　　法】

- 関数呼び出しと同様の形式でソース中に大文字で記述します。
- データ挿入用の関数名は，_ _OPCです。

［データ挿入関数一覧］

```
void _ _OPC (unsigned char x, ...);
```

引数に記述した定数値をカレント・アドレスに挿入します。
引数は定数しか記述できません。

- #pragma opc 指令によりデータ挿入用の関数の使用を宣言します。
  ただし，次の項目は #pragma opcの前に記述できます。

  - コメント
  - 他の #pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

- #pragma 以降に記述するキーワードは大文字でも小文字でも可能です。

データ挿入関数　　#pragma opc

【制　　限】

- **関数名としてデータ挿入用の関数名が使用できません（**#pragma opc**指定時）。**
- **_ _OPCは大文字で記述します。小文字は通常の関数扱いとなります。**

【使 用 例】

（Cソース）

```
#pragma opc
void main ( ) {
        _ _OPC(0xBF);
        _ _OPC(0xA1, 0x12);
        _ _OPC(0x10, 0x34, 0x12);
}
```

（コンパイラの出力オブジェクト）

```
_main:
;line    4 : _ _OPC (0xBF) ;
    DB      0BFH
;line    5 : _ _OPC (0xA1, 0x12) ;
    DB      0A1H
    DB      012H
;line    6 : _ _OPC (0x10, 0x34, 0x12) ;
    DB      010H
    DB      034H
    DB      012H
;line    7 : }
    ret
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **データ挿入用の関数を使用していなければ修正は必要ありません。**
- **データ挿入用の関数に変更したい場合は，上記の方法に従い変更を行います。**

＜本Cコンパイラから他のCコンパイラ＞

- **“** #pragma opc **”文を削除または** #ifdef **で切り分けます。関数名としてデータ挿入用の関数名を使用できます。**
- **データ挿入用の関数として使用する場合は，各コンパイラの仕様により変更が必要です（**#asm**，**#endasm **あるいは** asm()**；など）。**

（24）スタティック・モデル

スタティック・モデル

【機　　能】

- 引数はすべてレジスタ渡しとします（11. 7. 5　**スタティック・モデルの関数呼び出しインタフェース**を参照してください）。
- レジスタで渡ってきた関数引数を，関数固有の静的領域に割り付けます。
- オートマティック変数を関数固有の静的領域に割り付けます。
- leaf関数[注]の場合，引数およびオートマティック変数は，0FEFFH以下のsaddr領域に，記述順に上位アドレスから割り付けます。このsaddr領域は全モジュールのleaf関数で共有するため，共有領域と呼びます。共有領域の最大サイズは，-SMオプション指定時にパラメータで指定できます。

```
-SM［nn］：nn = 0-16
```

nnバイトを共有領域として割り付け，残りは関数固有の静的領域に割り付けます。
nn = 0の場合および省略時は，共有領域を持ちません。

注　関数を呼び出していない関数は，コンパイラが自動判別するので，norec / _ _leafを記述する必要はありません。

- 関数引数およびオートマティック変数に，sreg / _ _sregキーワードを付加できます。
  sreg / _ _sregキーワードを付加した関数引数，およびオートマティック変数は，saddrに割り付けられ，ビット操作が可能となります。
- -RKオプションを指定することにより，関数引数およびオートマティック変数（関数内static変数を除く）をsaddrに割り付け，ビット操作が可能となります（11. 5（3）saddr**領域利用**を参照してください）。
- 次のマクロ定義をコンパイラが自動的に行います。

```
#define _ _STATIC_MODEL_ _ 1
```

【効　　果】

- 通常，スタック・フレームをアクセスする命令よりも，静的領域をアクセスする命令の方が短く高速なので，オブジェクト・コードの短縮，実行速度の向上が図れます。
- ノーマル・モデルで行っている，saddr領域を使用している引数および変数（割り込み関数でのレジスタ変数，norec関数の引数／オートマティック変数，ランタイム・ライブラリの引数）の退避／復帰処理を行わないので，割り込み処理の高速化が図れます。
- 複数のleaf関数でデータ領域を共有するので，メモリを節約できます。

【方　　法】

- コンパイル時に-SMオプションを指定します。
  この際のオブジェクトをスタティック・モデルと呼び，これに対し，-SMオプション無指定時のオブジェクトをノーマル・モデルと呼びます。

スタティック・モデル

【使 用 例】

ここでは，-SM4指定時の例を示します。

（Cソース）

```
void sub(char, char, char);
void main()
{
        char i = 1;
        char j, k;
        j = 2;
        k = i+j;
        sub(i, j, k);
}
void sub(char p1, char p2, char p3)
{
        char a1, a2;
        a1 = 1<<p1;
        a2 = p2+p3;
}
```

（コンパイラの出力オブジェクト）

```
@@DATA  DSEG
?L0003: DS    (1)                          ；関数mainの自動変数i
?L0004: DS    (1)                          ；関数mainの自動変数j
?L0005: DS    (1)                          ；関数mainの自動変数k
?L0008: DS    (1)                          ；関数subの自動変数a2

;line    1 : void sub(char,char,char);
;line    2 : void main()
;line    3 : {

@@CODE   CSEG
_main:
;line    4 : char i=1;
        mov   a,#01H                       ;1
        mov   !?L0003,a                    ;i       ；自動変数i
;line    5 : char j,k;
```

スタティック・モデル

（コンパイラの出力オブジェクト～続き～）

```
;line    6 : j=2;
      inc     a
      mov     !?L0004,a                 ;j      ;自動変数j
;line    7 :  k=i+j;
      add     a,!?L0003                 ;i      ;iとjを加算
      mov     !?L0005,a                 ;k      ;kに代入
;line    8 : sub(i,j,k);
      movw    hl,ax                             ;kをレジスタHで渡す
      mov     a,!?L0004                 ;j
      movw    bc,ax                             ;jをレジスタBで渡す
      movw    a,!?L0003                 ;i      ;iをレジスタAで渡す
      call    !sub
;line    9 : }
      ret
;line    10: void sub(char p1,char p2,char p3)
;line    11: {
_sub:
      mov     _@KREG15,a                        ;第1引数を共有領域に割り当てる
      movw    ax,bc
      mov     _@KREG14,a                        ;第2引数を共有領域に割り当てる
      movw    ax,hl
      mov     _@KREG13,a                        ;第3引数を共有領域に割り当てる
;line    12: char a1,a2;
;line    13: a1=p1;
      mov     a,_@KREG15                ;p1     ;第1引数p1
      mov     _@KREG12,a                ;a1     ;自動変数a1は共有領域
;line    14: a2 = p2+p3;
      mov     a,_@KREG14                ;p2     ;第2引数p2
      add     a,_@KREG13                ;p3     ;第3引数p3を加算
      mov     !?L0008,a                 ;a2     ;自動変数a2は関数固有の領域
;line    15: }
      ret
```

スタティック・モデル

【制　　限】

- ノーマル・モデルのモジュールとはリンクできません。ただし，スタティック・モデルのモジュール同士であれば，共有領域の最大サイズは異なっていてもリンクできます。
- 浮動小数点数はサポートしません。floatおよびdoubleのキーワードが記述された場合は，フェータル・エラーとします。
- 引数は最大3引数，合計6バイトまでとします。
- 引数がスタック渡しでないため，可変長引数は使用できません。可変長引数はエラーとなります。
- 構造体／共用体の引数および返り値を使用できません。これらの記述はエラーとなります。
- noauto / norec / _ _leaf関数は使用できません。これらの記述に対しワーニング・メッセージを出力し無視します（11. 5（5）noauto**関数**，11. 5（6）norec**関数**を参照してください）。
- 再帰関数は使用できません。関数引数，オートマティック変数領域を静的に確保するため，再帰関数は使用できません。コンパイラが検出可能な再帰関数に対してはエラーとします。
- プロトタイプ宣言を省略できません。関数呼び出しがあるにもかかわらず，その関数の実体定義もプロトタイプ宣言もない場合は，エラーとします。
- 引数および返り値の制限，再帰関数である関数が使用できないため，一部の標準ライブラリを使用できません。
- -ZLオプションが指定されていない場合は，ワーニングを出力して，-ZLオプションが指定されたものとして処理します。したがって，常にlong型をint型とみなします（11. 5（25）**型変更**を参照してください）。

【互 換 性】

〈他のCのコンパイラから本Cのコンパイラ〉

- ノーマル・モデルのオブジェクトを作成する場合は，-SMオプションを指定しなければソースの修正は必要ありません。
- スタティック・モデルのオブジェクトを作成する場合は，上記の方法に従い変更します。

〈本Cコンパイラから他のCコンパイラ〉

- 他のコンパイラでそのまま再コンパイルすれば，ソースの修正は必要ありません。

【注　　意】

- 引数／オートマティック変数を静的に確保しているので，再帰関数は引数／オートマティック変数の内容が破壊される可能性があります。直接自分自身を呼び出す場合はエラーとしますが，他の関数を呼び出した先で自分自身が呼び出された場合，コンパイラはそれを検出できずエラーとなりません。
- 割り込み時に，処理中の関数が割り込み処理（割り込み関数，および割り込み関数が呼び出す関数）により呼び出された場合，引数／オートマティック変数の内容が破壊される可能性があります。
- 割り込み時に，処理中の関数が共有領域を使用している場合でも，共有領域の退避／復帰は行われません。

(25) 型 変 更

型 変 更　　-ZI

(1) int, short型のchar型への変更

【機　　能】

- int型／short型をchar型とみなします。つまり，charと記述したのとまったく同等となります。
- 型変更の詳細を次に示します（一部-QUオプションに影響を受けます）。

表11－14　型変更の詳細（int, short型のchar型への変更）

Cソース上で記述された型	オプション	変更後の型
short, short int, int	-QUあり	unsigned char
short, short int, int	-QUなし	signed char
unsigned short, unsigned short int unsigned, unsigned int	—	unsigned char
signed short, signed short int signed, signed int	—	signed char

- Cソース上で，最初にintまたはshortキーワードが出現した行に対し，ワーニング・メッセージを出力します。
- -QCオプションは指定の有無にかかわらず有効とします。-QCオプションの指定がない場合ワーニング・メッセージを出力し，-QCオプションを有効とします。
- -ZAオプションと同時に指定（-ZAIなど）した場合，ワーニング・メッセージを出力します（-W2指定時のみ）。
- 次に示す，型指定子が記述可能な構文で省略できるものは，char型とみなします。
  - 関数の引数および返却値
  - 型指定子省略の変数／関数宣言
- 次のマクロ定義をコンパイラが自動的に行います。

```
#define _ _FROM_INT_TO_CHAR_ _ 1
```

- 一部の標準ライブラリが使用できなくなります。

【方　　法】

- -ZIオプションを指定します。

【制　　限】

- -ZIを指定したモジュールと指定しないモジュールは，リンクできません。

型 変 更 -ZL

（2）long型のint型への変更

【機　　能】

- long型をint型とみなします。つまり，intと記述したのとまったく同等となります。
- 型変更の詳細を次に示します。

表11－15　型変更の詳細（long型のint型への変更）

Cソース上で記述された型	変更後の型
unsigned long, unsigned long int	unsigned int
long, long int, signed long, signed long int	signed int

- Cソース上で，最初にlongキーワードが出現した行に対し，ワーニング・メッセージを出力します。
- -ZAオプションと同時に指定（-ZALなど）した場合，ワーニング・メッセージを出力します（-W2指定時のみ）。
- 次のマクロ定義をコンパイラが自動的に行います。

```
#define _ _FROM_LONG_TO_INT_ _ 1
```

- 一部の標準ライブラリが使用できなくなります。

【方　　法】

- -ZLオプションを指定します。

【制　　限】

- -ZLを指定したモジュールと指定しないモジュールは，リンクできません。

(26) パスカル関数

パスカル関数 _ _pascal

【機　　能】

・関数呼び出し時に引数の積み込みによって使用したスタックの修正を，関数呼び出し側では行わずに，呼ばれた関数側で行うコードを生成します。

【効　　果】

・関数呼び出し箇所が多い場合に，オブジェクト・コードの短縮が図れます。

【方　　法】

・関数の宣言時に，_ _pascal属性を先頭に追加します。

【制　　限】

・パスカル関数は，可変長引数をサポートしません。可変長引数を定義した場合は，ワーニングを出力して_ _pascalキーワードを無視します。
・パスカル関数は，norec/_ _interruptキーワードを指定できません。指定した場合，norecキーワードの場合は，_ _pascalキーワードを無視し，_ _interrupt/_ _interrupt_brk/_ _rtos_interruptキーワードの場合は，エラーを出力します。
・プロトタイプ宣言が不完全な場合，正常作動しないことがあるため，パスカル関数の実体定義や，プロトタイプ宣言がないものに対しワーニング・メッセージを出力します。
・スタティック・モデル指定オプション（-SM）指定時は，パスカル関数をサポートしません。パスカル関数使用時に-SMを指定した場合は，_ _pascalキーワードが最初に出現した箇所に対し，ワーニング・メッセージを出力して，入力ファイル中の_ _pascalキーワードを無視します。

【説　　明】

・-ZRオプションにより，すべての関数をパスカル関数にできますが，呼び出し箇所が少ない関数に使用する場合，オブジェクト・コードが増加することがあります。

【使 用 例】

（Cソース）

```
_ _pascal int func(int a, int b, int c);
void main()
{
        int ret_val;

        ret_val = func(5, 10, 15);
```

パスカル関数　　_ _pascal

（Cソース ~続き~）

```
}
_ _pascal int func(int a, int b, int c)
{
        return (a + b + c);
}
```

（コンパイラの出力オブジェクト）

```
_main:
        push    hl
        movw    ax,#02H
        callt   [_@cprep]
        movw    ax,#0FH             ;15
        push    ax
        mov     x,#0AH              ;10
        push    ax
        mov     x,#05H              ;5
        call    !_func
        movw    ax,bc               ;ここでスタックの修正をしない
        mov     [hl+1],a            ;ret_val
        xch     a,x
        mov     [hl],a              ;ret_val
        pop     ax
        pop     hl
        ret

_func:
        push    hl
        push    ax
        movw    ax,sp
        movw    hl,ax
        mov     a,[hl]              ;a
        mov     a,[hl+6]            ;b
        xch     a,x
        mov     a,[hl+1]            ;a
        addc    a,[hl+7]            ;b
        xch     a,x
        add     a,[hl+8]            ;c
```

パスカル関数　　_ _pascal

（コンパイラの出力オブジェクト～続き～）

```
        xch     a,x
        addc    a,[hl+9]            ;c
        movw    bc,ax
        pop     ax
        pop     hl
        pop     de                  ;リターン・アドレスを取得
        pop     ax                  ;
        pop     ax                  ;呼び出し側で消費した4バイトのスタックを修正
        push    de                  ;リターン・アドレスの積み直し
```

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

- 予約語_ _pascalを使用していなければ，修正は必要ありません。
- パスカル関数に変更したい場合は，上記の方法に従って変更します。

〈本Cコンパイラから他のCコンパイラ〉

- #defineにより可能です。
- この変更により，パスカル関数は通常の関数として扱われます。

（27）関数呼び出しインタフェースの自動パスカル関数化

関数呼び出しインタフェースの自動パスカル関数化 -ZR

【機　　能】

・norec/_ _interrupt/可変長引数の関数を除くすべての関数に対して_ _pascal属性を付加します。

【方　　法】

・コンパイル時に-ZRオプションを指定します。

【制　　限】

・-ZRオプションを指定したモジュールと指定しないモジュールはリンクできません。リンクを行った場合，リンク・エラーとなります。
・スタティック・モデル指定オプション（-SM）は，同時に指定できません。
　指定した場合，ワーニング・メッセージを出力して-ZRオプションを無視します。
・数学関数標準ライブラリは，パスカル関数に未対応のため，数学関数標準ライブラリ使用時は，-ZRオプションを使用できません。

備考　パスカル関数呼び出しインタフェースに関しては，11. 7. 6　**パスカル関数呼び出しインタフェース**を参照してください。

（28）引数／返り値のint拡張抑制方法

引数／返り値のint拡張抑制方法 -ZB

【機　　能】

- 関数返り値の型定義がchar/unsigned char型の場合に，返り値のint拡張コードを生成しません。
- 関数引数のプロトタイプが定義されていて，かつそのプロトタイプの引数定義がchar/unsigned char型の場合に，引数のint拡張コードを生成しません。

【効　　果】

- int拡張コードが生成されないため，オブジェクト・コードの短縮，実行速度の向上が図れます。

【方　　法】

- コンパイル時に-ZBオプションを指定します。

【使 用 例】

（Cソース）

```
unsigned char func1(unsigned char x, unsigned char y);
unsigned char c, d, e;
void main()
{
        c = func1(d, e);
        c = func2(d, e);
}
unsigned char func1(unsigned char x, unsigned char y)
{
        return x + y;
}
```

（コンパイラの出力オブジェクト）

-ZB指定あり

```
_main:
;line   5:  c=func1(d,e);
     mov    a,!_e
     xch    a,x                 ;int拡張しない
     push   ax
     mov    a,!_d
     xch    a,x                 ;int拡張しない
     call   !_func1
     pop    ax
```

引数／返り値のint拡張抑制方法　-ZB

（コンパイラの出力オブジェクト～続き～）

```
     mov    a, c
     mov    !_c,a
;line   6:  c=func2(d,e);
     mov    a,!_e
     xch    a,x
     xor    a,a                 ；プロトタイプ宣言がないのでint拡張する
     push   ax
     mov    a,!_d
     xch    a,x
     xor    a,a                 ；プロトタイプ宣言がないのでint拡張する
     call   !_func2
     pop    ax
     mov    a,c
     mov    !_c,a
;line   7:  }
     ret
;line   8:
;line   9:  unsigned char func1(unsinged char x,unsinged char y){
_func1:
     push   hl
     push   ax
     movw   ax,sp
     movw   hl,ax
;line  10:  return x+y;
     mov    a,[hl];x
     add    a,[hl+6];y
     mov    c,a
;line  11:  }
     pop    ax
     pop    hl
     ret
     END
```

引数／返り値のint拡張抑制方法　-ZB

【制　　限】

・関数本体の定義とその関数に対するプロトタイプ宣言がファイル間で異なる場合，不正動作となる場合があります。

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

・すべての関数本体の定義に対するプロトタイプ宣言が正しく行われていない場合は，プロトタイプ宣言を正しく行います。あるいは，-ZBオプションを指定しません。

〈本Cコンパイラから他のCコンパイラ〉

・修正は必要ありません。

（29）配列オフセット計算簡略化方法

配列オフセット計算簡略化方法	-QW2
	-QW3
	-QW4
	-QW5

【機　　能】

- char / unsigned char / int / unsigned int / short / unsigned short**型配列のオフセット（配列の先頭からの距離）を計算する際に，インデクスが**unsigned char**型変数の場合に，桁上がりが生じないと仮定して，下位バイトのみ計算するコードを生成します。**
- -QW2**オプション指定時は，**saddr**領域配置の配列を**unsigned char**変数で参照する場合のみ，オフセットを下位バイトのみ計算するコードをスピード優先で生成します。**
- -QW3**オプション指定時は，配置領域にかかわらず配列を**unsigned char**変数で参照する場合に，オフセットを下位バイトのみ計算するコードをスピード優先で生成します。**
- -QW4**オプション指定時は，**saddr**領域配置の配列を**unsigned char**変数で参照する場合のみ，オフセットを下位バイトのみ計算するコードをコード・サイズ優先で作成します。**
- -QW5**オプション指定時は，配列領域にかかわらず配列を**unsigned char**変数で参照する場合に，オフセットを下位バイトのみ計算するコードをコード・サイズ優先で作成します。**

【効　　果】

- **オフセット計算コードが簡略化され，オブジェクト・コードの短縮，実行速度の向上が図れます。**

【方　　法】

- **コンパイル時に**-QW2, -QW3, -QW4, -QW5**オプションを指定します。**

【使 用 例】

（C**ソース**）

```
unsigned char c;
unsigned char ary[10];
sreg unsigned char sary[10];
void main()
{
        unsigned char a;

        a = ary[c];
        a = sary[c];
}
```

配列オフセット計算簡略化方法

-QW2
-QW3
-QW4
-QW5

（コンパイラの出力オブジェクト）

-QW3指定あり

```
_main:
        push    hl
        push    ax
        movw    ax, sp
        movw    hl, ax
;line     6:  unsigned char a;
;line     7:
;line     8:  a = ary[c];
        mov     a, !_c
        add     a, #low(_ary)
        mov     e, a                    ;下位バイトのみ計算
        mov     d, #high(_ary)
        mov     a,[de]
        mov     [hl+1], a               ;a
;line     9:  a = sary[c];
        mov     a, !_c
        add     a, #low(_sary)
        mov     e, a                    ;下位バイトのみ計算
        mov     d, #0FEH ; 254
        mov     a,[de]
        mov     [hl+1], a               ;a
;line     10: }
        pop     ax
        pop     hl
        ret
```

配列オフセット計算簡略化方法	-QW2 -QW3 -QW4 -QW5

【制　　限】

・オフセット計算簡略化対象となった配列の配置アドレスが256バイト境界をまたがる場合は，不正動作となる場合があります。

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

・配列を256バイト境界にまたがらないように配置します。あるいは，-QW2, -QW3, -QW4, -QW5オプションを指定しません。

〈本Cコンパイラから他のCコンパイラ〉

・修正は必要ありません。

（30）レジスタ直接参照関数

レジスタ直接参照関数　　#pragma realregister

【機　　能】

- オブジェクトにレジスタをアクセスするコードを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma指令がない場合は，レジスタ直接参照関数は通常の関数とみなされます。

【効　　果】

- C記述により，レジスタのアクセスを簡単に行えます。

【方　　法】

- 関数呼び出しと同様の形式でソース中に記述します（後述の**レジスタ直接参照用の関数一覧**を参照してください）。

  レジスタ直接参照関数名は，次の21個です。

```
_ _geta, _ _seta, _ _getax, _ _setax, _ _getcy, _ _setcy, _ _set1cy, _ _clr1cy
_ _not1cy, _ _inca, _ _deca, _ _rora, _ _rorca, _ _rola, _ _rolca, _ _shla
_ _shra, _ _ashra, _ _nega, _ _coma, _ _absa
```

- モジュールの#pragma realregister指令によりレジスタ直接参照関数の使用を宣言します。

  ただし，次の項目は#pragma realregisterの前に記述できます。

  - コメント
  - 他の#pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

【使 用 例】

（Cソース）

```
#pragma realregister
unsigned char c = 0x88,  d, e;
void main()
{
     _ _seta（c）;              /*  Aレジスタに変数cの値をセット   */
     _ _shla();                 /*  1ビット論理左シフト            */
     d = _ _geta();             /*  変数dにAレジスタの値をセット   */
     if(_ _getcy()){            /*  CYを参照（桁あふれを見る）     */
         e = 1;                 /*  CY == 1ならeに1をセット        */
     }
}
```

レジスタ直接参照関数　　#pragma realregister

（コンパイラの出力オブジェクト）

```
_main:
;line     5:  _ _seta(c);          /*  Aレジスタに変数cの値をセット  */
       mov    a, !_c
;line     6:  _ _shla();           /*  1ビット論理左シフト           */
       add    a, a
;line     7:  d = _ _geta();       /*  変数dにAレジスタの値をセット  */
       mov    !_d, a
;line     8:  if(_ _getcy()){      /*  CYを参照（桁あふれを見る）    */
       bnc    $?L0003
;line     9:           e = 1;     /*  CY == 1ならeに1をセット       */
       mov    a, #01H     ;1
       mov    !_e, a
?L0003:
;line     10: }
;line     11: }
       ret
```

[レジスタ直接参照用の関数一覧]

(1) `unsigned char  _ _geta(void);`

Aレジスタの値を取得します。

(2) `void  _ _seta(unsigned char x);`

xをAレジスタに設定します。

(3) `unsigned int  _ _getax(void);`

AXレジスタの値を取得します。

(4) `void  _ _setax(unsigned int x);`

xをAXレジスタに設定します。

(5) `bit  _ _getcy(void);`

CYフラグの値を取得します。

(6) `void  _ _setcy(unsigned char x);`

xの下位1ビットをCYフラグに設定します。

レジスタ直接参照関数　#pragma realregister

(7) `void _ _set1cy(void);`

set1　CY命令を生成します。

(8) `void _ _clr1cy(void);`

clr1　CY命令を生成します。

(9) `void _ _not1cy(void);`

not1　CY命令を生成します。

(10) `void _ _inca(void);`

inc　a命令を生成します。

(11) `void _ _deca(void);`

dec　a命令を生成します。

(12) `void _ _rora(void);`

ror　a, 1命令を生成します。

(13) `void _ _rorca(void);`

rorc　a, 1命令を生成します。

(14) `void _ _rola(void);`

rol　a, 1命令を生成します。

(15) `void _ _rolca(void);`

rolc　a, 1命令を生成します。

(16) `void _ _shla(void);`

Aレジスタを1ビット論理左シフトするコードを生成します。

(17) `void _ _shra(void);`

Aレジスタを1ビット論理右シフトするコードを生成します。

(18) `void _ _ashra(void);`

Aレジスタを1ビット算術右シフトするコードを生成します。

(19) `void _ _nega(void);`

Aレジスタの2の補数を得るコードを生成します。

レジスタ直接参照関数　#pragma realregister

(20) `void _ _coma(void);`

Aレジスタの1の補数を得るコードを生成します。

(21) `void _ _absa(void);`

Aレジスタの絶対値を得るコードを生成します。

【制　限】

- レジスタ直接参照用の関数名は，関数名として使用できません。レジスタ直接参照用の関数は小文字で記述します。大文字は通常の関数扱いとなります。
- _ _seta, _ _setax, _ _setcy関数で設定したA, AXレジスタおよびCYフラグの値は，以後のコード生成において保持されません。
- _ _geta, _ _getax, _ _getcy関数でA, AXレジスタおよびCYフラグが参照されるタイミングは，式の評価順によります。

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

- レジスタ直接参照用の関数を使用していなければ，修正は必要ありません。
- レジスタ直接参照用の関数に変更したい場合は，上記の方法に従い変更します。

〈本Cコンパイラから他のCコンパイラ〉

- “#pragma realregister”指令を削除するか，#ifdefで切り分けます。レジスタ直接参照用の関数名を，関数名として使用できます。
- レジスタ直接参照用の関数として使用する場合は，各コンパイラの使用により変更が必要です（#asm, #endasmあるいはasm（）；など）。

【注　意】

- レジスタ直接参照関数を実行するまでに，CY, A, AXが意図通りに保存されている保証はありません。したがって，この関数は式の第1項に書くなど，値が変化する前に使用されることをお勧めします。

（31）メモリ操作関数

メモリ操作関数 #pragma inline

【機　　能】

- メモリ操作用標準ライブラリ関数memcpy, memsetを関数呼び出しではなく，直接インライン展開して出力し，オブジェクト・ファイルを生成します。
- #pragma指令がない場合は，標準ライブラリ関数を呼び出すコードを生成します。

【効　　果】

- 標準ライブラリ関数呼び出し時と比べて，実行速度の向上が図れます。
- 指定文字数に定数を指定した場合は，オブジェクト・コードの短縮も図れます。

【方　　法】

- 関数呼び出しと同様の形式で，ソース中に記述します。
- 次の項目は，#pragma inlineの前に記述できます。
  - コメント
  - 他の#pragma指令
  - プリプロセス指令のうち変数の定義／参照，関数の定義／参照を生成しないもの

【使 用 例】

（Cソース）

```
#pragma inline
char ary1[100], ary2[100];
void main()
{
     memset(ary1, 'A', 50);
     memcpy(ary1, ary2, 50);
}
```

メモリ操作関数　　#pragma inline

（コンパイラの出力オブジェクト）

-SM指定なし

```
_main:
     push   h1
;line   5:  memset(ary1, 'A', 50);
     movw   de,#_ary1
     mov    a, #041H  ; 65
     mov    c, #032H  ; 50
     mov    [de], a
     incw   de
     dbnz   c, $$-2
;line   6:  memcpy(ary1, ary2, 50);
     movw   de, #_ary1
     movw   h1, #_ary2
     mov    c, #032H  ; 50
     mov    a, [h1]
     mov    [de], a
     incw   de
     incw   h1
     dbnz   c, $$-4
;line   7:  }
     pop    h1
     ret
```

-SM指定あり

```
_main:
     push   de
;line   5:  memset(ary1, 'A', 50);
     movw   h1,#_ary1
     mov    a, #041H  ; 65
     mov    c, #032H  ; 50
     mov    [h1], a
     incw   h1
     dbnz   c, $$-2
;line   6:  memcpy(ary1, ary2, 50);
     movw   h1, #_ary1
     movw   de, #_ary2
     mov    c, #032H  ; 50
     mov    a, [de]
```

メモリ操作関数　　#pragma inline

（-SM指定あり～続き～）

```
      mov    [h1], a
      incw   de
      incw   h1
      dbnz   c, $$-4
;line   7:  }
      pop    de
      ret
```

【互 換 性】

〈他のCコンパイラから本Cコンパイラ〉

- **メモリ操作用の関数を使用していなければ，修正は必要ありません。**
- **メモリ操作用の関数に変更したい場合は，上記の方法に従い変更します。**

〈本Cコンパイラから他のCコンパイラ〉

- **“** #pragma inline **” 指令を削除，または**#ifdef**で切り分けます。**

(32) 絶対番地配置指定

絶対番地配置指定 __directmap

【機　　能】

- __directmap宣言された外部変数および関数内static変数の初期値を，配置アドレス指定とみなして，指定アドレスに変数を配置します。
- Cソース中における__directmap変数は，通常の変数と同様に扱います。
- 初期値を配置アドレス指定とみなすため，初期値を定義できず，初期値は不定となります。
- 指定可能なアドレス指定範囲，指定アドレスに対する領域確保用モジュールがリンクされる領域確保範囲，および変数の重複チェック範囲は，次のとおりです。

アドレス指定範囲	領域確保範囲	重複チェック範囲
0x80-0xffff	0xfd00-0xfeff	0xf000-0xfeff

- アドレス指定がアドレス指定範囲外の場合は，F799エラーを出力します。
- __directmap宣言された変数の配置アドレスが重複し，重複チェック範囲内であれば，W762ワーニング・メッセージを出力して，重なった変数名を表示します。
- アドレス指定範囲がsaddr領域内の場合は，__sreg宣言を自動的に付与し，saddr命令を生成します。
- __directmap宣言されたchar/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long型変数に対してビット参照を行う場合は，sreg/__sregを併用する必要があります。併用しない場合は，エラーとします。

【効　　果】

任意のアドレスに変数を配置でき，同じアドレスに複数の変数を重ねて配置できます。

【方　　法】

- 絶対番地に配置する変数を定義したいモジュール中で__directmap宣言を行います。

```
__directmap 型名      変数名              = 配置アドレス指定；
__directmap static  型名    変数名        = 配置アドレス指定；
__directmap __sreg 型名     変数名        = 配置アドレス指定；
__directmap __sreg static 型名  変数名 = 配置アドレス指定；
```

- 構造体／共用体／配列に対して，__directmap宣言を行う場合は，{} で囲んでアドレス指定を行います。
- __directmap外部変数を参照するモジュール中では__directmapの宣言は不要で，extern宣言のみ行います。

```
extern  型名        変数名；
extern  __sreg   型名       変数名；
```

- saddr領域内に配置した__directmap外部変数を参照するモジュール中でsaddr命令を生成するには，__sregを併用してextern __sreg 型名 変数名；とする必要があります。

絶対番地配置指定　　_ _directmap

【使 用 例】

（Cソース）

```
_ _directmap char c = 0xfe00;
_ _directmap _ _sreg char d = 0xfe20;
_ _directmap _ _sreg char e = 0xfe21;
_ _directmap struct x {
        char a;
        char b;
} xx = {0xfe30};
void main()
{
        c = 1;
        d = 0x12;
        e.5 = 1;
        xx.a = 5;
        xx.b = 10;
}
```

（出力オブジェクト）

```
        PUBLIC _c
        PUBLIC _d
        PUBLIC _e
        PUBLIC _xx
        PUBLIC _main
_c      EQU    0FE00H          ;_ _directmap宣言された変数は
_d      EQU    0FE20H          ;EQUでアドレスを定義
_e      EQU    0FE21H          ;
_xx     EQU    0FE30H          ;
        EXTRN  _ _mmfe00       ;領域確保モジュール・リンク用
        EXTRN  _ _mmfe20       ;EXTRN出力
        EXTRN  _ _mmfe21       ;
        EXTRN  _ _mmfe30       ;
        EXTRN  _ _mmfe31       ;
```

絶対番地配置指定　　__directmap

（出力オブジェクト～続き～）

```
@@CODE CSEG
_main:
;line     10: c = 1;
       mov    a,#01H ;1
       mov    !_c,a
;line     11: d = 0x12;
       mov    _d,#012H          ;アドレス指定がsaddr領域内のため，
                                ;saddr命令を出力
;line     12: e.5 = 1;
       set1   _e.5              ;__sregと併用しているため，ビット操作可能
;line     13: xx.a = 5;
       mov    _xx,#05H          ;アドレス指定がsaddr領域内のため，
                                ;saddr命令を出力
;line     14: xx.b = 10;
       mov    _xx+1,#0AH        ;アドレス指定がsaddr領域内のため，
                                ;saddr命令を出力
;line     15: }
       ret
```

絶対番地配置指定　　_ _directmap

【制　　限】

- **関数引数，返り値，およびオートマティック変数には指定できません。指定した場合は，エラーとなります。**
- short/unsigned short/int/unsigned int/long/unsigned long**型変数を奇数番地に配置した場合，**_ _directmap**宣言を行ったファイル内では正常なコードが生成されますが，別ファイルから**extern**宣言で参照した場合，不正コードとなります。**
- **領域確保範囲外のアドレス指定を行った場合，変数領域は確保されないので，ディレクティブ・ファイルを記述するか，領域確保用モジュールを別途作成する必要があります。**

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- **キーワード**_ _directmap**を使用していなければ，修正の必要はありません。**
- _ _directmap**変数に変更したい場合は，前記の方法に従い変更を行います。**

＜本Cコンパイラから他のＣコンパイラ＞

- #define**により可能です（**11. 6　**Ｃソースの修正を参照してください）。**
- **絶対番地配置指定として使用する場合は，各コンパイラの仕様により変更が必要です。**

（33）スタティック・モデル拡張仕様

スタティック・モデル拡張仕様 -ZM

【機　　能】

- _@NRAT00～_@NRAT07の8バイトのsaddr領域を，引数用，ワーク用として，コンパイラの予約領域として確保します。
- 引数とオートマティック変数に対して，_ _temp宣言を行うことにより，テンポラリ変数を使用可能とします（11. 5（34） テンポラリ変数を参照してください）。
- 引数の宣言数を3個から，intサイズで6個，charサイズで9個まで記述可能とします。第4引数以降は，呼び出し側で_@NRAT00～_@NRAT05の領域に引数を設定し，呼ばれた側で別領域にコピーします。ただし，呼ばれた側がleaf関数，または引数に対して_ _temp宣言が行われている場合は，コピーは行わず，引数を設定した_@NRATxxの領域をそのまま使用します。
- 引数に２バイト・サイズ以下の構造体／共用体を記述可能とします。
- 関数返り値に，構造体／共用体を記述可能とします。サイズが2バイト以下の場合は，値を返します。サイズが3バイト以上の場合は，返却値格納用静的領域を確保してこの領域に返り値を格納し，返却値格納用静的領域の先頭アドレスを返します。
- leaf関数の共有領域として，_@NRAT00～_@NRAT07の8バイトの領域も使用します。共有領域の割り当ては，-SM指定で確保した_@KREGxx領域より先に，_@NRAT00～_@NRAT07の8バイトの領域に割り当てます。
- 配列／共用体／構造体に対しても，_@NRATxx, -SM指定で確保した_@KREGxx領域に収まるサイズであれば，_@NRATxx, _@KREGxxに割り当てます。
- 割り込み関数の退避対象は，表11－16のとおりです。

表11－16　割り込み関数の退避対象

復帰／退避領域	NO BANK	関数コールあり		関数コールなし	
		-ZM1	-ZM2	-ZM1	-ZM2
使用レジスタ	×	×	×	○	○
全レジスタ	×	○	○	×	×
全_@NRATxx領域	×	○	○	×	×
全_@KREGxx領域	×	○	×	×	×
使用_@KREGxx領域	×	×	○	×	○

○：退避する

×：退避しない

ただし，#pragma interrupt指定時に次のように指定することにより，退避対象を限定できます。

SAVE_R（退避／復帰対象をレジスタに限定）

SAVE_RN（退避／復帰対象をレジスタ，_@NRATxxに限定）

スタティック・モデル拡張仕様　-ZM

- -ZM1オプションと-ZM2オプションの相違点は，-SM指定で確保した_@KREGxx領域の取り扱いのみです。
  -ZM1オプション指定時は，leaf関数の共有領域のみ，_@KREGxxを使用します。-ZM2オプション指定時は，_@KREGxx領域の退避／復帰を行い，_@KREGxx領域に引数，オートマティック変数を割り当てます（ノーマル・モデルの-QRオプション互換）。
- -SMオプション未指定時に，-ZMオプションが指定された場合は，W055ワーニング・メッセージを出力し，-ZMオプション指定を無視します。

【効　果】

既存スタティック・モデルの制限事項を緩和できるため，記述性が向上します。

【方　法】

コンパイル時に，-ZMオプションを指定します。

スタティック・モデル拡張仕様　-ZM

【使用例1】

（Cソース）

```
char funcl(char a, char b, char c, char d, char e);
char func2(char a, char b, char c, char d);
void main()
{
        char a = 1, b = 2, c = 3, d = 4, e = 5, r;
        r = func1(a, b, c, d, e);
}
char func1(char a, char b, char c, char d, char e)
{
        char r;

        r = func2(a, b, c, d);
        return e + r;
}
char func2(char a, char b, char c, char d)
{
        return a + b + c + d;
}
```

スタティック・モデル拡張仕様　-ZM

（出力オブジェクト）

-SM8 -ZM1 -QC**指定あり**

```
_main:
;line     5:  char a = 1, b = 2, c = 3, d = 4, e = 5, r;
       mov    a,#01H  ; 1
       mov    !L0003,a         ;a
       inc    a
       mov    !L0004,a         ;b
       inc    a
       mov    !L0005,a         ;c
       inc    a
       mov    !L0006,a         ;d
       inc    a
       mov    !L0007,a         ;e
;line     6:
;line     7:  r = func1(a, b, c, d, e);
       mov    _@NRAT01,a       ; 第5引数を引数受け渡し用saddr領域に設定
       mov    a,!L0006         ;d
       mov    _@NRAT00,a       ; 第4引数を引数受け渡し用saddr領域に設定
       mov    a,!L0005         ;c
       movw   hl,ax
       mov    a,!L0004         ;b
       movw   bc,ax
       mov    a,!L0003         ;a
       call   !_func1
       mov    !L0008,a         ;r
;line     8:  }
       ret
;line     9:  char func1(char a, char b, char c, char d, char e)
;line     10: {
_func1:
       mov    !L0011,a
       movw   ax,bc
       mov    !L0012,a
       movw   ax,hl
       mov    !L0013,a
       mov    a,_@NRAT00       ;静的領域にコピー
       mov    !L0014,a         ;
```

スタティック・モデル拡張仕様　-ZM

（出力オブジェクト～続き～）

```
        mov    a,_@NRAT01       ;静的領域にコピー
        mov    !L0015,a         ;
;line     11: char r;
;line     12:
;line     13: r = func2(a, b, c, d);
        mov    a,!L0014         ;d
        mov    _@NRAT00,a       ;第4引数を引数受け渡し用saddr領域に設定
        mov    a,!L0013         ;c
        movw   hl,ax
        mov    a,!L0012         ;b
        movw   bc,ax
        mov    a,!L0011         ;a
        call   !_func2
        mov    !L0016,a         ;r
;line     14: return e + r;
        add    a,!L0015         ;e
;line     15: }
        ret
;line     16: char func2(char a, char b, char c, char d)
;line     17: {
_func2:
        mov    _@NRAT01,a
        movw   ax,bc
        mov    _@NRAT02,a
        movw   ax,hl
        mov    _@NRAT03,a
;line     18: return a + b + c + d;
        mov    a,_@NRAT01       ;a
        add    a,_@NRAT02       ;b
        add    a,_@NRAT03       ;c
        add    a,_@NRAT00       ;d  leaf関数時は _@NRAT00をそのまま使用
;line     19: }
        ret
```

スタティック・モデル拡張仕様　-ZM

-SM8 -ZM2 -QC指定あり

```
@@CODE CSEG
_main:
       movw   ax,_@KREG10      ;
       push   ax               ;_@KREG10～_@KREG15領域の退避
       movw   ax,_@KREG12      ;
       push   ax               ;
       movw   ax,_@KREG14      ;
       push   ax               ;
;line     5:  char a = 1, b = 2, c = 3, d = 4, e = 5, r;
       mov    _@KREG15,#01H    ;a, 1 _@KREG11～_@KREG15に変数を配置
       mov    _@KREG14,#02H    ;b, 2
       mov    _@KREG13,#03H    ;c, 3
       mov    _@KREG12,#04H    ;d, 4
       mov    _@KREG11,#05H    ;e, 5
;line     6:
;line     7:  r = func1(a, b, c, d, e);
       mov    a,_@KREG11       ;e
       mov    _@NRAT01,a       ;第5引数を引数受け渡し用saddr領域に設定
       mov    a,_@KREG12       ;d
       mov    _@NRAT00,a       ;第4引数を引数受け渡し用saddr領域に設定
       mov    a,_@KREG13       ;c
       movw   hl,ax
       mov    a,_@KREG14       ;b
       movw   bc,ax
       mov    a,_@KREG15       ;a
       call   !_func1
       mov    _@KREG10,a       ;r
;line     8:  }
       pop    ax               ;
       movw   _@KREG14,ax      ;_@KREG10～_@KREG15領域の復帰
       pop    ax               ;
       movw   _@KREG12,ax      ;
       pop    ax               ;
       movw   _@KREG10,ax      ;
       ret
;line     9:  char func1(char a, char b, char c, char d, char e)
;line    10:  {
```

スタティック・モデル拡張仕様 -ZM

（出力オブジェクト～続き～）

```
_func1:
        mov     _@NRAT06,a      ;aレジスタの退避
        movw    ax,_@KREG10     ;
        push    ax              ;_@KREG10～_@KREG15領域の退避
        movw    ax,_@KREG12     ;
        push    ax              ;
        movw    ax,_@KREG14     ;
        push    ax              ;
        mov     a,_@NRAT06      ;aレジスタの復帰
        mov     _@KREG15,a
        movw    ax,bc
        mov     _@KREG14,a
        movw    ax,hl
        mov     _@KREG13,a
        mov     a,_@NRAT00      ;_@KREG12にコピー
        mov     _@KREG12,a      ;
        mov     a,_@NRAT01      ;_@KREG11にコピー
        mov     _@KREG11,a      ;
;line 11:      char r;
;line 12:
;line 13:      r = func2(a, b, c, d);
        mov     a,_@KREG12      ;d
        mov     _@NRAT00,a      ;第4引数を引数受け渡し用saddr領域に設定
        mov     a,_@KREG13      ;c
        movw    hl,ax
        mov     a,_@KREG14      ;b
        movw    bc,ax
        mov     a,_@KREG15      ;a
        call    !_func2
        mov     _@KREG10,a      ;r
;line 14:      return e + r;
        add     a,_@KREG11      ;e
L0004:
;line 15:      }
        movw    hl,ax           ;aレジスタの退避
        pop     ax              ;
        movw    _@KREG14,ax     ;_@KREG10～_@KREG15領域の復帰
        pop     ax              ;
```

スタティック・モデル拡張仕様 -ZM

（出力オブジェクト～続き～）

```
        movw    _@KREG12,ax     ;
        pop     ax              ;
        movw    _@KREG10,ax     ;
        movw    ax,hl           ;aレジスタの復帰
        ret
;line 16:      char func2(char a, char b, char c, char d)
;line 17:      {
_func2:
        mov     _@NRAT01,a
        movw    ax,bc
        mov     _@NRAT02,a
        movw    ax,hl
        mov     _@NRAT03,a
;line 18:      return a + b + c + d;
        mov     a,_@NRAT01      ;a
        add     a,_@NRAT02      ;b
        add     a,_@NRAT03      ;c
        add     a,_@NRAT00      ;d leaf関数は _@NRAT00をそのまま使用
L0006:
;line 19:      }
        ret
```

スタティック・モデル拡張仕様 -ZM

【使用例2】

（Cソース）

```
_ _sreg  struct  x {
        unsigned char a;
        unsigned char b:1;
        unsigned char c:1;
} xx,yy;
_ _sreg  struct  y {
        int a;
        int b;
} ss, tt;
struct x func1(struct x);
struct y func2();
void main()
{
        yy = func1(xx);
        tt = func2();
}
struct x func1(struct x aa)
{
        aa.a = 0x12;
        aa.b = 0;
        aa.c = 1;
        return aa;
}
struct y func2()
{
        return tt;
}
```

スタティック・モデル拡張仕様 -ZM

（出力オブジェクト）

-SM -ZM指定あり

```
@@CODE CSEG
_main:
;line     14: yy = func1(xx);
       movw   ax,_xx
       call   !_func1
       movw   _yy,ax
;line     15: tt = func2();
       call   !_func2
       movw   hl,ax
       push   de
       movw   de,#_tt
       mov    c,#04H ;4
       mov    a,[hl]
       mov    [de],a
       incw   hl
       incw   de
       dbnz   c,$$-4
       pop    de
;line     16: }
       ret
;line     17: struct x func1(struct x aa)
;line     18: {
_func1:
       movw   _@NRAT00,ax
;line     19: aa.a = 0x12;
       mov    _@NRAT00,#012H   ;aa,18
;line     20: aa.b = 0;
       clr1   _@NRAT01.0
;line     21: aa.c = 1;
       set1   _@NRAT01.1
;line     22: return aa;
       movw   ax,_@NRAT00      ;aa  2バイト以下なので値返し
;line     23:}
       ret
```

スタティック・モデル拡張仕様 -ZM

（出力オブジェクト～続き～）

```
;line     24: struct y func2()
;line     25: {
;line     26: return tt;
       movw   hl,#_tt          ;3バイト以上なので，静的領域を確保し，
       push   de               ;静的領域に返り値をコピー
       movw   de,#L0007
       mov    c,#04H ;4
       mov    a,[hl]
       mov    [de],a
       incw   hl
       incw   de
       dbnz   c,$$-4
       pop    de
       movw   ax,#L0007        ;静的領域の先頭アドレスを返す
;line     27: }
       ret
```

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・ソース修正は必要ありません。

＜本Cコンパイラから他のＣコンパイラ＞

・ソース修正は必要ありません。

(34) テンポラリ変数

テンポラリ変数 _ _temp

【機　　能】

- leaf関数に該当する／しないにかかわらず，引数，オートマティック変数を_@NRAT00～_@NRAT07の領域に割り当てます。_@NRAT00～_@NRAT07領域に割りあたらなかった場合は，_ _temp宣言がない場合と同じ扱いとします。
- _ _temp宣言された引数とオートマティック変数は，関数呼び出し時に値が破壊されます。
- 外部変数とstatic変数には，_ _tempは宣言できません。
- _ _sreg宣言を併用した場合，char/unsigned char/short/unsigned short/int/unsigned int変数をビット操作可能とします。
- -SM, -ZMオプションが指定されていない場合に_ _temp宣言を行うと，W339ワーニング・メッセージを出力して，ファイル中の_ _temp宣言を無視します。

【効　　果】

- _ _temp宣言された引数とオートマティック変数は，_@NRAT00～_@NRAT07領域で共有されるため，引数とオートマティック変数領域を節約できます。
- 引数とオートマティック変数の生存区間が明確に分かっていて，関数呼び出しの前後で値の一致が保証される必要がない変数に対して適用すると，メモリの節約になります。

【方　　法】

コンパイル時に-SM，-ZMオプションを指定し，引数とオートマティック変数に対して_ _temp宣言を行います。

【使 用 例】

(Cソース)

```
void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d);
void func2(char a);
void main()
{
        func1(1, 2, 3, 4);
}
void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d)
{
        _ _temp char r;


        d.1 = 0;
        r = a + b + c + d;
        func2(r);
```

テンポラリ変数　　_ _temp

（Cソース～続き～）

```
}
void func2(char r)
{
        int a = 1, b = 2;
        r++;
}
```

（出力オブジェクト）

-SM -ZM指定時

```
@@CODE CSEG
_main:
;line      5: func1(1, 2, 3, 4);
        mov    a,#04H  ; 4
        mov    _@NRAT00,a
        mov    h,#03H  ; 3
        mov    b,#02H  ; 2
        sub    a,#03H  ; 3
        call   !_func1
;line      6: }
        ret
;line      7: void func1(_ _temp char a, char b, char c, _ _sreg _ _temp char d)
;line      8: {
_func1:
        mov    _@NRAT01,a       ;_@NRAT01に割り当て
        movw   ax,bc
        mov    !L0005,a
        movw   ax,hl
        mov    !L0006,a
                                ;_@NRAT00割り当ての引数はそのまま
;line      9: _ _temp char r;
;line     10:
;line     11: d.1 = 0;
        clr1    _@NRAT00.1
```

テンポラリ変数　　_ _temp

（出力オブジェクト～続き～）

```
; line    12: r = a + b + c + d;
        mov    a,_@NRAT01       ;a
        add    a,!L0005         ;b
        add    a,!L0006         ;c
        add    a,_@NRAT00       ;d
        mov    _@NRAT02,a       ;r
;line    13: func2(r);
        call   !_func2
                                ;リターン後は _@NRAT00～_@NRAT02の値は
                                ;変化している
;line    14: }
        ret
;line    15: void func2(char r)
;line    16: {
_func2:
        mov    _@NRAT00,a
;line    17: int a = 1, b = 2;
        movw   ax,#01H ; 1
        movw   _@NRAT02,ax      ;a
        incw   ax
        movw   _@NRAT04,ax      ;b
;line    18 :  r++;
        inc    _@NRAT00
;line    19 : }
        ret
```

テンポラリ変数　　_ _temp

【制　　限】

関数呼び出し時の引数が3引数以下の場合は，関数呼び出し時の引数に，_ _temp宣言された引数とオートマティック変数を記述できます。4引数以上ある場合は，引数評価時に値が破壊される可能性があるため，記述した場合の値は保証しません。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

- 予約語_ _tempを使用していなければ，修正の必要はありません。
- テンポラリ変数に変更したい場合は，前記の方法に従い変更を行います。

＜本Cコンパイラから他のCコンパイラ＞

- #defineにより可能です（11. 6　C**ソースの修正**を参照してください）。
  この変更により，_ _temp変数は通常の変数として扱われます。

(35) プロローグ／エピローグ対応ライブラリ

プロローグ／エピローグ対応ライブラリ -ZD

【機　　能】

- **プロローグ／エピローグ・コードの特定パターンを，ライブラリ呼び出しに置換します。**
- **ユーザが使用できる**callt**の数が，ノーマル・モデル時に**2**個，スタティック・モデル時に最大**10**個減ります。**
- **ノーマル・モデル時のライブラリ置換パターンは，次のとおりです。**

  HL, _@KREGxx**退避／コピー，スタック・フレーム確保** → callt [@@cprep2]

  HL, _@KREGxx**復帰，スタック・フレーム解放** → callt [@@cdisp2]

- **スタティック・モデル時の引数に対する**_@NRATxx, _@KREGxx**の割り当ては，最初の**3**引数が次に述べるパターンにあてはまるように配置します。また，**char/int**混在の場合は，**int**型複数引数のパターンにあてはまるように配置間隔を調整します。**
- **スタティック・モデル時のライブラリ置換パターンは，次のとおりです。**

(char 2**引数用**)

```
mov    _@NRAT00,a  →  callt [@@nrcp2]
movw   ax,bc
mov    _@NRAT01,a

mov    _@KREG15,a  →  callt [@@krcp2]
movw   ax,bc
mov    _@KREG14,a
```

(char 3**引数用**)

```
mov    _@NRAT05,a  →  callt [@@nrcp3]
movw   ax,bc
mov    _@NRAT06,a
movw   ax,hl
mov    _@NRAT07,a

mov    _@KREG15,a  →  callt [@@krcp3]
movw   ax,bc
mov    _@KREG14,a
movw   ax,hl
mov    _@KREG13,a
```

プロローグ／エピローグ対応ライブラリ　-ZD

（char 3引数用～続き～）

```
mov    _@NRAT06,a   →   call !@@nkrc3
movw   ax,bc
mov    _@NRAT07,a
movw   ax,hl
mov    _@KREG15,a
```

（int 2引数用）

```
movw   _@NRAT00,ax →   callt [@@nrip2]
movw   ax,bc
movw   _@NRAT02,ax

movw   _@KREG14,ax →   callt [@@krip2]
movw   ax,bc
movw   _@KREG12,ax
```

（int 3引数用）

```
movw   _@NRAT02,ax →   callt [@@nrip3]
movw   ax,bc
movw   _@NRAT04,ax
movw   ax,hl
movw   _@NRAT06,ax

movw   _@KREG14,ax →   callt [@@krip3]
movw   ax,bc
movw   _@KREG12,ax
movw   ax,hl
movw   _@KREG10,ax

movw   _@NRAT04,ax →   call !@@nkri31
movw   ax,bc
movw   _@NRAT06,ax
movw   ax,hl
movw   _@KREG14,ax
```

プロローグ／エピローグ対応ライブラリ　-ZD

（int 3引数用～続き～）

```
movw   _@NRAT06,ax →  call !@@nkri32
movw   ax,bc
movw   _@KREG14,ax
movw   ax,hl
movw   _@KREG12,ax
```

（退避／復帰用）

```
_@NRAT00～_@NRAT07退避     →    callt [@@nrsave]

_@NRAT00～_@NRAT07復帰     →    callt [@@nrload]

_@KREG14～15退避           →    call !@@krs02

_@KREG12～15退避           →    call !@@krs04
                           →    call !@@krs04i

_@KREG10～15退避           →    call !@@krs06
                           →    call !@@krs06i

_@KREG08～15退避           →    call !@@krs08
                           →    call !@@krs08i

_@KREG06～15退避           →    call !@@krs10
                           →    call !@@krs10i

_@KREG04～15退避           →    call !@@krs12
                           →    call !@@krs12i

_@KREG02～15退避           →    call !@@krs14
                           →    call !@@krs14i

_@KREG00～15退避           →    call !@@krs16
                           →    call !@@krs16i

_@KREG14～15復帰           →    call !@@krl02
```

プロローグ／エピローグ対応ライブラリ　-ZD

（退避／復帰用～続き～）

```
        _@KREG12～15復帰          →    call !@@krl04
                                  →    call !@@krl04i

        _@KREG10～15復帰          →    call !@@krl06
                                  →    call !@@krl06i

        _@KREG08～15復帰          →    call !@@krl08
                                  →    call !@@krl08i

        _@KREG06～15復帰          →    call !@@krl10
                                  →    call !@@krl10i

        _@KREG04～15復帰          →    call !@@krl12
                                  →    call !@@krl12i

        _@KREG02～15復帰          →    call !@@krl14
                                  →    call !@@krl14i

        _@KREG00～15復帰          →    call !@@krl16
                                  →    call !@@krl16i
```

【効　果】

プロローグ／エピローグ・コードをライブラリに置換することにより，オブジェクト・コードを短縮できます。

【方　法】

コンパイル時に-ZDオプションを指定します。

【使 用 例1】

（Cソース）

```
int func1(int a, int b, int c);
int func2(int a, int b, int c);
void main()
{
        int r;


        r = func1(1, 2, 3);
```

プロローグ／エピローグ対応ライブラリ　-ZD

（Cソース～続き～）

```
}
int func1(int a, int b, int c)
{
        return func2(a+1, b+1, c+1);
}
int func2(int a, int b, int c)
{
        return a+b+c;
}
```

（出力オブジェクト）

-SM -ZM2D -QC指定あり

```
@@CODE CSEG
_main:
        movw   ax,_@KREG14
        push   ax
;line     5:  int r;
;line     6:
;line     7:  r = func1(1, 2, 3);
        movw   hl, #03H ; 3
        movw   bc, #02H ; 2
        movw   ax, #01H ; 1
        call   !_func1
        movw   _@KREG14,ax      ;r
;line     8:  }
        pop    ax
        movw   _@KREG14,ax
        ret
;line     9:  int func1(int a, int b, int c)
;line     10: {
_func1:
        call   !@@krs06
        callt  [@@krip3]
;line     11: return func2(a+1, b+1, c+1);
        movw    ax, _@KREG10    ;c
```

プロローグ／エピローグ対応ライブラリ　-ZD

（出力オブジェクト～続き～）

```
        incw   ax
        movw   hl,ax
        movw   ax,_@KREG12      ;b
        incw   ax
        movw   bc,ax
        movw   ax,_@KREG14      ;a
        incw   ax
        call   !_func2
L0004:
;line     12: }
        call   !@@krl06
        ret
;line     13: int func2(int a, int b, int c)
;line     14: {
_func2:
        callt  [@@nrip3]
;line     15: return a+b+c;
        movw   ax, _@NRAT02     ;a
        xch    a,x
        add    a, _@NRAT04      ;b
        xch    a,x
        addc   a, _@NRAT05      ;b
        xch    a,x
        add    a, _@NRAT06      ;c
        xch    a,x
        addc   a, _@NRAT07      ;c
L0006:
;line     16: }
        ret
```

プロローグ／エピローグ対応ライブラリ　-ZD

【使 用 例2】

（Cソース）

```
int func(register int a, register int b);
void main()
{
        register int a = 1, b = 2, c = 3,r;
        r = func(a, b);
}
int func(register int a, register int b)
{
        register int r;

        r = a + b;
        return r;
}
```

（出力オブジェクト）

-QR –ZD指定あり

```
@@CODE CSEG
_main:
        movw   de,#03100H
        callt  [@@cprep2]
;line     4:  register int a = 1, b = 2, c = 3, r;
        movw   hl,#01H ; 1
        movw   ax,hl
        incw   ax
        movw   _@KREG14,ax      ;b
        incw   ax
        movw   _@KREG12,ax      ;c
;line     5:
;line     6:  r = func(a, b);
        movw   ax,_@KREG14      ;b
        push   ax
        movw   ax,hl
        call   !_func
        pop    ax
        movw   ax,bc
```

プロローグ／エピローグ対応ライブラリ　-ZD

（出力オブジェクト～続き～）

```
        movw   _@KREG10,ax      ;r
;line     7:  }
        movw   ax,#03100H
        callt  [@@cdisp2]
        ret
;line     8:  int func(register int a, register int b)
;line     9:  {
_func:
        movw   de,#0E840H
        callt  [@@cprep2]
;line     10: register int r;
;line     11:
;line     12: r = a + b;
        movw   ax,hl
        xch    a,x
        add    a,_@KREG12       ;a
        xch    a,x
        addc   a,_@KREG13       ;a
        movw   _@KREG14,ax      ;r
L0004:
;line     14: }
        movw   ax,#0E840H
        callt  [@@cdisp2]
        ret
```

プロローグ／エピローグ対応ライブラリ　-ZD

【制　　限】

・最適化指定オプション-QL4は同時に指定できません。指定した場合はW052ワーニング・メッセージを出力して，-QL4オプションを-QL3オプション指定に置き換えて処理します。

【注　　意】

スタティック・モデル時の引数コピー・パターンは，最初の3引数以内に対してregister指定がない場合，または最初の3引数以内に対してすべて_ _temp指定を行っている場合のみ，パターン・マッチングします。したがって，-QVオプション指定，および最初の3引数以内で部分的にregister/_ _temp指定を行うとパターン・マッチングしないため，-ZDオプション指定の効果が得られなくなります。

【互 換 性】

＜他のCコンパイラから本Cコンパイラ＞

・ソースの修正は必要ありません。

・プロローグ／エピローグ・コードをライブラリに置換したい場合は，前記の方法に従い変更を行います。

＜本Cコンパイラから他のCコンパイラ＞

・ソースの修正は必要ありません。

11. 6　Cソースの修正

拡張機能を使用することにより，効率の良いオブジェクトを生成できます。しかし，拡張機能は 78K/0Sシリーズに即したもので，他に利用するためには修正が必要になる場合があります。ここでは，他のCコンパイラから本Cコンパイラへの移植と，本Cコンパイラから他のCコンパイラへの移植の2つの場合について，その方法を説明します。

＜他のCコンパイラから本Cコンパイラ＞

- #pragma[注]

  他のCコンパイラが #pragmaをサポートしている場合は，Cソースを修正する必要があります。修正方法はそのCコンパイラの仕様によって検討します。

- 拡張仕様

  他のCコンパイラがキーワードを追加するなどの仕様の拡張を行っている場合は，修正する必要があります。修正方法はそのCコンパイラの仕様によって検討します。

注　ANSIでサポートされている前処理指令の1つで，#pragmaに続く文字列をコンパイラへの指令として認識させるものです。その指令がコンパイラによってサポートされていなければ，#pragma指令は無視され，コンパイルが続けられて正常に終了します。

＜本Cコンパイラから他のCコンパイラ＞

本Cコンパイラは，拡張機能としてキーワードの追加を行っているため，他のCコンパイラへ移植するためには，キーワードを削除するか，#ifdefで切り分けなければなりません。

【例】

① キーワードを無効にする（callf，sreg，noauto，norecなども同様）

```
#ifndef _ _K0S_ _
    #define callt          /* callt関数を通常の関数にします。*/
#endif
```

② 他の型に変更する

```
#ifndef _ _K0S_ _
    #define bit char       /* bit型変数をchar型変数にします。*/
#endif
```

11. 7　関数呼び出しインタフェース

関数呼び出し時の関数間インタフェースについて次の内容を説明します。

1. **返り値（すべての関数で共通）**

2. **通常関数呼び出しインタフェース**
   - **（1）引数の渡し方**
   - **（2）引数の格納場所と順序**
   - **（3）自動変数の格納場所と順序**

3. noauto**関数呼び出しインタフェース**
   - **（1）引数の渡し方**
   - **（2）引数の格納場所と順序**
   - **（3）自動変数の格納場所と順序**

4. norec**関数呼び出しインタフェース**
   - **（1）引数の渡し方**
   - **（2）引数の格納場所と順序**
   - **（3）自動変数の格納場所と順序**

5. **スタティック・モデルの関数呼び出しインタフェース**
   - **（1）引数の渡し方**
   - **（2）引数の格納場所と順序**
   - **（3）自動変数の格納場所と順序**

6. **パスカル関数呼び出しインタフェース**

11. 7. 1　返 り 値

呼び出された関数は返り値を，表11－17のように，レジスタ，キャリー・フラグに格納します。

表11－17　返り値の格納場所

品種 / 型	ノーマル・モデル	スタティック・モデル
1バイト整数	BC	A
2バイト整数		AX
4バイト整数	BC（下位），DE（上位）	サポートしない
ポインタ	BC	AX
構造体，共用体	BC（関数固有の領域にコピーした場合，構造体，共用体の先頭アドレス）	サポートしない
1ビット	CY（キャリー・フラグ）	CY（キャリー・フラグ）
浮動小数点数（float型）	BC（下位），DE（上位）	サポートしない
浮動小数点数（double型）	BC（下位），DE（上位）	サポートしない

11. 7. 2　通常関数呼び出しインタフェース

引数の割り当て場所がすべてレジスタで，自動変数が存在しない関数の場合は，noauto関数呼び出しインタフェースと同様です。

(1) 引数の渡し方

- 引数には，レジスタに割り当てる引数と通常の引数があります。
- レジスタに割り当てる引数は，レジスタ宣言した引数であり，割り当て可能なレジスタ，_@KREGxxがある間，レジスタ，_@KREGxxに割り当たります。ただし，_@KREGxxへの割り当ては，-QR指定時のみ行います。以下，レジスタ，_@KREGxxに割り当たる引数をレジスタ引数と呼びます。
- _@KREGxxについては，**付録**A　saddr**領域のレーベル一覧**を参照してください。
- 残りの引数は，スタックに割り当たります。
- 関数呼び出し側では，レジスタ宣言された引数，通常の引数ともに同じ方法で渡します。第2引数以降は，スタックで渡し，第1引数はレジスタまたはスタックで渡します。
- 関数定義側では，レジスタまたはスタックで渡ってきた引数を，引数割り当て場所に格納します。
- レジスタ引数は，レジスタまたは_@KREGxxにコピーします。受け渡しがレジスタの場合でも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。
- 通常の引数は，スタックに積みます。受け渡しがスタックの場合は，受け渡し場所がそのまま引数割り当て場所になります。
- 引数を割り当てるレジスタの退避，復帰は，関数定義側で行います。
- 第1引数の渡し場所については，**表**11－18に示します。

表11－18　**第1引数の渡し場所（関数呼び出し側）**

オプション / 型	ノーマル・モデルの場合
1**バイト・データ**[注] 2**バイト・データ**[注]	AX
3**バイト・データ**[注]	AX, BC
4**バイト・データ**[注]	AX, BC
浮動小数点数（float**型**）	AX, BC
浮動小数点数（double**型**）	AX, BC
その他	**スタック渡し**

注　1-4**バイト・データ**には，構造体，共用体，ポインタを含みます。

(2) 引数の格納場所と順序

・引数には，レジスタに割り当てる引数と通常の引数があります。レジスタに割り当てる引数は，レジスタ宣言した引数，および-QV指定時の引数です。
・レジスタに割り当てられない引数は，スタックに割り当てます。スタックに割り当たる引数は，最後の引数から順番にスタックに積みます。
・引数を割り当てるレジスタの退避，復帰は，関数定義側で行います。
・通常の引数は，スタックに積みます。受け渡しがスタックの場合は，受け渡し場所がそのまま引数の割り当て場所になります。
・関数定義側では，レジスタまたはスタックで渡ってきた引数を，引数割り当て場所に格納します。レジスタ引数は，レジスタまたは_@KREGxxにコピーします。_@KREGxxへのコピーは，-QR指定時のみ行います。受け渡しがレジスタの場合でも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。
・関数呼び出し側では，レジスタ引数，通常の引数ともに同じ方法で渡します。
第2引数以降はスタックで渡し，第1引数はレジスタまたはスタックで渡します。
第1引数の渡し場所については，表11－18を参照してください。

（使用するレジスタ）

HL
ただし，スタック・フレームがある場合はHLには割り当てません。

（使用するsaddr領域）

_@KREG12〜15

（割り当て順序）

・レジスタの場合

char型	：L, Hの順
int, short, enum型	：HL

・saddr領域の場合

char型	：_@KREG12, _@KREG13, _@KREG14, _@KREG15の順
int, short, enum型	：_@KREG12〜13, _@KREG14〜15の順
long, float, double型	：_@KREG12〜13（下位），_@KREG14〜15（上位）

(3) 自動変数の格納場所と順序

・自動変数には，レジスタに割り当てる自動変数と通常の自動変数があります。レジスタに割り当てる自動変数は，レジスタ宣言した自動変数，-QV指定時の自動変数であり，割り当て可能なレジスタ，_@KREGxxがある間，レジスタ，_@KREGxxに割り当たります。ただし，_@KREGxxへの割り当ては，-QR指定時のみ行います。
以降，レジスタ，_@KREGxxに割り当たる自動変数をレジスタ変数と呼びます。
・_@KREGxxについては，付録A　saddr領域のレーベル一覧を参照してください。
・レジスタ変数は，レジスタ引数を割り当てたあとに割り当てを行います。このため，レジスタ変数がレジスタに割り当たるのは，レジスタ引数の割り当て後にレジスタが余ったときです。
・レジスタに割り当たらなかった自動変数は，スタックに割り当たります。
・自動変数を割り当てるレジスタ，_@KREGxxの退避，復帰は，関数定義側で行います。

(a) 自動変数の割り当て順序

自動変数のレジスタ，_@KREGxxへの割り当て順序は，次のとおりです。

(使用するレジスタ)

HL
ただし，スタック・フレームがある場合はHLには割り当てません。

(使用するsaddr領域)

_@KREG00〜15

(割り当て順序)

・レジスタの場合

char型	：L, Hの順
int, short, enum型	：HL

・saddr領域の場合

char型	：_@KREG00, _@KREG01…_@KREG11の順
int, short, enum型	：_@KREG00〜01, _@KREG02〜03…_@KREG10〜15の順
long, float, double型	：_@KREG00〜03, _@KREG04〜07, _@KREG12〜15の順

・スタックに割り当たる自動変数は，宣言順にスタックに積みます。

【例】

ノーマル・モデルの場合

（Cソース1）

```
void func0 (register int,int);
void main ()
{
        func0 (0x1234,0x5678);
}
void func0 (register int p1,int p2)
{
        register int r;
        int a;
        r=p2;
        a=p1;
}
```

（出力コード）

```
_main:
;line   4:  func0(0x1234,0x5678);
     movw        ax,#05678H        ;22136
     push        ax                          ；スタック受け渡し引数
     movw        ax,#01234H        ;4660     ；第1引数はレジスタ渡し
     call        !_func0                     ；関数呼び出し
     pop         ax                          ；スタック受け渡し引数
;line   5:  }
     ret
;line   6:  void func0(register int p1,int p2)
;line   7:  {
_func0:
     push        hl
     xch         a,x
     xch         a,_@KREG12
     xch         a,x
     xch         a,_@KREG13                  ；レジスタ引数p1を_@KREG12に割り当てる
     push        ax                          ；レジスタ引数用saddr領域退避
     movw        ax,_@KREG14
     push        ax                          ；レジスタ変数用saddr領域退避
     push        ax                          ；自動変数aの領域確保
     movw        ax,sp
     movw        hl,ax
```

（出力コード～続き～）

```
;line    8:  register int r;
;line    9:  int a;
;line   10:  r=p2;
     mov          a,[hl+10]          ;p2        ；スタック受け渡し引数p2を
     xch          a,x
     mov          a,[hl+11]          ;p2
     movw         _@KREG14,ax        ;r         ；レジスタ変数_@KREG14に代入
;line   11:  a=p1;
     movw         ax,_@KREG12        ;p1        ；レジスタ引数_@KREG12を
     mov          [hl+1],a           ;a
     xch          a,x
     mov          [hl],a             ;a         ；自動変数aに代入
;line   12:  }
     pop          ax                            ；自動変数aの領域解放
     pop          ax
     movw         _@KREG14,ax                   ；レジスタ変数用saddr領域復帰
     pop          ax
     movw         _@KREG12,ax                   ；レジスタ引数用saddr領域復帰
     pop          hl
     ret
```

(Cソース2)

```
void func1 (int,register int);
void main ()
{
        func1 (0x1234,0x5678);
}
void func1 (int p1,register int p2)
{
        register int r;
        int a;
        r=p2;
        a=p1;
}
```

(出力コード)

```
_main:
;line   4:  func1(0x1234,0x5678);
     movw        ax,#05678H        ;22136
     push        ax                      ;スタック受け渡し引数
     movw        ax,#01234H        ;4660 ;第1引数はレジスタ渡し
     call        !_func1                 ;関数呼び出し
     pop         ax                      ;スタック受け渡し引数
;line   5:  }
     ret
;line   6:  void func1(int p1,register int p2)
;line   7:  {
_func1:
     push        hl
     push        ax                      ;第1引数p1をスタックに積む
     movw        ax,_@KREG12
     push        ax                      ;レジスタ引数用saddr領域退避
     movw        ax,_@KREG14
     push        ax                      ;レジスタ変数用saddr領域退避
     push        ax                      ;自動変数aの領域確保
     movw        ax,sp
     movw        hl,ax
     mov         a,[hl+12]               ;スタック渡しsaddr領域受け引数p2
     xch         a,x
     mov         a,[hl+13]
     movw        _@KREG12,ax             ;レジスタ引数を_@KREG12に割り当てる
;line   8:  register int r;
;line   9:  int a;
```

（出力コード～続き～）

```
;line   10:  r=p2;
     movw        ax,_@KREG12        ;p2
     movw        _@KREG14,ax        ;r    ；レジスタ変数_@KREG14
;line   11:  a=p1;
     mov         a,[hl+6]           ;p1   ；レジスタ渡しスタック受け引数p1（下位）
     mov         [hl],a             ;a    ；自動変数a（下位）
     xch         a,x
     mov         a,[hl+7]           ;p1   ；レジスタ渡しスタック受け引数p1（上位）
     mov         [hl+1],a           ;a    ；自動変数a（上位）
;line   12:  }
     pop         ax                        ；自動変数aの領域解放
     pop         ax
     movw        _@KREG14,ax               ；レジスタ変数用saddr領域復帰
     pop         ax
     movw        _@KREG12,ax               ；レジスタ引数用saddr領域復帰
     pop         ax
     pop         hl
     ret
```

11. 7. 3　noauto関数呼び出しインタフェース（ノーマル・モデルのみ）

(1) 引数の渡し方

- 関数呼び出し側では，通常関数と同じ方法で渡します（11. 7. 2　**通常関数呼び出しインタフェース**を参照してください）。
- 関数定義側では，レジスタまたはスタックで渡ってきた引数を，レジスタおよび_@KREG12～15にコピーします。_@KREG12～15へのコピーは，-QR指定時のみ行います。受け渡しがレジスタの場合でも，関数呼び出し側（渡す側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。
- 引数を割り当てるレジスタおよび_@KREG12～15の退避，復帰は，関数定義側で行います。

(2) 引数の格納場所と順序

- 関数定義側では，引数はすべて，レジスタおよび_@KREG12～15に割り当たります。ただし，_@KREG12～15への割り当ては，-QR指定時のみ行います。
- レジスタおよび_@KREG12～15に割り当てることができない引数があればエラーとします。
- 関数呼び出し側では，通常関数と同じ方法で渡します（11. 7. 2　**通常関数呼び出しインタフェース**を参照してください）。
- 関数定義側では，レジスタまたはスタックで渡ってきた引数を，レジスタおよび_@KREG12～15にコピーします。受け渡しがレジスタの場合でも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。
- 引数を割り当てるレジスタおよび_@KREG12～15の退避，復帰は，関数定義側で行います。

（割り当て順序）

- 通常関数と同様です（11. 7. 2　**通常関数呼び出しインタフェース**を参照してください）。

(3) 自動変数の格納場所と順序

自動変数は，割り当て可能なレジスタ，_@KREG12～15に割り当たります。

ただし，_@KREG12～15への割り当ては，-QR指定時のみです。_@KREG12～15については，付録Asaddr領域のレーベル一覧を参照してください。

自動変数は，引数を割り当てた後，レジスタが余っていればレジスタに割り当てます。また，-QR指定時は _@KREG12～15にも割り当てます。

レジスタ，_@KREG12～15に割り当てることができない自動変数があればエラーとします。

自動変数を割り当てるレジスタ，_@KREG12～15の退避，復帰は，関数定義側で行います。

(割り当て順序)

・自動変数のレジスタへの割り当て順序は，引数の割り当て順序と同じです。

・_@KREG12～15に割り当たる自動変数は，宣言順に割り当てます。

【例】

(Cソース)

```
noauto void func2(int, int);
void main()
{
        func2(0x1234, 0x5678);
}
noauto void func2(int p1, int p2)
{
   ⋮
}
```

（出力コード）

```
_main:
;line     4: func2 (0x1234,0x5678);
     movw    ax,#05678H       ;22136
     push    ax                          ;スタック渡し引数
     movw    ax,#01234H       ;4660      ;第1引数はレジスタ渡し
     call    !_func2                     ;関数呼び出し
     pop     ax                          ;スタック渡し引数
;line     5: }
     ret
;line     6: noauto void func2 (int p1,int p2)
;line     7: {
_func2:
     push    hl                          ;引数用レジスタ退避
     xch     a,x
     xch     a,_@KREG12                  ;引数p1を_@KREG12に割り当てる（下位）
     xch     a,x
     xch     a,_@KREG13                  ;引数p1を_@KREG13に割り当てる（上位）
     push    ax                          ;引数用saddr領域退避
     movw    ax,sp
     movw    hl,ax
     mov     a,[hl+6]                    ;スタック渡しレジスタ受け引数p2（下位）
     xch     a,x
     mov     a,[hl+7]                    ;スタック渡しレジスタ受け引数p2（上位）
     movw    hl,ax                       ;引数をHLに割り当てる
        ⋮
     pop     ax
     movw    _@KREG12,ax                 ;引数用saddr領域復帰
     pop     hl                          ;引数用レジスタ復帰
     ret
```

11. 7. 4　norec関数呼び出しインタフェース（ノーマル・モデルのみ）

（1）引数の渡し方

引数はすべて，レジスタ，_@NRARGx, _@RTARG6, 7**に割り当たります。関数呼び出し側では，引数をレジスタ，**_@NRARGx**で渡します。**

関数定義側では，レジスタで渡ってきた引数を，レジスタまたは_@RTARG6, 7**にコピーします（付録**A　saddr**領域のレーベル一覧を参照してください）。**

（2）引数の格納場所と順序

- **関数定義側では，引数はすべて，レジスタ，**_@NRARGx, _@RTARG6, 7**に割り当たります。ただし，**_@NRARGx**への割り当ては，**-QR**指定時のみ行われます。**
- _@RTARG6～7**への割り当ては，**DE**に格納された引数がある場合のみです（付録**A　saddr**領域のレーベル一覧を参照してください）。**
- **レジスタ，**_@NRARGx, _@RTARG6, 7**に割り当てることができない引数があれば，エラーとします。**
- **関数呼び出し側では，引数をレジスタ，**_@NRARGx**で渡します。**
- **関数定義側では，レジスタで渡ってきた引数をレジスタまたは**_@RTARG6, 7**にコピーします。受け渡しがレジスタの場合でも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。**
  受け渡しが_@NRARGx**の場合は，受け渡し場所がそのまま引数の割り当て場所になります。**
- **レジスタでの受け渡しができなくなったときは，**_@NRARGx**にも割り当てて受け渡します。レジスタと**_@NRARGx**を混在して受け渡すことになります。**

（引数の割り当て順序）

- _@NRARGx**に割り当たる引数は，宣言順に割り当てます。**
- **レジスタに割り当たる引数は，次の規則でレジスタ，**_@RTARG6, 7**に割り当てます。**

（使用するレジスタ）

- **引数が**char, int, short, enum, **ポインタ型1個の場合　　：**AX**渡し，**DE**受け**
- **引数が**char, int, short, enum, **ポインタ型2個以上の場合　：**AX, DE**渡し**
  _@RTARG6, 7
  DE**受け**

（割り当て順序）

- char, int, short, enum, **ポインタ型　　：**DE, _@RTARG6, 7**の順**

(3) 自動変数の格納場所と順序

自動変数は，割り当て可能なレジスタ，_@NRARGxがある間，レジスタ，_@NRARGxに割り当たり，なくなれば_@NRATxxに割り当たります。

ただし，_@NRARGx，_@NRATxxへの割り当ては，-QR指定時のみ行われます。_@NRATxxについては，付録A　saddr領域のレーベル一覧を参照してください。

レジスタ，_@NRARGx，_@NRATxxに割り当てることができない自動変数があればエラーとします。

自動変数を割り当てるレジスタの退避，復帰は，関数定義側で行います。

（割り当て順序）

- 自動変数のレジスタ，_@RTARG6～7への割り当て順序は，引数の割り当て順序と同じです。
- _@NRARGx, _@NRATxxに割り当たる自動変数は，宣言順に割り当てます。

【例】

ノーマル・モデルの場合

（Cソース）

```
norec void func3(char, int, char, int);
void main()
{
        func3(0x12, 0x34, 0x56, 0x78);
}
norec void func3(char p1, int p2, char p3, int p4)
{
        int a;
        a = p2;
}
```

（出力コード）

-QR指定

```
_main:
;line     4: func3(0x12, 0x34, 0x56, 0x78);
     movw    ax, #078H                  ;引数を_@NRARG1で渡す
     movw    _@NRARG1, ax
     mov     _@NRARG0, #056H  ;86       ;引数を_@NRARG0で渡す
     movw    de, #034H        ;52       ;引数をレジスタDEで渡す
     mov     a, #012H         ;18       ;引数をレジスタAで渡す
     call    !_func3                    ;関数呼び出し
     ret
;line     6: norec void func3(char p1, int p2, char p3, int p4)
;line     7: {
_func3:
     mov     _@RTARG6, a                ;引数p1を_@RTARG6に割り当てる
;line     8: int a;
;line     9: a = p2;
     movw    ax, de                     ;引数p2
     movw    _@NRARG2, ax     ;a        ;自動変数a
     ret
```

11. 7. 5　スタティック・モデルの関数呼び出しインタフェース

(1) 引数の渡し方

・関数呼び出し側では，レジスタ引数，通常の引数ともに同じ方法で渡します。
　引数は最大3引数，6バイトまでとし，すべてレジスタで渡します。
・関数定義側では，レジスタで渡ってきた引数を，引数割り当て場所に格納します。
　レジスタ引数は，レジスタにコピーします。受け渡しがすべてレジスタでも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。
・通常の引数は，関数固有の領域に割り当てます。

(2) 引数の格納場所と順序

(a) 引数の格納場所

・引数には，レジスタに割り当てる引数と通常の引数があります。
・レジスタに割り当てる引数は，レジスタ宣言した引数であり，レジスタに割り当て可能なかぎり，レジスタに割り当たります。
・関数定義側では，レジスタで渡ってきた引数を，引数割り当て場所に格納します。
　レジスタ引数は，レジスタにコピーします。受け渡しがすべてレジスタでも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが必要です。通常の引数は，関数固有の領域に割り当てます。
・引数／オートマティック変数を割り当てるレジスタの退避，復帰は，関数定義側で行います。
・残りの引数は，関数固有に確保した領域に割り当たります。
・関数呼び出し側では，レジスタ引数，通常の引数ともに同じ方法で渡します。
　引数は最大3引数，6バイトまでとし，すべてレジスタで渡します。
　引数の渡し場所を，表11－19に示します。

表11－19　スタティック・モデルの引数の渡し場所

データ・サイズ	第1引数	第2引数	第3引数
1バイト・データ[注]	A	B	H
2バイト・データ[注]	AX	BC	HL
4バイト・データ[注]	AX, BCに割り当て，残りをHまたはHLに割り当てます。		

注　1-4バイト・データには，構造体，共用体は含みません。

(b) 引数の割り当て順序

・関数固有の領域に割り当たる引数は，最後の引数から順番に割り当てます。
・レジスタ引数は，次の規則でレジスタDEに割り当てます。

（使用するレジスタ）

DE

（割り当て順序）

char型　　：D, Eの順
int, short, enum型：DE

(3) 自動変数の格納場所と順序

(a) 自動変数の格納場所

- 自動変数には，レジスタに割り当てる自動変数と通常の自動変数があります。
- レジスタに割り当てる自動変数は，レジスタ宣言した自動変数，-QV指定時の自動変数です。
- レジスタ変数は，レジスタ引数を割り当てたあとに割り当てを行います。このため，レジスタ変数がレジスタに割り当たるのは，レジスタ引数の割り当て後にレジスタが余ったときです。
- 残りの自動変数は，関数固有の領域に割り当たります。
- 自動変数を割り当てるレジスタの退避，復帰は，関数定義側で行います。

(b) 自動変数の割り当て順序

- 自動変数のレジスタへの割り当て順序は，次の規則でレジスタDEに割り当てます。

（使用するレジスタ）

DE

（割り当て順序）

char	：E, Dの順
int, short, enum型	：DE

- 関数固有の領域に割り当てられる自動変数は，宣言順に割り当てます。

【例1】

（Cソース）

```
void func4(register int, char);
void main()
{
        func4(0x1234, 0x56);
}
void func4(register int p1, char p2)
{
        register char r;
        int a;
        r = p2;
        a = p1;
}
```

（出力コード）

```
@@DATA      DSEG
L0005:      DS      (1)             ;引数p2
L0006:      DS      (1)             ;自動変数r
L0007:      DS      (2)             ;自動変数a

;line    1: void func4(register int, char);
;line    2: void main()
;line    3:  {

@@CODE      CSEG
_main:
;line    4: func4(0x1234, 0x56);
    mov     b, #056H        ;86     ;第2引数をレジスタBで渡す
    movw    ax, #01234H     ;4660   ;第1引数をレジスタAXで渡す
    call    !_func4                 ;関数呼び出し
;line    5: }
    ret
;line    6: void func4(register int p1, char p2)
;line    7: {
_func4:
    push    de                      ;レジスタ引数用レジスタ退避
    movw    de, ax                  ;レジスタ引数p1をDEに割り当てる
    movw    ax, bc
    mov     !L0005, a               ;引数p2をL0005にコピー
;line    8: register char r;
;line    9: int a;
;line   10: r = p2;
    mov     !L0006, a               ;r      ;自動変数r
;line   11: a = p1
    movw    ax, de                          ;レジスタ引数p1
    movw    hl, #L0007              ;a      ;自動変数a
    callt   [@@hlist]
;line   12: }
    pop     de                      ;レジスタ引数用レジスタ復帰
    ret
```

【例2】

（Cソース）

```
void func5(int, register char);  void func();
void main()
{
        func5(0x1234, 0x56);
}
void func5(int p1, register char p2)
{
     register char r;
     int a;
     r = p2;
     a = p1; func();
}
```

（出力コード）

```
@@DATA  DSEG
L0005:  DS      (2)
L0006:  DS      (2)

;line     1: void func5(int, register char);     void func();
;line     2: void main()
;line     3: {

@@CODE       CSEG
_main:
;line     4: func5(0x1234, 0x56);
              mov     b, #056H        ;86     ;第2引数をレジスタBで渡す
              movw    ax, #01234H     ;4660   ;第1引数をレジスタAXで渡す
              call    !_func5                 ;関数呼び出し
;line     5: }
              ret
;line     6: void func5(int p1, register char p2)
;line     7: {
_func5:
              push    de                      ;レジスタ変数，レジスタ引数用レジスタ退避
              movw    hl, #L0005              ;引数p1をL0005にコピー
              callt   [@@hlist]
              movw    ax, bc
```

（出力コード～続き～）

```
                mov     de, ax                  ;レジスタ引数p2をdに割り当てる
    ;line     8: register char r;
    ;line     9: int a;
    ;line    10: r = p2;
                movw    ax, de                  ;レジスタ引数p2
                mov     e, a                    ;レジスタ変数r
    ;line    11: a = p1; func();
                movw    hl, #L0005      ;p1     ;引数p1
                callt   [@@hlilo]
                movw    hl, #L0006      ;a      ;自動変数a
                callt   [@@hlist]
                call    !_func
    ;line    12: }
                pop     de                      ;レジスタ引数用レジスタ復帰
                ret
```

11. 7. 6　パスカル関数呼び出しインタフェース

関数呼び出し時に引数の積み込みによって使用したスタックの修正を，関数呼び出し側で行わずに，呼ばれた関数側で行う点のみが他関数インタフェースと異なる点であり，それ以外の点は同時に指定された関数属性と同じです。

[引数の割り当て場所]
[引数の割り当て順序]
[自動変数の割り当て場所]
[自動変数の割り当て順序]

- noauto属性が同時に指定されている場合は，noauto関数呼び出しと同じです（11. 7. 3　noauto関数呼び出しインタフェースを参照してください）。
- noauto属性が同時に指定されていない場合は，通常関数呼び出しと同じです（11. 7. 2　通常関数呼び出しインタフェースを参照してください）。

【例1】

（Cソース）

```
_ _pascal void func0(register int, int);
void main()
{
        func0(0x1234, 0x5678);
}
_ _pascal void func0(register int p1, int p2)
{
        register int r;
        int a;
        r = p2;
        a = p1;
}
```

（出力コード）

-QR指定あり

```
_main:
;line     4: func0(0x1234, 0x5678);
     movw    ax, #05678H      ;22136
     push    ax                          ;スタック受け渡し引数
     movw    ax, #01234H      ;4660      ;第1引数はレジスタ渡し
     call    !_func0                     ;関数呼び出し
                                         ;ここでスタックの修正をしない
;line     5: }
     ret
;line     6: _ _pascal void func0(register int p1, int p2)
;line     7: {
_func0:
     push    hl
     xch     a, x
     xch     a,_@KREG12
     xch     a, x
     xch     a,_@KREG13                  ;レジスタ引数p1を_@KREG12に割り当てる
     push    ax                          ;レジスタ引数用saddr領域退避
     movw    ax,_@KREG14
     push    ax                          ;レジスタ変数用saddr領域退避
     push    ax                          ;自動変数aの領域確保
     movw    ax, sp
     movw    hl, ax
;line     8: register int r;
;line     9: int a;
;line    10: r = p2;
     mov     a,[hl+10]        ;p2        ;スタック受け渡し引数p2を
     xch     a, x
     mov     a,[hl+11]        ;p2
     movw    _@KREG14, ax     ;r         ;レジスタ変数_@KREG14に代入
;line    11: a = p1;
     movw    ax,_@KREG12      ;p1        ;レジスタ引数_@KREG12を
     mov     [hl+1], a        ;a
     xch     a, x
     mov     [hl], a          ;a         ;自動変数aに代入
;line    12: }
     pop     ax                          ;自動変数aの領域解放
     pop     ax
     movw    _@KREG14, ax                ;レジスタ変数用saddr領域復帰
```

（出力コード～続き～）

```
        pop     ax
        movw    _@KREG12, ax                ;レジスタ引数用saddr領域復帰
        pop     hl
        pop     de                          ;リターン・アドレスを取得
        pop     ax                          ;スタック受け渡し引数で消費したスタックを修正
        push    de                          ;リターン・アドレスの積み直し
        ret
```

【例2】

（Cソース）

```
_ _pascal noauto void func2(int, int);
void main()
{
        func2(0x1234, 0x5678);
}
_ _pascal noauto void func2(int p1, int p2)
{
        ⋮
}
```

（出力コード）

-QR指定あり

```
_main:
;line     4: func2(0x1234, 0x5678);
            movw   ax, #05678H ;22136
            push   ax                       ;スタック渡し引数
            movw   ax, #01234H ;4660        ;第1引数はレジスタ渡し
            call   !_func2                  ;関数呼び出し
                                            ;ここでスタックの修正をしない
;line     5: }
            ret
;line     6: _ _pascal noauto void func2(int p1, int p2)
;line     7:{
_func2:
            push   hl                       ;引数用レジスタ退避
            xch    a, x
```

（出力コード～続き～）

```
        xch     a,_@KREG12              ;引数p1を_@KREG12に割り当てる（下位）
        xch     a, x
        xch     a,_@KREG13              ;引数p1を_@KREG13に割り当てる（上位）
        push    ax                      ;引数用saddr領域退避
        movw    ax, sp
        movw    hl, ax
        mov     a,[hl+6]                ;スタック渡しレジスタ受け引数p2（下位）
        xch     a, x
        mov     a,[hl+7]                ;スタック渡しレジスタ受け引数p2（上位）
        movw    hl, ax                  ;引数をHLに割り当てる
          ⋮
        pop     ax
        movw    _@KREG12, ax            ;引数用saddr領域復帰
        pop     hl                      ;引数用レジスタ復帰
        pop     de                      ;リターン・アドレスを取得
        pop     ax                      ;スタック受け渡し引数で消費したスタックを修正
        push    de                      ;リターン・アドレスの積み直し
        ret
```

〔メ　モ〕

第12章　アセンブラとの相互参照

この章では，アセンブリ言語で作成したプログラムとのリンク方法について説明します。

Cソース・プログラムから呼び出す関数が他言語で記述されている場合，双方のオブジェクト・モジュールをリンカで結合します。この章では，C言語で記述されたプログラムが他言語で記述されたプログラムを呼び出す手順，および他言語で記述されたプログラムからC言語で記述されたプログラムを呼び出す手順を説明します。

他言語とのインタフェースの方法について，本CコンパイラとRA78K0S アセンブラ・パッケージを使い次の順序で説明します。

(1) C言語からアセンブリ言語ルーチンの呼び出し
(2) アセンブリ言語からC言語関数の呼び出し
(3) C言語で定義した変数を参照する方法
(4) アセンブリ言語で定義した変数をC言語側で参照する方法
(5) その他注意事項

12. 1　引数／オートマティック変数のアクセス方法

本Cコンパイラの引数／オートマティック変数のアクセス方法は次の通りです。

12. 1. 1　ノーマル・モデルの場合

・関数呼び出し側では，レジスタ引数，通常の引数ともに同じ方法で渡します。
第1引数は次に示すレジスタおよびスタックを使用し，第2引数以降はスタックで渡します。

表12－1　引数の引き渡し方法（関数呼び出し側）

型	渡し場所（第1引数）	渡し場所（第2引数～）
1バイト，2バイト・データ	AX	スタック渡し
3バイト，4バイト・データ	AX, BC	スタック渡し
浮動小数点数	AX, BC	スタック渡し
その他	スタック渡し	スタック渡し

備考　1-4バイト・データには，構造体，共用体を含みます。

・関数定義側では，レジスタまたはスタックで渡ってきた引数を，引数割り当て場所に格納します。
レジスタ引数は，レジスタまたはsaddr領域（_@KREGxx）にコピーされます。受け渡しがレジスタの場合でも，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが行われます。
レジスタで渡ってきた通常の引数は，関数定義側でスタックに積みます。受け渡しがスタックの場合は，受け渡し場所がそのまま引数の割り当て場所になります。
引数を割り当てるレジスタの退避，復帰は，関数定義側で行います。
・関数の引数，および関数内で宣言されたオートマティック変数の値をオプションにより，次のレジスタ，saddr領域，あるいはスタック・フレームに格納します。スタック・フレームに格納する際のベース・ポインタは，HLレジスタを使用します。
関数の引数は，register宣言または-QVオプションが指定されていて，かつ-QRオプションが指定されている場合に，saddr領域に割り付けられます。

表12－2　引数／オートマティック変数の格納一覧（呼ばれる関数内）

オプション	引数／auto変数	格納場所	優先順位
-QV （レジスタ割り当てオプション）	宣言された引数またはオートマティック変数	HLレジスタ （ベース・ポインタが必要ない場合のみ）	char型：L, Hの順 int, short, enum型：HL
-QR （saddr割り当てオプション）	register宣言された引数またはオートマティック変数	HLレジスタ （ベース・ポインタが必要ない場合のみ） 引数： _@KREG12～15 [0FEE4H～0FEE7H] オートマティック変数： _@KREG00～11 [0FED8H～0FEE3H] 引数に割り当たらなかった _@KREG12～15	参照回数に基づきサイズ分のみ割り当てる。 レジスタには， char型：L, Hの順 int, short, enum型：HL のように割り当てる。
-QRV	宣言された引数またはオートマティック変数	HLレジスタ （ベース・ポインタが必要ない場合のみ） 引数： _@KREG12～15 [0FEE4H～0FEE7H] オートマティック変数： _@KREG00～11 [0FED8H～0FEE3H] 引数に割り当たらなかった _@KREG12～15	参照回数に基づきサイズ分のみ割り当てる。 レジスタには， char型：L, Hの順 int, short, enum型：HL のように割り当てる。
ディフォールト	宣言された引数， オートマティック変数	スタック・フレーム	出現順

次に関数呼び出し例を示します。

（Cソース：ノーマル・モデル，-QRV指定時）

```
      void func0(register  int,  int);
      void main(){
               func0(0x1234,0x5678);
      }
      void func0(register int p1,int p2){
               register  int  r;
               int  a;
               r=p2;
               a=p1;
      }
```

（出力アセンブラ・ソース）

```
      EXTRN    _@KREG12
      EXTRN    _@KREG13
      EXTRN    _@KREG10
      EXTRN    _@KREG14
      PUBLIC   _func0
      PUBLIC   _main

@@CODE         CSEG
_main:
      movw     ax,#05678H     ;22136
      push     ax                          ；スタック受け渡し引数
      movw     ax,#01234H     ;4660        ；第1引数はレジスタ渡し
      call     !_func0                     ；関数呼び出し
      pop      ax                          ；スタック受け渡し引数
      ret
_func0:
      push     hl                          ；引数用レジスタ退避
      xch      a,x
      xch      a,_@KREG12
      xch      a,x
      xch      a,_@KREG13                  ；レジスタ引数p1を _@KREG12に割り当てる
      push     ax                          ；レジスタ引数用saddr領域退避
      movw     ax,_@KREG10
      push     ax                          ；レジスタ変数用saddr領域退避
      movw     ax,_@KREG14
      push     ax                          ；自動変数用saddr領域退避
```

（出力アセンブラ・ソース　～続き～）

```
        movw    ax,sp
        movw    hl,ax
        mov     a,[hl+10]               ；スタック受け渡し引数p2を
        xch     a,x
        mov     a,[hl+11]
        movw    hl,ax                   ；HLに割り当てる
        movw    ax,hl                   ；引数p2を
        movw    _@KREG14,ax     ;r      ；レジスタ変数rに代入
        movw    ax,_@KREG12     ;p1     ；レジスタ引数p1を
        movw    _@KREG10,ax     ;a      ；自動変数aに代入
        pop     ax
        movw    _@KREG14,ax             ；レジスタ変数用saddr領域復帰
        pop     ax
        movw    _@KREG10,ax             ；自動変数用saddr領域復帰
        pop     ax
        movw    _@KREG12,ax             ；レジスタ引数用saddr領域復帰
        pop     hl                      ；引数用レジスタ復帰
        ret
        END
```

12. 1. 2　スタティック・モデルの場合

・関数呼び出し側では，レジスタ引数，通常の引数ともに同じ方法で渡します。
・引数は最大3引数，6バイトまでとし，すべてレジスタで渡します。

表12－3　引数の引き渡し方法（関数呼び出し側）

型	渡し場所（第1引数）	渡し場所（第2引数）	渡し場所（第3引数）
1バイト・データ	A	B	H
2バイト・データ	AX	BC	HL
4バイト・データ	AX, BCに割り当て，残りをHまたはHLに割り当てる		

備考　1-4バイト・データには，構造体，共用体は含みません。

・関数定義側では，レジスタで渡ってきた引数を，引数割り当て場所に格納します。
　register宣言された引数（レジスタ引数）は，可能なかぎりレジスタに割り当て，通常の引数は，関数固有に確保した領域に割り当てます。
・レジスタ引数は，すべてレジスタで受け渡しが行われますが，関数呼び出し側（渡し側）と関数定義側（受け側）のレジスタが異なるため，レジスタのコピーが行われます。
・引数／オートマティック変数を割り当てるレジスタの退避，復帰は，関数定義側で行います。
・関数の引数，および関数内で宣言されたオートマティック変数の値をオプションにより，次のレジスタ，関数固有の領域に格納します。関数固有の領域は，関数ごとにRAM内の領域を任意に確保した静的領域です。

表12－4　引数／オートマティック変数の格納一覧（呼ばれる関数内）

オプション	引数／auto変数	格納場所	優先順位
-QV （レジスタ割り当てオプション）	宣言された引数またはオートマティック変数	DEレジスタ	引数： char型：D, Eの順 int, short, enum型：DE オートマティック変数： char型：E, Dの順 int, short, enum型：DE
ディフォールト	宣言された引数， オートマティック変数	関数固有の領域	引数は，第1引数から順，自動変数は出現順に割り当てる
ディフォールト	register宣言された引数， register変数	DEレジスタ	参照回数に基づきサイズ分のみ割り当てる。 サイズ分以上は，関数固有の領域に割り当てる。

次に関数呼び出し例を示します。

（Cソース：スタティック・モデル -SM，-QV指定時）

```
      void sub();
      void func(register  int,  char);
      void main(){
               func(0x1234,0x56);
      }
      void func(register int p1,char p2){
               register  char  r;
               int  a;
               r=p2;
               a=p1;
               sub();
      }
```

（出力アセンブラ・ソース）

```
      PUBLIC   _func
      PUBLIC   _main
               ⋮
@@DATA         DSEG
?L0005:        DS    (1)                ;引数p2
?L0006:        DS    (1)                ;レジスタ変数r
?L0007:        DS    (2)                ;自動変数a
               ⋮
@@CODE         CSEG
_main:
      mov      b,#056H        ;86       ;第2引数をレジスタBで渡す
      movw     ax,#01234H     ;4660     ;第1引数をレジスタAXで渡す
      call     !_func                   ;関数呼び出し
      ret
_func:
      push     de                       ;レジスタ引数用レジスタ退避
      movw     de,ax                    ;レジスタ引数p1をDEに割り当てる
      movw     ax,bc
      mov      !?L0005,a                ;引数p2を?L0005にコピー
      mov      !?L0006,a      ;r        ;レジスタ変数rに代入
      movw     ax,de                    ;レジスタ引数p1を
      mov      !?L0007+1,a    ;a
      xch      a,x
      mov      !?L0007,a      ;a        ;自動変数aに代入
      call     !_sub
      pop      de                       ;レジスタ引数用レジスタ復帰
      ret
      END
```

12. 2　返り値の格納方法

関数呼び出し時の返り値は，レジスタ，キャリー・フラグに格納します。

次に返り値の格納方法を示します。

表12－5　返り値の格納場所

型	ノーマル・モデル	スタティック・モデル
1バイト整数	BC	A
2バイト整数		AX
4バイト整数	BC（下位）DE（上位）	サポートしません
ポインタ	BC	AX
構造体，共用体	BC（関数固有の領域にコピーした構造体，共用体の先頭アドレス）	サポートしません
1ビット	CY（キャリー・フラグ）	CY（キャリー・フラグ）
浮動小数点数	BC（下位）DE（上位）	サポートしません

12. 3　C言語からアセンブリ言語ルーチンの呼び出し

ここでは，ノーマル・モデルを使用した場合（ディフォールト）の例を示します。-QVオプション，-QRオプション，および -QRVオプションを指定した場合は，表12－2に従って格納されます。ただし，HLレジスタは，ベース・ポインタが必要のない場合（使用されていない場合）にのみ割り当てます。

C言語からアセンブリ言語ルーチンの呼び出しを次の順序で説明します。

- C言語の関数呼び出し手順
- アセンブリ言語ルーチンの情報退避とリターン

(1) C言語の関数呼び出し手順

アセンブリ言語ルーチンを呼び出すC言語のプログラム例を次に示します。

```
extern int  FUNC(int,long);          /* 関数プロトタイプ */

void main()
{
        int       i,j;
        long      l;

        i=1;
        l=0x54321;
        j=FUNC(i,l);                 /* 関数コール */
}
```

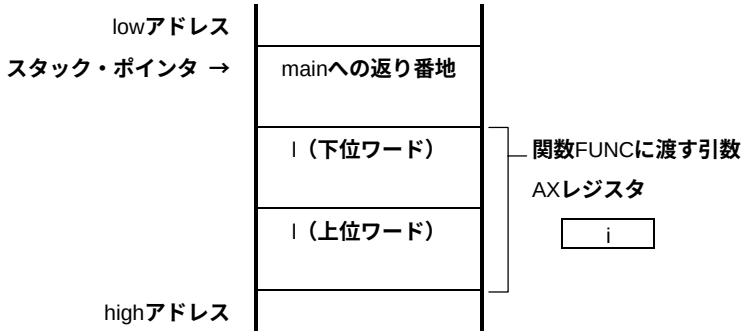
このプログラム例で，実行時に行われるプログラム間のインタフェースと制御の流れを次に示します。

① 関数mainから関数FUNCへ渡す第1引数をレジスタに入れ，第2引数以降をスタックに積む

② CALL命令により関数FUNCに制御を渡す

上記のプログラム例により関数FUNCに制御を移した直後のスタックは，次のようになります。

図12－1　コール後のスタック領域



スタック領域

(2) アセンブリ言語ルーチンの情報退避とリターン

main関数から呼び出される関数FUNCでは，次の処理を行います。

① ベース・ポインタ，ワーク・レジスタを退避する
② スタック・ポインタ（SP）をベース・ポインタ（HL）へコピーする
③ 関数FUNC本来の処理を行う
④ 返り値をセットする
⑤ 退避したレジスタを復帰する
⑥ 関数mainへリターンする

アセンブリ言語のプログラム例を次に示します。

```
$PROCESSOR(9024)

        PUBLIC   _FUNC
        PUBLIC   _DT1
        PUBLIC   _DT2

@@DATA          DSEG
_DT1:    DS     (2)
_DT2:    DS     (4)

@@CODE          CSEG
_FUNC:
        PUSH     HL                    ;save base pointer ・・・・・・・・・・・・・・・・・・ ①
        PUSH     AX
        MOVW     AX,SP                 ;copy stack pointer ・・・・・・・・・・・・・・・・・ ②
        MOVW     HL,AX
        MOV      A,[HL]                ;arg1
        MOV      !_DT1,A               ;move 1st argument(i)
        XCH      A,X
        MOV      A,[HL+1]              ;arg1
        MOV      !_DT1+1,A
        MOV      A,[HL+8]              ;arg2
        XCH      A,X
        MOV      A,[HL+9]              ;arg2
        MOVW     BC,AX
        MOV      A,[HL+6]              ;arg2
        XCH      A,X
        MOV      A,[HL+7]              ;arg2
        MOVW     DE,#_DT2
```

```
        XCH     A,X
        MOV     [DE],A                  ;move 2nd argument(l)
        XCH     A,X
        INCW    DE
        MOV     [DE],A
        XCHW    AX,BC
        INCW    DE
        XCH     A,X
        MOV     [DE],A
        XCH     A,X
        INCW    DE
        MOV     [DE],A
        XCHW    AX,BC
        MOVW    BC,#0AH                 ;set return value ····· ④
        POP     AX
        POP     HL                      ;restore base pointer ····· ⑤
        RET ····· ⑥
        END
```

① ベース・ポインタ，ワーク・レジスタの退避

Cソースで記述した関数名の先頭に，‘ _ ’を付加したレーベルを記述します。Cソース中で記述した関数名と同じ名前になります。

レーベルを記述したあと，HLレジスタ（ベース・ポインタ）を退避します。

Cコンパイラが生成するプログラムでは，レジスタ変数用レジスタを退避せずに他の関数を呼び出します。このため，呼ばれる関数でこれらのレジスタの値を変更する場合は，事前に値の退避を行わなければなりません。ただし，呼び出し側でレジスタ変数を使っていない場合，ワーク・レジスタを退避する必要はありません。

② スタック・ポインタ（SP）のベース・ポインタ（HL）へのコピー

関数内の‘ PUSH，POP ’によりスタック・ポインタ（SP）は変わります。このため，スタック・ポインタを‘ HL ’レジスタにコピーして，引数のベース・ポインタとして使用します。

③ 関数FUNC本来の処理を行う

‘ ①-② ’の処理を行ったあと，呼び出される関数の本来の処理を行います。

④ 返り値のセット

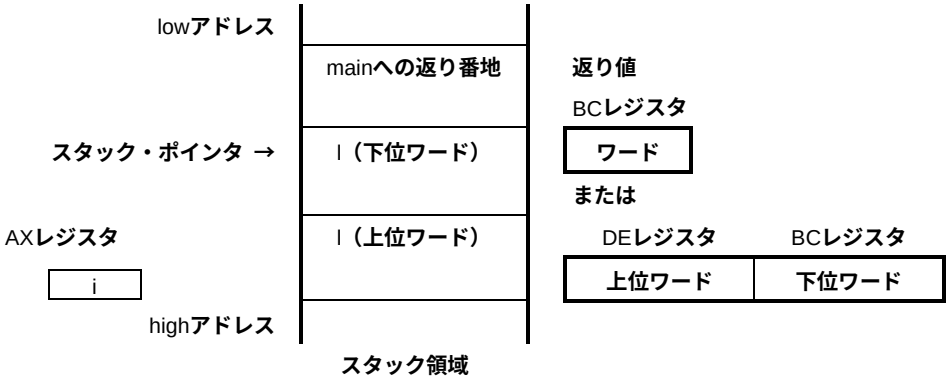
返り値がある場合，返り値を‘ BC ’，‘ DE ’レジスタへセットします。返り値がない場合，セットする必要はありません。

⑤ レジスタの復帰

退避したベース・ポインタとワーク・レジスタを復帰します。

⑥ 関数mainへのリターン

図12－2　リターン後のスタック領域



12. 4　アセンブリ言語からC言語ルーチンの呼び出し

(1) アセンブリ言語の関数呼び出し

C言語により記述された関数を，アセンブリ言語ルーチンから呼び出す手順は，次のようになります。

① 引数をスタックに積む
② Cのワーク・レジスタ（AX, BC, DE）を退避する
③ C言語関数をコールする
④ 引数のバイト数分スタック・ポインタ（SP）の値を修正する
⑤ C言語関数の返り値（BCまたはDE, BC）を参照する

アセンブリ言語のプログラム例を次に示します。

```
$PROCESSOR(9024)

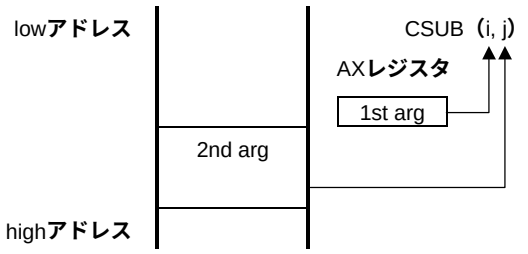
        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2

@@CODE  CSEG
_FUNC2:
        movw    ax,#20H         ;set 2nd argument(j)
        push    ax              ;
        movw    ax,#21H         ;set 1st argument(i)
        call    !_CSUB          ;call “CSUB(i, j)”
        pop     ax              ;
        ret
        END
```

① 引数の積み込み

引数があれば引数をスタックに積みます。引数の受け渡しは，次の図12－3のようになります。

図12－3　スタックへの引数の積み込み



② ワーク・レジスタ（AX, BC, DE）の退避

C言語では，‘ AX, BC, DEの3つのレジスタ・ペアを作業用として使用し，戻り時に値の復帰を行いません。このため，レジスタ内の値が必要な場合は，呼び出し側で退避します。レジスタの退避／復帰は，引数受け渡しコードの前後で行ってください。なお，HLレジスタについては，C言語側で使用している場合，常にC言語側で退避されます。

③ C言語関数のコール

C言語関数の呼び出しは，CALL命令で行います。C言語関数がcallt関数の場合，‘ callt ’命令で呼び出します。

④ スタック・ポインタ（SP）の復帰

引数を積んだバイト数分スタックを復帰します。

⑤返り値（BC, DE）の参照

C言語からの返り値は次のように返されます。

	BCレジスタ
返り値が16ビット以内：	ワード

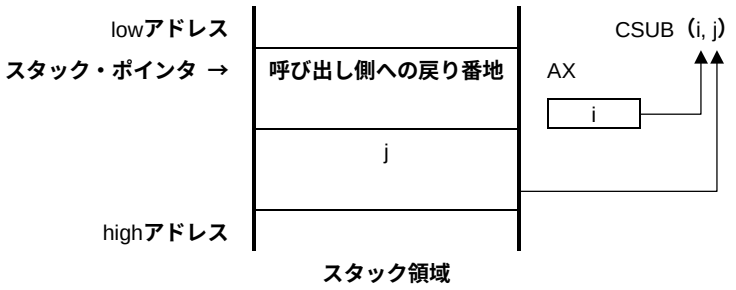
	DEレジスタ	BCレジスタ
返り値が17ビット以上：	上位ワード	下位ワード

(2) C言語関数の引数参照方法

次に示すC言語プログラムに引数‘ i, j ’を正しく渡すには，スタックに図12－4のように積みます。

```
void CSUB(i,j)
int     i,j ;
{
        i+=j ;
}
```

図12－4　C言語への引数の受け渡し



12. 5　他言語で定義された変数の参照

(1) C言語で定義した変数を参照する方法

C言語プログラム中で定義した外部変数をアセンブリ言語ルーチン中で参照する場合，extrn宣言します。アセンブリ言語ルーチン中では，定義した変数の先頭に‘ _ ’（アンダスコア）を付けます。

C言語のプログラム例

```
extern void subf();

char     c=0 ;
int      i=0 ;
void main()
{
         subf() ;
}
```

RA78K0S アセンブラでは次のように行います。

```
$PROCESSOR(9024)

      PUBLIC   _subf
      EXTRN    _c
      EXTRN    _i

@@CODE  CSEG
_subf:
      MOV      a, #04H
      MOV      !_c, a
      MOVW     ax,#07H        ;7
      MOVW     de, #_i
      INCW     DE
      MOV      [DE], A
      DECW     DE
      XCH      A, X
      MOV      [DE], A
      RET
      END
```

(2) アセンブリ言語で定義した変数をC言語側で参照する方法

アセンブリ言語で定義した変数をC言語側で参照するには，次のように行います。

C言語のプログラム例

```
extern char c ;
extern int  i ;

void subf()
{
        c='A' ;
        i=4 ;
}
```

RA78K0Sアセンブラでは次のように行います。

```
        NAME  ASMSUB

        PUBLIC        _c
        PUBLIC        _i

ABC     CSEG
_c:     DB    0
_i:     DW    0

        END
```

12. 6　その他注意事項

(1)‘ _ ’（アンダスコア）

本Cコンパイラは，出力するオブジェクト・モジュールの外部定義および参照名に‘ _ ’（アンダスコア，ASCIIコード‘ 5FH ’）を付けます。次のCプログラム例で，“j = FUNC（i, l）；**”は，“**_FUNC**という外部名を参照する”と訳されます。**

```
extern int FUNC(int,long);          /* 関数プロトタイプ */

void main()
{
        int       i, j;
        long      l;

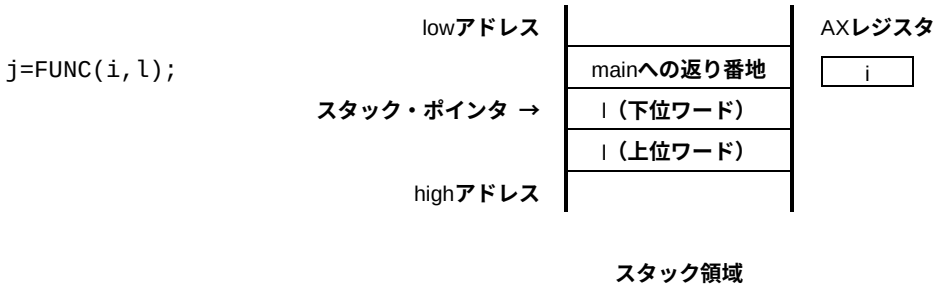
        i=1;
        l=0x54321;
        j=FUNC(i,l);                /* 関数コール */
}
```

RA78K0Sでは，ルーチン名を‘ _FUNC ’と記述します。

(2) スタックに積む引数の配置

スタックに積まれる引数は，後位置引数から前位置引数へとhighアドレスから**low**アドレス方向に積まれます。

図12－5　**引数のスタック配置**



〔メ　モ〕

第13章　効率の良いコンパイラの活用法

本章では，本Cコンパイラを有効に利用する方法を紹介します。

13. 1　効率の良いコーディング

78K/0Sシリーズ応用製品の開発を行う場合，本Cコンパイラではデバイスのsaddr領域，callt領域を利用することにより，効率の良いオブジェクトを生成できます。

- 外部変数を使用する
  - if（saddr領域が使用可能） —— sreg/__sreg変数を使用する／コンパイラ・オプション（-RD）を使用する
- 1ビットのデータを使用する
  - if（saddr領域が使用可能） —— bit/boolean/__boolean型変数を使用する
- 関数の定義
  - if（何回も呼ばれる関数）
    - if（callt領域が使用可能）
      - __callt/callt関数とする（コード・サイズ削減に有効）
    - if（再帰的に使用しない）
      - __leaf/norec関数とする
  - if（オートマティック変数を使用しない）
    - noauto関数とする
  - if（オートマティック変数を使用する&& saddr領域が使用可能）
    - register宣言する

(1) 外部変数の使用

外部変数を定義する時にsaddr領域が利用可能であれば，定義する外部変数をsreg/_ _sreg変数にします。

sreg/_ _sreg変数は，メモリに対する命令と比べ命令コードが短く，オブジェクト・コードを縮小でき，実行速度も向上します（sreg変数にするかわりにオプション -RDによっても同様のことができます）。

```
sreg/_ _sreg変数の定義：  extern sreg int 変数名;
                          extern _ _sreg int 変数名;
```

備考　11. 5（3）saddr**領域利用**を参照してください。

(2) 1ビット・データの使用

1ビットのデータしか使用しないオブジェクトは，bit型変数（または boolean/_ _boolean型変数） にします。bit/boolean/_ _boolean型変数に対する操作には，ビット操作命令が生成されます。また，sreg変数と同様，saddr領域を使用しますので，コードが縮小でき，実行速度も向上します。

```
bit/boolean型変数の宣言：  bit 変数名;
                           boolean 変数名;
                           _ _boolean  変数名;
```

備考　11. 5（7）bit**型変数**を参照してください。

(3) 関数定義の工夫

何回も呼ばれる関数は，オブジェクト・コードを短縮するか，高速に呼び出せる構造にする必要があります。したがって，何回も呼ばれる関数で，callt領域を利用できる場合はcallt関数にします。callt関数は，デバイスのcallt領域を利用して呼び出されるので通常の呼び出しよりも速く，かつ短いコードで呼び出せます。

```
callt関数の定義：  callt int tsub() {
                              ⋮
                   }
```

備考　11. 5（1）callt**関数**，11. 5（6）norec**関数**を参照してください。

saddr領域の使用に加え，最適化オプションを使用してコンパイルすることにより，Cソースの修正を行わずに良いオブジェクトを生成することができます。なお，各 -Qサブオプションの効果については，CC78K0S C**コンパイラ ユーザーズ・マニュアル 操作編**（U14871J）を参照してください。

(4) 最適化オプション

オブジェクト・コード・サイズを重視した最適化オプションは次のとおりです。

【オブジェクト・コード優先】

```
-QX3
```

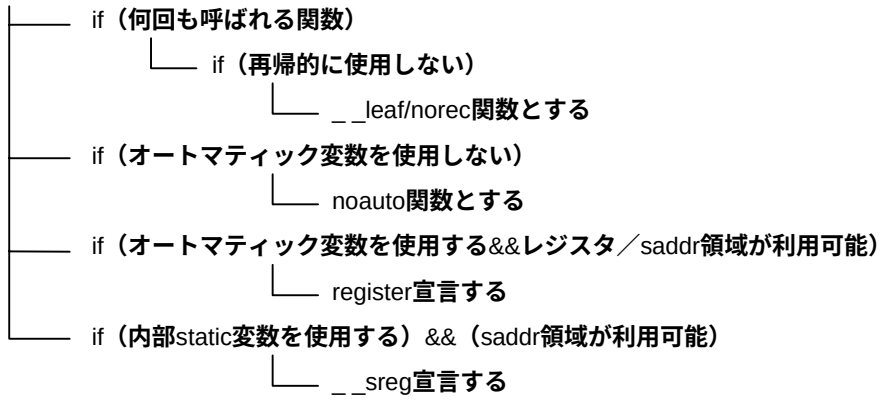
__sregを変数に付加することで，さらに，コード・サイズの縮小，実行スピード向上が見込めます。ただし，saddr領域が使用できる場合に限ります。領域が不足し使用できなくなった場合は，コンパイル・エラーとなります。

実行スピードも重視する場合は-QX2ディフォールトを指定してください。

さらに，オブジェクト効率を向上させたい場合は，Cソースに本コンパイラがサポートしている拡張機能を加えてください。

(5) 拡張機能の使用

・関数の定義



・再帰的に使用しない関数

何回も呼ばれる関数の中で再帰的に使用しないものは，__leaf／norec関数にします。norec関数は，関数の前後処理（スタック・フレーム）のない関数になります。このため，通常の関数に比べオブジェクト・コードが短縮でき，実行速度も向上します。

備考　norec関数の定義（norec int rout () ...）については，11. 5（6）norec関数，11. 7. 4　norec関数呼び出しインタフェースを参照してください。

・オートマティック変数を使用しない関数

オートマティック変数を使用しない関数は，noauto 関数にします。noauto 関数は，スタック・フレームのない関数です。また，引数も可能なかぎりレジスタ渡しとなります。オブジェクト・コードの短縮ができ，実行速度が向上します。

備考　noauto関数の定義（noauto int sub1 (int i) ...）については，11. 5（5）noauto関数，11. 7. 3 noauto関数呼び出しインタフェースを参照してください。

・オートマティック変数を使用する関数

オートマティック変数を使用する関数で，saddr領域が使用可能であればregister宣言します。register宣言は，宣言されたオブジェクトをレジスタに割り当てます。レジスタを用いたプログラムは，メモリを使ったプログラムと比べ高速に動作し，またオブジェクト・コードも短縮されます。

備考　register変数の定義（register int i; ...）については，11. 5（2）レジスタ変数を参照してください。

・内部static変数を使用する関数

内部static変数を使用する関数で，saddr領域が使用可能であれば，_ _sreg宣言，または，-RSオプションを指定します。sreg変数と同様，オブジェクト・コードを縮小でき，実行速度も向上します。

備考　11. 5（3）saddr領域利用を参照してください。

その他，次のような方法で，コード効率または実行スピードを向上できます。

- SFR名（またはSFRビット名称）の使用

```
#pragma sfr
```

- 1ビットのメンバのみからなるビット・フィールドには，_ _sreg宣言を使用（メンバにはunsigned char型も使用可能）

```
_ _sreg  struct  bf {
         unsigned char a:1;
         unsigned char b:1;
         unsigned char c:1;
         unsigned char d:1;
         unsigned char e:1;
         unsigned char f:1;
 } bf_1;
```

- 乗算，除算組み込み関数の使用

```
#pragma mul
#pragma div
```

- 高速化したいモジュールのみ，アセンブリ言語で記述

〔メ　モ〕

付録A　saddr領域のレーベル一覧

CC78K0Sでは，次に示すレーベル名によってsaddr領域を参照しています。したがって，Cソース・プログラム，またはアセンブラ・ソース・プログラム中で次のレーベルと同じ名前を使用できません。

A. 1　ノーマル・モデル

(a) レジスタ変数

レーベル名	アドレス
_@KREG00	0FED8H
_@KREG01	0FED9H
_@KREG02	0FEDAH
_@KREG03	0FEDBH
_@KREG04	0FEDCH
_@KREG05	0FEDDH
_@KREG06	0FEDEH
_@KREG07	0FEDFH
_@KREG08	0FEE0H
_@KREG09	0FEE1H
_@KREG10	0FEE2H
_@KREG11	0FEE3H
_@KREG12	0FEE4H注
_@KREG13	0FEE5H注
_@KREG14	0FEE6H注
_@KREG15	0FEE7H注

注　関数の引数がregister宣言，または-QVオプションが指定され，かつ-QRオプションが指定されている場合に，引数をsaddr領域に割り当てます。

(b) norec関数の引数

レーベル名	アドレス
_@NRARG0	0FEE8H
_@NRARG1	0FEEAH
_@NRARG2	0FEECH
_@NRARG3	0FEEEH

(c) norec関数のオートマティック変数

レーベル名	アドレス
_@NRAT00	0FEF0H
_@NRAT01	0FEF1H
_@NRAT02	0FEF2H
_@NRAT03	0FEF3H
_@NRAT04	0FEF4H
_@NRAT05	0FEF5H
_@NRAT06	0FEF6H
_@NRAT07	0FEF7H

(d) ランタイム・ライブラリの引数

レーベル名	アドレス
_@RTARG0	0FEF8H
_@RTARG1	0FEF9H
_@RTARG2	0FEFAH
_@RTARG3	0FEFBH
_@RTARG4	0FEFCH
_@RTARG5	0FEFDH
_@RTARG6	0FEFEH
_@RTARG7	0FEFFH

A. 2 スタティック・モデル

(a) 共有領域

レーベル名	アドレス
_@KREG00	0FEF0H
_@KREG01	0FEF1H
_@KREG02	0FEF2H
_@KREG03	0FEF3H
_@KREG04	0FEF4H
_@KREG05	0FEF5H
_@KREG06	0FEF6H
_@KREG07	0FEF7H
_@KREG08	0FEF8H
_@KREG09	0FEF9H
_@KREG10	0FEFAH
_@KREG11	0FEFBH
_@KREG12	0FEFCH
_@KREG13	0FEFDH
_@KREG14	0FEFEH
_@KREG15	0FEFFH

(b) 引数，オートマティック変数，ワーク用

レーベル名	アドレス
_@NRAT00	0FExxH[注]
_@NRAT01	_@NRAT00＋1
_@NRAT02	_@NRAT00＋2
_@NRAT03	_@NRAT00＋3
_@NRAT04	_@NRAT00＋4
_@NRAT05	_@NRAT00＋5
_@NRAT06	_@NRAT00＋6
_@NRAT07	_@NRAT00＋7

注　saddr領域内の任意のアドレス

〔メ　モ〕

付録B　セグメント名一覧

コンパイラが出力する全セグメントと配置について説明をします。
なお，表に使用されているオプション，再配置属性は次の①，②のとおりです。

コンパイラが出力する全セグメントの説明をします。

① CSEGの再配置属性

CALLT0	：指定セグメントを40H-7FH番地で先頭が2の倍数になるように配置します。
AT　絶対式	：指定セグメントを絶対番地に配置します（0000H-FEFFH内）。
FIXED	：指定セグメントの先頭を800H-0FFFHに配置することを指定します。
UNITP	：指定セグメントを任意の位置へ先頭が2の倍数になるように配置します（80H-0FA7EH内）。

② DSEGの再配置属性

SADDRP	：指定セグメントをsaddr領域内のFE20H-FEFFHに先頭が2の倍数になるように配置します。
UNITP	：指定セグメントを任意の位置へ先頭が2の倍数になるように配置します（ディフォールトはRAM領域内）。

B. 1　セグメント名一覧

セクション名	セグメント・タイプ	再配置属性	説　　明
@@CODE	CSEG		コード部用セグメント
@@CNST	CSEG		const変数用セグメント
@@R_INIT	CSEG		初期化データ用セグメント（初期値あり）
@@R_INIS	CSEG	UNITP	初期化データ用セグメント（初期値ありsreg変数）
@@CALT	CSEG	CALLT0	callt関数のテーブル用セグメント
@@VECTnn	CSEG	AT 00nnH	ベクタ・テーブル用セグメント[注]
@@INIT	DSEG		データ領域用セグメント（初期値あり）
@@DATA	DSEG		データ領域用セグメント（初期値なし）
@@INIS	DSEG	SADDRP	データ領域用セグメント（初期値ありsreg変数）
@@DATS	DSEG	SADDRP	データ領域用セグメント（初期値なしsreg変数）
@@BITS	BSEG		boolean型変数，bit型変数用セグメント

注　割り込みの種類により，nnの値が変わります。

B. 2　セグメントの配置

セグメント・タイプ	配置先（ディフォールト）
CSEG	ROM
BSEG	RAMのsaddr領域
DSEG	RAM

B. 3　Cソース例

```
#pragma INTERRUPT INTP0 inter          /*割り込みベクタ       */

void inter(void);                      /*割り込み関数プロトタイプ宣言*/
const int i_cnst = 1;                  /*const変数            */
callt void f_clt(void);                /*callt関数プロトタイプ宣言*/
boolean b_bit;                         /*boolean型変数        */
long l_init = 2;                       /*初期値あり外部変数   */
int i_data;                            /*初期値なし外部変数   */
sreg int sr_inis = 3;                  /*初期値ありsreg変数   */
sreg int sr_dats;                      /*初期値なしsreg変数   */

void main()                            /*関数定義             */
{
     int i;
     i = 100;
}

void inter()                           /*割り込み関数定義     */
{
     unsigned char  uc = 0;
     uc++;
     if (b_bit)
     b_bit = 0;
}

callt void f_clt()                     /*callt関数定義        */
{
}
```

B. 4　出力アセンブラ・モジュール例

アセンブラ・ソース中の疑似命令，命令セットは，各デバイスにより異なります。
詳細は，RA78K0Sのオンライン・ヘルプを参照してください。

```
;78K/0S Series C Compiler V1.30 Assembler Source
;                                       Date:xx xxx xxxx Time:xx:xx:xx

;Command  : -c9026 sampk0s.c -sa -ng
;In-file  : sampk0s.c
;Asm-file : sampk0s.asm
;Para-file:

$PROCESSOR(9026)
$NODEBUG
$NODEBUGA
$KANJICODE SJIS
$TOL_INF          03FH, 0130H, 00H, 00H

        EXTRN     _@cprep
        PUBLIC    _inter
        PUBLIC    ?f_clt
        PUBLIC    _i_cnst
        PUBLIC    _b_bit
        PUBLIC    _l_init
        PUBLIC    _i_data
        PUBLIC    _sr_inis
        PUBLIC    _sr_dats
        PUBLIC    _main
        PUBLIC    _f_clt
        PUBLIC    _@vect06

@@BITS  BSEG                            ;boolean型変数用セグメント
_b_bit  DBIT

@@CNST  CSEG                            ;const変数用セグメント
_i_cnst:          DW     01H       ; 1

@@R_INIT          CSEG                  ;初期化データ用セグメント（初期値あり外部変数）
        DW        00002H,00000H   ; 2

@@INIT  DSEG                            ;データ領域用セグメント（初期値あり外部変数）
_l_init:          DS     (4)
```

```
@@DATA  DSEG                            ;データ領域用セグメント（初期値なし外部変数）
_i_data:        DS      (2)

@@R_INIS        CSEG    UNITP           ;初期化データ用セグメント（初期値ありsreg変数）
        DW      03H     ; 3

@@INIS  DSEG    SADDRP                  ;データ領域用セグメント（初期値ありsreg変数）
_sr_inis:       DS      (2)

@@DATS  DSEG    SADDRP                  ;データ領域用セグメント（初期値なしsreg変数）
_sr_dats:       DS      (2)

@@CALT  CSEG    CALLT0                  ;callt関数用セグメント
?f_clt: DW      _f_clt

;line   1 :  #pragma INTERRUPT INTP0 inter      /*割り込みベクタ*/
;line   2 :
;line   3 :  void inter(void);                  /*割り込み関数プロトタイプ宣言*/
;line   4 :  const int i_cnst=1;                /*const変数*/
;line   5 :  callt void f_clt(void);            /*callt関数プロトタイプ宣言*/
;line   6 :  boolean b_bit;                     /*boolean型変数*/
;line   7 :  long l_init=2;                     /*初期値あり外部変数*/
;line   8 :  int i_data;                        /*初期値なし外部変数*/
;line   9 :  sreg int sr_inis=3;                /*初期値ありsreg変数*/
;line   10:  sreg int sr_dats;                  /*初期値なしsreg変数*/
;line   11:
;line   12:  void main()                        /*関数定義*/
;line   13:  {

@@CODE  CSEG                            ;コード部用セグメント
_main:
       push hl                                  ;[INF] 1, 4
       movw ax,#02H                             ;[INF] 3, 6
       callt[_@cprep]                           ;[INF] 1, 8
;line   14:  int  i;
;line   15:  i=100;
       movw ax,#064H        ;100                ;[INF] 3, 6
       mov  [hl+1],a        ;i                  ;[INF] 2, 6
       xch  a,x                                 ;[INF] 1, 4
       mov  [hl],a  ; i                         ;[INF] 1, 6
; line  16:  }
       pop  ax                                  ;[INF] 1, 6
       pop  hl                                  ;[INF] 1, 6
```

```
        ret                                             ;[INF] 1, 6
;line   17:
;line   18: void inter()                                /*割り込み関数定義*/
;line   19:  {
_inter:
        push ax                                         ;[INF] 1, 4
        push de                                         ;[INF] 1, 4
        push hl                                         ;[INF] 1, 4
        movw ax,#02H                                    ;[INF] 3, 6
        callt[_@cprep]                                  ;[INF] 1, 8
;line   20:  unsigned char uc=0;
        xor  a,a                                        ;[INF] 2, 4
        mov  [hl+1],a          ; uc                     ;[INF] 2, 6
;line   21:  uc++;
        inc  a                                          ;[INF] 2, 4
        xch  a,[hl+1]          ; uc                     ;[INF] 2, 8
;line   22:  if(b_bit)
        bf   _b_bit,$L0005                              ;[INF] 4,10
;line   23:  b_bit=0;
        clr1 _b_bit                                     ;[INF] 3, 6
L0005:
;line   24:  }
        pop  ax                                         ;[INF] 1, 6
        pop  hl                                         ;[INF] 1, 6
        pop  de                                         ;[INF] 1, 6
        pop  ax                                         ;[INF] 1, 6
        reti                                            ;[INF] 1, 8
;line   25:
;line   26:  callt void f_clt()                         /*callt関数定義*/
;line   27: {
_f_clt:
;line   28:  }
        ret                                             ;[INF] 1, 6

@@VECT06    CSEG    AT    0006H       ;割り込みベクタ
_@vect06:
        DW   _inter
        END
```

```
; *** Code Information ***
;
; $FILE C:¥NECTools32¥work¥sampk0s.c
;
; $FUNC main(13)
;       void=(void)
;       CODE SIZE= 15 bytes, CLOCK_SIZE= 58 clocks, STACK_SIZE= 6 bytes
;
; $FUNC inter(19)
;       void=(void)
;       CODE SIZE= 27 bytes, CLOCK_SIZE= 96 clocks, STACK_SIZE= 10 bytes
;
; $FUNC f_clt(27)
;       void=(void)
;       CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes

; Target chip : uPD789026
; Device file : Vx.xx
```

〔メ　モ〕

付録C　ランタイム・ライブラリー覧

表C－1にランタイム・ライブラリの一覧を示します。

これらの演算の命令は，@@などを関数名の頭につけた形式で呼び出されます。

ただし，cstart, cprep, cdispは，先頭に_@を付加した形式で呼び出されます。

なお，表C－1にない演算については，ライブラリのサポートはありません。コンパイラがインライン展開を行います。

longの加減算，and/or/xor，シフトはインライン展開される場合があります。

表C－1　ランタイム・ライブラリー覧（1/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
インクリメント	lsinc	○	－	signed longをインクリメントする
	luinc	○	－	unsigned longをインクリメントする
	finc	○	－	floatをインクリメントする
デクリメント	lsdec	○	－	signed longをデクリメントする
	ludec	○	－	unsigned longをデクリメントする
	fdec	○	－	floatをデクリメントする
符号反転	lsrev	○	－	signed longを符号反転する
	lurev	○	－	unsigned longを符号反転する
	frev	○	－	floatを符号反転する
1の補数	lscom	○	－	signed longの1の補数を求める
	lucom	○	－	unsigned longの1の補数を求める
論理否定	lsnot	○	－	signed longの否定を求める
	lunot	○	－	unsigned longの否定を求める
	fnot	○	－	floatの否定を求める
乗算	csmul	○	○	signed char同士の乗算
	cumul	○	○	unsigned char同士の乗算
	ismul	○	○	signed int同士の乗算
	iumul	○	○	unsigned int同士の乗算
	lsmul	○	－	signed long同士の乗算
	lumul	○	－	unsigned long同士の乗算
	fmul	○	－	float同士の乗算
除算	csdiv	○	○	signed char同士の除算
	cudiv	○	○	unsigned char同士の除算
	isdiv	○	○	signed int同士の除算
	iudiv	○	○	unsigned int同士の除算
	lsdiv	○	－	signed long同士の除算
	ludiv	○	－	unsigned long同士の除算
	fdiv	○	－	float同士の除算

表C－1　ランタイム・ライブラリー覧（2/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
剰余算	csrem	○	○	signed char同士の剰余算
	curem	○	○	unsigned char同士の剰余算
	isrem	○	○	signed int同士の剰余算
	iurem	○	○	unsigned int同士の剰余算
	lsrem	○	－	signed long同士の剰余算
	lurem	○	－	unignedint同士の剰余算
加算	lsadd	○	－	signed long同士の加算
	luadd	○	－	unsigned long同士の加算
	fadd	○	－	float同士の加算
減算	lssub	○	－	signed long同士の減算
	lusub	○	－	unsigned long同士の減算
	fsub	○	－	float同士の減算
左シフト	islsh	○	○	signed intの左シフト
	iulsh	○	○	unsigned intの左シフト
	lslsh	○	－	signed longの左シフト
	lulsh	○	－	unsigned longの左シフト
右シフト	islsh	○	○	signed intの右シフト
	iulsh	○	○	unsigned intの右シフト
	lsrsh	○	－	signed longの右シフト
	lursh	○	－	unsigned longの右シフト
比較	cscmp	○	○	signed char同士の比較
	iscmp	○	○	signed int同士の比較
	lscmp	○	－	signed long同士の比較
	lucmp	○	－	unsigned long同士の比較
	fcmp	○	－	float同士の比較
ビットAND	lsband	○	－	signed long同士のAND
	luband	○	－	unsigned long同士のAND
ビットOR	lsbor	○	－	signed long同士のOR
	lubor	○	－	unsigned long同士のOR
ビットXOR	lsbxor	○	－	signed long同士のXOR
	lubxor	○	－	unsigned long同士のXOR
論理AND	fand	○	－	float同士の論理AND
論理OR	for	○	－	float同士の論理OR
浮動小数点数からの変換	ftols	○	－	floatからsigned longに変換する
	ftolu	○	－	floatからunsigned longに変換する
浮動小数点への変換	lstof	○	－	signed longからfloatに変換する
	lutof	○	－	unsigned longからfloatに変換する
bitからの変換	btol	○	－	bitをlongに変換する

表C－1　ランタイム・ライブラリー覧（3/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
スタートアップ・ルーチン	cstart	○	○	スタートアップ・モジュール ・atexit関数で関数を登録する領域（2×32バイト）を確保し，先頭のレーベル名を_@FNCTBLとする。 ・ブレーク領域（32バイト）を確保し，先頭のレーベル名を_@MEMTOPとし，領域の次のアドレスのレーベル名を_@MEMBTMとする。 ・リセット・ベクタ・テーブルのセグメントを次のように定義し，スタートアップ・モジュールの先頭アドレスを指定する。 @@VECT00　CSEG　AT　0000H DW　_@cstart ・エラー番号を入れる変数_errnoに0を設定する。 ・atexit関数で登録した関数の数を入れる変数_@FNCENTに0を設定する。 ・ブレーク値の初期値として，_@MEMTOPのアドレスを変数_@BRKADRに設定する。 ・rand関数の疑似乱数の発生元となる変数_@SEEDに初期値1を設定する。 ・初期化データのコピー処理，および初期値なし外部データの0クリアを行う。 ・main関数（ユーザ・プログラム）を呼び出す。 ・exit関数をパラメータ0で呼び出す
関数前後処理	cprep	○	－	関数の前処理
	cdisp	○	－	関数の後処理
	cprep2	○	－	関数の前処理（レジスタ変数用saddr領域を含む）
	cdisp2	○	－	関数の後処理（レジスタ変数用saddr領域を含む）
	nrcp2	－	○	引数コピー用
	nrcp3	－	○	
	krcp2	－	○	
	krcp3	－	○	
	nkrc3	－	○	
	nrip2	－	○	
	nrip3	－	○	
	krip2	－	○	
	krip3	－	○	
	nkri31	－	○	
	nkri32	－	○	
	nrsave	－	○	_@NRATxx退避用
	nrload	－	○	_@NRATxx復帰用

表C－1　ランタイム・ライブラリー覧（4/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
関数前後処理	krs02	－	○	_@KREGxx退避用
	krs04	－	○	
	krs04i	－	○	
	krs06	－	○	
	krs06i	－	○	
	krs08	－	○	
	krs08i	－	○	
	krs10	－	○	
	krs10i	－	○	
	krs12	－	○	
	krs12i	－	○	
	krs14	－	○	
	krs14i	－	○	
	krs16	－	○	
	krs16i	－	○	
	krl02	－	○	_@KREGxx復帰用
	krl04	－	○	
	krl04i	－	○	
	krl06	－	○	
	krl06i	－	○	
	krl08	－	○	
	krl08i	－	○	
	krl10	－	○	
	krl10i	－	○	
	krl12	－	○	
	krl12i	－	○	
	krl14	－	○	
	krl14i	－	○	
	krl16	－	○	
	krl16i	－	○	
	hdwinit	○	○	CPUリセット直後に周辺装置（sfr）の初期化処理を行う
BCD型変換	bcdtob	○	○	1バイトbcdを1バイトbinaryに変換する
	btobcd	○	○	1バイトbinaryを2バイトbcdに変換する
	bcdtow	○	○	2バイトbcdを2バイトbinaryに変換する
	wtobcd	○	○	2バイトbinaryを2バイトbcdに変換する
	bbcd	○	○	1バイトbinaryを1バイトbcdに変換する
補助	mulu	○	○	K0mulu命令互換
	divuw	○	○	K0divuw命令互換

表C－1　ランタイム・ライブラリー覧（5/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
補助	clra0	○	○	定型命令パターン置き換え用
	clra1	○	○	
	clrax0	○	○	
	clrax1	○	○	
	clrbc0	○	○	
	clrbc1	○	○	
	cmpa0	○	○	
	cmpa1	○	○	
	cmpc0	○	○	
	cmpax0	○	○	
	cmpax1	○	○	
	movca	○	○	
	movac	○	○	
	ctoi	○	○	
	uctoi	○	○	
	adjba	○	○	
	adjbs	○	○	
	addrde	○	○	
	addrhl	○	○	
	shl4	○	○	
	shr4	○	○	
	tabled	○	○	
	tableh	○	○	
	apdecd	○	○	
	apdech	○	○	
	apincd	○	○	
	apinch	○	○	
	deilo	○	○	
	deist	○	○	
	deiinc	○	○	
	deidec	○	○	
	hlilo	○	○	
	hlist	○	○	
	hliinc	○	○	
	hlidec	○	○	
	dellab	○	－	
	dell03	○	－	
	della4	○	－	
	delsab	○	－	
	dels03	○	－	
	hlllab	○	－	

表C－1　ランタイム・ライブラリー覧（6/6）

分　類	関数名	サポートされるモデル		機　能
		ノーマル・モデル	スタティック・モデル	
補助	hlll03	○	－	定型命令パターン置き換え用
	hllla4	○	－	
	hllsab	○	－	
	hlls03	○	－	
	hliadd	○	○	
	hlisub	○	○	
	hlicmp	○	○	
	hliand	○	○	
	hlior	○	○	
	hlixor	○	○	
	imule	○	○	
	isdive	○	○	
	iudive	○	○	
	isreme	○	○	
	iureme	○	○	
	iadde	○	○	
	isube	○	○	
	iande	○	○	
	iore	○	○	
	ixore	○	○	

付録D　ライブラリ消費スタック一覧

表D－1に標準ライブラリのスタック消費量一覧を示します。

表D－1　標準ライブラリのスタック消費量一覧（1/4）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
ctype.h	isalnum	0	0
	isalpha	0	0
	iscntrl	0	0
	isdigit	0	0
	isgraph	0	0
	islower	0	0
	isprint	0	0
	ispunct	0	0
	isspace	0	0
	isupper	0	0
	isxdigit	0	0
	tolower	0	0
	toupper	0	0
	isascii	0	0
	toascii	0	0
	_tolower	0	0
	_toupper	0	0
	tolow	0	0
	toup	0	0
setjmp.h	setjmp	4	4
	longimp	2	2
stdarg.h	va_arg	0	－
	va_start	0	－
	va_end	0	－
stdio.h	sprintf	52（72）注1	－
	sscanf	290（304）注1	－
	printf	54（72）注1	－
	scanf	294（304）注1	－
	vprintf	52（72）注1	－
	vsprintf	52（72）注1	－
	getchar	0	0
	gets	6	6
	putchar	0	0
	puts	4	4

表D－1　標準ライブラリのスタック消費量一覧（2/4）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
stdlib.h	atoi	4	4
	atol	10	－
	strtol	20	－
	strtoul	20	－
	calloc	14	14
	free	8	8
	malloc	6	6
	realloc	12	12
	abort	0	0
	atexit	0	0
	exit	2＋n[注2]	2＋n[注2]
	abs	0	0
	div	6	－
	labs	2	－
	ldiv	16	－
	brk	0	0
	sbrk	4	4
	atof	33	－
	strtod	33	－
	itoa	10	10
	ltoa	16	－
	ultoa	16	－
	rand	16	－
	srand	0	－
	bsearch	32＋n[注3]	－
	qsort	16＋n[注4]	－
	strbrk	0	0
	strsbrk	4	4
	stritoa	10	10
	strltoa	16	－
	strultoa	16	－
string.h	memcpy	4	6
	memmove	4	8
	strcpy	2	4
	strncpy	4	6
	strcat	2	4
	strncat	4	6
	memcmp	2	4
	strcmp	2	2
	strncmp	2	4
	memchr	2	2
	strchr	2	0
	strcspn	6	6
	strpbrk	4	4

表D－1　標準ライブラリのスタック消費量一覧（3/4）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
string.h	strrchr	4	4
	strspn	6	6
	strstr	4	4
	strtok	4	4
	memset	4	4
	strerror	0	0
	strlen	0	0
	strcoll	2	2
	strxfrm	4	4
math.h	acos	24	－
	asin	24	－
	atan	20	－
	atan2	21	－
	cos	24（34）注5	－
	sin	24（34）注5	－
	tan	26（34）注5	－
	cosh	24	－
	sinh	25	－
	tanh	30	－
	exp	22	－
	frexp	2（10）注5	－
	ldexp	2（10）注5	－
	log	24（34）注5	－
	log10	24（34）注5	－
	modf	2（10）注5	－
	pow	25（35）注5	－
	sqrt	18	－
	ceil	2	－
	fabs	0	－
	floor	2	－
	frmod	2（10）注5	－
	matherr	0	－
	acosf	24	－
	asinf	24	－
	atanf	20	－
	atan2f	21	－
	cosf	24（34）注5	－
	sinf	24（34）注5	－
	tanf	26（34）注5	－
	coshf	24	－
	sinhf	25	－
	tanhf	30	－
	expf	22	－
	frexpf	2（10）注5	－

表D－1　標準ライブラリのスタック消費量一覧（4/4）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
math.h	ldexpf	2（10）注5	－
	logf	24（34）注5	－
	log10f	24（34）注5	－
	modff	2（10）注5	－
	powf	25（35）注5	－
	sqrtf	18	－
	ceilf	2	－
	fabsf	0	－
	floorf	2	－
	fmodf	2（10）注5	－
assert.h	_ _assertfail	66（84）注6	－

注1.　（　）内は浮動小数点対応版使用時の値

2. nはatexit関数で登録された外部関数中の最大スタック消費量
3. nはbsearchから呼び出される外部関数のスタック消費量
4. nは（20＋qsortから呼び出される外部関数のスタック消費量）×（1＋再帰呼び出しの発生回数）
5. （　）内は演算例外発生時
6. （　）内は浮動小数点数対応版printf使用時

表D－2にランタイム・ライブラリのスタック消費量一覧を示します。

表D－2　ランタイム・ライブラリのスタック消費量一覧（1/5）

分　類	関数名	ノーマル・モデル	スタティック・モデル
インクリメント	lsinc	0	－
	luinc	0	－
	finc	12（22）注1	－
デクリメント	lsdec	0	－
	ludec	0	－
	fdec	12（22）注1	－
符号反転	lsrev	0	－
	lurev	0	－
	frev	0	－
1の補数	lscom	0	－
	lucom	0	－
論理否定	lsnot	0	－
	lunot	0	－
	fnot	0	－
乗算	csmul	4	4
	cumul	4	4
	ismul	6	6
	iumul	6	6
	lsmul	6	－
	lumul	6	－
	fmul	8（18）注1	－
除算	csdiv	8	8
	cudiv	4	4
	isdiv	8	12
	iudiv	4	6
	lsdiv	10	－
	ludiv	6	－
	fdiv	8（18）注1	－
剰余算	csrem	8	8
	curem	4	4
	isrem	8	12
	iurem	4	6
	lsrem	10	－
	lurem	6	－
加算	lsadd	0	－
	luadd	0	－
	fadd	8（18）注1	－
減算	lssub	0	－
	lusub	0	－
	fsub	8（18）注1	－

表D－2　ランタイム・ライブラリluaddのスタック消費量一覧（2/5）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
左シフト	islsh	0	0
	iulsh	0	0
	lslsh	2	－
	lulsh	2	－
右シフト	isrsh	0	0
	iursh	0	0
	lsrsh	2	－
	lursh	2	－
比較	cscmp	0	2
	iscmp	0	2
	lscmp	2	－
	lucmp	2	－
	fcmp	4（14）注1	－
ビットAND	lsband	0	－
	luband	0	－
ビットOR	lsbor	0	－
	lubor	0	－
ビットXOR	lsbxor	0	－
	lubxor	0	－
論理AND	fand	0	－
論理OR	for	0	－
浮動小数点数からの変換	ftols	4	－
	ftolu	4	－
浮動小数点数への変換	lstof	12（22）注1	－
	lutof	12（22）注1	－
bitからの変換	btol	0	－
スタートアップ・ルーチン	cstart	2	2
関数前後処理	cprep	2＋n注2	－
	cdisp	0	－
	cprep2	自動変数＋レジスタ変数のサイズ	－
	cdisp2	0	－
	nrcp2	－	0
	nrcp3	－	0
	krcp2	－	0
	krcp3	－	0
	nkrc3	－	0
	nrip2	－	0
	nrip3	－	0
	krip2	－	0
	krip3	－	0
	nkri31	－	0
	nkri32	－	0
	nrsave	－	8
	nrload	－	0

表D－2　ランタイム・ライブラリluaddのスタック消費量一覧（3/5）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
関数前後処理	krs02	－	2
	krs04	－	4
	krs04i	－	4
	krs06	－	6
	krs06i	－	6
	krs08	－	8
	krs08i	－	8
	krs10	－	10
	krs10i	－	10
	krs12	－	12
	krs12i	－	12
	krs14	－	14
	krs14i	－	14
	krs16	－	16
	krs16i	－	16
	krl02	－	0
	krl04	－	0
	krl04i	－	0
	krl06	－	0
	krl06i	－	0
	krl08	－	0
	krl08i	－	0
	krl10	－	0
	krl10i	－	0
	krl12	－	0
	krl12i	－	0
	krl14	－	0
	krl14i	－	0
	krl16	－	0
	krl16i	－	0
	hdwinit	0	0
BCD型変換	bcdtob	4	4
	btobcd	8	8
	bcdtow	4	4
	wtobcd	10	10
	bbcd	8	8
補助	mulu	4	4
	divuw	6	6
	clra0	0	0
	clra1	0	0
	clrax0	0	0
	clrax1	0	0

表D－2　ランタイム・ライブラリのスタック消費量一覧（4/5）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
補助	clrbc0	0	0
	clrbc1	0	0
	cmpa0	0	0
	cmpa1	0	0
	cmpc0	0	0
	cmpax0	0	0
	cmpax1	0	0
	movca	0	0
	movac	0	0
	ctoi	0	0
	uctoi	0	0
	adjba	2	2
	adjbs	1	1
	addrde	0	0
	addrhl	0	0
	shl4	0	0
	shr4	0	0
	tabled	0	0
	tableh	0	0
	apdecd	0	0
	apdech	0	0
	apincd	0	0
	apinch	0	0
	deilo	0	0
	deist	0	0
	deiinc	0	0
	deidec	0	0
	hlilo	0	0
	hlist	0	0
	hliinc	0	0
	hlidec	0	0
	dellab	2	－
	dell03	0	－
	della4	0	－
	delsab	0	－
	dels03	2	－
	hlllab	0	－
	hlll03	0	－
	hllla4	0	－
	hllsab	0	－
	hlls03	0	－
	hliadd	0	0
	hlisub	0	0

表D－2　ランタイム・ライブラリのスタック消費量一覧（5/5）

分　　類	関数名	ノーマル・モデル	スタティック・モデル
補助	hlicmp	0	0
	hliand	0	0
	hlior	0	0
	hlixor	0	0
	imule	10	10
	isdive	12	16
	iudive	8	10
	isreme	12	16
	iureme	8	10
	iadde	0	0
	isube	2	2
	iande	0	0
	iore	0	0
	ixore	0	0

注1.　（　）内は演算例外発生時（コンパイラ付属のmatherr関数を使用した場合）

2. nは確保するオートマティック変数のサイズ

〔メ　モ〕

付録E　索　　引

【アルファベットで始まる語句】

〔メ　モ〕

お問い合わせ先

【技術的なお問い合わせ先】

ＮＥＣ半導体テクニカルホットライン
（電話：午前 9:00～12:00，午後 1:00～5:00）

電　話　：044−435−9494
FAX　：044−435−9608
E-mail　：s-info@saed.tmg.nec.co.jp

【営業関係お問い合わせ先】

第一販売事業部
東　京　(03)3798−6106, 6107, 6108
名古屋　(052)222−2375
大　阪　(06)6945−3178, 3200, 3208, 3212
仙　台　(022)267−8740
郡　山　(024)923−5591
千　葉　(043)238−8116

第二販売事業部
東　京　(03)3798−6110, 6111, 6112
立　川　(042)526−5981, 6167
松　本　(0263)35−1662
静　岡　(054)254−4794
金　沢　(076)232−7303
松　山　(089)945−4149

第三販売事業部
東　京　(03)3798−6151, 6155, 6586, 1622, 1623, 6156
水　戸　(029)226−1702
広　島　(082)242−5504
高　崎　(027)326−1303
鳥　取　(0857)27−5313
太　田　(0276)46−4014
名古屋　(052)222−2170, 2190
福　岡　(092)261−2806

【資料の請求先】

上記営業関係お問い合わせ先またはNEC特約店へお申しつけください。

【インターネット電子デバイス・ニュース】

NECエレクトロンデバイスの情報がインターネットでご覧になれます。　URL（アドレス）　http://www.ic.nec.co.jp/

アンケート記入のお願い

お手数ですが，このドキュメントに対するご意見をお寄せください。今後のドキュメント作成の参考にさせていただきます。

［ドキュメント名］CC78K0S　Cコンパイラ　Ver.1.30以上　言語編　ユーザーズ・マニュアル
（U14872JJ1V0UM00（第1版））

［お名前など］（さしつかえのない範囲で）

御社名（学校名，その他）　（　　）
ご住所　（　　）
お電話番号　（　　）
お仕事の内容　（　　）
お名前　（　　）

1．ご評価（各欄に○をご記入ください）

項　　目	大変良い	良　い	普　通	悪　い	大変悪い
全体の構成					
説明内容					
用語解説					
調べやすさ					
デザイン，字の大きさなど					
その他（　　）					
（　　）					

2．わかりやすい所（第　章，第　章，第　章，第　章，その他　　）
理由［　　］

3．わかりにくい所（第　章，第　章，第　章，第　章，その他　　）
理由［　　］

4．ご意見，ご要望

5．このドキュメントをお届けしたのは
NEC販売員，特約店販売員，その他（　　）

ご協力ありがとうございました。
下記あてにFAXで送信いただくか，最寄りの販売員にコピーをお渡しください。

日本電気(株) NEC エレクトロンデバイス
半導体テクニカルホットライン
FAX：（044）435-9608

2000.6

キリトリ