User's Manual

CubeSuite+ V2.01.00

統合開発環境

ユーザーズマニュアル　RL78,78K0Rコーディング編

対象デバイス

RL78ファミリ

78K0Rマイクロコントローラ

ルネサス エレクトロニクス
www.renesas.com

Rev.1.00　2013.12

ご注意書き

1. 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・システムの設計において、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て、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。
2. 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。
3. 本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害に関し、当社は、何らの責任を負うものではありません。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。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
   各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   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、産業用ロボット等

   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、各種安全装置等

   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。 たとえ、意図し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損害が生じても、当社は一切その責任を負いません。 なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の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。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、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、社会的損害等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、お客様の機器・システムとしての出荷保証を行ってください。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、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の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。
9. 本資料に記載されている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・システムに使用することはできません。また、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に使用しないでください。当社製品または技術を輸出する場合は、「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。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
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禁じます。

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。

(2012.4)

このマニュアルの使い方

このマニュアルは，RL78 ファミリ，78K0R マイクロコントローラ用アプリケーション・システムを開発する際の統合開発環境である CubeSuite+ について説明します。

CubeSuite+ は，RL78 ファミリ，78K0R マイクロコントローラの統合開発環境（ソフトウエア開発における，設計，実装，デバッグなどの各開発フェーズに必要なツールをプラットフォームである IDE に統合）です。統合することで，さまざまなツールを使い分ける必要がなく，本製品のみを使用して開発のすべてを行うことができます。

対 象 者　このマニュアルは，CubeSuite+ を使用してアプリケーション・システムを開発するユーザを対象としています。

目　　的　このマニュアルは，CubeSuite+ の持つソフトウエア機能をユーザに理解していただき，これらのデバイスを使用するシステムのハードウエア，ソフトウエア開発の参照用資料として役立つことを目的としています。

構　　成　このマニュアルは，大きく分けて次の内容で構成しています。

読 み 方　このマニュアルを読むにあたっては，電気，論理回路，マイクロコンピュータに関する一般知識が必要となります。

凡　　例

データ表記の重み	：	左が上位桁，右が下位桁
アクティブ・ロウの表記	：	$\overline{\text{XXX}}$（端子，信号名称に上線）
注	：	本文中につけた注の説明
注意	：	気をつけて読んでいただきたい内容
備考	：	本文中の補足説明

数の表記	:	10 進数　...　XXXX
		16 進数　...　0xXXXX

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

資料名		資料番号	
		和文	英文
CubeSuite+ 統合開発環境 ユーザーズ・マニュアル	起動編	R20UT2682J	R20UT2682E
	RX 設計編	R20UT2683J	R20UT2683E
	V850 設計編	R20UT2134J	R20UT2134E
	RL78 設計編	R20UT2684J	R20UT2684E
	78K0R 設計編	R20UT2137J	R20UT2137E
	78K0 設計編	R20UT2138J	R20UT2138E
	RH850 コーディング編	R20UT2584J	R20UT2584E
	RX コーディング編	R20UT2470J	R20UT2470E
	V850 コーディング編	R20UT0553J	R20UT0553E
	コーディング編（CX コンパイラ）	R20UT2659J	R20UT2659E
	RL78,78K0R コーディング編	このマニュアル	R20UT2774E
	78K0 コーディング編	R20UT2141J	R20UT2141E
	RH850 ビルド編	R20UT2585J	R20UT2585E
	RX ビルド編	R20UT2472J	R20UT2472E
	V850 ビルド編	R20UT0557J	R20UT0557E
	ビルド編（CX コンパイラ）	R20UT2142J	R20UT2142E
	RL78,78K0R ビルド編	R20UT2623J	R20UT2623E
	78K0 ビルド編	R20UT0783J	R20UT0783E
	RH850 デバッグ編	R20UT2685J	R20UT2685E
	RX デバッグ編	R20UT2702J	R20UT2702E
	V850 デバッグ編	R20UT2446J	R20UT2446E
	RL78 デバッグ編	R20UT2445J	R20UT2445E
	78K0R デバッグ編	R20UT0732J	R20UT0732E
	78K0 デバッグ編	R20UT0731J	R20UT0731E
	解析編	R20UT2686J	R20UT2686E
	メッセージ編	R20UT2687J	R20UT2687E

注意　上記関連資料は，予告なしに内容を変更することがあります。設計などには，必ず最新の資料を使用してください。

目　　次

第 1 章 概 説

この章では，システム開発時における RL78,78K0R C コンパイラ・パッケージ（CA78K0R）の役割，および，機能概要について説明します。

1. 1 概 要

RL78,78K0R C コンパイラは，RL78,78K0R の C 言語，または ANSI-C で記述されたソースを機械語に変換する言語処理プログラムです。RL78,78K0R C コンパイラにより，RL78,78K0R のオブジェクト・ファイル，またはアセンブラ・ソース・ファイルを得ることができます。

RL78,78K0R アセンブラは，RL78,78K0R のアセンブリ言語で記述されたソースを機械語に変換する言語処理プログラムの総称です。

1. 1. 1 C コンパイラとアセンブラ

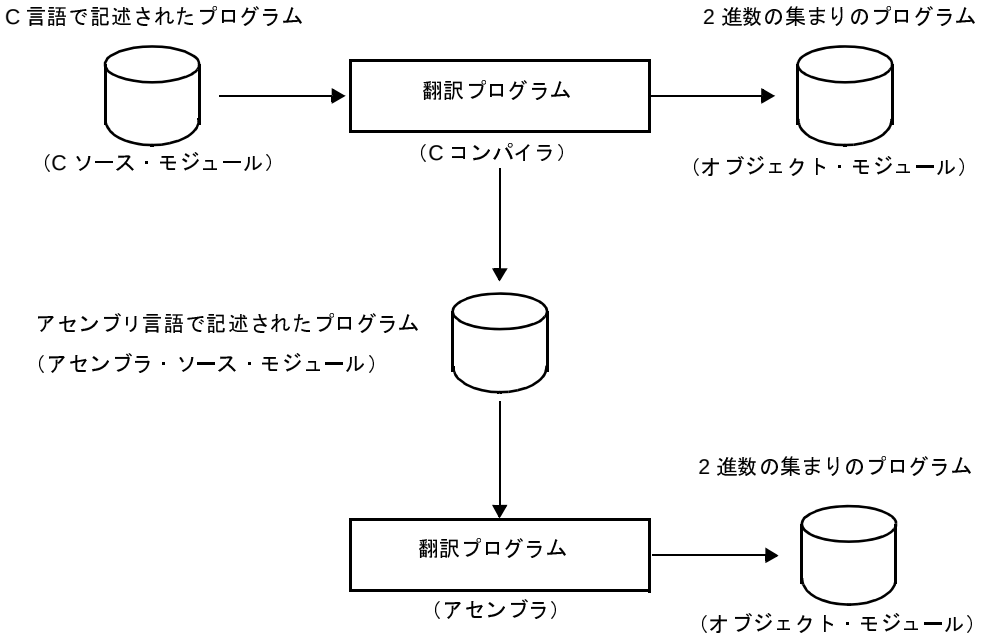
（1）C 言語とアセンブリ言語

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，ユーザは 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することができます。

また，アセンブラは，アセンブラ・ソース・モジュールを入力し，オブジェクト・モジュールを出力します。

C コンパイラ／アセンブラの流れを次に示します。

図 1―1 C コンパイラ／アセンブラの流れ



(2) リロケータブル・アセンブラ

アセンブラが変換した機械語は，マイクロコンピュータのメモリに書き込まれて使用されます。このとき，変換された機械語がメモリのどこに書き込まれるかが，決定されていなければなりません。

したがって，アセンブラの変換する機械語には，「各機械語がメモリのどのアドレスに配置されるべきか」という情報が付加されています。

機械語を配置するアドレスの決定方法により，アセンブラは，“アブソリュート・アセンブラ”と“リロケータブル・アセンブラ”に大別され，RL78,78K0R アセンブラでは，リロケータブル・アセンブラを採用しています。

- アブソリュート・アセンブラ

  アセンブリ言語から変換した機械語は，絶対的なアドレスに配置されます。

- リロケータブル・アセンブラ

  アセンブリ言語から変換した機械語には，一時的なアドレスが与えられます。

  なお，リンカにより，絶対的なアドレスが配置されます。

アブソリュート・アセンブラで１つのプログラムを作成する際には，原則として一度にプログラミングしなければなりませんでした。しかし，大きなプログラムを１つのまとまりとして作成した場合，プログラムが複雑になり，また保守する際にもプログラムの解析が困難になります。

そこで，プログラムを１つ１つの機能単位ごとにいくつかのサブ・プログラム（モジュール）に分割して，プログラム開発を行います。これをモジュラ・プログラミングと呼びます。

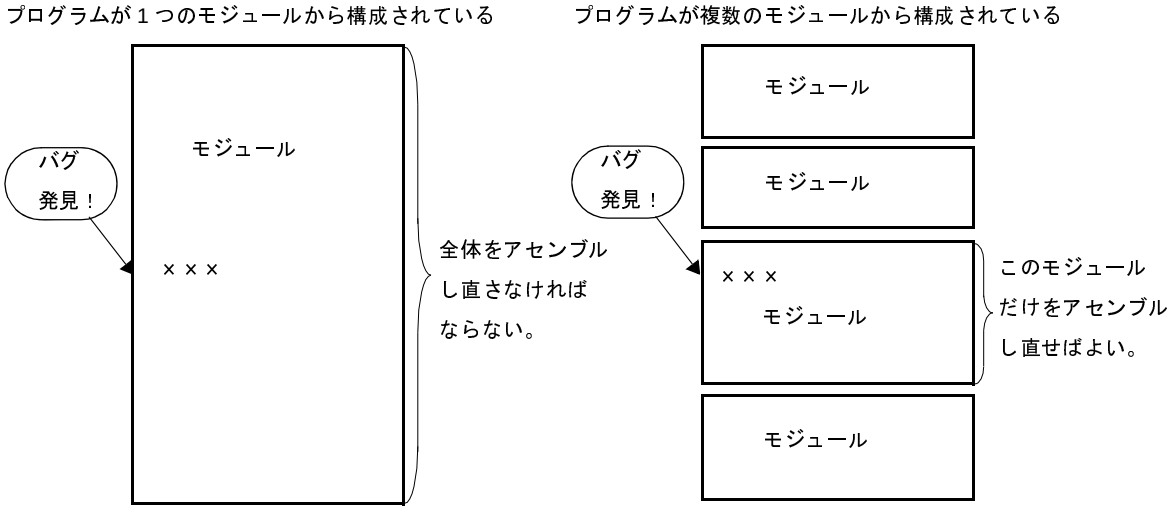
リロケータブル・アセンブラは，モジュラ・プログラミングに適したアセンブラであり，モジュラ・プログラミングを行うことにより，次の利点があげられます。

(a) 開発効率が上がる

大きなプログラムを一度にプログラミングすることは困難です。このような場合，プログラムを１つ１つの機能単位ごとにモジュール分割すれば，複数の人数でプログラムの並行開発ができ，開発効率が上がります。

また，プログラム中にバグが発見された場合，一部の修正を行うために全プログラムをアセンブルすることなく，修正が必要なモジュールだけアセンブルし直すことができ，デバッグ時間を短くできます。

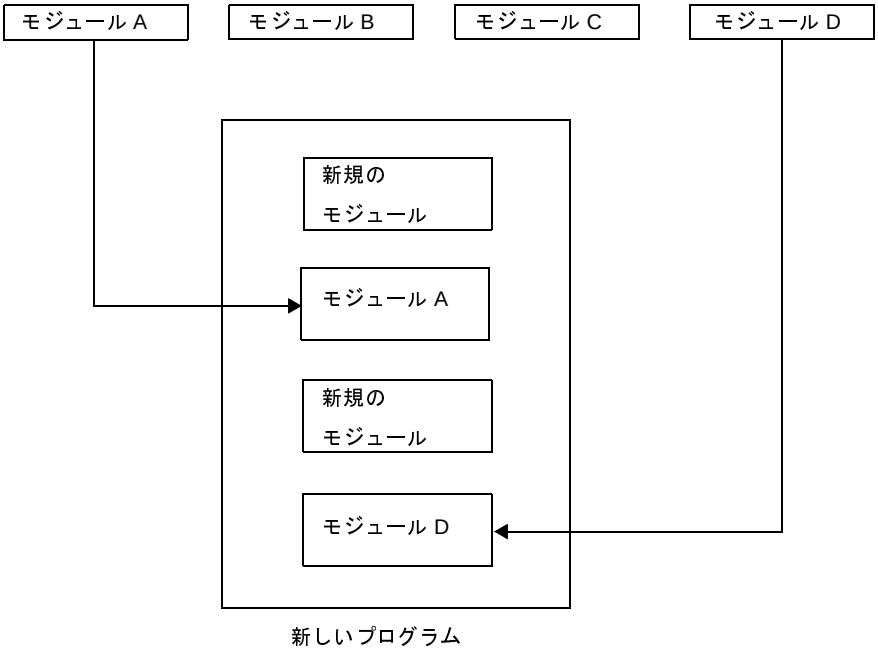
図 1―2　モジュール分割



（b）資源の活用ができる

以前に作成された信頼性，汎用性の高いモジュールをほかのプログラムの開発に再利用できます。このような汎用性の高いモジュールを蓄積することにより，新規にプログラム開発する部分を少なくすることができます。

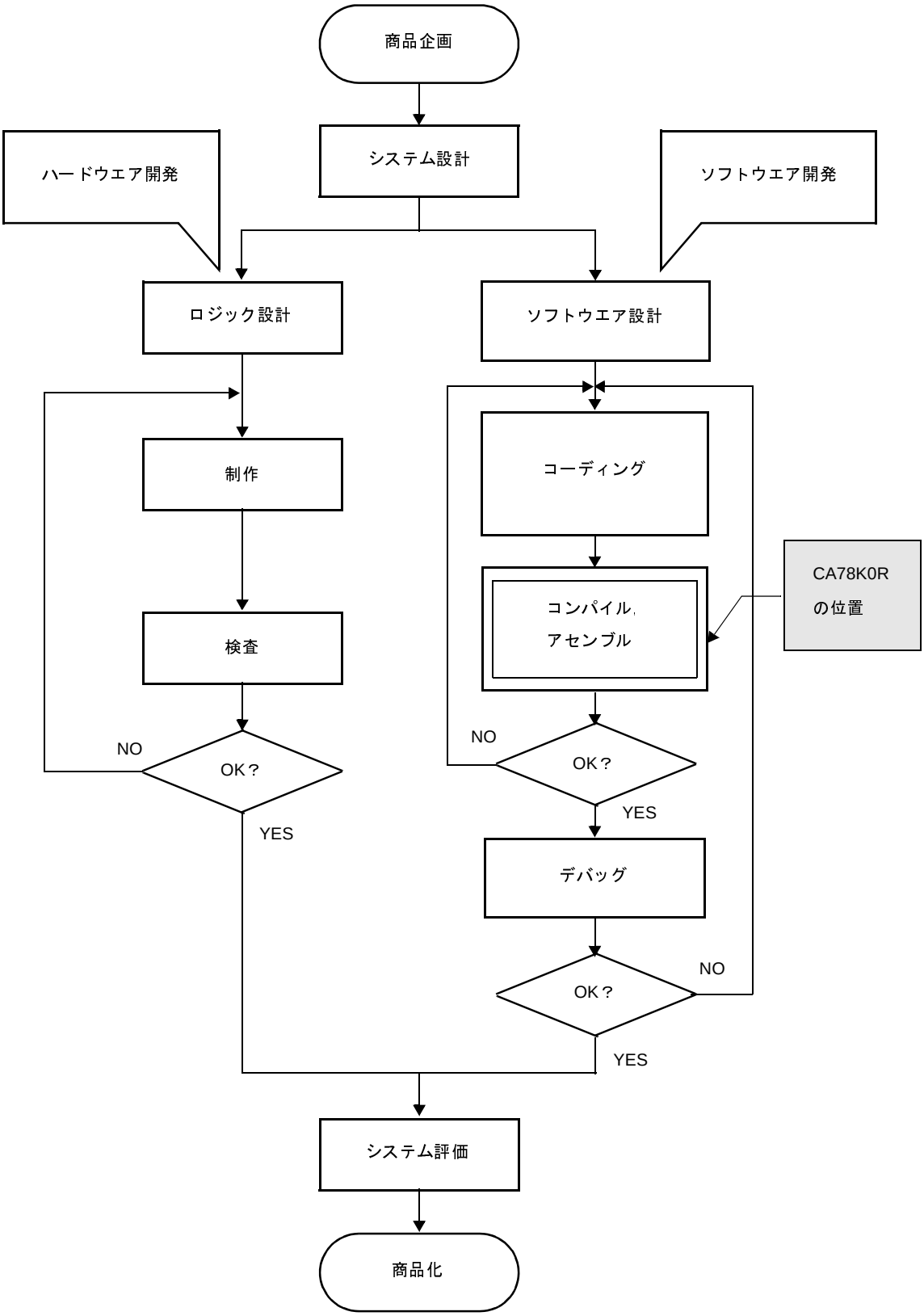
図 1—3 資源の活用



1. 1. 2 C コンパイラ／アセンブラの位置づけ

製品開発における“コンパイラ”，“アセンブラ”の位置づけを次に示します。

図 1—4 マイクロコンピュータ応用製品の開発工程



1. 1. 3 処理の流れ

プログラム開発手順は，次のようになります。

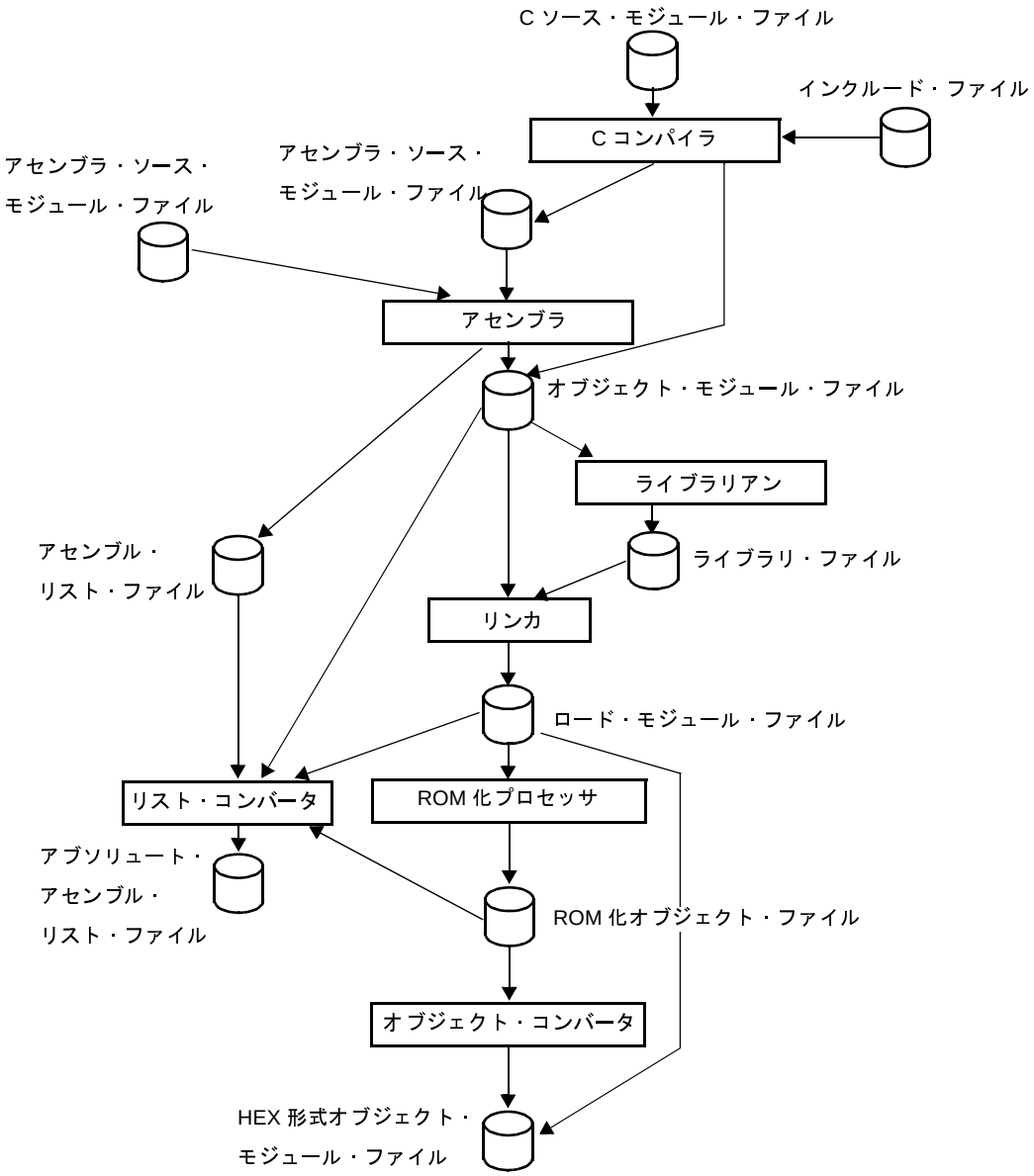
Cコンパイラは，Cソース・モジュール・ファイルをコンパイルしてオブジェクト・モジュール・ファイル，またはアセンブラ・ソース・モジュール・ファイルを生成します。生成されたアセンブラ・ソース・モジュール・ファイルにより，手作業による最適化（ハンド・オプティマイズ）を行うことができ，効率のよいオブジェクト・モジュールを作成することができます。特に，高速な処理を必要とする場合，またはオブジェクト・モジュールをコンパクトにしたい場合などに有効です。

なお，以下のプログラムから構成されています。

表 1—1 プログラム構成

プログラム名	機能
コンパイラ	C ソース・モジュール・ファイルのコンパイル
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 形式オブジェクト・モジュール・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
リスト・コンバータ	アブソリュート・アセンブル・リスト・ファイルの生成

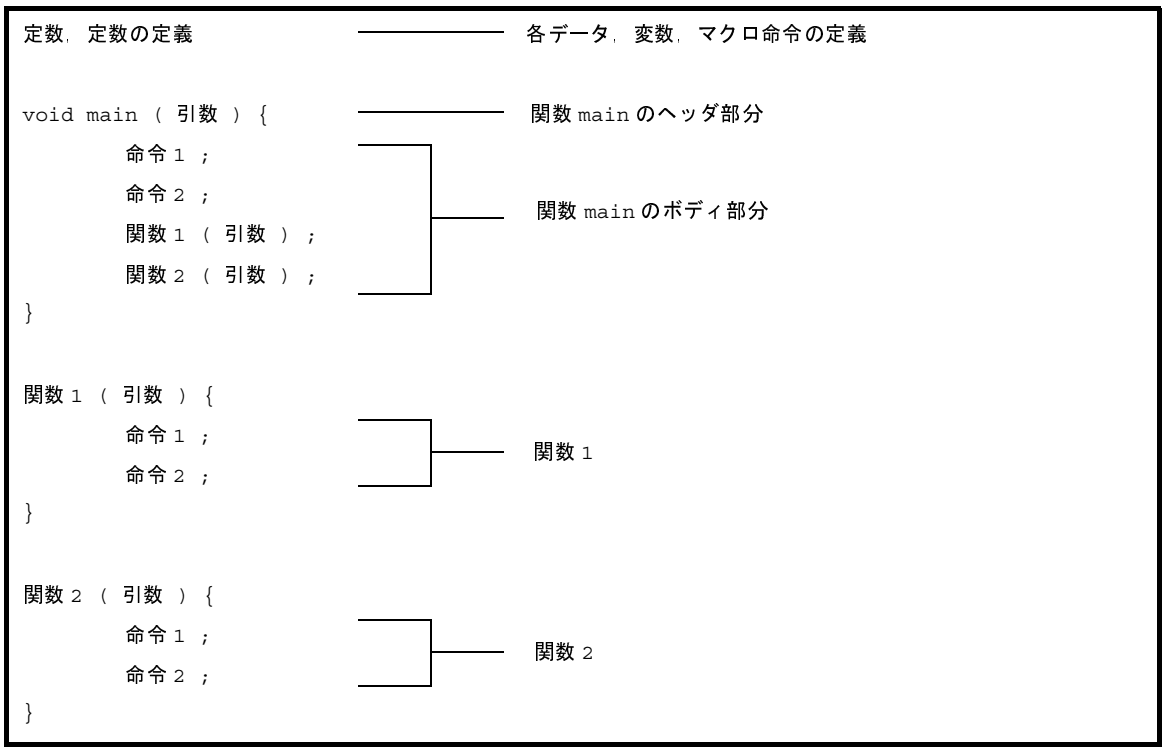
図 1—5 コンパイラ／アセンブラの処理の流れ



1. 1. 4 C ソース・プログラムの基本構成

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 xxx xxx  前処理指令（マクロ定義）*/
#define FALSE   0                       /* #define xxx xxx  前処理指令（マクロ定義）*/
#define SIZE    200                     /* #define xxx xxx  前処理指令（マクロ定義）*/

void    displaystring ( char *  , int ) ;  /* xxx xxxx ( xxx, xxx ) 関数プロトタイプ宣言 */
void    displaychar ( char ) ;          /* xxx xxxx ( xxx )      関数プロトタイプ宣言 */

char    mark[SIZE + 1];                 /* char xxx          型宣言，外部定義 */
                                        /* xx[xx]            演算子 */
void main ( void ) {
    int     i, prime, k, count ;        /* int xxx           型宣言 */

    count = 0 ;                         /* xx = xx           演算子 */

    for ( i = 0 ; i <= SIZE ; i ++ )    /* for ( xx ; xx ; xx ) xxx ; 制御構造 */
        mark[i] = TRUE ;
    for ( i = 0 ; i <= SIZE ; i ++ ) {
```

```
        if ( mark[i] ) {
            prime = i + i + 3 ;              /* xxx = xxx + xxx + xxx    演算子 */
            displaystring ( "%6d", prime ) ; /* xxx ( xxx ) ;            演算子 */

            count ++ ;
            if ( ( count%8 ) == 0 )
                displaychar ( '¥n' ) ;       /* if ( xxx ) xxx ;         制御構造 */
             for ( k = i + prime ; k <= SIZE ; k += prime )
                mark[k] = FALSE ;
        }
    }
    displaystring ( "¥n%d primes found.", count ) ;
                                             /* xxx ( xxx ) ;            演算子 */
}

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

    j = i ;
    ss = s ;
}

void    displaychar ( 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。

(2) 演算子，式

算術演算，論理演算，代入などを行います。

(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。

(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。

(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。

(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。

(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。

1.2 特 長

CA78K0R の特長を次に示します。

1.2.1 C コンパイラの特長

(1) ANSI-C 準拠

C 言語の標準的な規格である ANSI-C 規格に準拠しています。

備考 ANSI: American National Standards Institute

(2) ROM/RAM 効率を重視

外部変数をショート・ダイレクト・アドレッシング領域に割り付けることができます。また，関数引数，自動変数をショート・ダイレクト・アドレッシング領域やレジスタへ割り付けることができます。

ビット操作命令を活用した，1 ビットのデータ定義，操作が可能です。

(3) 組み込みを制御を意識

RL78,78K0R が持つ周辺ハードウェアを C 言語で直接制御できます。

割り込み処理を C 言語で直接記述できます。

(4) RL78,78K0R 固有の拡張仕様をサポート

RL78,78K0R C コンパイラは，ANSI にない CPU のコードを生成する次の拡張機能を備えています。RL78,78K0R 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機能，オブジェクト・コードを短縮し，実行速度の向上を図る機能があります。

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。

表 1—2 実行速度の向上方法一覧

方法	拡張機能
変数をレジスタに割り当てる	レジスタ変数

方法	拡張機能
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
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
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
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
memcpy，memset を直接インライン展開して出力する	メモリ操作関数

RL78,78K0R C コンパイラの各拡張機能の詳細については，「3. 2 拡張言語仕様」を参照してください。

1. 2. 2 アセンブラの特長

RL78,78K0R アセンブラは，次の特長的な機能を備えています。

（1）マクロ機能

ソース中で同じ命令群を何回も記述する場合，その一連の命令群を 1 つのマクロ名に対応させてマクロ定義をすることができます。

マクロ機能を利用することにより，コーディングの効率を上げることができます。

（2）分岐命令の最適化機能

分岐命令自動選択疑似命令として，「BR」，「CALL」を備えています。

メモリ効率のよいプログラムを生成するためには，分岐命令の分岐先範囲に応じたバイトの分岐命令を記述する必要があります。しかし，分岐先範囲をいちいち意識して分岐命令を記述することは面倒です。BR 疑似命令，または CALL 疑似命令を記述することにより，アセンブラが分岐先範囲に応じて適切な分岐命令のコードを生成します。これを分岐命令の最適化機能と呼びます。

（3）条件付きアセンブル機能

ソースの一部分を条件によりアセンブルする／しないに設定することができます。

ソース中にデバッグ文などを記述した場合，デバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。デバッグ文が必要なくなった場合でも，ソースに大幅な変更を加えることなく，アセンブルを行うことができます。

（4）78K0 の互換マクロ機能

78K0 用アセンブラで作成したアセンブラ・ソース・ファイルをアセンブル可能にします。

下記の RL78,78K0R で使用できない 78K0 の命令をソースの記述を変更せずにアセンブルしたい場合，-compati オプションを指定します。

RL78,78K0R で使用できない 78K0 の命令：DIVUW/ROR4/ROL4/ADJBA/ADJBS/CALLF/DBNZ

1. 2. 3 最大値

(1) コンパイラの最大値

コンパイラの最大値については，「(9) 翻訳限界」を参照してください。

(2) アセンブラの最大値

アセンブラの最大値を以下に示します。

表 1—3 アセンブラの最大値

項目	最大値
シンボル数（ローカル＋パブリック）	65,535 個
クロスリファレンス・リスト出力可能なシンボル数	65,534 個注 1
1 マクロ参照のマクロ・ボディ最大サイズ	1M バイト
全マクロ・ボディ合計のサイズ	10M バイト
1 ファイル中のセグメント数	256 個
1 ファイル中のマクロ，インクルード指定	10,000 個
1 インクルード・ファイル中のマクロ，インクルード指定	10,000 個
リロケーション情報注 2	65,535 個
行番号情報	65,535 個
1 ファイル中の BR/CALL 疑似命令数	32,767 個
ソース 1 行の文字数	2,048 文字注 3
シンボル長	256 文字
スイッチ名の定義数注 4	1,000 個
スイッチ名の文字長注 4	31 文字
セグメント名の文字長	8 文字
モジュール名（NAME 疑似命令）の文字長	256 文字
MACRO 疑似命令の仮引数の数	16 個
マクロ参照の実引数の数	16 個
IRP 疑似命令の実引数の数	16 個
マクロ・ボディ内のローカル・シンボル数	64 個
マクロ展開のローカル・シンボル数合計	65,535 個
マクロ（マクロ参照，REPT 疑似命令，IRP 疑似命令）のネスト数	8 レベル
TITLE 制御命令，-lh オプションで指定可能な文字数	60 文字注 5
SUBTITLE 制御命令で指定可能な文字数	72 文字
1 ファイル中のインクルード・ファイルのネスト数	16 レベル
条件アセンブルのネスト数	8 レベル
-i オプションで，指定可能なインクルード・ファイル・パス数	64 個
-d オプションで定義可能なシンボル数	30 個

注 1. モジュール名，セクション名の個数を引いた数です。
メモリを使用します。メモリがなければファイルを使用します。

2. アセンブラでシンボル値が解決できない場合に，リンカに渡す情報のことです。
たとえば，MOV 命令で外部参照シンボルを参照した場合，リロケーション情報が 2 個，.rel ファイルに生成されます。

3. 復帰 / 改行コードを含みます。1 行に 2049 文字以上記述した場合，警告メッセージが出力され，2049 文字以降は無視されます。

4. スイッチ名は，SET/RESET 疑似命令で真 / 偽に設定され，$IF などで使用されるものです。

5. アセンブル・リスト・ファイルの 1 行の文字数指定（X とします）が 119 文字以下の場合，“X - 60” 文字以内とします。

(3) リンカの最大値

リンカの最大値を以下に示します。

表 1—4 リンカの最大値

項目	最大値
シンボル数（ローカル＋パブリック＋内部生成シンボル[注 1]）	2,147,483,647 個
同一セグメントの行番号情報	1,048,575 個
セグメント数	65,535 個[注 2]
入力モジュール	1,024 個
メモリ領域名の文字長	256 文字
メモリ領域数	100 個[注 2]
-b オプションで指定可能なライブラリ・ファイル数	64 個
-i オプションで指定可能なライブラリ・ファイル・パス数	64 個

注 1. 各プログラムが内部で生成するシンボル（デバッグ情報用含む）

2. デフォルトで定義されているものを含みます。

第 2 章 機 能

この章では，CA78K0R をより効果的に用いるためのプログラミング技法，および拡張機能の利用方法について説明します。

2. 1 変数（アセンブリ言語）

この節では，変数（アセンブリ言語）について説明します。

2. 1. 1 初期値なし変数を定義する

データ・セグメント内にメモリ領域を確保します。

データ・セグメントを定義するには，DSEG 疑似命令を使用し，メモリ領域には DS 疑似命令を使用します。

例 10 バイトの初期値なし変数の定義

```
            DSEG
_table :    DS      10
```

備考 「DSEG」，「DS」を参照してください。

2. 1. 2 初期値あり const 定数を定義する

コード・セグメント内のメモリ領域を初期化します。

コード・セグメントを定義するには，CSEG 疑似命令を使用し，メモリ初期化には 1 バイトの場合には DB 疑似命令を，2 バイトの場合には DW 疑似命令を，4 バイトの場合には DG 疑似命令を使用します。

例 初期値あり定数の定義

```
            CSEG
_val1 :     DB      0F0H    ; 1 バイト
_val2 :     DW      1234H   ; 2 バイト
_val3 :     DG      56789H  ; 4 バイト（20 ビット）
```

備考 「CSEG」，「DB」，「DW」，「DG」を参照してください。

2.1.3 1ビット変数を定義する

ビット・セグメント内に1ビットメモリ領域を確保します。

ビット・セグメントを定義するには，BSEG 疑似命令を使用し，1ビットのメモリ領域には DBIT 疑似命令を使用します。

例 初期値なしビット変数の定義

```
        BSEG
_bit1   DBIT
_bit2   DBIT
_bit3   DBIT
```

備考 「BSEG」,「DBIT」を参照してください。

2. 1. 4 変数の 1/8 ビット・アクセスを行う

アセンブラの記述で，saddr 上のアドレスに 2 つのシンボル名を与え，そのシンボル名をそれぞれビット・アクセス用とバイト・アクセス用とするためには，セグメント DSEG の再配置属性を saddr とし，バイト・アクセス用シンボルのビット名を EQU 疑似命令で，ビット・アクセス用シンボル名として定義してください。

例 バイト・アクセス用シンボル名：FLAGBYTE,
ビット・アクセス用シンボル名：FLAGBIT の場合

- smp1.asm

```
        NAME    SMP1
        PUBLIC  FLAGBYTE, FLAGBIT

FLAGS           DSEG    SADDR           ; DSEG の再配置属性は SADDR
FLAGBYTE :      DS ( 1 )                ; FLAGBYTE の定義
FLAGBIT         EQU     FLAGBYTE.0      ; FLAGBIT の定義
                END
```

- smp2.asm

```
        NAME    SMP2

        EXTRN   FLAGBYTE
        EXTBIT  FLAGBIT         ; FLAGBIT は EXTBIT 宣言して使用

        CSEG
C1 :
        MOV     FLAGBYTE, #0FFH
        CLR1    FLAGBIT
        END
```

備考 「DSEG」,「EQU」を参照してください。

2. 1. 5 短い命令長でアクセスできる領域へ配置する

ショート・ダイレクト・アドレッシング領域は，ほかのデータ・メモリに比べ，短いバイト数の命令でアクセスすることのできる領域です。この領域を効率的に使うことで，メモリ効率のよいプログラムを開発することができます。

ショート・ダイレクト・アドレッシング領域に配置するには，DSEG 疑似命令の再配置属性を saddr，または，saddrp に指定します。

アセンブラソースの使用例を以下に示します。

- モジュール 1

```
PUBLIC  TMP1, TMP2
WORK    DSEG saddrp
TMP1 :  DS 2 ; word
TMP2 :  DS 1 ; byte
```

- モジュール 2

```
EXTRN   TMP1, TMP2
SAB     CSEG
MOVW    TMP1, #1234H
MOV     TMP2, #56H
        :
```

備考 「DSEG」を参照してください。

2.2 変数（C 言語）

この節では，変数（C 言語）について説明します。

2.2.1 参照のみのデータを ROM へ配置する

（1）初期値あり変数の ROM 配置

参照のみの初期値あり変数を ROM に配置するには，const 修飾子を指定してください。

例 参照のみの初期値あり変数 a を ROM に配置します。

```
const int       a = 0x12 ;      /* ROM に配置 */
int             b = 0x12 ;      /* ROM/RAM に配置 */
```

変数 a は，ROM に配置されます。
変数 b の場合には，初期値が ROM に配置されて，変数自体は RAM に配置されます（ROM ／ RAM 両方の領域が必要になります）。
スタートアップ・ルーチンの ROM 化処理で，ROM の初期値を RAM の変数にコピーします。
ROM 化により，ROM と RAM の両方に領域が必要になります。

（2）ROM 領域へのテーブル・データの割り当て

テーブル・データを ROM 領域だけに割り当てるには，以下のように型修飾子 const を使用して宣言してください。

```
const unsigned char table_data[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 } ;
```

2.2.2 短い命令長でアクセスできる領域へ配置する

ショート・ダイレクト・アドレッシング領域は，ほかのデータ・メモリに比べ，短いバイト数の命令でアクセスすることのできる領域です。この領域を効率的に使うことで，メモリ効率のよいプログラムが開発できます。

使用例を以下に示します。

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，自動的にショート・ダイレクト・アドレッシング領域 [FFE20H ～ FFEB3H]，にリロケータブルに割り当てられます。

```
extern sreg int hsmm0 ;
extern sreg int hsmm1 ;
extern sreg int *hsptr ;

void main ( void ) {
        hsmm0 -= hsmm1 ;
}
```

備考 「sreg 宣言による saddr 領域利用（sreg/__sreg）」を参照してください。

2. 2. 3 near 領域へ配置する

スモール・モデルを使うと，アドレス長を 16 ビットとしてコード生成を行います。

コードやデータが 64K バイト以内ということがあらかじめわかっている場合は，ラージ・モデルよりもスモール・モデルを使うほうがコード・サイズを小さくすることができます。

コンパイラオプションにて，スモール・モデル（-ms オプション）を指定することでデータと関数を near 領域へ配置します。

または，変数，関数に __near 型修飾子を付けます。

```
__near int              func ( void ) ; /* near 領域に配置 */
__near const int        a = 0 x 12 ;    /* near 領域に配置 */
__near int              b = 0 x 12 ;    /* near 領域に配置 */

__near int func ( void ) {              /* near 領域に配置 */
        /* 関数処理 */
        return 0 ;
}
```

備考 「near/far 領域指定（__near/__far）」を参照してください。

2. 2. 4 far 領域へ配置する

ラージ・モデルを使うと，アドレス長を 20 ビットとしてコード生成を行います。

データ 64K バイト以内，コード 1M バイト以内であればミディアム・モデルを使用してください。

コンパイラオプションにて，ミディアム・モデル（-mm オプション）を指定することでデータを near 領域，関数を far 領域へ配置します。

または，変数，関数にそれぞれ __near，__far 型修飾子を付けます。

```
__far int                func ( void ) ; /* far 領域に配置 */
__near const int         a = 0 x 12 ;    /* near 領域に配置 */
__near int               b = 0 x 12 ;    /* near 領域に配置 */

__far int func ( void ) {                /* far 領域に配置 */
        /* 関数処理 */
        return 0 ;
}
```

データ 1M バイト以内，コード 1M バイト以内であればラージ・モデルを使用してください。

コンパイラオプションにて，ラージ・モデル（-ml オプション）を指定することでデータと関数を far 領域へ配置します。

または，変数，関数にそれぞれ __far 型修飾子を付けます。

```
__far int                func ( void ) ; /* far 領域に配置 */
__far const int          a = 0 x 12 ;    /* far 領域に配置 */
__far int                b = 0 x 12 ;    /* far 領域に配置 */

__far int func ( void ) {                /* far 領域に配置 */
        /* 関数処理 */
        return 0 ;
}
```

備考 「near/far 領域指定（__near/__far）」を参照してください。

2. 2. 5 直接アドレスへ配置する

（1）direcrtmap

__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。配置アドレスは整数で指定してください。

C ソース中における __directmap 変数は，static 変数と同様に扱います。

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 char        c = 0xffe00 ;
__directmap __sreg char d = 0xffe20 ;
__directmap __sreg char e = 0xffe21 ;

__directmap struct x {
        char    a ;
        char    b ;
} xx = { 0xffe30 } ;
void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

備考 「絶対番地配置指定（__directmap）」を参照してください。

（2）セクションを使用した方法

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。

#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。

セクション名 @@CODEL を CC1 に変更し，開始アドレスを 2400H 番地に指定します。

```
#pragma section @@CODEL CC1 AT  2400H

void main ( void ) {
        /* 関数本体 */
}
```

備考 「コンパイラ出力セクション名の変更（#pragma section）」を参照してください。

2. 2. 6 1 ビット変数を定義する

変数を bit，boolean 型にすることで，1 ビットのデータとして扱われ，ショート・ダイレクト・アドレッシング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。

- MOV1, AND1, OR1, XOR1, SET1, CLR1, NOT1, BT, BF

C 記述で，ショート・ダイレクト・アドレッシング領域へのビット・アクセスが可能になります。

```
#define ON      1
#define OFF     0

extern bit      data1 ;
extern bit      data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while ( data1 ) {
                data1 = data2 ;
                testb ( ) ;
        }
        if ( data1 && data2 )
                chgb ( ) ;
}
```

備考 「bit 型変数（bit），boolean 型変数（boolean/__boolean）」を参照してください。

2. 2. 7 構造体の空き領域を詰める

構造体のメンバ変数を 2 バイト・アラインしないようにするには，-rc オプションを指定してください。

ただし，構造体以外の変数をアラインメントしないようにする方法は，サポートされていません。

2.3 関　　数

この節では，関数について説明します。

2.3.1 短い命令長でアクセスできる領域へ配置する

callt 関数を利用することで通常の呼び出し命令 call に比べ，オブジェクト・コードを短縮することができます。

callt 命令は，callt テーブルと呼ばれる領域 [80H ～ 0BFH] 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 void    func1 ( void ) ;

__callt void    func1 ( void ) {
        /* 関数本体 */
}
```

備考 「callt 関数（callt/__callt）」を参照してください。

2.3.2 near 領域へ配置する

スモール・モデルを使うと，アドレス長を 16 ビットとしてコード生成を行います。

コードやデータが 64K バイト以内ということがあらかじめわかっている場合は，ラージ・モデルよりもスモール・モデルを使うほうがコード・サイズを小さくすることができます。

コンパイラオプションにて，スモール・モデル（-ms オプション）を指定することで関数を near 領域へ配置します。

または，関数に __near 型修飾子を付けます。

```
__near int              func ( void ) ; /* near 領域に配置 */
__near const int        a = 0 x 12 ;    /* near 領域に配置 */
__near int              b = 0 x 12 ;    /* near 領域に配置 */

__near int func ( void ) {              /* near 領域に配置 */
        /* 関数処理 */
        return 0 ;
}

void main ( void ) {
        int     a ;
        a = func ( ) ;
}
```

備考 「near/far 領域指定（__near/__far）」を参照してください。

2. 3. 3 far 領域へ配置する

データ 64K バイト以内，コード 1M バイト以内であればミディアム・モデルを使用してください。

コンパイラオプションにて，ミディアム・モデル（-mm オプション）を指定することで関数を far 領域へ配置します。

データ 1M バイト以内，コード 1M バイト以内であればラージ・モデルを使用してください。

コンパイラオプションにて，ラージ・モデル（-ml オプション）を指定することでデータを far 領域，関数を far 領域へ配置します。

または，関数に __far 型修飾子を付けます。

```
__far int               func ( void ) ; /* far 領域に配置 */
__near const int        a = 0 x 12 ;    /* near 領域に配置 */
__near int              b = 0 x 12 ;    /* near 領域に配置 */

__far int func ( void ) {               /* far 領域に配置 */
        /* 関数処理 */
        return 0 ;
}

void main ( void ) {
        int     a ;
        a = func ( ) ;
}
```

備考 「near/far 領域指定（__near/__far）」を参照してください。

2. 3. 4 直接アドレスへ配置する

(1) セクションを使用した方法

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。

#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。

```
#pragma section @@DATA  ??DATA  AT 0FDE00H

int             a1 ;                            // ??DATA
int             a2 ;                            // ??DATA

#pragma section @@DATS  ??DATS  AT 0FFE30H

sreg int        b1 ;                            // ??DATS
sreg int        b2 ;                            // ??DATS
```

備考 「コンパイラ出力セクション名の変更（#pragma section）」を参照してください。

2. 3. 5 関数のインライン展開を行う

#pragma inline は，メモリ操作用標準ライブラリ memcpy，memset を関数呼び出しではなく，直接インライン展開してコードを出力する機能です。

また，実行速度を速くするために，いくつかの関数をインライン展開したい場合，関数ごとにインライン展開できるような命令はありませんが，memcpy，memset 以外の関数をインライン展開する場合は，以下のような関数形式マクロ等を使用して記述してください。

```
#define MEMCOPY ( a, b, c ) \
        { \
                struct st { unsigned char d[ ( c ) ]; } ; \
                * ( ( struct st * ) ( a ) ) = * ( ( struct st * ) ( b ) ) ; \
        }
```

備考 「メモリ操作関数（#pragma inline）」を参照してください。

2. 3. 6 アセンブラ命令を埋め込む

C コンパイラが出力するアセンブラ・ソース・ファイルに，ユーザが記述したアセンブラ・ソースを埋め込みます。

（1）#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
```

プロパティ パネル の［コンパイル・オプション］タブで「アセンブリ・ファイルを出力する」を「はい」に設定してください。（設定方法は，「CubeSuite+ 統合開発環境 ユーザーズマニュアル RL78,78K0R ビルド編」を参照してください。）

備考 「ASM 文（#asm ～ #endasm/__asm）」を参照してください。

（2）__asm

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可能です。

プロパティ パネル の［コンパイル・オプション］タブで「アセンブリ・ファイルを出力する」を「はい」に設定してください。（設定方法は，「CubeSuite+ 統合開発環境 ユーザーズマニュアル RL78,78K0R ビルド編」を参照してください。）

備考 「ASM 文（#asm ～ #endasm/__asm）」を参照してください。

2. 4 マイコン機能の使用

この節では，マイコン機能の使用について説明します。

2. 4. 1 C 言語で特殊機能レジスタ（SFR）へアクセスする

（1）SFR の各レジスタの設定

sfr 領域は，RL78 ファミリ，78K0R マイクロコントローラ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（PM1，P1，TMC80 など）の領域です。

sfr 領域を使用するには，C ソースの先頭に，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することができます。

```
#pragma sfr
```

宣言がない場合は，定義されていないというエラー・メッセージが表示されます。

```
E0711 Undeclared ' 変数名 ' ; function ' 関数名 '
```

#pragma sfr 指定で使用可能になる SFR の記号は，特殊機能レジスター覧中の略号と同じです。

また，次のものは，#pragma sfr の前に記述することができます。

- 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 名は，初期値なし（不定）の外部変数となります。

SFR 名に不正な定数データを代入した場合は，コンパイル・エラーとなります。

備考 「sfr 領域利用（#pragma sfr）」を参照してください。

（2）SFR の各レジスタ内のビット指定

SFR の各レジスタ内のビット指定の記述方法は，以下のように予約語，または，“レジスタ名 . ビット位置”になります。

例 1. TM1 を開始させる場合

```
TCE1 = 1 ;
または
TMC1.0 = 1 ;
```

2. TM1 を停止させる場合

```
TCE1 = 0 ;
または
TMC1.0 = 0 ;
```

2. 4. 2 C 言語で割り込み処理を行う

(1) 割り込み関数

割り込み関数を指定する場合には，次の 2 つの指令があります。

- #pragma interrupt
- #pragma vect

これらはどちらを使用してもかまいません。また，ベクタ・テーブルも生成され，出力アセンブラ・ソースで確認することができます。

#pragma 指令は C ソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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

例 INTP0 端子入力に対する割り込み処理

```
#pragma interrupt INTP0 inter rb1

void    inter ( void ) {
        /* INTP0 端子入力に対する割り込み処理 */
}
```

備考 「割り込み関数（#pragma vect/interrupt）」を参照してください。

(2) OS 管理割り込みハンドラ

OS 管理割り込みハンドラは，“#pragma rtos_interrupt”を使用し以下のように記述します。

#pragma 指令は C ソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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 rtos_interrupt INTP0 inthdr1

#include "kernel.h"
#include "kernel_id.h"

void    inthdr1 ( void ) {
        /* 本体処理を記述 */
        return ;
}
```

備考 「RTOS 用割り込みハンドラ（#pragma rtos_interrupt）」を参照してください。

（3）スタック領域の確保

コンパイラの拡張機能の割り込み関数を使用する場合で，スタック切り替えの指定を使わなかった場合は，コンパイラは必要なスタック・サイズを別途確保せず，デフォルトのスタックを使用します。

2. 4. 3 C 言語で CPU 制御命令を使用する

（1）halt 命令

マイクロコンピュータのスタンバイ機能の一つである halt 命令を使用するには，#pragma HALT を使用し次のように記述します。

#pragma 指令は C ソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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

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。

例 halt 命令の使用

```
#pragma HALT
     :
void    func ( void ) {
         :
        HALT ( ) ;
}
```

備考 「CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」を参照してください。

（2）stop 命令

マイクロコンピュータのスタンバイ機能の一つである stop 命令を使用するには，#pragma STOP を使用し次のように記述します。

#pragma 指令はＣソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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

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。

例 stop 命令の使用

```
#pragma STOP
    :
void    func ( void ) {
         :
        STOP ( ) ;
}
```

備考 「CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」を参照してください。

（3）brk 命令

マイクロコンピュータのソフトウエア割り込みを使用するには，#pragma BRK を使用し次のように 記述します。

#pragma 指令はＣソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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

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。

例 brk 命令の使用

```
#pragma BRK
    :
void    func ( void ) {
         :
        BRK ( ) ;
}
```

備考 「CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」を参照してください。

（4）nop 命令

マイクロコンピュータの何も動作をせずにクロックを進める nop 命令を使用するには，#pragma NOP を使用し次のように記述します。

#pragma 指令はＣソースの先頭に記述します。

ただし，次の項目はこの #pragma 指令の前に記述することができます。

- 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

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。

例 nop 命令の使用

```
#pragma NOP
     :
void    func ( void ) {
         :
        NOP ( ) ;
}
```

備考 「CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」を参照してください。

2.5 スタートアップ・ルーチン

この節では，スタートアップ・ルーチンについて説明します。

2.5.1 スタートアップ・ルーチン内の関数／領域を削除する

（1）exit 関数の削除

スタートアップ・ルーチンの中で，EQU シンボル EXITSW を 0 に設定することにより，exit 関数を削除することができます。

（2）未使用領域の削除

標準ライブラリが使用する _@FNCTBL 等の領域については，使用しているライブラリを確認し，スタートアップ・ルーチン cstart.asm 中の EXITSW 等の EQU シンボルの値を変更することにより，未使用の領域を削除することができます。

制御する EQU シンボルとライブラリ関数名，シンボル名はつぎのようになります。

EQU シンボル	ライブラリ関数名	シンボル名
BRKSW	brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR
EXITSW	exit atexit	_@FNCTBL _@FNCENT
RANDSW	rand srand	_@SEED
DIVSW	div	_@DIVR
LDIVSW	ldiv	_@LDIVR
STRTOKSW	strtok	_@TOKPTR
FLOATSW	atof strtod strtol strtoul 数学関数 浮動小数点ランタイム・ライブラリ	_errno

備考 「7.4 スタートアップ・ルーチン」を参照してください。

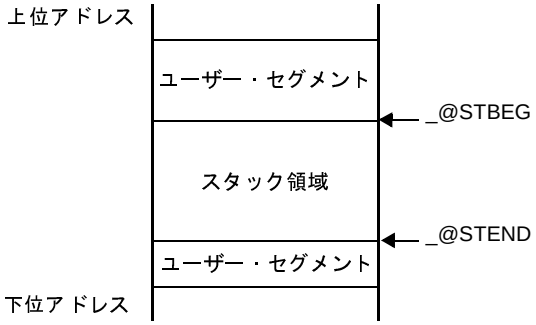
2. 5. 2 スタック領域を確保する

(1) スタックの設定

リンクする際に，スタック解決用シンボル生成指定オプション -s を指定すると，スタックの最下位アドレスの値を持つ「_@STEND」シンボルと，最上位アドレス＋１の値を持つ「_@STBEG」シンボルが生成されます。

```
MOVW    SP, #LOWW       _@STBEG
```

図 2—1 スタックの設定



この場合，スタック・ポインタは，以下のように設定してください。

```
MOVW    SP, #LOWW       _@STBEG
```

(2) スタック領域の確認

リンカの -kp オプションを指定して，リンク・リスト・ファイル中にパブリック・シンボル・リストを出力することで，スタック領域を確認することができます。

「_@STEND」シンボルと「_@STBEG」シンボルの間が，スタック領域になります。

例 パブリック・シンボル・リスト

```
*** Public symbol list ***

MODULE  ATTR    VALUE   NAME

        NUM     0FFE20H  _@STBEG
        NUM     0FFB7EH  _@STEND
```

2. 5. 3 RAM の初期化を行う

スタートアップ・ルーチンで初期化を行うのは以下です。

- saddr 領域（汎用レジスタ領域は除く）
- “_@STEND”シンボルと“_@STBEG”シンボルで挟まれたスタック領域

また，デフォルトのスタートアップ・ルーチンで初期値をコピーされる領域は次のようになります。

- @@INIT セグメント
- @@INITL セグメント
- @@INIS セグメント

0 クリアされる領域は次のようになります。

- saddr 領域（0FFE20H ～ 0FFEDFH）
- @@DATA セグメント
- @@DATAL セグメント
- @@DATS セグメント

上記以外の領域を初期化したい場合には，スタートアップ・ルーチンに初期化する処理を追加してください。

備考 「7. 4 スタートアップ・ルーチン」を参照してください。

2.6 リンク・ディレクティブ

この節では，リンク・ディレクティブについて説明します。

2.6.1 デフォルト領域を分割する

リンク・ディレクティブにて定義するメモリ領域の名前を指定することができますが，特殊機能レジスタ領域の配置には注意が必要です。

たとえば，RAM 領域を 2 箇所指定する場合のメモリ領域名をデフォルトで定義されている RAM とユーザ定義の STACK とした場合，必ず「RAM」という領域に SFR が含まれるようにリンク・ディレクティブを記述してください。

例 リンク・ディレクティブ

```
MEMORY  STACK : ( 0FEF00H, 00100H )
MEMORY  RAM :   ( 0FF000H, 01000H )
```

備考 「5.1.1 リンク・ディレクティブ」を参照してください。

2.6.2 セクションの配置を指定する

(1) 領域指定

セクションの配置を指定する際に，メモリ領域を指定することができます。

MERGE 疑似命令で，対象セクションをメモリ領域に配置指定します。

例 入力セグメント SEG1 をメモリ領域 MEM1 中に割り付ける

```
MEMORY  ROM : ( 0000H, 1000H )
MEMORY  MEM1 : ( 1000H, 2000H )
MERGE   SEG1 : = MEM1
```

備考 「5.1.1 リンク・ディレクティブ」を参照してください。

(2) アドレス指定

セクションの配置を指定する際に，アドレスを指定することができます。

MERGE 疑似命令で，対象セクション配置アドレスを指定します。

例 入力セグメント SEG1 を 500H 番地に割り付ける

```
MEMORY  ROM : ( 0000H, 10000H )
MERGE   SEG1 : AT ( 500H )
```

備考 「5.1.1 リンク・ディレクティブ」を参照してください。

2.7 コード・サイズの削減

この節では，コード・サイズの削減について説明します。

2.7.1 拡張機能でオブジェクト生成の効率化を行う

RL78,78K0R 応用製品の開発を行う場合，RL78,78K0R 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することができます。

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（オートマチック変数を使用する && saddr 領域が使用可能）

（1）外部変数の使用

外部変数を定義するときに 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・コードを縮小することができ，実行速度も向上します（sreg 変数にする代わりに，オプション -rd によっても同様のことを行うことができます）。

```
sreg/__sreg 変数の定義 :  extern sreg int   変数名 ;
                          extern __sreg int 変数名 ;
```

備考 「sreg 宣言による saddr 領域利用（sreg/__sreg）」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，コードを縮小することができ，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 「bit 型変数（bit），boolean 型変数（boolean/__boolean）」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短いコードで呼び出すことができます。

```
callt 関数の定義 : callt   int     tsub ( ) {
                   :
                   }
```

備考 「callt 関数（callt/__callt）」を参照してください。

-qx3 は，-qx2 に加え”共通コードのサブルーチン化”と”スタックアクセス用ライブラリ”の呼び出しなどを行うことで，コード・サイズの削減を行います。したがって，-qx2 と比較して実行スピードは遅くなる可能性があります。

(4) 拡張機能の使用

関数の定義
- if （オートマチック変数を使用する && saddr 領域が使用可能）
  - register 宣言する
- if （内部 static 変数を使用する）&& （saddr 領域が利用可能）
  - __sreg 宣言する

(a) オートマチック変数を使用する関数

オートマチ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，「レジスタ変数（register）」を参照してください。

（b）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・コードを縮小することができ，実行速度も向上します。

備考 「sreg 宣言による saddr 領域利用（sreg/__sreg）」を参照してください。

（5）その他の機能

その他，次のような方法で，コード効率，または実行速度を向上することができます。

（a）SFR 名（または SFR ビット名称）の使用

#pragma sfr

備考 「sfr 領域利用（#pragma sfr）」を参照してください。

（b）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 ;
```

備考 「sreg 宣言による saddr 領域利用（sreg/__sreg）」を参照してください。

（c）割り込み処理にはレジスタ・バンク切り替えを使用

#pragma interrupt INTP0 inter RB1

備考 「割り込み関数（#pragma vect/interrupt）」を参照してください。

（d）乗算，除算組み込み関数の使用

#pragma mul

#pragma div

備考 「乗算関数（#pragma mul）」，「除算関数（#pragma div）」を参照してください。

（e）高速化したいモジュールのみ，アセンブリ言語で記述

2. 7. 2 複雑な式の計算を行う

計算結果がバイト型で収まる数値になるとき，計算過程の途中でダブル・ワード型が必要な場合の最も合理的なプログラムの記述方法は，以下のようになります。

例 a の b における百分率 c を四捨五入して求めます。

```
c = ( a × 100 + b ÷ 2 ) ÷ b
```

以下のような記述した場合，答え c を long int 宣言することになり，領域の確保が 1 バイトで済むところを 4 バイトも確保しなければなりません。

```
void    _x ( ) {
         c = ( ( unsigned long int ) a * ( unsigned long int ) 100 + ( unsigned long int ) b / ( unsigned long int ) 2 ) / ( unsigned long int ) b ;
}
```

計算過程の途中の数値だけをダブル・ワード型にするには，以下のように記述してください。

```
#pragma mul
#pragma div

unsigned int    a, b ;
unsigned char   c ;

void    _x ( ) {
        c = ( unsigned char ) divux ( ( unsigned long ) ( b / 2 ) + muluw ( a, 100 ), b ) ;
}
```

2. 8 コンパイラとアセンブラの相互参照

この節では，コンパイラとアセンブラの相互参照について説明します。

2. 8. 1 変数の相互参照を行う

(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，extern 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“_”（アンダースコア）を付けます。

例 C ソース

```
extern void     subf ( void ) ;
char    c = 0 ;
int     i = 0 ;

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

例 アセンブラ・ソース

```
$PROCESSOR ( F1166A0 )
        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i
@@CODE CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

備考 「9. 5 C 言語で定義した変数をアセンブリ言語側で参照する方法」を参照してください。

(2) アセンブリ 言語で定義した変数を参照する方法

アセンブリ 言語プログラム中で定義した外部変数を C 言語ルーチン中で参照する場合，extern 宣言します。

アセンブリ言語ルーチン中で定義する変数の先頭に“_”（アンダースコア）を付けます。

例 C ソース

```
extern char     c ;
extern int      i ;

void    subf ( void ) {
        c = 'A' ;
        i = 4 ;
}
```

例 アセンブラ・ソース

```
NAME ASMSUB
                PUBLIC  _i
                PUBLIC  _c
ABC DSEG        BASEP
_i : DW         0
_c : DB         0
                END
```

備考 「9. 6 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」を参照してください。

2. 8. 2 関数の相互参照を行う

(1) C 言語で定義した関数を参照する方法

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。

(a) ワーク・レジスタ（AX，BC，DE）を退避する

(b) 引数をスタックに積む

(c) C 言語ルーチンをコールする

(d) 引数のバイト数分スタック・ポインタ（SP）の値を修正する

(e) C 言語ルーチンの戻り値（BC，または DE，BC）を参照する

例 アセンブリ言語

```
$PROCESSOR ( F1166A0 )
        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2
@@CODE CSEG
_FUNC2 :
        movw    ax, #20H        ; 第２引数 "j" を設定
        push    ax              ;
        movw    ax, #21H        ; 第１引数 "i" を設定
        call    !_CSUB          ; 関数 "CSUB ( i, j )" の呼び出し
        pop     ax              ;
        ret
END
```

備考 詳細は，「9. 3 アセンブリ言語から C 言語ルーチンの呼び出し」を参照してください。

(2) アセンブリ 言語で定義した関数を参照する方法

C 言語ルーチンから呼び出されるアセンブリ言語で定義した関数では，次の手順で処理を行います。

(a) ベース・ポインタ，レジスタ変数用 saddr 領域を退避する

(b) スタック・ポインタ（SP）をベース・ポインタ（HL）へコピーする

(c) 関数 FUNC 本来の処理を行う

(d) 戻り値をセットする

(e) 退避したレジスタを復帰する

(f) 関数 main へリターンする

例 アセンブリ言語

```
$PROCESSOR ( F1166A0 )
        PUBLIC _FUNC
        PUBLIC _DT1
        PUBLIC _DT2
@@DATA DSEG BASEP
_DT1 : DS ( 2 )
_DT2 : DS ( 4 )
@@CODE CSEG
_FUNC :
        PUSH    HL              ; ベース・ポインタを退避
        PUSH    AX
        MOVW    HL, SP          ; スタック・ポインタをコピー
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; 第1引数 "i" を移動
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]    ; arg2
        MOVW    !_DT2, AX       ; 第2引数 "l" を移動
        MOVW    BC, #0AH        ; 戻り値を設定
        POP     AX
        POP     HL              ; ベース・ポインタを復帰
        RET
        END
```

備考 詳細は,「9.4 C 言語からアセンブリ言語ルーチンの呼び出し」を参照してください。

第 3 章　コンパイラ言語仕様

この章では，RL78,78K0R C コンパイラがサポートするコンパイラ言語仕様について説明します。

3. 1　基本言語仕様

この節では，C コンパイラがサポートする基本言語仕様について説明します。

C コンパイラは，ANSI 規格で規定された言語仕様をサポートしていますが，その中には処理系定義として規定されている項目があります。ここでは，RL78 ファミリ，78K0R マイクロコントローラの処理系に依存した項目の言語仕様について説明します。

また，厳密な ANSI 準拠処理のオプションを指定した場合と指定しない場合の差分についても説明します。

なお，RL78,78K0R C コンパイラで独自に追加されている拡張言語仕様については，「3. 2　拡張言語仕様」を参照してください。

3. 1. 1　処理系依存

この節では，ANSI 規格における処理系依存項目について説明します。

（1）データ型とサイズ

複数バイトで構成されるデータ型の中のバイト順序は，“下位から上位”です。また，符号付き整数は，2 の補数で表現します。最上位ビットには，符号（正，または 0 の場合 0，負の場合 1）が入ります。

- 1 バイト中のビット数は，8 ビットとします。
- オブジェクト中のバイト数，バイト順序，符号化は，次のように規定します。

表 3—1　データ型とサイズ

データ型	サイズ
char 型	1 バイト
int，short 型	2 バイト
long，float，double 型	4 バイト
ポインタ	near　：2 バイト far　：4 バイト

（2）翻訳段階

ANSI 規格では，翻訳における構文規則間の優先順位を 8 つの翻訳段階（翻訳フェーズ）に規定しています。3 段階目の“前処理字句と空白類文字の並びへの分割”で処理系定義となっている，“改行文字以外の空白類文字の空でない並び”は 1 つに置き換えられずそのまま保持されます。

ただし，タブについては -lt オプションの指定した空白文字に置き換えられます。

(3) 診断メッセージ

何らかの構文規則違反，および制約違反を含む翻訳単位に対して，ソース・ファイル名，行番号（特定可能な場合のみ）を含むエラー・メッセージを出力します。なお，エラー・メッセージの書式は“警告”，“致命的エラー”，“その他のエラー”の 3 種類に区別されます。

(4) フリー・スタンディング環境

(a) フリー・スタンディング環境注においては，プログラム開始処理時に呼び出される関数の名前，および型は特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

注 オペレーティング・システムの機能を使用せずに C 言語ソース・プログラムを実行する環境のこと。
ANSI 規格では，実行環境にはフリー・スタンディング環境とホスト環境の 2 つが規定されていますが，RL78,78K0R C コンパイラでは，ホスト環境は現在提供されていません。

(b) フリー・スタンディング環境におけるプログラム終了処理の効果は，特に規定しません。したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(5) プログラムの実行

対話型装置の構成については，特に規定しません。
したがって，ユーザ・オウン・コーディング，またはターゲット・システムに依存します。

(6) 文字集合

実行環境文字集合の要素の値は，ASCII コードです。

(7) 多バイト文字

多バイト文字は，文字定数，文字列ではサポートしていません。
ただし，コメントにおける日本語記述はサポートしています。

(8) 文字表示の意味

拡張表記の値は，次のように規定します。

表 3—2 拡張表記と意味

拡張表記	値（ASCII）	意味
¥ a	07	アラート（警告音）
¥ b	08	バックスペース
¥ f	0C	フォーム・フィード（改ページ）
¥ n	0A	ニュー・ライン（改行）
¥ r	0D	キャリッジ・リターン（復帰）

拡張表記	値（ASCII）	意味
¥ t	09	水平タブ
¥ v	0B	垂直タブ

（9）翻訳限界

次に，翻訳に際しての限界値を示します。

表 3—3 翻訳限界値

内容	限界値
複文，繰り返し制御構造，および選択制御構造の入れ子のレベル数（ただし，“case”のラベル数に依存）	45
条件組み込みの入れ子のレベル数	255
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（任意の組み合わせの）ポインタ，配列，および関数宣言子の数	12
完全な式の中の，かっこで囲まれた式の入れ子のレベル数	1024
マクロ名における有効先頭文字数	256
外部識別子における有効先頭文字数	249
内部識別子における有効先頭文字数	249
1 つの翻訳単位内の外部識別子の数	1024 **注 1**
1 つの基本ブロック内で宣言可能なブロック有効範囲をもつ識別子の数	255
1 つの翻訳単位内で同時に定義可能なマクロ識別子の数	60000
1 つの関数定義内の仮引数，および 1 つの関数呼び出し内の実引数の数	39 **注 1**
1 つのマクロ定義内の仮引数の数	31
1 つのマクロ呼び出し内の実引数の数	31
1 つの論理ソース行内の文字数	32767 **注 1**
連結後の 1 つの文字列定数，またはワイド文字列定数内の文字数	509 **注 1**
1 つのオブジェクト・サイズ（データを示す）	65535
インクルード・ファイルに対する入れ子のレベル数	50
1 つの“switch”文に対する“case”ラベルの数（ネストされている場合，それも含める）	1024
1 コンパイル単位のソース行数	65535 **注 1**
関数コールのネスト	40 **注 1**
1 関数内のラベル数	33
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	メモリ・モデルによる **注 2**
単一構造体，または単一共用体内のメンバ数	1024
単一列挙型における列挙型定数の数	255
単一構造体宣言の並び内の，構造体，または共用体定義の入れ子のレベル数	15
初期化子要素のネスト	15

内容	限界値
1 ソース・モジュール・ファイル中の関数定義数	4095
完全な宣言子の中の，かっこで囲まれた宣言子の入れ子のレベル数	591 注 1
マクロのネスト	10000
インクルード・ファイル・パス指定数	64

注 1. 値は保証値であり，それ以上の値でも可能な場合もありますが，動作は保証されません。

2. メモリ・モデルにより，拡張機能を使用しない場合の最大値は以下のようになります。

メモリ・モデル	最大値
スモール・モデル	コード部 64K バイト，データ部 64K バイトの合計 128K バイト
ミディアム・モデル	コード部 1M バイト，データ部 64K バイトの合計 1M バイト
ラージ・モデル	コード部 1M バイト，データ部 1M バイトの合計 1M バイト

（10）数量的限界

（a）汎整数型の限界値（limits.h ファイル）

汎整数型（char 型，符号付き／符号なし整数型，および列挙型）で表現できる値の各種限界値を limits.h ファイルに定義しています。

なお，多バイト文字はサポートしていないため，MB_LEN_MAX は該当する限界値を持ちません。そこで，MB_LEN_MAX には 1 として，定義のみ行っています。

また，-qu オプションが指定された場合，CHAR_MIN は 0，CHAR_MAX は UCHAR_MAX と同値となります。次に，limits.h ファイルで定義されている各種限界値を示します。

表 3—4 汎整数型の各種限界値（limits.h ファイル）

名前	値	意味
CHAR_BIT	+8	ビット・フィールドではない最小のオブジェクトのビット数（＝1 バイト）
SCHAR_MIN	-128	signed char 型の最小値
SCHAR_MAX	+127	signed char 型の最大値
UCHAR_MAX	+255	unsigned char 型の最大値
CHAR_MIN	-128	char 型の最小値
CHAR_MAX	+127	char 型の最大値
SHRT_MIN	-32768	short int 型の最小値
SHRT_MAX	+32767	short int 型の最大値
USHRT_MAX	+65535	unsigned short int 型の最大値
INT_MIN	-32768	int 型の最小値
INT_MAX	+32767	int 型の最大値
UINT_MAX	+65535	unsigned int 型の最大値

名前	値	意味
LONG_MIN	-2147483648	long int 型の最小値
LONG_MAX	+2147483647	long int 型の最大値
ULONG_MAX	+4294967295	unsigned long int 型の最大値

（b）浮動小数点型の各種限界値（float.h ファイル）

浮動小数点型の特性に関する各種限界値を float.h ファイルに定義しています。

次に，float.h ファイルで定義されている各種限界値を示します。

表 3―5 浮動小数点型の各種限界値の定義（float.h ファイル）

名前	値	意味
FLT_ROUNDS	+1	浮動小数点加算に対する丸めのモード RL78 ファミリ，78K0R マイクロコントローラでは，1（もっとも近い方向へ丸める）とします。
FLT_RADIX	+2	指数表現の基数（b）
FLT_MANT_DIG	+24	浮動小数点仮数部における FLT_RADIX を底とする数字の数（p）
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	+6	q 桁の 10 進数の浮動小数点数を基数 b の p 桁をもつ浮動小数点数に丸めることができ，再び変更なしに q 桁の 10 進数値に戻すことができるような 10 進数の桁数**注1**（q）
DBL_DIG		
LDBL_DIG		
FLT_MIN_EXP	-125	FLT_RADIX をその値から 1 引いた値でべき乗したとき，正規化された浮動小数点数となるような最小の負の整数 （e_{min}）
DBL_MIN_EXP		
LDBL_MIN_EXP		
FLT_MIN_10_EXP	-37	10 をその値でべき乗したとき，正規化された浮動小数点数の範囲内になるような最小の負の整数 $\log_{10} b^{e_{min-1}}$
DBL_MIN_10_EXP		
LDBL_MIN_10_EXP		
FLT_MAX_EXP	+128	FLT_RADIX をその値から 1 引いた値でべき乗したとき，表現可能な有限浮動小数点数となるような最大の整数 （e_{max}）
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	+38	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP		
LDBL_MAX_10_EXP		
FLT_MAX	3.40282347E + 38F	表現可能な有限浮動小数点数の最大値 $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX		
LDBL_MAX		

名前	値	意味
FLT_EPSILON	1.19209290E - 07F	指定された浮動小数点型で表現できる 1.0 と，1.0 より大きい最も小さい値との差異注2 b^{1-p}
DBL_EPSILON		
LDBL_EPSILON		
FLT_MIN	1.17549435E - 38F	正規化された正の浮動小数点数の最小値 $b^{e_{min}-1}$
DBL_MIN		
LDBL_MIN		

注 1. DBL_DIG，LDBL_DIG は，ANSI 規格では，10 以上となっていますが，RL78 ファミリ，78K0R マイクロコントローラでは，double 型も long double 型も 32 ビットであるため 6 となります。

2. DBL_EPSILON と LDBL_EPSILON は，ANSI 規格では，1E - 9 以下となっていますが，RL78 ファミリ，78K0R マイクロコントローラにおいては 1.19209290E - 07F となります。

（11）識別子

識別子で認識することができるのは，最初の 249 文字です。

なお，英字の大文字と小文字は区別されます。

（12）char 型

型指定子（signed，unsigned）の付かない単なる char 型は，符号付き整数として扱います。

ただし，C コンパイラの -qu オプションを指定することにより，符号なし整数として扱うこともできます。

ANSI 規格において要求されるソース・プログラムの文字集合に含まれないもの（エスケープ・シーケンス）は，char 型以外を char 型へ代入する場合と同様に，型変換して格納されます。

```
char    c = '¥777';     /* c の値は -1 となる */
```

（13）浮動小数点定数

浮動小数点定数は，IEEE754 注に準拠しています。

注 IEEE：Institute of Electrical and Electronics Engineers（電気電信学会）の略称です。
また，IEEE754 とは，浮動小数点演算を扱うシステムにおいて，扱うデータ形式や数値範囲などの仕様の統一化を図った標準です。

（14）文字定数

（a）ソース・プログラムの文字集合と実行環境における文字集合は，基本的に両者とも ASCII コードで，同一の値をもつメンバと対応します。

（b）2 つ以上の文字を含む整数文字定数の値は，すべての文字が有効値となります。

（c）基本的な実行環境文字集合で表現されない文字やエスケープ・シーケンスを含む場合，次のようになります。

- 8 進数エスケープ・シーケンス，および 16 進数エスケープ・シーケンスは，その 8 進数表記，および 16 進数表記で示される値となります。

¥077	63

- 単純エスケープ・シーケンスは，次のようになります。

¥'	'
¥"	"
¥?	?
¥¥	¥

- ¥a，¥b，¥f，¥n，¥r，¥t，¥v については，「（8）文字表示の意味」で示されている値と同値になります。

（d）多バイト文字の文字定数はサポートしていません。

（15）ヘッダ・ファイル名

ヘッダ・ファイル名の 2 つの形式（< >，" "）内の列をヘッダ・ファイル，または外部ソース・ファイル名に反映する方法は，「（32）ヘッダ・ファイル取り込み」で規定します。

（16）コメント

コメント中に日本語が記述できます。デフォルトの文字コードは，シフト JIS となります。

入力ソース・ファイルの中の文字コードは，C コンパイラの -z オプション，または環境変数で選択できます。

オプション指定は環境変数よりも優先されます。ただし，none を指定すると，文字コードは保証されません。

（a）オプション指定

```
-ze | -zn | -zs
```

（b）環境変数

```
LANG78K [ euc | none | sjis ]
```

なお，設定方法は，使用する環境の設定方法に従います。

（17）符号付き定数と符号なし定数

汎整数型の値がよりサイズの小さい符号付き整数に変換される場合，上位ビットを切り捨てて，ビット列イメージをコピーします。

また，符号なし整数が，対応する符号付き整数に変換される場合，内部表現は変化しません。

（18）浮動小数点と汎整数

汎整数型の値が浮動小数点型に型変換される際，型変換される値が，表現しうる値の範囲内にはあるが正確に表現することができない場合，その結果は，表現しうる最も近い値へ丸められます。

なお，結果がちょうど中央の値である場合には，偶数（仮数の最下位ビットが 0 のもの）に丸められます。

（19）double 型と float 型

RL78 ファミリ，78K0R マイクロコントローラ処理系では，double 型は float 型と同じ浮動小数点表現であり，32 ビット・データ（単精度）として扱われます。

（20）ビット単位の演算子における符号付き型

ビット単位の演算子における符号付き型に対する特性は，シフト演算子については，「（26）ビット単位のシフト演算子」の規定に準じます。

また，その他の演算子については，符号なしの値として（ビット・イメージのままで）計算するものとします。

（21）構造体と共用体のメンバ

共用体のメンバの値がそれと異なるメンバに格納される場合，整列条件に従って格納されるため，その共用体のあるメンバへのアクセスは，整列条件に従って行われます（「（b）構造体型」，および「（c）共用体型」を参照）。

ただし，共通の先頭メンバの並びを共有している構造体だけをメンバとして含んでいる共用体の場合，内部表現は同じであるため，どの構造体の共通の先頭メンバを参照しても同じになります。

（22）sizeof 演算子

“sizeof”演算子の結果は，「（1）データ型とサイズ」におけるオブジェクト中のバイトに関する規定に準じます。

なお，構造体と共用体については，パディング領域を含んだバイト数とします。

（23）キャスト演算子

ポインタを汎整数型に変換する場合，要求される変数のサイズは，次のサイズです。変換結果は，ビット列がそのまま保存されます。

また，任意の整数はポインタに型変換できますが，int 型よりも小さい整数の場合，結果はその型に従って拡張されます。

- near：2 バイト
- far：　4 バイト

near ポインタ，または int から far ポインタへのキャスト，near ポインタから long へのキャストは，以下の動作となります。

- 変数ポインタは，上位に 0xf を追加します（0 は例外で，0 拡張します）。
- 関数ポインタは，0 拡張します。

（24）乗除／剰余演算子

整数同士の除算で割り切れず，オペランドが負の値をもつ場合，“/”演算子の結果は，除数，または被除数のいずれか一方が負の場合は，代数的な商よりも大きい最小の整数となります。

ただし，どちらも負の場合は，代数的な商よりも小さい最大の整数となります。

また，オペランドが負の値をもつ場合，“%”演算子の結果の符号は第一オペランドの符号とします。

（25）加減演算子

同一配列の要素を指す 2 つのポインタが減算される場合，結果の型は int 型とし，サイズは 2 バイトとします。

（26）ビット単位のシフト演算子

“E1 >> E2”において，E1 が符号付きの型で負の値をもつ場合，算術シフトを行います。

（27）記憶域クラス指定子

記憶域クラス指定子“register”の宣言は，可能なかぎり高速にアクセスするために行いますが，必ずしも有効であるとはかぎりません。

（28）構造体と共用体指定子

（a）signed，unsigned の付かない int 型ビット・フィールドは，符号なしとして扱います。

（b）ビット・フィールドを保持するために，十分な大きさの任意のアドレス付け可能な記憶域単位を割り付けることができますが，十分な領域がなかった場合，合わなかったビット・フィールドはフィールドの型の整列条件に合わせて次の単位に詰め込まれます。

（c）単位内のビット・フィールドの割り付け順序は下位から上位です。
ただし，-rb オプションにより，割り付け順序を上位から下位にすることができます。

（d）1 つの構造体，または共用体の非ビット・フィールドの各メンバは，次のように境界整列されます。

- char，unsignd char 型，およびその配列： バイト境界
- その他（ポインタを含む）： 2 バイト境界

（29）列挙型指定子

列挙型は，次の型の中ですべての列挙定数を表現可能な最初のものとなります。

- signed char
- unsigned char
- signed int

（30）型修飾子

“volatile”修飾された型をもつデータへのアクセスは，データがマッピングされているアドレス（I/O ポートなど）に依存します。

（31）条件組み込み

（a）条件組み込みで指定される文字定数に対する値と，その他の式中に現れる文字定数の値とは等しくなります。

（b）単一文字の文字定数は，負の値を持たないようにしてください。

（32）ヘッダ・ファイル取り込み

（a）“#include < 文字列 >”という形式の前処理指示

“#include < 文字列 >”という形式の前処理指示は，“文字列”が“¥”で始まらない場合[注]，指定されたフォルダ（-i オプション）からヘッダ・ファイルを検索し，次に INC78K0R 環境変数で指定されているフォルダを検索し，最後に，cc78k0r.exe が置かれた bin フォルダからの相対パスでの ..¥inc78k0r フォルダを検索します。

なお，“<”と“>”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “¥”と“/”の両者がフォルダの区切りとしてみなされます。

例

```
#include        <header.h>
```

検索順は，次のとおりです。

- -i で指定したフォルダ
- 環境変数 INC78K0R で指定されているフォルダ
- 標準のフォルダ

(b)“#include " 文字列 "”という形式の前処理指示

“#include " 文字列 "”という形式の前処理指示は，“文字列”が“¥”で始まらない場合[注]，ソース・ファイルがあるフォルダからヘッダ・ファイルを検索し，次に，指定したフォルダ（-i オプション），INC78K0R 環境変数で指定されているフォルダ，最後に cc78k0r.exe が置かれた bin フォルダからの相対パスでの ..¥inc78k0r フォルダを検索します。

なお，“ " ”“ " ”の区切り記号の間に指定された文字列で一意に識別されるヘッダ・ファイルを探し出すと，そのヘッダ・ファイルの内容全体で置き換えます。

注 “¥”と“/”の両者がフォルダの区切りとしてみなされます。

例

```
#include        "header.h"
```

検索順は，次のとおりです。

- ソース・ファイルがあるフォルダ
- -i で指定したフォルダ
- 環境変数 INC78K0R で指定されているフォルダ
- 標準のフォルダ

(c)“#include 前処理字句列”という形式

“#include 前処理字句列”という形式において，前処理字句列が単一で < 文字列 >，または " 文字列 " の形式に置換されるマクロである場合にのみ，単一のヘッダ・ファイル名の前処理字句として扱われます。

(d)“#include < 文字列 >”という形式の前処理指示

（最終的に）区切られた列とヘッダ・ファイル名との間においては，列中の英文字の長さを判別し，

```
コンパイラ動作環境において有効なファイル名長までが有効
```

となります。ファイルを探すフォルダについては，上記の規定に準じます。

(33) #pragma 指令

#pragma 指令は，ANSI でサポートされている前処理指令の 1 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を続行します。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。

（34）あらかじめ定義されたマクロ名

次に，サポートしているマクロ名を示します。

なお，“__”で終わらないマクロは，従来の C 言語仕様（K&R 仕様）のために提供しているものです。ANSI 規格に厳密な処理を行う場合，前後に“__”のある形式のマクロを利用するようにしてください。

表 3―6 サポートしているマクロ

マクロ名	定義
__LINE__	その時点でのソース行の行番号（10 進数）
__FILE__	仮定されたソース・ファイルの名前（文字列定数）
__DATE__	ソース・ファイルの翻訳日付（“Mmm dd yyyy”の形式をもつ文字列定数 ここで，月の名前は ANSI 規格で規定されている asctime 関数で生成されるもの（英字 3 文字の並びで最初の 1 文字のみ大文字）と同じもの。dd の最初の文字は値が 10 より小さい場合空白とします）
__TIME__	ソース・ファイルの翻訳時間（asctime 関数で生成される時間と同じような“hh : mm : ss”の型式をもつ文字列定数）
__STDC__	10 進定数 1（ANSI に厳密な処理を指定時に定義）注
__K0R__	10 進定数 1
__K0R_SMALL__	10 進定数 1（スモール・モデル指定時）
__K0R_MEDIUM__	10 進定数 1（ミディアム・モデル指定時）
__K0R_LARGE__	10 進定数 1（ラージ・モデル指定時）
__CHAR_UNSIGNED__	10 進定数 1（-qu オプションを指定した場合に定義）
__RL78__	10 進定数 1（RL78 ファミリのデバイス種別指定時）
__RL78_1__	10 進定数 1（乗除積和算拡張命令非搭載の RL78 ファミリでレジスタ・バンクありのデバイス種別指定時）
__RL78_2__	10 進定数 1（乗除積和算拡張命令搭載の RL78 ファミリのデバイス種別指定時）
__RL78_3__	10 進定数 1（乗除積和算拡張命令非搭載の RL78 ファミリでレジスタ・バンクなしのデバイス種別指定時）
__CA78K0R__	10 進定数 1
CPU マクロ	ターゲット CPU を示すマクロで 10 進定数 1 デバイス・ファイル中の「品種指定名」で示される文字列の先頭に”__”と末尾に“_”を付けたものが定義されます（英字は大文字で記述してください）

注 -za オプション指定時に定義します。

（35）特別なデータ型の定義

次に，stddef.h ファイルにおける NULL，size_t，ptrdiff_t の定義を示します。

表 3—7 NULL，size_t，ptrdiff_t の定義（stddef.h ファイル）

NULL ／ size_t ／ ptrdiff_t	定義
NULL	((void *) 0)
size_t	unsigned int
ptrdiff_t	int

3. 1. 2 データの内部表現と領域

RL78,78K0R C コンパイラが扱うデータのそれぞれの型における，内部表現と領域について説明します。

（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

- 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，オーバフローが生じるとオーバフローした部分は無視されます。

- 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。

- 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。

- 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，RL78,78K0R C コンパイラでは，double，long double 型は，float と同様に ANSI/IEEE754-1985 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 3—8 型による値の範囲

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

備考 1. signed は省略可能です。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。

2. 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。

3. 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。

4. 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。

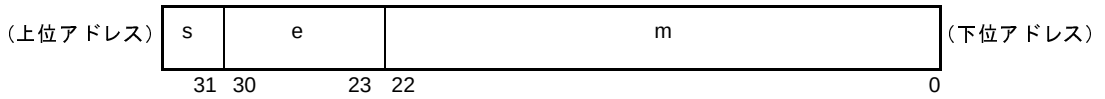
5. float，double，long double の値の範囲は，絶対値の範囲です。

浮動小数点数（float 型）の仕様を以下に示します。

- 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。

図 3—1 浮動小数点数のフォーマット



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） 正数の場合は 0，負数の場合は 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。これらの関係を次に示します。
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。

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

- 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 = 0FFH，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 = 0FFH，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 3—9 演算例外

例外	戻り値
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）

各例外発生時は，matherr 関数を呼び出して警告します。

（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 5 種類の型があります。

- 配列型
- 構造体型
- 共用体型
- 関数型
- ポインタ型

（a）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，不完全型の配列を作ることはできません。

（b）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，名前によって指定することができます。

備考 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。

（c）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，個別に指定することができます。

(d) 関数型

関数型は，指定された型の戻り値を持つ関数を示します。

戻り値の型，引数の数，および引数の型を指定します。

戻り値の型がＴであれば，その関数はＴを返す関数と呼ばれます。

(e) 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。

3.1.3 メモリ

この項では，メモリについて説明します。

(1) メモリ・モデル

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。

メモリ・モデルには，次のものがあります。

表 3—10 メモリ・モデル

メモリ・モデル	説明
スモール・モデル (-ms オプション指定時)	コード部最大 64K バイト，データ部 64K バイトの合計 128K バイト
ミディアム・モデル (-mm オプション指定時)	コード部最大 1M バイト，データ部最大 64K バイトの合計 1M バイト
ラージ・モデル (-ml オプション指定時)	コード部最大 1M バイト，データ部最大 1M バイトの合計 1M バイト

なお，データ部には，ROM データを含みます。上記は，拡張機能を使用しない場合のサイズとなります。

(2) レジスタ・バンク

- スタートアップ時に，レジスタ・バンクが“RB0”に設定されます（RL78,78K0R C コンパイラのスタートアップ・ルーチンの中で設定されています）。この設定により，通常（レジスタ・バンクの変更をしないかぎり）レジスタ・バンク 0 は，常に使用されます。
- レジスタ・バンク変更指定をした割り込み関数の先頭で，指定されたレジスタ・バンクに設定されます。

(3) メモリ空間

RL78,78K0R C コンパイラは，次のようにメモリ空間を利用します。

図 3—2 メモリ空間の利用

アドレス		用途		サイズ（バイト）
00	080 - 0BFH	CALLT テーブル		64
FF	E20 - EB3H	sreg 変数，boolean 型変数		148
FF	EB4 - EC3H	レジスタ変数		16
FF	EC4 - ED3H	コンパイラ予約領域		16
FF	ED4 - ED7H	セグメント情報格納		4
FF	ED8 - EDFH	ランタイム・ライブラリの引数		8
FF	EE0 - EF7H	RB3 - RB1	ワーク・レジスタ 注1	24
	EF8 - EFFH	RB0	ワーク・レジスタ	8
FF	F00 - FFFH	sfr 領域		256
F0	000 - 7FFH	2nd sfr 領域		最大 2048 注2

注 1. レジスタ・バンク指定をしたときに使用します。

2. デバイスにより異なります。

3. 2　拡張言語仕様

この章では，ANSI（American National Standards Institute）規格に規定されていない，RL78,78K0R C コンパイラ特有の拡張機能について説明します。

RL78,78K0R C コンパイラの拡張機能は，ターゲット・デバイスである RL78,78K0R 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らないので，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を「第 2 章　機　　能」で説明しているので，この章とあわせて参照してください。

RL78,78K0R 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することができます。

備考　この節の説明において，“RTOS”は，78K0R リアルタイム OS の意味です。

3. 2. 1　マクロ名

RL78,78K0R C コンパイラは，ターゲット・デバイスのマイクロコントローラ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。以下の例では，__K0R__ と，__F1166A0_ 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，「（34）あらかじめ定義されたマクロ名」を参照してください。

コンパイル時のオプション：

```
>cc78k0r -cf1166a0 prime.c …
```

3. 2. 2　予 約 語

RL78,78K0R 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予約語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，ラベルや変数名として使用することはできません。

予約語は，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予約語と判断されません。

次に，RL78,78K0R C コンパイラで追加されている予約語一覧を示します。これらの予約語のうち，“__”で始まらない予約語は，ANSI-C 言語仕様のみを許可するオプション（-za）指定により，無効にすることができます。

表 3—11　追加予約語一覧

追加予約語		用途
常に有効	-za オプション指定時は無効	
__callt	callt	callt 領域を利用した関数呼び出し
__callf	callf	78K0 互換用
__sreg	sreg	saddr 領域に変数割り当て
—	noauto	78K0 互換用

追加予約語		用途
常に有効	-za オプション指定時は無効	
__leaf	norec	78K0 互換用
__boolean	boolean	saddr，sfr 領域へのビット・アクセス
−	bit	saddr，sfr 領域へのビット・アクセス
__interrupt	−	ハードウエア割り込み
__interrupt_brk	−	ソフトウエア割り込み
__asm	−	ASM 文
__rtos_interrupt	−	RTOS 用割り込みハンドラ
__pascal	−	78K0 互換用
__flash	−	ファーム ROM 関数
__flashf	−	__flashf 関数
__directmap	−	絶対番地配置指定
__temp	−	78K0 互換用
__near，__far	−	メモリ配置領域指定
__mxcall	−	78K0 互換用

(1) 関数

callt，__callt，__interrupt，__interrupt_brk，__rtos_interrupt，__flash，__flashf 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以下に示します。

```
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 ）
```

記述例を以下に示します。

```
__callt  int  func  ( int ) ;
```

修飾属性子の指定は，次のものにかぎります。

なお，callt と __callt 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，__interrupt，__interrupt_brk，__rtos_interrupt，__flash，__flashf

注意 callf，__callf，noauto，__pascal，__mxcall，norec，__leaf の記述に対しては，ワーニングを出力し無視します。

(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，「sreg 宣言による saddr 領域利用（sreg/__sreg）」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）にのみ指定することができます。

__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，「絶対番地配置指定（__directmap）」を参照してください）。

__near，__far 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，「near/far 領域指定（__near/__far）」を参照してください）。

注意 __temp の記述に対しては，ワーニングを出力し無視します。

3. 2. 3 #pragma 指令

#pragma 指令は，ANSI でサポートされている前処理指令の 1 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ることができます。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。

RL78,78K0R 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可能です。

この指令を使用した拡張機能については，「3. 2. 4 拡張機能の使用方法」を参照してください。

表 3—12 #pragma 指令リスト

#pragma 指令	用途
#pragma sfr	SFR 名を C ソース・レベルで記述する →「sfr 領域利用（#pragma sfr）」参照
#pragma vect #pragma interrupt	割り込み処理を C ソース・レベルで記述する →「割り込み関数（#pragma vect/interrupt）」参照
#pragma di #pragma ei	DI／EI 命令を C ソース・レベルで記述する →「割り込み機能（#pragma DI/EI）」参照
#pragma halt #pragma stop #pragma brk #pragma nop	CPU 制御命令を C ソース・レベルで記述する →「CPU 制御命令（#pragma HALT/STOP/BRK/NOP）」参照
#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 →「コンパイラ出力セクション名の変更（#pragma section）」参照
#pragma name	モジュール名を変更する →「モジュール名変更機能（#pragma name）」参照
#pragma rot	ローテート関数を使用する →「ローテート関数（#pragma rot）」参照

#pragma 指令	用途
#pragma mul	乗算関数を使用する →「乗算関数（#pragma mul）」参照
#pragma div	除算関数を使用する →「除算関数（#pragma div）」参照
#pragma mac	積和演算関数を使用する →「積和演算関数（#pragma mac）」参照
#pragma opc	データ挿入関数を使用する →「データ挿入関数（#pragma opc）」参照
#pragma rtos_interrupt	RI78V4（リアルタイム OS）用割り込みハンドラを C ソース・レベルで記述する →「RTOS 用割り込みハンドラ（#pragma rtos_interrupt）」参照
#pragma rtos_task	RI78V4（リアルタイム OS）用タスクを C ソース・レベルで記述する →「RTOS 用タスク（#pragma rtos_task）」参照
#pragma ext_func	ブート領域からフラッシュ領域への関数呼び出しを行う →「ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）」参照
#pragma inline	標準ライブラリ関数 memcpy，memset をインライン展開する →「メモリ操作関数（#pragma inline）」参照

3. 2. 4 拡張機能の使用方法

拡張機能には，次のものがあります。

表 3—13 拡張機能一覧

拡張機能	概要
callt 関数（callt/__callt）	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・コードを短縮することができます。
レジスタ変数（register）	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・コードを短縮することができます。
sreg 宣言による saddr 領域利用（sreg/__sreg）	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・コードを短縮することができます。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。
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（-rd）	外部変数／外部 static 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・コードを短縮することができます。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。

拡張機能	概要
内部 static 変数の saddr 自動割り当てオプションによる利用（-rs）	内部 static 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・コードを短縮することができます。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。
sfr 領域利用（#pragma sfr）	特殊機能レジスタ（sfr）を sfr の略号（sfr 名）によって C ソース・ファイル中で使用することができます。
bit 型変数（bit），boolean 型変数（boolean/__boolean）	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 領域へビット・アクセスすることができます。 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。
ASM 文（#asm ～ #endasm/__asm）	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。
漢字（/* 漢字 */，// 漢字）	C ソース・ファイルのコメント中に，漢字を記述することができます。 漢字コードには，シフト JIS コード，EUC コードを選択することができます。また，漢字コードなしも選択することができます。
割り込み関数（#pragma vect/interrupt）	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。
割り込み関数修飾子（__interrupt/__interrupt_brk）	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することができます。
割り込み機能（#pragma DI/EI）	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。
CPU 制御命令（#pragma HALT/STOP/BRK/NOP）	オブジェクトに次の各命令を埋め込みます。 halt 命令 stop 命令 brk 命令 nop 命令
ビット・フィールド宣言（型指定子の拡張）	ビット・フィールドを unsigned char，signed char，signed int，unsigned short，signed short 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，実行速度の向上を図ることができます。
ビット・フィールド宣言（ビット・フィールドの割り付け方向）	ビット・フィールドの割り付け方向を -rb オプションの指定／無指定により変更します。
コンパイラ出力セクション名の変更（#pragma section）	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することができます。
2 進定数（0b）	C ソース中で，2 進数を記述することができます。
モジュール名変更機能（#pragma name）	C ソース中で，オブジェクト・モジュール名を任意に変更することができます。
ローテート関数（#pragma rot）	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。

拡張機能	概要
乗算関数（#pragma mul）	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，実行速度を向上することができます。
除算関数（#pragma div）	78K0 コンパイラと互換性があり，除算命令の入出力のデータ・サイズを生かしたコードを出力します。 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を生成することができます。
積和演算関数（#pragma mac）	積和演算式の記述より，実行スピードが速く，かつサイズが小さいコードを生成することができます。
BCD 演算関数（#pragma bcd）	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。
データ挿入関数（#pragma opc）	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。
RTOS 用割り込みハンドラ（#pragma rtos_interrupt）	RI78V4（リアルタイム OS）用の割り込みハンドラを記述することができます。
RTOS 用割り込みハンドラ修飾子（__rtos_interrupt）	RI78V4（リアルタイム OS）用の割り込みハンドラ記述とベクタ設定を別ファイルにするための修飾子です。
RTOS 用タスク（#pragma rtos_task）	#pragma 指令により，指定された関数を RI78V4（リアルタイム OS）用のタスクと解釈します。 これにより，C ソース・レベルで，コード効率の良いリアルタイム OS 用タスクの記述が可能となります。
フラッシュ領域配置方法（-zf）	コンパイル時に -zf オプションを指定することにより，プログラムをフラッシュ領域に配置したり，-zf オプションを指定せずに作成したブート領域のオブジェクトと結合して使用できるようになります。
フラッシュ領域分岐テーブル，フラッシュ領域指定（-zt/-zz）	フラッシュ領域分岐テーブルの先頭アドレスを -zt オプションにより指定することにより，スタートアップ・ルーチン，割り込み関数をフラッシュ領域に配置したり，ブート領域からフラッシュ領域への関数呼び出しを行うことができます。
ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）	ブート領域から呼び出すフラッシュ領域中の関数名，および ID 値を #pragma 指令により指定することにより，ブート領域からフラッシュ領域中の関数を呼び出せるようになります。
ミラー元領域指定（-mi0/-mi1）	コンパイル時に，-mi0/-mi1 オプションを指定することにより，ミラー元の領域を考慮したコードを出力することができます。
引数／戻り値の int 拡張抑制方法（-zb）	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，実行速度の向上を図ることができます。
メモリ操作関数（#pragma inline）	#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，実行速度の向上を図ることができます。
絶対番地配置指定（__directmap）	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することができ，同じアドレスに複数の変数を重ねて配置することができます。
near/far 領域指定（__near/__far）	関数，変数の宣言時に，__near/__far 型修飾子を追加することにより，関数，変数の配置場所を明示的に指定することができます。

拡張機能	概要
メモリ・モデル指定（-ms/-mm/-ml）	コンパイル時に，-ms/-mm/-ml オプションを指定することにより，関数，変数の配置場所をメモリ・モデルで指定することができます。
ROM データ配置先指定（-rf/-rn）	ROM データを far 領域，または near 領域の任意の領域に配置することができます。
セルフ・プログラミングにおける RAM 配置先指定（-zx）	コードおよび ROM データを RAM 領域に配置することができます。

callt 関数（callt/__callt）

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。

［機能］

- callt 命令は，callt テーブルと呼ばれる領域 [80H ～ BFH] 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・コードを短縮することができます。

［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
```

［制限］

- callt 関数は，メモリ・モデルによらず，[C0H ～ 0FFFFH] の領域に配置します。
- 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。
- -zf オプション指定時は，callt 関数は定義できません。定義した場合は，エラーとなります。
- callt テーブルは 80H ～ BFH 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，次に示すとおりとなります。

オプション	-ql1	-ql2 ～ -ql3
callt 属性の関数の数	32	30

- -ql オプション未使用時，およびデフォルトの制限値は，次のようになります。

callt 関数	制限値
1 ロード・モジュール当たりの個数	最大 32
リンクするモジュールでトータルの個数	最大 32

[使用例]

（C ソース）

```
============ ca1.c ============
__callt extern int      tsub ( void ) ;

void main ( void ) {
        int     ret_val ;
        ret_val = tsub ( ) ;
}
```

```
 ============ ca2.c ============


__callt int     tsub ( void ) {
        int val ;
        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
@@BASE  CSEG    BASE
_tsub :                         ; 関数定義
                :
                                ; 関数本体
                :
```

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]

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 予約語 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。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #define を用います。詳細については，「3. 2. 5　C ソースの修正」を参照してください。

レジスタ変数（register）

変数をレジスタ，saddr 領域に割り当てます。

［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。
- レジスタ変数の割り当て方法の詳細については，「3. 3　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，参照頻度順にレジスタ HL，saddr 領域 [FFEB4H ～ FFEC3H] 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。

［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，実行速度の向上を図ることができます。

［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
```

［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することができます。
- char は他の型に対して 1/2 の領域を long/float/double/long double/far ポインタは 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，near 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，near 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。

［使用例］

C ソースを以下に示します。

```
void    func ( ) ;

void main ( ) {
        register int    i, j ;
        i = 0 ;
        j = 1 ;
        i += j ;
        func ( ) ;
}
```

(1) 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

次のラベルはスタートアップ・ルーチンで宣言されます（「3. 4 saddr 領域のラベル一覧」を参照してください）。

コンパイラの出力オブジェクトは，以下のようになります。

```
        EXTRN   _@KREG00        ; 使用する saddr 領域の参照を行う
@@CODEL CSEG
_main :
        push    hl              ; 関数の先頭でレジスタの内容を退避する
        movw    ax, _@KREG00    ; 関数の先頭で saddr の内容を退避する
        push    ax
; line  3 :     register int i, j ;
; line  4 :     i = 0 ;
; line  5 :     j = 1 ;
        movw    hl, #00H        ; 関数中では次のようなコードを出力する
        onew    ax
        movw    _@KREG00, ax    ; j
; line  6 :     i += j ;
        addw    ax, hl
        movw    hl, ax
; line  7 :
        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
        END
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

sreg 宣言による saddr 領域利用（sreg/__sreg）

sreg 宣言，あるいは __sreg 宣言された外部変数，および関数内 static 変数を saddr 領域に割り当てます。

[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 領域 [FFE20H ～ FFEB3H] にリロケータブルに割り当てられます。前記領域を越える場合は，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 [FFE20H ～ FFF1FH] です。ただし，[FFEB4H ～ FFEDFH] はコンパイラが使用するので，注意が必要です（「図 3—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 変数にすることができます。

```
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。関数内でも記述することができます。

```
extern sreg     型名    変数名 / extern __sreg   型名    変数名
```

[制限]

- const 型，または関数に sreg/__sreg を指定した場合は，警告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/far ポインタ型は 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，near ポインタの場合で 1 ロード・モジュールあたり 74 変数まで使用可能とします（saddr 領域［FFE20H ～ FFEB3H ］を使用した場合）。
  ただし 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 ソースを以下に示します。

```
extern sreg     int     hsmm0 ;
extern sreg     int     hsmm1 ;
extern sreg     int     *hsptr ;

void main ( ) {
        hsmm0 -= hsmm1 ;
}
```

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，RL78,78K0R 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 )
```

関数中では，次のようなコードを出力します。

```
movw    ax, _hsmm0
subw    ax, _hsmm1
movw    _hsmm0, ax
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 予約語 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。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #define を用います。詳細については，「3. 2. 5　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。

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（-rd）

外部変数／外部 static 変数を saddr 領域に割り当てます。

［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* の値と m の指定により，割り当てる外部変数，外部 static 変数を次のように指定することができます。

n，m の指定	saddr 領域に割り当てる変数
n	（1）*n* = 1 の場合 char，unsigned char 型の変数 （2）*n* = 2 の場合 *n* = 1 の場合の変数と short，unsigned short，int，unsigned int，enum，near ポインタ型の変数 （3）*n* = 4 の場合 *n* = 2 の場合の変数と long，unsigned long，float，double，long double，far ポインタ型の変数
m	構造体，共用体，配列
省略した場合	すべての変数

- 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「sreg 宣言による saddr 領域利用（sreg/__sreg）」で記述したとおりとなります。

［方法］

- -rd[*n*][m]（*n* は 1，2，または 4）オプションを指定します。

［制限］

- -rd[*n*][m] オプションで異なる *n*，m を指定したモジュール同士は，リンクすることはできません。

内部 static 変数の saddr 自動割り当てオプションによる利用（-rs）

内部 static 変数を saddr 領域に割り当てます。

［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* の値と m の指定により，割り当てる内部 static 変数を次のように指定することができます。

n，m の指定	saddr 領域に割り当てる変数
n	（1）*n* = 1 の場合 char，unsigned char 型の変数 （2）*n* = 2 の場合 *n* = 1 の場合の変数と short，unsigned short，int，unsigned int，enum，near ポインタ型の変数 （3）*n* = 4 の場合 *n* = 2 の場合の変数と long，unsigned long，float，double，long double，far ポインタ型の変数
m	構造体，共用体，配列
省略した場合	すべての変数

- 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「sreg 宣言による saddr 領域利用（sreg/__sreg）」で記述したとおりとなります。

［方法］

- -rs[*n*][m]（*n* は 1，2，または 4）オプションを指定します。

備考　-rs[*n*][m] オプションで異なる *n*，m を指定したモジュール同士も，リンクすることができます。

sfr 領域利用（#pragma sfr）

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

［機能］

- sfr 領域は，RL78,78K0R 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・レベルで記述することができます。
- 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，[FFF00H ～ FFFFFH，および F0000H ～ F07FFH 注] 中に割り付けてあるものです。

注　デバイスにより異なります。

［効果］

- sfr 領域に関する操作を C ソース・レベルで記述することができます。
- 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ることができます。

［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可能です）。

```
#pragma sfr
```

#pragma sfr は，C ソースの先頭に記述します。
次のものは，#pragma sfr の前に記述することができます。
 - 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。

［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 ソースを以下に示します。

```
#ifdef  __K0R__
#pragma sfr
#endif

void main ( void ) {
        PL0 -= ADCR ;
        /* ADCR = 10 ;  ==> error */
}
```

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, PL0
sub     a, ADCR
mov     PL0, a
```

[互換性]

(1) 他の C コンパイラから RL78,78K0R 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。

(2) RL78,78K0R 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  __K0R__
#pragma sfr
#else

unsigned char   P0 ;    /* 変数の宣言 */
#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。

bit 型変数（bit），boolean 型変数（boolean/__boolean）

1 ビットの記憶領域を持つ変数を生成します。

［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。

```
MOV1, AND1, OR1, XOR1, SET1, CLR1, NOT1, BT, BF
```

［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 を記述することも可能です。

```
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）
```

［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されません。
- 配列の定義／参照を行うことはできません。
- 構造体，共用体のメンバとして使用することはできません。
- 関数の引数の型として使用することはできません。
- オートマチック変数の型として使用することはできません。
- bit 型変数のみで，1 ロード・モジュール当たり最大 1184 変数まで使用することができます（saddr 領域 [FFE20H ～ FFEB3H] を使用した場合）。
- 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。
- 次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。

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	!=

- *，&（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。
- sreg 変数を使用した場合と，-rd，-rs（saddr 自動割り当てオプション）指定時には，使用可能な数は減ります。

［使用例］

C ソースを以下に示します。

```
#define ON      1
#define OFF     0

extern bit      data1 ;
extern bit      data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while ( data1 )  {
                data1 = data2 ;
                testb ( ) ;
        }

        if ( data1 && data2 )
                chgb ( ) ;
}
```

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。このときには，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
```

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, $?L0004          （判断）
mov1    CY, _data2              （代入）
mov1    _data1, CY              （代入）
bf      _data1, $?L0005         （論理 AND 式）
bf      _data2, $?L0005         （論理 AND 式）
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 予約語 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。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #define を用います。詳細については，「3. 2. 5　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）。

ASM 文（#asm ～ #endasm/__asm）

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・ソースが埋め込みます。

［機能］

（1）#asm ～ #endasm

- RL78,78K0R 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。

（2）__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・ソースで操作することができます。
- C ソースには記述することができない機能を実現可能です。
- 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を得ることができます。

［方法］

（1）#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
```

（2）__asm

- 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可能です。

［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。
  プロパティパネルの［コンパイル・オプション］タブで「アセンブリ・ファイルを出力する」を「はい」に設定してください。（設定方法は，「CubeSuite+ 統合開発環境 ユーザーズマニュアル RL78,78K0R ビルド編」を参照してください。）
- __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することができません。したがって，アセンブラ・ソースはセグメント名 @@CODE，または @@CODEL の CSEG に出力されます。

［使用例］

（1）#asm ～ #endasm

C ソースを次に示します。

```
void main ( void ) {
#asm
        callt [init]
#endasm
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@CODEL CSEG
_main :
        callt [init]
        ret
        END
```

#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。

（2）__asm

C ソースを次に示します。

```
int     a, b ;

void main ( void ) {
        __asm ( "¥tmovw ax, !_a ¥t ; ax <- a" ) ;
        __asm ( "¥tmovw !_b, ax ¥t ; b <- ax" ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@CODEL 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。

漢字（/* 漢字 */，// 漢字）

C ソースのコメント中に漢字を記述することができます。

[機能]

- C ソースのコメント中に漢字を記述することができます。
- 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することができます。
  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 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 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 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。
- デフォルトは，SJIS を指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。

[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（デフォルトの設定でよい場合は，設定の必要はありません）。

（1）コンパイラ・オプションによる設定

次のオプションのうち，いずれかを指定します。

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）

（2）環境変数 LANG78K による設定

- SJIS，EUC，または NONE のいずれかを設定します。
- SJIS，EUC，NONE は，大文字でも小文字でも記述可能です。
- C ソースのコメント中に漢字（環境変数 LANG78K に SJIS を設定した場合はシフト JIS コード，EUC を設定した場合は EUC コード）を記述します。

```
SET     LANG78K = SJIS   ; シフト JIS コードの場合
SET     LANG78K = EUC    ;EUC コードの場合
SET     LANG78K = NONE   ; 漢字コードなしの場合
```

[制限]

- コメント中に記述することができるのは，シフト JIS コード，EUC コードです。
  コメント以外で記述することができるのは，ASCII コードが 0x7f 以下の文字です。
  具体的には，全角文字のすべて，半角カタカナ（半角の句読点等を含む）をコメント以外には記述することができません。
  なお，記述した場合，意図したコードにならない場合があります。

[使用例]

C ソースを以下に示します。

```
// main 関数
void main ( void ) {
        /* コメント */
}
```

アセンブラ・ソース中に漢字種別情報を出力します。
コンパイラの出力オブジェクトは，以下のようになります。

```
$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 : // main 関数
; line          2 : void main ( void ) {
@@CODEL CSEG
_main :
; line          3 :             /* コメント */
; line          4 : }
```

[説明]

- C ソースのコメント中にのみ漢字を使うことができます。
- “// コメント”を使用する場合は，コンパイラ・オプション -zp を指定してください。

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- コメント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。

（2）RL78,78K0R 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くことができる 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くことができない 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。

割り込み関数（#pragma vect/interrupt）

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。

［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（レジスタ・バンク指定の場合は，そのコードの後ろ）と終わりに出力します。
  - レジスタ
  - レジスタ変数用 saddr 領域
  - ワーク用 saddr 領域
  - ランタイム・ライブラリ用 saddr 領域
  - セグメント情報格納用 saddr 領域
  - ES，CS レジスタ

  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・バンクの変更，またはレジスタの退避／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。
  - レジスタ・バンク指定がある場合は，指定されたレジスタ・バンクに変更するためのコードを割り込み関数の先頭に出力するため，レジスタの退避／復帰は行いません。
  - 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。

- 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，ワーク用の 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，次のようになります。

退避／復帰領域	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	
		スタック	RBn	スタック	RBn	スタック	RBn	スタック	RBn
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，ES，CS レジスタ，セグメント情報格納用 saddr 領域	－	－	－	－	－	○	○	○	○
全ランタイム・ライブラリ用 saddr 領域，ES，CS レジスタ，セグメント情報格納用 saddr 領域	－	○	○	○	○	－	－	－	－
使用レジスタ変数用 saddr 領域	－	－	－	○	○	－	－	○	○
コンパイラ予約領域用 saddr 領域	－	－	－	○注	○注	－	－	－	－

スタック　：スタック使用指定
RBn　　：レジスタ・バンク指定
○　　：退避する
－　　：退避しない

注　スピード優先時（-ql 未指定時）は退避しません。

［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。
- レジスタ・バンクを変更できるため，レジスタの退避処理を行うコードを出力せず，オブジェクト・コードを縮小，実行速度を向上することができます。
- 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，コンパイラが使用するレジスタ，および 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，デバイスのユーザーズ・マニュアルを参照してください）。ただし，ソフトウエア割り込み BRK の場合は，BRK_I と記述してください。
- 次の項目はこの #pragma 指令の前に記述することができます。
  - 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
                                          レジスタ・バンク指定]
```

- *割り込み要求名*

  大文字で記述します。

  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）。

  ただし，ソフトウエア割り込み BRK の場合は，BRK_I と記述してください。

- *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 short で定義してください（例：unsigned short buff [5] ;）。

  オフセット位置は，buff のサイズ以下の偶数の値を指定してください（例：unsigned short buff[5] の場合，サイズは 10 バイトとなるので，10 以下の偶数値）。

- *スタック使用指定*

  STACK（デフォルト）

- *無変更指定*

  NOBANK

- *レジスタ・バンク指定*

  RB0/RB1/RB2/RB3

注意 1. RL78,78K0R C コンパイラのスタートアップ・ルーチンでは，レジスタ・バンク 0 に初期指定されているので，レジスタ・バンク 1 ～ 3 を指定するようにしてください。

2. 関数コールがある割り込み関数の退避／復帰において，スピード優先時（-ql 未指定時）は，-qr 指定時であってもコンパイラ予約領域用 saddr 領域を退避しません。

［制限］

- -zf 無指定時，割り込み関数はメモリ・モデルによらず，[C0H ～ 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- スタック切り替え指定は，near 領域以外の配列を指定することはできません。指定した場合，エラーとなります。
- オフセット位置の指定は，偶数以外を指定することはできません。奇数を指定した場合，エラーとなります。
- スタック・ポインタを切り替えるために確保する配列は，ほかのマイクロコントローラと異なり，unsigned short 型とします。
- 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。
- 以下の 3 つ条件を満たすときに，レジスタの内容を書き換えてしまう可能性がありますが，コンパイラはこれをチェックすることはできません。
  レジスタ・バンク切り換えの設定がある場合は，レジスタ・バンクが重複しないように設定してください。また，レジスタ・バンクが重複するような設定を行う場合は，それらの割り込みが重ならないように，制御してください。
  NOBANK（無変更指定）を指定した場合も，レジスタの退避を行わないので，レジスタを破壊しないように制御する必要があります。
  - 複数の割り込みが発生
  - 発生した割り込みの中に，同じ BANK を使用する割り込みが複数ある
  - #pragma interrupt ～の記述で，NOBANK，またはレジスタ・バンク指定がある
- レジスタ・バンク指定時に指定可能なレジスタ・バンク番号は，デバイスが有するレジスタ・バンク数に依存します。
  ただし，RL78 8 ビット CPU はレジスタ・バンクが 1 つしかないため，レジスタ・バンク指定はできません。
- 割り込み関数は，callt/__callt/__rtos_interrupt/__flash/__flashf を指定することができません。
  __far は，-zf オプション指定時のみ指定可能です。
- 割り込み関数は，引数，戻り値を持つことができないため，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などを使用する場合，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。
- leafwork1 ～ 16 を指定した場合は，ワーニングを出力し，無視します。
- 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 指令では行わないため，別途グローバルの unsigned short 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 予約語を付加した場合，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することはできません。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／レジスタ・バンク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。
- #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，デフォルトのスタックが使用されます。
- -zx 指定時に #pragma vect/#pragma interrupt 指令を記述した場合はエラーとなります。割り込み関数を定義する時は，__interrupt/__interrupt_brk 修飾子を使用してください。RL78 ファミリでは，セルフ・プログラミング・ライブラリを用いて，セルフ・プログラミングにおける割り込みベクタテーブルを配置します。

［使用例］

(1) レジスタ・バンク指定がある場合

C ソースを次に示します。

```
#pragma interrupt INTP0 inter rb1

void    inter ( void ) {
        /* INTP0 端子入力に対する割り込み処理 */
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@VECT08        CSEG    AT      0008H ; INTP0
_@vect08 :
        DW      _inter
@@BASE          CSEG    BASE
_inter :

        ; レジスタ・バンクの切り替えコード
        ; コンパイラが使用する saddr 領域の退避コード
        ; ES，CS レジスタの退避コード
        ; INTP0 端子入力に対する割り込み処理（関数本体）
        ; ES，CS レジスタの復帰コード
        ; コンパイラが使用する saddr 領域の復帰コード
        reti
```

(2) スタック切り替え指定とレジスタ・バンク指定がある場合

C ソースを以下に示します。

```
#pragma interrupt INTP0 inter sp = buff + 10 rb2

unsigned short  buff[5] ;
void    func ( void ) ;

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

コンパイラの出力オブジェクトは，以下のようになります。

【コードサイズ優先時】

```
@@BASE      CSEG    BASE
_inter :
        sel     RB2                     ; レジスタ・バンクの切り替え
        movw    ax, sp                  ; スタック・ポインタの切り替え
        movw    sp, #_buff + 10         ;       〃
        push    ax                      ;       〃
        movw    c, #0CH                 ; コンパイラが使用する saddr の退避
        dec     c                       ;       〃
        dec     c                       ;       〃
        movw    ax, _@SEGAX[c]          ;       〃
        push    ax                      ;       〃
        bnz     $$ - 6                  ;       〃
        mov     a, ES                   ; ES，CS レジスタの退避
        mov     x, a                    ;       〃
        mov     a, CS                   ;       〃
        push    ax                      ;       〃
        call    !!_func
        pop     ax                      ; ES，CS レジスタの復帰
        mov     CS, a                   ;       〃
        mov     a, x                    ;       〃
        mov     ES, a                   ;       〃
        movw    de, #_@SEGAX            ; コンパイラが使用する saddr 領域の復帰
        mov     c, #06H                 ;       〃
        pop     ax                      ;       〃
        movw    [de], ax                ;       〃
        incw    de                      ;       〃
        incw    de                      ;       〃
        dec     c                       ;       〃
        bnz     $$ - 5                  ;       〃
        pop     ax                      ; スタック・ポインタを元に戻す
        movw    sp, ax                  ;       〃
        reti

@@VECT08    CSEG    AT      0008H
_@vect08 :
            DW      _inter
```

【スピード優先時】

```
@@BASE      CSEG    BASE
_inter :
        sel     RB2                     ; レジスタ・バンクの切り替え
        movw    ax, sp                  ; スタック・ポインタの切り替え
```

```
        movw    sp, #_buff + 10         ;       〃
        push    ax                      ;       〃
        movw    ax, _@RTARG6            ; コンパイラが使用する saddr の退避
        push    ax                      ;       〃
        movw    ax, _@RTARG4            ;       〃
        push    ax                      ;       〃
        movw    ax, _@RTARG2            ;       〃
        push    ax                      ;       〃
        movw    ax, _@RTARG0            ;       〃
        push    ax                      ;       〃
        movw    ax, _@SEGDE             ;       〃
        push    ax                      ;       〃
        movw    ax, _@SEGAX             ;       〃
        push    ax                      ;       〃
        mov     a, ES                   ; ES，CS レジスタの退避
        mov     x, a                    ;       〃
        mov     a, CS                   ;       〃
        push    ax                      ;       〃
        call    !!_func
        pop     ax                      ; ES，CS レジスタの復帰
        mov     CS, a                   ;       〃
        mov     a, x                    ;       〃
        mov     ES, a                   ;       〃
        pop     ax                      ; コンパイラが使用する saddr 領域の復帰
        movw    _@SEGAX, ax             ;       〃
        pop     ax                      ;       〃
        movw    _@SEGDE, ax             ;       〃
        pop     ax                      ;       〃
        movw    _@RTARG0, ax            ;       〃
        pop     ax                      ;       〃
        movw    _@RTARG2, ax            ;       〃
        pop     ax                      ;       〃
        movw    _@RTARG4, ax            ;       〃
        pop     ax                      ;       〃
        movw    _@RTARG6, ax            ;       〃
        pop     ax                      ; スタック・ポインタを元に戻す
        movw    sp, ax                  ;       〃
        reti


@@VECT08    CSEG    AT      0008H
_@vect08 :
            DW      _inter
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

割り込み関数修飾子（__interrupt/__interrupt_brk）

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することができます。

［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。
- 関数を __interrupt_brk 修飾子で宣言することにより，その関数はソフトウエア割り込み関数とみなされ，ソフトウエア割り込み関数のためのリターン命令 RETB 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／ソフトウエア）割り込み関数とみなされ，次の（1）～（6）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）ワーク用 saddr 領域

（4）ランタイム・ライブラリ用 saddr 領域

（5）セグメント情報格納用 saddr 領域

（6）ES，CS レジスタ

備考　コンパイル時に -qr オプションを指定しない場合（デフォルト）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。

［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することができます。

[方法]

- 割り込み関数の修飾子に __interrupt/__interrupt_brk のいずれかを付加します。

(1) 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 }
```

(2) ソフトウエア割り込み関数の場合

```
__interrupt_brk void    func ( ) { 処理 }
```

[制限]

- -zf 無指定時，割り込み関数はメモリ・モデルによらず，［C0H ～ 0FFFFH］の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- 割り込み関数は，callt/__callt/__rtos_interrupt/__flash/__flashf を指定することができません。
- -zx 指定時，割り込み関数は -zf オプションの有無やメモリ・モデルによらず，［C0H ～ FFEFFH］の領域に配置します。セルフ・プログラミング・モード時は，セルフ・プログラミング・ライブラリを用いて，割り込みベクタテーブルを配置します。

[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   RB1     /* ソフトウエア割り込みの割り込み */
#pragma interrupt       BRK_I   inter_b RB2     /* 要求名は“BRK_I”です。*/

__interrupt void        inter ( ) ;             /* プロトタイプ宣言 */
__interrupt_brk void    inter_b ( ) ;           /* プロトタイプ宣言 */
__interrupt void        inter ( ) { 処理 } ;     /* 関数本体 */
__interrupt_brk void    inter_b ( ) { 処理 } ;   /* 関数本体 */
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

［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，デフォ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，「割り込み関数（#pragma vect/interrupt）」を参照してください。

割り込み機能（#pragma DI/EI）

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。

[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（オートマチック変数の宣言，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（関数名のラベルの直後）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 の前に記述することができます。
  - 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可能です。

[制限]

- この機能を使用する場合は，関数名として DI，EI を使用することはできません。
- 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  __K0R__
#pragma DI
#pragma EI
#endif
```

C ソースを以下に示します。

```
#pragma DI
#pragma EI

void main ( void ) {
        DI ( ) ;
        ; 関数本体
        EI ( ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
_main :
        di
        ; 前処理
        ; 関数本体
        ; 後処理
        ei
        ret
```

（1）DI，EI を前／後処理の後と前に出力する場合

C ソースを以下に示します。

```
#pragma DI
#pragma EI

void main ( void ) {
        {
                DI ( ) ;
                ; 関数本体
                EI ( ) ;
        }
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
_main :
        ; 前処理
        di
        ; 関数本体
        ei
        ; 後処理
        ret
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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 を使用することができます（例：#ifdef __K0R__ ～ #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。

CPU 制御命令（#pragma HALT/STOP/BRK/NOP）

オブジェクトに halt 命令，stop 命令，brk 命令，nop 命令を埋め込みます。

［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。
  - HALT 動作用の命令（HALT）
  - STOP 動作用の命令（STOP）
  - BRK 命令
  - NOP 命令

［効果］

- マイクロコンピュータのスタンバイ機能を C プログラムで使用することができます。
- ソフトウエア割り込みを発生することができます。
- CPU を動作させずに，クロックを進めることができます。

［方法］

- #pragma HALT，#pragma STOP，#pragma NOP，#pragma BRK 命令を C ソースの先頭に記述します。
- 次の項目は，#pragma 指令の前に記述することができます。
  - 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可能です。
- 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。

```
HALT ( ) ;
STOP ( ) ;
BRK ( ) ;
NOP ( ) ;
```

［制限］

- この機能を使用する場合は，関数名として HALT，STOP，BRK，NOP を使用することができません。
- HALT，STOP，BRK，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 ソースを以下に示します。

```
#pragma HALT
#pragma STOP
#pragma BRK
#pragma NOP

void main ( void ) {
        HALT ( ) ;
        STOP ( ) ;
        BRK ( ) ;
        NOP ( ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@CODEL CSEG
_main :
        halt
        stop
        brk
        nop
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- “#pragma HALT”，“#pragma STOP”，“#pragma BRK”，“#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，BRK，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。

ビット・フィールド宣言（型指定子の拡張）

ビット・フィールドを unsigned char，signed char，unsigned int，signed int，unsigned short，signed short 型で指定することができます。

[機能]

- unsigned char，signed 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，signed int，unsigned short，signed short 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。ただし，-rc オプション指定時は，ワード境界をまたがって割り付けることが可能となります。
- サイズの同じ型のビット・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。
  サイズの違う型の場合は，違うバイト単位（またはワード単位）に割り付けられます。
- unsigned short，signed short 型は，それぞれ unsigned int，signed int 型と同じに扱います。

[効果]

- メモリの節約を図ることができます。

[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，signed char，signed int，unsigned short，signed short 型の指定を行うことができます。
  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］

（1）他の C コンパイラから RL78,78K0R 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，signed char，unsigned short，signed short を使用したい場合は，型指定子を変更します。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- 型指定子に unsigned char，signed char，signed int，unsigned short，signed short 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，signed char，signed int，unsigned short，signed short を使用している場合は，unsigned int に変更します。

ビット・フィールド宣言（ビット・フィールドの割り付け方向）

ビット・フィールドの割り付け方向を -rb オプションの指定／無指定により変更します。

[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。

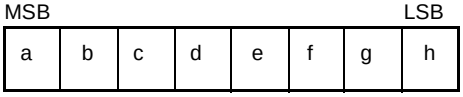
[使用例]

（1）ビット・フィールドの宣言 1

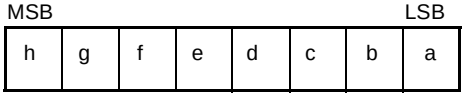
```
struct t {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
        unsigned char   g : 1 ;
        unsigned char   h : 1 ;
} ;
```

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。

-rb オプション指定時の
MSB から割り付けたビット配置



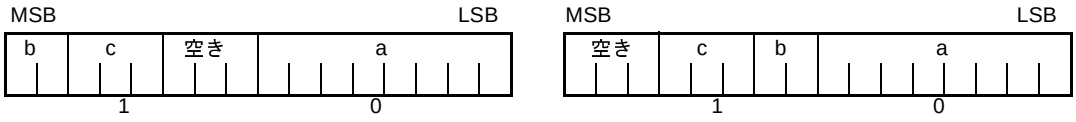
-rb オプション無指定時の
LSB から割り付けたビット配置



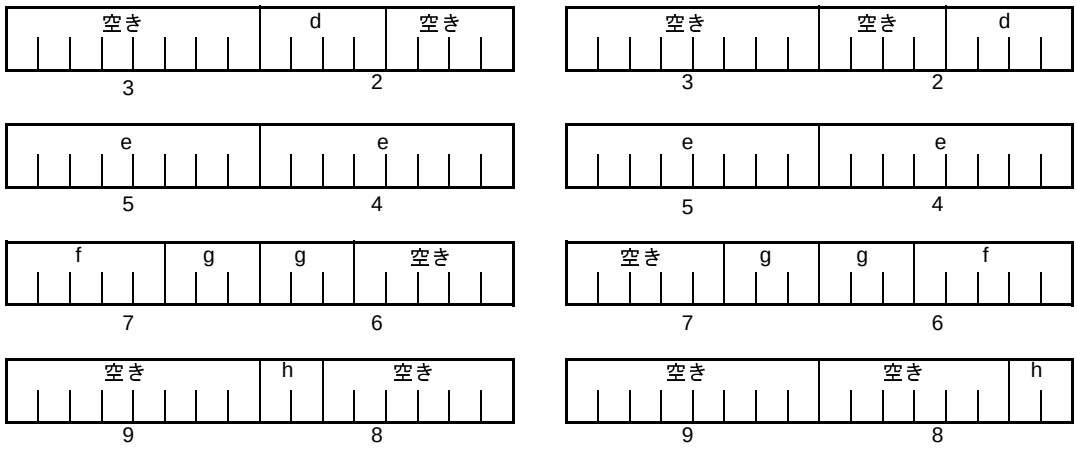
(2) ビット・フィールドの宣言2

```
struct  t {
        char            a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned int    f : 5 ;
        unsigned int    g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 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

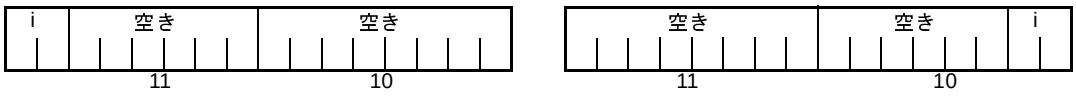


char 型のメンバ a を最初のバイト単位に割り付けます。b，c は次のバイト単位から割り付けます。十分な空きがなくなれば，次のバイト単位に割り付けます。ここでは，空きが 3 ビットで，d が 4 ビットなので，d は次のバイト単位に割り付けます。



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。

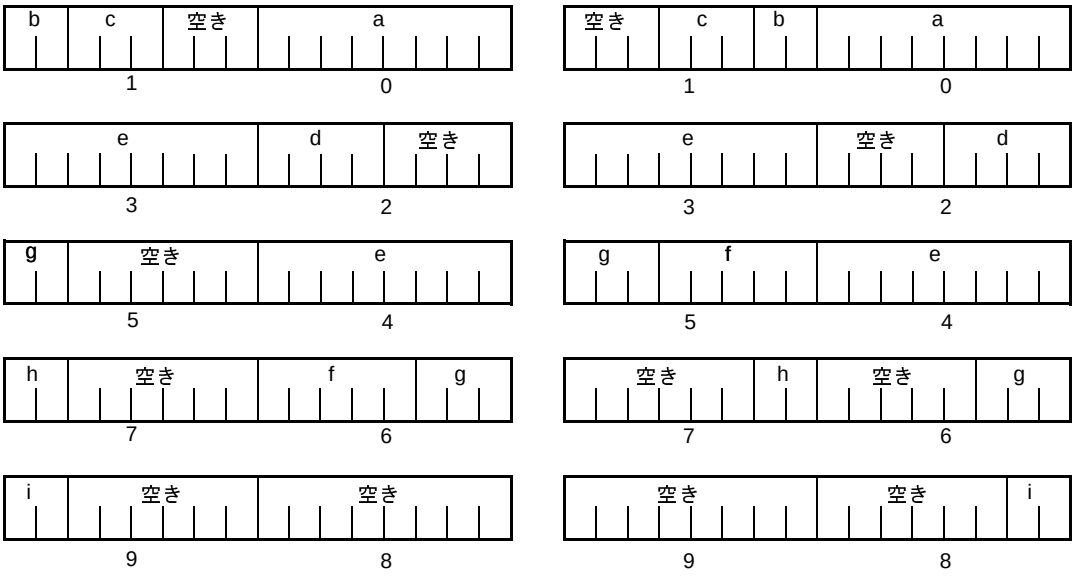
h は unsigned char 型のビット・フィールドなので，unsigned int 型のビット・フィールドの g と同じバイト単位ではなく，次のバイト単位に割り付けます。



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。

なお，コンパイラでは，配列の内部データはポインタとして処理しているため，-rc 指定時はバイトアクセスとなります。

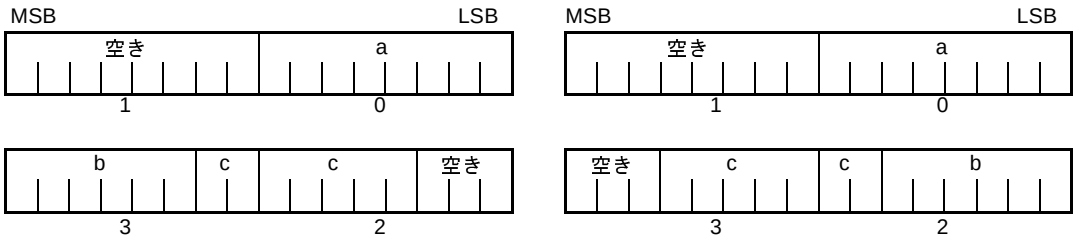


備考 ビット配置図の下の数字は，構造体の先頭からのバイト・オフセット値を示します。

(3) ビット・フィールドの宣言 3

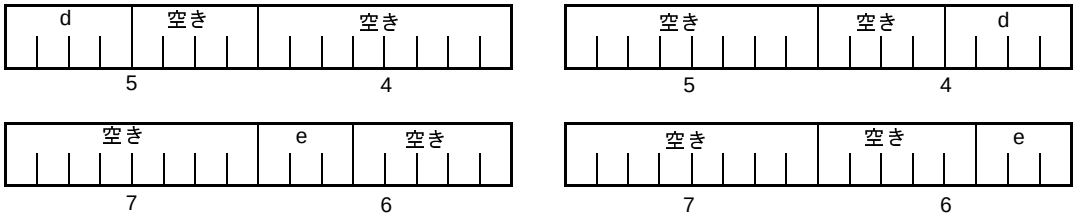
```
struct  t {
        char            a ;
        unsigned int    b : 6 ;
        unsigned int    c : 7 ;
        unsigned int    d : 4 ;
        unsigned char   e : 3 ;
        unsigned int    f : 10 ;
        unsigned int    g : 2 ;
        unsigned int    h : 5 ;
        unsigned int    i : 6 ;
} ;
```

-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

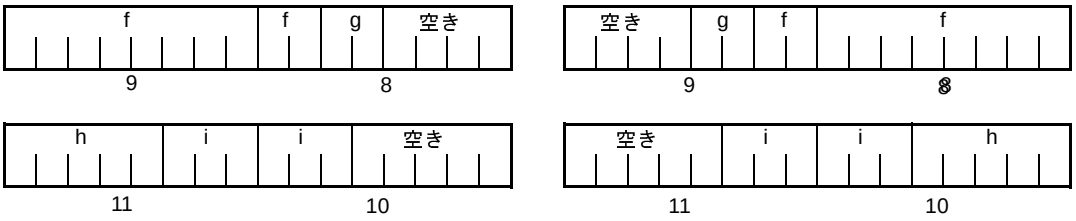


b，c は unsigned int 型のビット・フィールドなので，次のワード単位から割り付けます。

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。



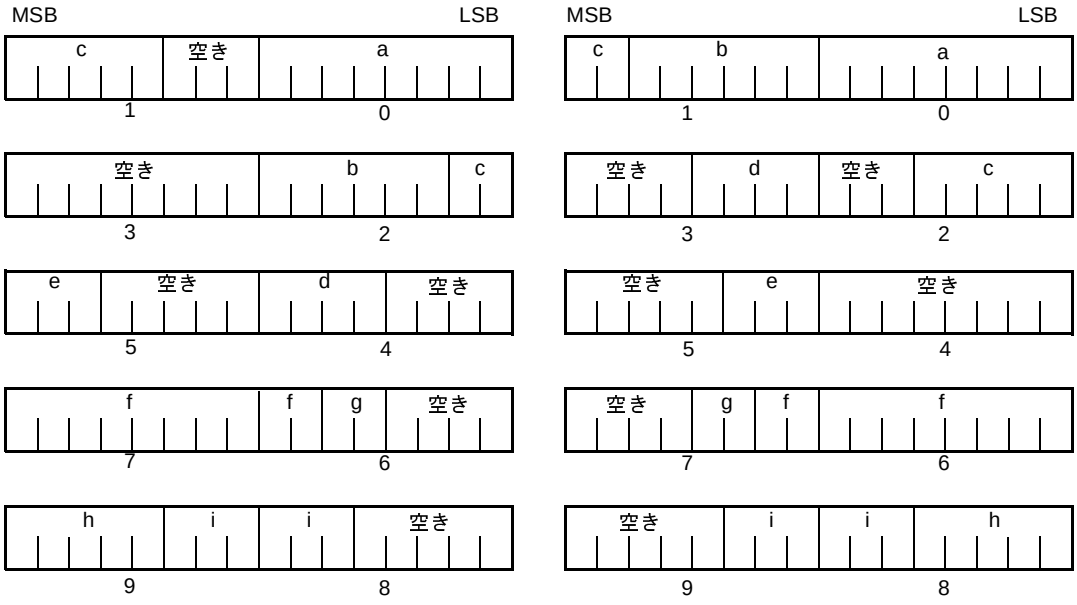
e は unsigned char 型のビット・フィールドなので，次のバイト単位に割り付けます。



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。

なお，コンパイラでは，配列の内部データはポインタとして処理しているため，-rc 指定時はバイトアクセスとなります。



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]

(1) 他の C コンパイラから RL78,78K0R C コンパイラ

- 修正は必要ありません。

(2) RL78,78K0R 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。

コンパイラ出力セクション名の変更（#pragma section）

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。

［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，デフォ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フォルトの配置については，「3.5 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することができます。リンク・ディレクティブについては，「5.1.1 リンク・ディレクティブ」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することが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，データを配置することができます。

［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。
  次の項目は，この #pragma 指令の前に記述することができます。
  - 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，@@CNSTL 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を行うことができます。元のセクション名に戻す場合は，変更セクション名にコンパイラ出力セクション名を記述します。
  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可能です。
- 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 進数および，アセンブラの 16 進数のみ記述することができます。

（1）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 )
```

（2）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。
例えば，値が 255 の数値を 16 進数で表現する場合，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，@@CNSTL 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 ソース記述後）にこの #pragma 指令を記述することはできません。ワーニングを出力し無視します。
- 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。コンパイラのオプションにてアセンブラ・ソース・モジュール・ファイル作成指定“-a, -sa”を行ってください。
- 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）のいっさいないファイルはインクルードすることはできません。エラーになります（後述の「エラー記述例 1」を参照）。
- 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 文を記述することはできません。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 指令を記述することはできません。エラーになります（後述の「エラー記述例 3」を参照）。
  ただし，C の本文がヘッダファイルの中にある場合はエラーになりません。

```
d1.h
        extern int      a ;


d2.h
        #define VAR 1


d.c
        #include "d1.h"                 // C の本文があり，#include の中の場合，
        #include "d2.h"                 // d.c の #pragma 指令はエラーではない
        #pragma section @@DATA ??DATA1
```

［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 など）を変更することはできません。
- 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 の指定アドレスは FFE20H ～ FFEB3H の範囲内に，@@CALT の指定アドレスは 0x80 ～ 0xbf の範囲内に，@@CODE，@@BASE の指定アドレスは 0x0 ～ 0xffff の範囲内に，@@CNST の指定アドレスはミラー領域の範囲内に，その他のセクションの指定アドレスは 0x0 ～ 0xffeff の範囲内にしてください。

［使用例］

セクション名 @@CODEL を CC1 に変更し，開始アドレスを 2400H 番地に指定します。
C ソースを以下に示します。

```
#pragma section @@CODEL CC1     AT      2400H

void main ( void ) {
        ; 関数本体
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
CC1     CSEG    AT      2400H
_main :
        ; 前処理
        ; 関数本体
        ; 後処理
        ret
```

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記述例を示します。
// 以降に配置するセクションを示します。

（1）記述例 1

```
#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，デフォルト @@DATA に戻る
#pragma section @@INIT          ??INIT
#pragma section @@R_INIT        ??R_INIT

int             a4 ;                    // @@DATA
sreg int        b4 ;                    // ??DATS
// @@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             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，デフォルトに戻る
#pragma section @@BITS          ??BITS
__boolean e4 ;                          // ??BITS
#pragma section @@CNST          ??CNST

char            *const p = "Hello" ;    // p も "Hello" も ??CNST
```

（2）記述例 2

```
#pragma section     @@INIT      ??INIT1
#pragma section     @@R_INIT    ??RINIT1
#pragma section     @@DATA      ??DATA1

char        c1 ;
int         i2 ;
#pragma section     @@INIT      ??INIT2
#pragma section     @@R_INIT    ??RINIT2
#pragma section     @@DATA      ??DATA2

char        c1 ;
int         i2 = 1 ;
#pragma section     @@DATA      ??DATA3
#pragma section     @@INIT      ??INIT3
#pragma section     @@R_INIT    ??RINIT3

extern char c1 ;                    // ??DATA1
```

```
int         i2 ;                    // ??INIT1 と ??RINIT1
#pragma section     @@DATA      ??DATA4
#pragma section     @@INIT      ??INIT4
#pragma section     @@R_INIT    ??RINIT4
```

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。

（3）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
                                // 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"
```

（4）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
                                // 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
```

（5）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
                                        // 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
                                        // 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
                                        // 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]

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

［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。
- -zf オプション指定時には，セクション名の先頭から 2 番目の“@”を“E”に変更したセクション名となります。
- #pragma section の使用により ROM 化関連のセクション名[注]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。

注 ROM 化関連セクション名
@@R_INIT, @@R_INIS, @@RLINIT, @@INITL, @@INIT, @@INIS

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），終端ルーチン（rom.asm）の変更例について，次に示します。

C ソースを以下に示します。

```
#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ラベルの宣言と，初期値のコピーを行う部分を追加し，終端ルーチンには終端ラ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 の先頭と終端のラベルの名前であり，TT1_S，TT1_E は，セクション TT1 の先頭と終端のラベルの名前です。

（1）スタートアップ・ルーチン cstart*.asm の変更点

（a）名前を変更したセクションの終端ラベルの宣言を追加します。

```
  :
EXTRN   RTT1_E, TT1_E           ; RTT1_E, TT1_E の EXTRN 宣言を追加する
  :
```

C ソースに，#pragma section 指令を追加します。

```
  :
#pragma section @@R_INIT    RTT1
#pragma section @@INIT      TT1
  :
```

（b）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, #LOW _?DATS
        BZ      $LDATS2
        MOV     [HL + 0], #0
        INCW    HL
        BR      $LDATS1
LDATS2 :
        MOV     ES, #HIGH RTT1_S
        MOV     HL, #LOWW RTT1_S
        MOV     DE, #LOWW TT1_S
LTT1 :
        MOVW    AX, HL
        CMPW    AX, #LOWW RTT1_E
        BZ      $LTT2
        MOV     A, ES:[HL]
        MOV     [DE], A
        INCW    HL
        INCW    DE
        BR      $LTT1
LTT2 :
;
        CALL    !!_main                 ; main ( ) ;
        CLRW    AX
        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

（c）名前を変更したセクションの先頭のラベルを設定します。

```
  :
@@R_INIT        CSEG    UNIT64KP
_@R_INIT :
@@R_INIS        CSEG    UNIT64KP
_@R_INIS :
@@INIT          DSEG
_@INIT :
@@DATA          DSEG
_@DATA :
@@INIS          DSEG    SADDRP
_@INIS :
@@DATS          DSEG    SADDRP
_@DATS :

RTT1            CSEG    UNIT64KP        ; セクション RTT1 の先頭を示す
RTT1_S :                                ; ラベルの設定を追加
TT1             DSEG    BASEP           ; セクション TT1 の先頭を示す
TT1_S :                                 ; ラベルの設定を追加

@@CODEL         CSEG
@@CALT          CSEG    CALLT0
@@CNST          CSEG    MIRRORP
@@BITS          BSEG
;
                END
```

（2）終端ルーチン rom.asm の変更点

注意 オブジェクト・モジュール名“@rom”，“@rome”は変更しないでください。

（a）名前を変更したセクションの終端を示すラベルの宣言

```
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    UNIT64KP
_?R_INIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG
_?INIT :
@@DATA          DSEG
_?DATA :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS
        :
```

（b）終端を示すラベルの設定

```
  :
RTT1    CSEG    UNIT64KP        ; セクション RTT1 の終端を示す
RTT1_E :                        ; ラベルの設定を追加

TT1     DSEG    BASEP           ; セクション TT1 の終端を示す
TT1_E :                         ; ラベルの設定を追加

;
        END
```

2 進定数（0b）

C ソース中で，2 進数を記述することができます。

[機能]

- 整数定数が記述可能な位置に，2 進定数を記述することができます。

[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することができ，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することの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

[使用例]

C ソースを以下に示します。

```
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］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 修正する必要はありません。

（2）RL78,78K0R 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。

モジュール名変更機能（#pragma name）

C ソース中で，オブジェクト・モジュール名を任意に変更することができます。

[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 254 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 254 文字を出力します。
- 255 文字以上のモジュール名が指定された場合は，警告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，警告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することができます。

[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
```

[互換性]

（1）他の C コンパイラから RL78,78K0R 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，修正する必要はありません。
- 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します。

（2）RL78,78K0R 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。

ローテート関数（#pragma rot）

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。

［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することができます。

［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

（1）unsigned char rorb (*x*, *y*) ;
unsigned char *x* ;
unsigned char *y* ;
x を *y* 回右ローテートします。

（2）unsigned char rolb (*x*, *y*) ;
unsigned char *x* ;
unsigned char *y* ;
x を *y* 回左ローテートします。

（3）unsigned int rorw (*x*, *y*) ;
unsigned int *x* ;
unsigned char *y* ;
x を *y* 回右ローテートします。

（4）unsigned int rolw (*x*, *y*) ;
unsigned int *x* ;
unsigned char *y* ;
x を *y* 回左ローテートします。

- 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 の前に記述することができます。
  - 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］

- 関数名として，ローテート用の関数名を使用することはできません。
- 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。

［使用例］

C ソースを以下に示します。

```
#pragma rot

unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned char   c ;

void main ( void ) {
        c = rorb ( a, b ) ;
}
```

出力アセンブラ・ソースは，以下のようになります。

```
        mov     x, !_b
        mov     a, !_a
L0003 :
        ror     a, 1
        dec     x
        bnz     $L0003
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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します。

（2）RL78,78K0R 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）。

乗算関数（#pragma mul）

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。

［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（mulu 関数）。
- #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］

- 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実行スピードが速く，かつサイズが小さいコードを生成することができます（mulu 関数）。
- 乗算器，または RL78 拡張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（muluw/mulsw 関数）。

［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。
  乗算用の関数名を次に示します。
  mulu, muluw, mulsw

（1）unsigned int mulu (*x*, *y*) ;
unsigned char *x* ;
unsigned char *y* ;
x と *y* を符号なし乗算します。

（2）unsigned long muluw (x, y) ;
unsigned int x ;
unsigned int y ;
x と *y* を符号なし乗算します。

（3）signed long mulsw (x, y) ;
signed int x ;
signed int y ;
x と *y* を符号付き乗算します。

- 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 の前に記述することができます。
  - 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］

- 関数名として，乗算用の関数名を使用することはできません（#pragma mul 宣言時）。
- 乗算用の関数は，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。
- インライン展開をせず，ライブラリ呼び出しとなります（muluw/mulsw 関数）。

［使用例］

C ソースを以下に示します。

```
#pragma mul

unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned int    i ;

void main ( void ) {
        i = mulu ( a, b ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
mov     x, !_b
mov     a, !_a
mulu    x
movw    !_i, ax
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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します。

（2）RL78,78K0R 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）。

除算関数（#pragma div）

オブジェクトに式の値を除算するコードを生成します。

［機能］

- オブジェクトに式の値を除算するコードを生成します。
- #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］

- 78K0 コンパイラと互換性があり，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を生成することができます。

［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

（1）**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。

（2）**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。

- 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 の前に記述することができます。
  - 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］

- インライン展開をせず，ライブラリ呼び出しとなります。
- 関数名として，除算用の関数名を使用することはできません。
- 除算用の関数は，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。

［使用例］

C ソースを以下に示します。

```
#pragma div

unsigned int    a = 0x1234 ;
unsigned char   b = 0x12 ;
unsigned char   c ;
unsigned int    i ;

void main ( void ) {
        i = divuw ( a, b ) ;
        c = moduw ( a, b ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
mov     c, !_b
movw    ax, !_a
call    !@@divuw
movw    !_i, ax
mov     c, !_b
movw    ax, !_a
call    !@@divuw
mov     a, c
mov     !_c, a
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 除算用の関数を使用していなければ，修正する必要はありません。
- 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します。

（2）RL78,78K0R 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することができます。
- 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）。

積和演算関数（#pragma mac）

オブジェクトに式の値を積和演算するコードを生成します。

［機能］

- オブジェクトに式の値を積和演算するコードを生成します。
- #pragma の指令がない場合は，積和演算用の関数は通常の関数とみなされます。

［効果］

- 積和演算器，または RL78 拡張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，通常の積和演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を生成することができます。

［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。
  積和演算用の関数名を次に示します。

```
macuw, macsw
```

（1）unsigned long macuw (x, y, z) ;
unsigned long x ;
unsigned int y ;
unsigned int z ;
x + (*y* * *z*) の符号なし積和演算を行い，演算結果を返します。

（2）signed long macsw (x, y, z) ;
signed long x ;
signed int y ;
signed int z ;
x + (*y* * *z*) の符号付き積和演算を行い，演算結果を返します。

- モジュールの #pragma mac 指令により，積和演算用の関数の使用を宣言します。
  ただし，次の項目は，#pragma mac の前に記述することができます。
  - 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］

- インライン展開をせず，ライブラリ呼び出しとなります。
- 関数名として，積和演算用の関数名を使用することはできません
- 積和演算用の関数は，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。

［使用例］

C ソースを以下に示します。

```
#pragma mac

unsigned long   a = 100000 ;
unsigned int    b = 1000 ;
unsigned int    c = 100 ;
singed long     d = 100000 ;
signed int      e = 1000 ;
signed int      f = -100 ;
unsigned long   ul ;
signed long     sl ;

void main( ){
        ul = macuw( a, b, c ) ;
        sl = macsw( d, e, f ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
movw    ax, !_a
movw    _@RTARG0, ax
movw    ax, !_a+2
movw    _@RTARG2, ax
movw    ax, !_b
movw    _@RTARG4, ax
movw    ax, !_c
call    !@@macuw
movw    ax, _@RTARG2
movw    !_ul+2, ax
movw    ax, _@RTARG0
movw    !_ul, ax
movw    ax, !_d
movw    _@RTARG0, ax
movw    ax, !_d+2
movw    _@RTARG2, ax
movw    ax, !_e
movw    _@RTARG4, ax
```

```
movw    ax, !_f
call    !@@macsw
movw    ax, _@RTARG2
movw    !_sl+2, ax
movw    ax, _@RTARG0
movw    !_sl, ax
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 積和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。
- 積和演算用の関数に変更したい場合は，前記の方法に従い変更します。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- “#pragma mac”文を削除，または #ifdef で切り分けます。関数名として，積和演算用の関数名を使用できます。
- 積和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）。

BCD 演算関数（#pragma bcd）

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。

［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。
  ただし，bcdtob，btobcde，bcdtow，wtobcd，btobcd 関数は，インライン展開されません。
- #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 演算機能を実現することができます。

［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。

（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）。

（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。

（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，オーバフローは切り捨てます。

- モジュールの #pragma bcd 指令により，BCD 演算用の関数の使用を宣言します。ただし，次の項目は #pragma bcd の前に記述することができます。
  - 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，関数名として使用することはできません。
- 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。

［使用例］

C ソースを以下に示します。

```
#pragma  bcd

unsigned char   a = 0x12 ;
unsigned char   b = 0x34 ;
unsigned char   c ;

void main ( void ) {
        c = adbcdb ( a, b ) ;
        c = sbbcdb ( b, a ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
mov     a, !_a
add     a, !_b
add     a, !BCDADJ
mov     !_c, a
mov     a, !_b
sub     a, !_a
sub     a, !BCDADJ
mov     !_c, a
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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します。

（2）RL78,78K0R 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することができます。
- 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）。

データ挿入関数（#pragma opc）

カレント・アドレスに定数データを挿入します。

［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。

（1）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，定数しか記述することができません。

- #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 の前に記述することができます。
  - 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］

- 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 ソースを以下に示します。

```
#pragma opc

void main ( void ) {
        __OPC ( 0xa7 ) ;
        __OPC ( 0x51, 0x12 ) ;
        __OPC ( 0x30, 0x34, 0x12 ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
_main :
; line  4 : __OPC ( 0xa7 ) ;
        DB      0AFH
; line  5 : __OPC ( 0x51, 0x12 ) ;
        DB      051H
        DB      012H
; line  6 : __OPC ( 0x30, 0x34, 0x12 ) ;
        DB      030H
        DB      034H
        DB      012H
; line  7 : }
        ret
```

[互換性]

(1) 他の C コンパイラから RL78,78K0R 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します。

(2) RL78,78K0R 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）。

RTOS 用割り込みハンドラ（#pragma rtos_interrupt）

RI78V4 用の割り込みハンドラを記述することができます。

［機能］

- #pragma rtos_interrupt 指令で指定された関数名を RL78,78K0R RTOS RI78V4 用割り込みハンドラと解釈します。
- 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する割り込みベクタ・テーブルに登録します。
- RTOS 用割り込みハンドラは，次の順番でコード生成を行います。

（1）call !!addr20 命令によるカーネル・シンボル __kernel_int_entry の呼び出し

（2）コンパイラが使用する saddr 領域の退避

（3）ローカル変数領域の確保（ローカル変数があるときのみ）

（4）関数本体

（5）ローカル変数領域の解放（ローカル変数があるときのみ）

（6）コンパイラが使用する saddr 領域の復帰

（7）ラベル _ret_int に br !!addr20 命令で無条件分岐

［効果］

- C ソース・レベルで，RTOS 用割り込みハンドラ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］

- 次の #pragma 指令により，割り込み要求名と関数名を指定します。

```
#pragma rtos_interrupt[ 割り込み要求名 関数名 ]
```

- #pragma 指令は，C ソースの先頭に記述します。
  次の項目は，この #pragma 指令の前に記述することができます。
  - 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 以降に記述するキーワードのうち，割り込み要求名は必ず大文字で記述してください。その他のキーワードは，大文字でも小文字でも可能です。

［制限］

- -zf 無指定時，RTOS 用割り込みハンドラはメモリ・モデルによらず，[C0H ～ 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- 割り込み要求名は，大文字で記述します。
- 割り込み要求名にソフトウエア割り込み，ノンマスカブル割り込みを指定することはできません。指定した場合は，エラーとします。
- 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。
- RTOS 用割り込みハンドラは，callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf を指定することができません。
  __far は，-zf オプション指定時のみ指定可能です。
- 関数名として，ret_int/_kernel_int_entry を使用することはできません。
- -zx 指定時に #pragma rtos_interrupt 指令を記述した場合はエラーとなります。RTOS 割り込みハンドラを定義する時は，__rtos_interrupt 修飾子を使用してください。RL78 ファミリでは，セルフ・プログラミング・ライブラリを用いて，セルフ・プログラミングにおける割り込みベクタテーブルを配置します。

［使用例］

C ソースを以下に示します。

```
#pragma rtos_interrupt  INTP0   intp

int     i ;

void    intp ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        func ( ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@BASE          CSEG    BASE
_intp :
        call    !!__kernel_int_entry
        movw    ax, _@RTARG0    ; コンパイラが使用する saddr 領域の退避
        push    ax              ;
        movw    ax, _@RTARG2    ;
        push    ax              ;
        movw    ax, _@RTARG4    ;
        push    ax              ;
        movw    ax, _@RTARG6    ;
        push    ax              ;
```

```
        movw    ax, _@SEGAX     ;
        push    ax              ;
        movw    ax, _@SEGDE     ;
        push    ax              ;
        subw    sp, #06H        ; ローカル変数の領域確保
        movw    hl, sp
; line 5 :      int     a[3]    ;
; line 6 :      a[0] = 1 ;
        onew    ax
        movw    [hl], ax        ; a
; line 7 :      func ( ) ;
         call    !!_func
; line 8 :      }
        addw    sp, #06H        ; ローカル変数の領域解放
        pop     ax              ; コンパイラが使用する saddr 領域の復帰
        movw    _@SEGDE, ax     ;
        pop     ax              ;
        movw    _@SEGAX, ax     ;
        pop     ax              ;
        movw    _@RTARG6, ax    ;
        pop     ax              ;
        movw    _@RTARG4, ax    ;
        pop     ax              ;
        movw    _@RTARG2, ax    ;
        pop     ax              ;
        movw    _@RTARG0, ax    ;
        br      !!_ret_int

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- RTOS 用割り込みハンドラをサポートしていなければ，修正は必要ありません。
- RTOS 用割り込みハンドラに変更したい場合は，上記の方法に従い変更を行います。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #pragma rtos_interrupt 指定を削除すれば，通常の関数として扱われます。
- RTOS 用割り込みハンドラ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。

RTOS 用割り込みハンドラ修飾子（__rtos_interrupt）

RI78V4 用の割り込みハンドラ記述とベクタ設定を別ファイルにします。

［機能］

- __rtos_interrupt 修飾子で宣言された関数は，RTOS 用割り込みハンドラと解釈します。RTOS 用割り込みハンドラでのレジスタ，saddr の退避／復帰については，「RTOS 用割り込みハンドラ（#pragma rtos_interrupt）」を参照してください。

［効果］

- ベクタ・テーブルの設定と RTOS 用割り込みハンドラ関数定義を別ファイルに記述することができます。

［方法］

- RTOS 用割り込みハンドラの修飾子に，__rtos_interrupt を付加します。

```
__rtos_interrupt void   func ( )        { 処理 }
```

［制限］

- -zf 無指定時，RTOS 用割り込みハンドラはメモリ・モデルによらず，[C0H ～ 0FFFFH] の領域に配置します。
  -zf 指定時は，メモリ・モデルに依存した配置となります。また，__near/__far の指定による配置指定も有効となります。
- RTOS 用割り込みハンドラは，callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf を指定することができません。
  __far は，-zf オプション指定時のみ指定可能です。
- 関数名として，ret_int/__kernel_int_entry を使用することはできません。
- -zx 指定時，割り込み関数は -zf オプションの有無やメモリ・モデルによらず，［C0H ～ FFEFFH］の領域に配置します。セルフ・プログラミング・モード時は，セルフ・プログラミング・ライブラリを用いて，割り込みベクタテーブルを配置します。

［注意］

- この修飾子の宣言だけでは，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 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。
- #pragma rtos_interrupt …の指定と同一ファイルに RTOS 用割り込みハンドラを定義する場合は，この修飾子を記述しなくても，#pragma rtos_interrupt で指定された関数名を RTOS 用割り込みハンドラと判断します。

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- RTOS 用割り込みハンドラをサポートしていなければ，修正は必要ありません。
- RTOS 用割り込みハンドラに変更したい場合は，上記の方法に従って変更を行います。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #define により可能です（「3. 2. 5　C ソースの修正」を参照してください）。
  この変更により，通常の関数として扱われます。
- RTOS 用割り込みハンドラ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となります。

RTOS 用タスク（#pragma rtos_task）

#pragma 指令により，指定された関数を RI78V4 用のタスクと解釈します。

［機能］

- #pragma rtos_task で指定された関数名を RTOS 用のタスクと解釈します。
- 関数名指定がある場合，その実体定義が同一ファイル中にない場合はエラーとなります。
- RTOS 用タスクの前処理では，フレーム・ポインタ／レジスタ変数用レジスタの退避は行いません。また，後処理を出力しません。
- RTOS 用タスクの最後で，常に RTOS のシステム・コール ext_tsk を呼びます。
- 次の RTOS システム・コール呼び出し関数を使用可能とします。

  void ext_tsk (void) ;

  RTOS のシステム・コール ext_tsk を呼ぶ。

  ただし，ext_tsk の実体定義，および割り込み関数，RTOS 用割り込みハンドラ内での ext_tsk の呼び出しは，エラーとします。
- RTOS システム・コール ext_tsk は，br !!addr20 命令で呼びます。通常関数の最後で ext_tsk を発行した場合は，エピローグを出力しません。
- タスクは，引数なし，または 1 ～ 4 バイトまでの 1 引数を使用可能とし，戻り値の記述はできません。

  2 つ以上の引数を記述した場合，4 バイトを越える引数を記述した場合，または戻り値を記述した場合，エラーとなります。

［効果］

- C ソース・レベルで，RTOS 用タスクを記述することができます。
- フレーム・ポインタ／レジスタ変数用レジスタの退避，および後処理が出力されなくなるため，コード効率が良くなります。

［方法］

- 次の #pragma 指令に，関数名を指定します。

```
#pragma rtos_task[ タスク名 ]
```

なお，この #pragma 指令は C ソースの先頭に記述します。

ただし，次の項目は，この #pragma 指令の前に記述することができます。

- 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 以降に記述するキーワードは，大文字でも小文字でも記述可能です。

［制限］

- RTOS 用タスクは，callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf を指定することができません。
  __far は，-zf オプション指定時のみ指定可能です。
- RTOS 用タスクを通常の関数のように呼び出すことはできません。
- RTOS システム・コール呼び出し関数名 ext_tsk を関数名として使用することはできません。
- C ソース中に #pragma rtos_task を記述しないと，ext_tsk を RTOS 用システム・コールと認識しないため，
  RTOS 用割り込みハンドラから ext_tsk を呼び出しても以下のエラーが出力されません。
  E0778: Cannot call ext_tsk in interrupt function
  回避策を以下に示します。
  - #pragma rtos_task でタスクを使用することを明示してください。
  - RTOS 割り込みハンドラからは，ext_tsk を呼ばないでください。

［使用例］

C ソースを以下に示します。

```
#pragma rtos_task       func
#pragma rtos_task       func2

void    func ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( ) ;
}

void    func2 ( int x ) {
        int     a[3] ;
        a[0] = 1 ;
}

void    func3 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( ) ;
}

void    func4 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        if ( a[0] )
                ext_tsk ( ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
@@CODEL CSEG
_func :
        subw    sp, #06H        ; フレーム・ポインタは退避されない
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; ext_tsk 関数の記述による ext_tsk 呼び出し
        br      !!_ext_tsk      ; タスクが常に出力する ext_tsk 呼び出し
                                ; エピローグを出力しない
_func2 :
        push    ax              ; フレーム・ポインタは退避されない
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; タスクが常に出力する ext_tsk 呼び出し
                                ; エピローグを出力しない
_func3 :
        push    hl              ; フレーム・ポインタを退避する
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; ext_tsk が関数の最後で呼び出された場合，
                                ; エピローグを出力しない
_func4 :
        push    hl              ; フレーム・ポインタを退避する
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        clrw    bc
        cmpw    ax, bc
        skz
        br      !!_ext_tsk      ; ext_tsk が関数の途中で呼び出された場合，
        addw    sp, #06H        ; エピローグを出力する
        pop     hl
        ret
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- RTOS 用タスクをサポートしていなければ，修正は必要ありません。
- RTOS 用タスクに変更したい場合は，上記の方法に従い変更を行います。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #pragma rtos_task 指定を削除すれば通常の関数として扱われます。
- RTOS 用タスク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。

フラッシュ領域配置方法（-zf）

コンパイル時に -zf オプションを指定することにより，プログラムをフラッシュ領域に配置したり，-zf オプションを指定せずに作成したブート領域のオブジェクトと結合して使用できるようになります。

［機能］

- フラッシュ領域に配置するオブジェクト・ファイルを生成します。
- ブート領域からは，フラッシュ領域の外部変数を参照することはできません。
- フラッシュ領域からは，ブート領域の外部変数を参照することができます。
- ブート領域のプログラムとフラッシュ領域のプログラムでは，同じ外部変数，および同じグローバル関数を定義することはできません。

［効果］

- プログラムをフラッシュ領域に配置することができるようになります。
- -zf オプションを指定せずに作成したブート領域のオブジェクトと結合して使用できるようになります。

［方法］

- コンパイル時に -zf オプションを指定します。

［制限］

- スタートアップ・ルーチン，ライブラリはフラッシュ領域用のものを使用してください。

フラッシュ領域分岐テーブル，フラッシュ領域指定（-zt/-zz）

フラッシュ領域分岐テーブルの先頭アドレスを -zt オプションにより指定することにより，スタートアップ・ルーチン，割り込み関数をフラッシュ領域に配置したり，ブート領域からフラッシュ領域への関数呼び出しを行うことができます。

[機能]

- -zt オプション指定により，スタートアップ・ルーチンへの分岐テーブル，割り込み関数への分岐テーブル，およびブート領域からフラッシュ領域への関数呼び出しのための分岐テーブルの先頭アドレスを決定します。
- 分岐テーブルの先頭から 64 個分は，割り込み関数専用（スタートアップ・ルーチンを含む）とし，それぞれ 4 バイトの領域を占有します。
- 通常関数の分岐テーブルは「分岐テーブルの先頭アドレス + 4 * 64」以降に配置し，それぞれ 4 バイトの領域を占有します。ext_func の ID 値については，「ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）」を参照してください。
- -zz オプションの指定によりフラッシュ領域の先頭アドレスを決定します。
- -zt オプションのみを指定した場合，同アドレスを -zz オプションにも指定したと見なします。
- -zz オプションのみを指定した場合，同アドレスを -zt オプションにも指定したと見なします。

[効果]

- スタートアップ・ルーチン，割り込み関数をフラッシュ領域に配置することができます。
- ブート領域からフラッシュ領域への呼び出しを行うことができます。

[方法]

- 次の -zt オプション指定により，フラッシュ領域分岐テーブルの先頭アドレスを指定します。

```
-ztxxxxxH : xxxxx = 0c0H ～ 0edfffH 注
```

- 次の -zz オプション指定により，フラッシュ領域分岐テーブルの先頭アドレスを指定します。

```
-zzxxxxxH : xxxxx = 0c0H ～ 0edfffH 注
```

注　デバイスにより異なります。

[制限]

- フラッシュ領域の先頭アドレス値，分岐テーブルの先頭アドレス値は，0C0H ～ 0EDFFFH とします。0EDFFFH は，デバイスにより異なります。
- #pragma ext_func，- zf 指定時の #pragma vect，#pragma interrupt，または #pragma rtos_interrupt を記述し，-zz，または -zt オプションの指定がない場合，エラーとなります。

- 指定した分岐テーブルの先頭アドレス値に従って，割り込みベクタ用ライブラリ（_@vect00 ～ _@vect7e）を再構築する必要があります。割り込みベクタ用ライブラリ中の分岐テーブルの先頭アドレス値のデフォルトは，2000H です。
- フラッシュ領域の先頭アドレス値は，-zb リンカ・オプションで指定するフラッシュ・スタート・アドレスと一致させてください。アドレスが一致していない場合は，リンク・エラーとなります。
- フラッシュ領域分岐テーブルの配置アドレスがフラッシュ領域の先頭アドレスより小さい時は，エラーとなります。
- ブート領域，またはフラッシュ領域配置のプログラムを作成する場合は，-zt オプション，または -zz オプションによりフラッシュ領域，およびフラッシュ領域分岐テーブルの配置アドレスを指定する必要があります。
- -zt，および -zz オプションの指定アドレスが異なるモジュールをリンクした場合，リンク・エラーとなります。
- ブート領域またはフラッシュ領域の ROM データを near 領域に配置できない場合，ROM データを指すポインタは強制的に far ポインタとなります（［注意］を参照してください）。この時，スモールおよびミディアム・モデルにて「const *」を引数に持つ標準ライブラリ関数を呼ぶには，末尾に“_f”を付けた関数名で呼び出さなければなりません（ワーニング W0072 を常に出力します）。

「const *」を引数に持つ標準ライブラリ関数は，以下のとおりです。

sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/strtoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

［使用例］

分岐テーブルを 2000H 番地以降に生成し，割り込み関数を配置します。

C ソースを以下に示します。

```
#pragma interrupt       INTP0   intp

void    intp ( void ) {

}
```

（1）割り込み関数をブート領域に配置する場合（-zf 指定なし，-zt2000H 指定あり）

コンパイラの出力オブジェクトは，以下のようになります。

```
                PUBLIC  _intp
                PUBLIC  _@vect06
@@BASE          CSEG    BASE
_intp :
                reti

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

割り込みベクタ・テーブルに，割り込み関数の先頭アドレスを設定します。

（2）割り込み関数をフラッシュ領域に配置する場合（-zf 指定あり，-zt2000H 指定あり）

コンパイラの出力オブジェクトは，以下のようになります。

```
                PUBLIC  _intp
@ECODE          CSEG    BASE
_intp :
                reti

@EVECT06        CSEG    AT      0200CH
                br      !!_intp
```

分岐テーブルに，割り込み関数の先頭アドレスを設定します。
分岐テーブルの先頭アドレスが 2000H，割り込みベクタ・アドレス（2 バイト）が 0006H なので，分岐テーブルのアドレス値は，2000H + 4 *（0006H / 2）番地となります。
割り込みベクタ・テーブルへの 200CH 番地の設定は，割り込みベクタ用ライブラリが行います。

割り込みベクタ 06 用ライブラリを以下に示します。

```
                PUBLIC  _@vect06

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      200CH
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- フラッシュ領域分岐テーブルの先頭アドレスを指定したい場合は，上記の方法に従って変更します。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- フラッシュ領域分岐テーブルの先頭アドレスを指定する場合は，各コンパイラの仕様により変更が必要です。

［注意］

- フラッシュ領域の先頭アドレスとミラー元領域のアドレスによって，near/far 領域指定の扱いが変わります。near/far 領域指定が実際の配置先と異なる時は，コマンドライン解析時に一度だけ W0070，W0071 を出力します。
- フラッシュ領域の先頭アドレスがミラー元領域内にあり，64K バイト未満の時は，near/far 領域指定をそのまま適用します。（「図 3—3 メモリマップ例 1」参照）

- フラッシュ領域の先頭アドレスがミラー元領域内にあり，64K バイト以上の時，フラッシュ領域の関数を far 領域に配置します。(「図 3—4 メモリマップ例 2」参照)
- フラッシュ領域の先頭アドレスがミラー元領域エンドアドレスより大きく，64K バイト未満の時，フラッシュ領域の ROM データを far 領域に配置します。(「図 3—5 メモリマップ例 3」参照)
- フラッシュ領域の先頭アドレスがミラー元領域エンドアドレスより大きく，64K バイト以上の時，フラッシュ領域の関数を far 領域に配置します。フラッシュ領域の ROM データを far 領域に配置します。(「図 3—6 メモリマップ例 4」参照)
- フラッシュ領域の先頭アドレスがミラー元領域スタートアドレスより小さく，64K バイト未満の時，ブート領域の ROM データを far 領域に配置します。(「図 3—7 メモリマップ例 5」参照)
- フラッシュ領域の先頭アドレスがミラー元領域スタートアドレスより小さく，64K バイト以上の時，ブート領域の ROM データを far 領域に配置します。フラッシュ領域の関数を far 領域に配置します。(「図 3—8 メモリマップ例 6」参照)
- ブート領域，またはフラッシュ領域の ROM データを near 領域に配置できない場合，ROM データを指すポインタは強制的に far ポインタとなるため，スモールおよびミディアム・モデルで ANSI 準拠ではなくなります。
  ANSI 準拠オプション -za 指定時は，ワーニング W0073 を出力します。
- ブート領域，またはフラッシュ領域の ROM データを near 領域に配置できない場合や，フラッシュ領域の関数を near 領域に配置できない場合は，以下の制限が生じます。

表 3—14 ブート領域にミラー元領域がない場合の ROM データの扱い

領域	定義	extern 宣言	ポインタの指す先
ブート	far 固定	far 固定	far 固定
フラッシュ	near/far 指定可	far 固定	far 固定

表 3—15 フラッシュ領域にミラー元領域がない場合の ROM データの扱い

領域	定義	extern 宣言	ポインタの指す先
ブート	near/far 指定可	near/far 指定可	far 固定
フラッシュ	far 固定	far 固定	far 固定

表 3—16 フラッシュ領域の先頭が 64K バイト以内にない場合の関数の扱い

領域	定義	extern 宣言	ポインタの指す先
ブート	near/far 指定可	near/far 指定可[注]	far 固定
フラッシュ	far 固定	far 固定	far 固定

注 #pragma ext_func で指定した関数は，関数本体がフラッシュ領域にあるので far 固定となります。

図 3—3 メモリマップ例 1

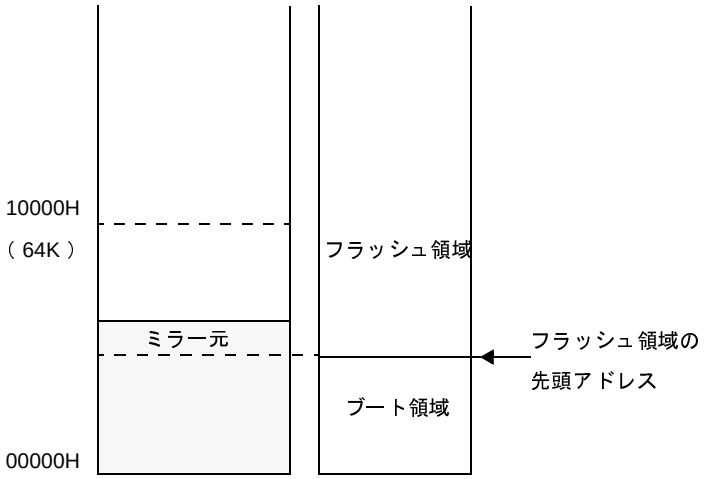


図 3—4 メモリマップ例 2

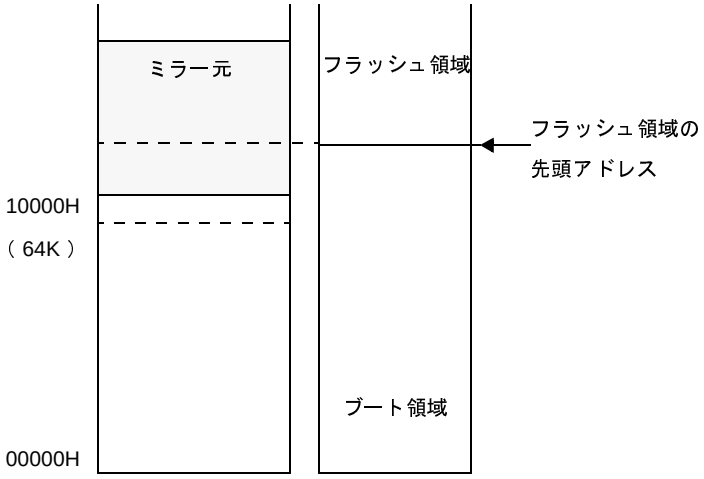


図 3—5 メモリマップ例 3

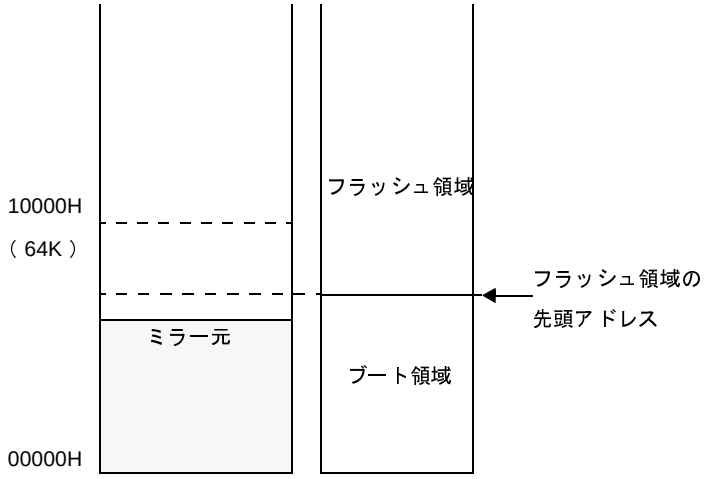


図 3—6 メモリマップ例 4

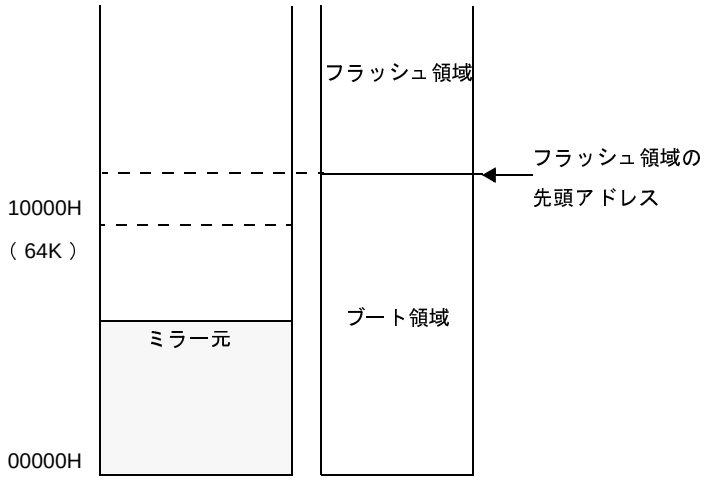


図 3—7 メモリマップ例 5

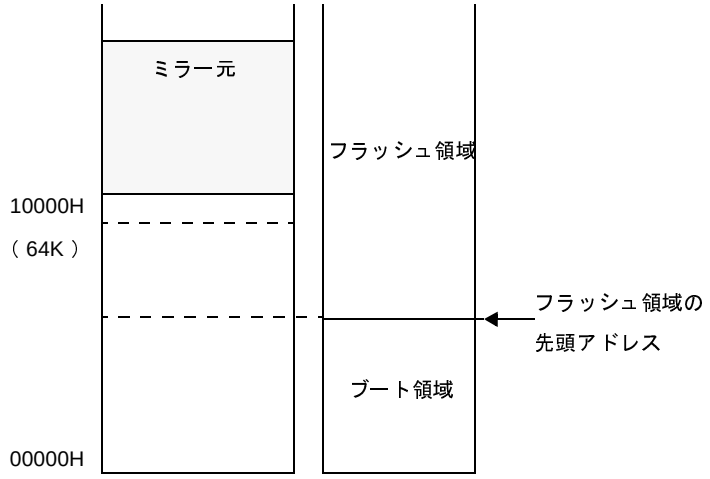
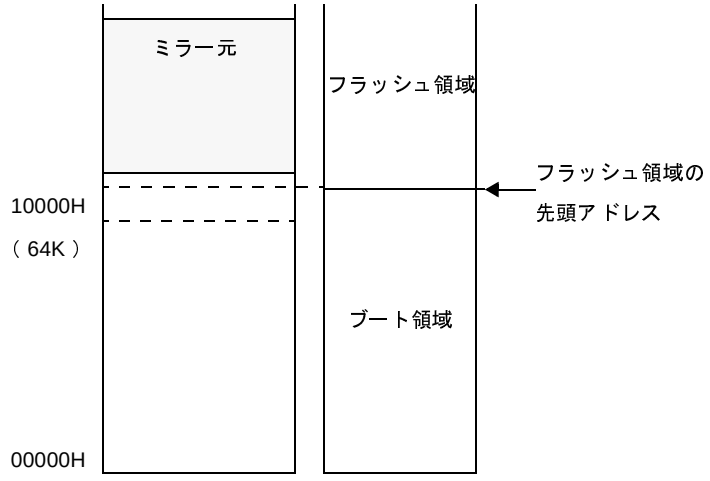


図 3—8 メモリマップ例 6



ブート領域からフラッシュ領域への関数呼び出し機能（#pragma ext_func）

ブート領域から呼び出すフラッシュ領域中の関数名，および ID 値を #pragma 指令により指定することにより，ブート領域からフラッシュ領域中の関数を呼び出せるようになります。

［機能］

- ブート領域からフラッシュ領域への関数呼び出しをフラッシュ領域分岐テーブルを介して行います。
- フラッシュ領域からは，ブート領域中の関数を直接呼び出せます。

［効果］

- ブート領域からフラッシュ領域中の関数を呼び出すことができるようになります。

［方法］

- 次の #pragma 指令により，ブート領域から呼び出すフラッシュ領域中の関数名，および ID 値を指定します。

```
#pragma ext_func　　関数名　ID 値
```

なお，この #pragma 指令は，C ソースの先頭に記述します。
次の項目は，この #pragma 指令の前に記述されても問題ありません。

- 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。

［制限］

- ID 値は，0 ～ 255 （ 0xff ）とします。
- -zt オプション，または -zz オプションを指定せず #pragma ext_func を記述した場合，エラーとなります。
- 同じ関数名で ID 値が異なる場合，および異なる関数名で ID 値が同じ場合は，エラーとなります。
  次の（1），（2）は，エラーとなります。

（1）#pragma ext_func f1 3
**　　#pragma ext_func f1 4**

（2）#pragma ext_func f1 3
**　　#pragma ext_func f2 3**

- ブート領域からフラッシュ領域へ関数呼び出しを行い，フラッシュ領域に対応する関数定義がない場合，リンカはチェックすることができませんので，注意してください。

- callt 関数は，ブート領域内のみ配置可能とし，フラッシュ領域（-zf オプション指定時）に callt 関数を定義したときは，エラーとなります。
- スモールまたはミディアム・モデルにて -rf オプションを指定するか，ラージ・モデルにて -rn オプションを指定した場合，「const *」を引数に持つ標準ライブラリ関数を呼ぶには，末尾に“_f”を付けた関数名で呼び出さなければなりません（ワーニング W0072 を常に出力します）。「const *」を引数に持つ標準ライブラリ関数は，以下のとおりです。

  sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/strtoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

［使用例］

分岐テーブルを 2000H 番地以降に生成し，フラッシュ領域中の関数 f1，f2 をブート領域から呼び出します。
C ソースを以下に示します。

- ブート領域側

```
#pragma interrupt INTP0 intf0
#pragma ext_func f1 3
#pragma ext_func f2 4

void    f1 ( ), f2 ( ) ;

void    func ( ) {
        f1 ( ) ;
        f2 ( ) ;
}
```

- フラッシュ領域側

```
#pragma interrupt INTP1 intf1
#pragma ext_func f1 3
#pragma ext_func f2 4

void    f1 ( ) {
}

void    f2 ( ) {
}

void    intf1 ( ) {
}
```

備考 1. #pragma ext_func f1 3 は，関数 f1 への飛び先を分岐テーブルの先頭アドレス + 4 * 64 + 4 * 3 番地に配置することを意味します。

2. #pragma ext_func f2 4 は，関数 f2 への飛び先を分岐テーブルの先頭アドレス + 4 * 64 + 4 * 4 番地に配置することを意味します。
3. 分岐テーブルの先頭から 4 * 64 バイトは，割り込み関数専用（スタートアップ・ルーチンを含む）です。

コンパイラの出力オブジェクトは，以下のようになります。

（1）フラッシュ領域分岐テーブルの配置アドレスが 64K バイト未満の場合

- ブート領域側（-zf 指定なし，-zt2000H 指定あり）

```
@@CODEL         CSEG
_func :
        call    !0210CH
        call    !02110H
        ret
@@VECT08        CSEG    AT    0008H
_@vect08 :
                DW      _intf0
```

- フラッシュ領域側（-zf 指定あり）

```
@ECODEL         CSEG
_f1 :
        ret
_f2 :
        ret
_intf1 :
        reti

@EVECT0A        CSEG    AT    02014H
        br      !!_intf1

@EXT03          CSEG    AT    0210CH
        br      !!_f1
        br      !!_f2
```

- 割り込みベクタ 0A 用ライブラリ

```
@@VECT0A        CSEG    AT    000AH
_@vect0a :
                DW      2014H
```

(2) フラッシュ領域分岐テーブルの配置アドレスが 64K バイト以上の場合

フラッシュ領域分岐テーブルの配置アドレスが 13000H の時

- ブート領域側（-zf 指定なし，-zt13000H 指定あり）

```
@@CODEL         CSEG
_func :
        call    !!01310CH
        call    !!013110H
        ret

@@VECT08        CSEG    AT    0008H
_@vect08 :
                DW      _intf0
```

- フラッシュ領域側（-zf 指定あり，-zt13000H 指定あり）

```
@ECODEL         CSEG
_f1 :
        ret
_f2 :
        ret
_intf1 :
        reti

@EVECT0A        CSEG    AT    013014H
        br      !!_intf1

@EXT03          CSEG    AT    01310CH
        br      !!_f1
        br      !!_f2
```

- 割り込みベクタ 0A 用ライブラリ

```
@@BASE          CSEG    BASE
?@vect0a :
        br      !!013014H

@@VECT0A        CSEG    AT    000AH
_@vect0a :
                DW      ?@vect0a
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- #pragma ext_func を使用していなければ，修正は必要ありません。
- ブート領域からフラッシュ領域への関数呼び出しを行いたい場合は，上記の方法に従って変更します。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #pragma ext_func 指令を削除，または #ifdef で切り分けます。
- ブート領域からフラッシュ領域への関数呼び出しを行う場合は，各コンパイラの仕様により変更が必要です。

［注意］

- スモールまたはミディアム・モデルにて -rf オプションを指定するか，ラージ・モデルにて -rn オプションを指定した場合，ANSI 準拠ではなくなります。ANSI 準拠オプション -za 指定時は，ワーニング W0073 を出力します。

ミラー元領域指定（-mi0/-mi1）

コンパイル時に，-mi0/-mi1 オプションを指定することにより，ミラー元の領域を考慮したコードを出力することができます。

[機能]

- -mi0 オプション指定時は，MAA に 0 を指定したものとしてコードを出力します。
- -mi1 オプション指定時は，MAA に 1 を指定したものとしてコードを出力します。
- -mi0，-mi1 オプションが異なるモジュールをリンクした場合，リンク・エラーとなります。
- -mi オプション未指定は，MAA0 としてコードを出力します。
- リンカの -mi オプションのデフォルト値は，コンパイラの -mi オプションの値となります。
- コンパイラの -mi オプション指定がなければ 0 となります。
- コンパイラとリンカの -mi オプション指定が異なるモジュールをリンクした場合，リンク・エラーとなります。
- ミラー領域，MAA の詳細については，デバイスのユーザーズ・マニュアルを参照してください。

[効果]

- ミラー元の領域を考慮したコードを出力することができます。

[方法]

- コンパイル時に，-mi0/-mi1 オプションを指定します。

[互換性]

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- -mi オプションでミラー元の領域を選択し再コンパイルすればソースの修正は必要ありません。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- 他の C 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。

引数／戻り値の int 拡張抑制方法（-zb）

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，実行速度の向上を図ることができます。

[機能]

- 関数戻り値の型定義が char/unsigned char 型の場合に，戻り値の 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，実行速度の向上を図ることができます。

[方法]

- コンパイル時に -zb オプションを指定します。

[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。

[使用例]

C ソースを以下に示します。

```
unsigned char   func1 ( unsigned char x, unsigned char y ) ;
unsigned char   c, d, e ;

void main ( void ) {
        c = func1 ( d, e ) ;
        c = func2 ( d, e ) ;
}

unsigned char   func1 ( unsigned char x, unsigned char y ) {
        return x + y ;
}
```

(1) -zb 指定ありの場合

コンパイラの出力オブジェクトは，以下のようになります。

```
_main :
; line  5 :             c = func1 ( d, e ) ;
        mov     x, !_e
        push    ax
        mov     x, !_d                  ; int 拡張しない
        call    !_func1
        pop     ax
        mov     a, c
        mov     !_c, a
; line  6 :             c = func2 ( d, e ) ;
        mov     x, !_e
        clrb    a                        ; プロトタイプ宣言がないので int 拡張する
        push    ax
        mov     x, !_d
        mov     x, #00H
        xch     a, x                     ; 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 ( unsigned char x, unsigned char y ) {
_func1 :
        push    hl
        push    ax
        movw    ax, sp
        movw    hl, ax
        mov     a, [hl]
        mov     x, a
        mov     a, [hl + 6]
        movw    hl, ax
; line  10 :            return x + y ;
        mov     a, l
        add     a, h
        mov     c, a                    ; int 拡張しない
; line 11 :     }
        pop     ax
        pop     hl
        ret
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

メモリ操作関数（#pragma inline）

標準ライブラリ関数 memcpy，memset を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。

[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，実行速度の向上を図ることができます。
- 指定文字数に定数を指定した場合は，オブジェクト・コードの短縮も図ることができます。

[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。
- 次の項目は，#pragma inline の前に記述することができます。
  - 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 ソースを以下に示します。

```
#pragma inline

char    ary1[100], ary2[100] ;

void main ( void ) {
        memset ( ary1, 'A', 50 ) ;
        memcpy ( ary1, ary2, 50 ) ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
_main :
        push    hl
; line  5 :     memset ( ary1, 'A', 50 ) ;
        movw    de, #loww ( _ary1 )
        mov     a, #041H        ; 65
        mov     c, #032H        ; 50
L0003 :
        mov     [de], a
        incw    de
        dec     c
        bnz     $L0003
; line  6 :     memcpy ( ary1, ary2, 50 ) ;
        movw    de, #loww ( _ary1 )
        movw    hl, #loww ( _ary2 )
        mov     c, #032H        ; 50
L0005 :
        mov     a, [hl]
        mov     [de], a
        incw    de
        incw    hl
        dec     c
        bnz     $L0005
; line  7 : }
        pop     hl
        ret
```

［互換性］

（1）他の C コンパイラから RL78,78K0R 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。

（2）RL78,78K0R 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。

絶対番地配置指定（__directmap）

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することができます。

［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。
  配置アドレスは整数で指定してください。
- C ソース中における __directmap 変数は，static 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，初期値を定義することができず，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。

項目	範囲	
	スモール・モデル， ミディアム・モデルの場合	ラージ・モデルの場合
アドレス指定範囲	0xf0000 ～ 0xfffff	0x00000 ～ 0xfffff
領域確保範囲	0xffd00 ～ 0xffeff	0xffd00 ～ 0xffeff
重複チェック範囲	デバイスの内蔵 RAM の スタート・アドレス～エンド・アドレス	デバイスの内蔵 RAM の スタート・アドレス～エンド・アドレス

- アドレス指定がアドレス指定範囲外の場合は，エラーを出力します。
- __directmap で宣言された変数は，以下の領域の境界をまたいで配置することはできません。配置した場合，エラーを出力します。
  - saddr 領域（0xffe20 ～ 0xffeff）
  - sfr 領域，saddr 領域が重なる領域（0xfff00 ～ 0xfff1f）
  - sfr 領域（0xfff20 ～ 0xfffff）
  - 2nd sfr 領域（デバイスにより異なります）
- __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，警告メッセージ（W0762）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。
- アドレス指定範囲が near 領域内である場合は，near 領域の変数としてアクセスします。
- アドレス指定範囲が saddr 領域でも near 領域でもない場合は，far 領域の変数としてアクセスします。
- __near，__far 型修飾子を指定しない場合は，メモリ・モデルに従ったアクセスをします。

- 型修飾子の指定がある場合，その指定に従ったアクセスをします。ただし，アドレス指定範囲と型修飾子に矛盾がある場合はエラーとします。

アドレス指定範囲，メモリ・モデル，型修飾子指定の関係をまとめると，次のとおりとなります。

アドレス指定範囲		型修飾子					
		__near __sreg	__far __sreg	__sreg	__near	__far	無指定
saddr領域内	アクセス方法	sreg	sreg	sreg	sreg	sreg	sreg
	ポインタ長	2バイト	4バイト	スモール : 2バイト ミディアム: 2バイト ラージ : 4バイト	2バイト	4バイト	スモール : 2バイト ミディアム: 2バイト ラージ : 4バイト
near領域内	アクセス方法	エラー	エラー	エラー	near	far	スモール : near ミディアム: near ラージ : far
	ポインタ長				2バイト	4バイト	スモール : 2バイト ミディアム: 2バイト ラージ : 4バイト
far領域内	アクセス方法	エラー	エラー	エラー	エラー	far	スモール : エラー ミディアム: エラー ラージ : far
	ポインタ長					4バイト	スモール : エラー ミディアム: エラー ラージ : 4バイト

［効果］

- 任意のアドレスに変数を配置することができ，同じアドレスに複数の変数を重ねて配置することができます。

[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。

[制限]

- 関数引数，戻り値，およびオートマチック変数には指定することができません。指定した場合は，エラーとなります。
- 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。
- __directmap 変数は，static 変数と同様に扱うため，extern 宣言することはできません。

[使用例]

C ソースを以下に示します。

```
__directmap char        c = 0xffe00 ;
__directmap __sreg char d = 0xffe20 ;
__directmap __sreg char e = 0xffe21 ;
__directmap struct  x {
        char    a ;
        char    b ;
} xx = { 0xffe30 } ;

void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

コンパイラの出力オブジェクトは，以下のようになります。

```
        PUBLIC  _main
_c      EQU     0FFE00H                 ; __directmap 宣言された変数は
_d      EQU     0FFE20H                 ; EQU でアドレスを定義
_e      EQU     0FFE21H                 ;
_xx     EQU     0FFE30H                 ;
        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                ;
@@CODEL CSEG
_main :
; line 10 :
        oneb    !loww ( _c )
; line 11 :
        mov     _d, #012H               ; アドレス指定が saddr 領域内のため，
; line 12 :                             ; saddr 命令を出力
        set1    _e.5                    ; __sreg と併用しているため，ビット操作可能
; line 13 :
        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 :
        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
        END
```

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- 予約語 __directmap を使用していなければ，修正する必要はありません。
- __directmap 変数に変更したい場合は，前記の方法に従い変更します。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- #define により可能です（「3. 2. 5　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。

near/far 領域指定（__near/__far）

関数，変数の宣言時に，__near/__far 型修飾子を追加することにより，関数，変数の配置場所を明示的に指定することができます。

［機能］

- __near/__far 型修飾子を指定することによって，関数，変数の配置場所を明示的に指定します。

修飾子	配置場所
__near 型修飾子	near 領域 （データ：0F0000H ～ 0FFFFFH，コード：000000H ～ 00FFFFH）
__far 型修飾子	far 領域 （000000H ～ 0FFFFFH）

- near 領域へのポインタは 2 バイト，far 領域へのポインタは 4 バイトとします。
- 同じ変数，関数の宣言において，__near/__far 型修飾子が混在する場合は，エラーとします。
- __near/__far 型修飾子は，文法的に型修飾子と同様に扱います。
- __near/__far 型修飾子を __callt/__interrupt/__rtos_interrupt/__interrupt_brk/__sreg/__boolean と同時に指定した場合，__near/__far 型修飾子は無視されます。
- __near/__far 型修飾子が同時に指定された場合は，エラーとなります。
- __near/__far 型修飾子をオートマチック変数，引数，レジスタ変数に指定した場合，__near/__far 型修飾子は無視されます。
- near 領域の変数は，ES レジスタを用いずにアクセスします。
  ポインタ長は，2 バイトです。
- far 領域の変数は，ES レジスタを設定してアクセスします。
  ポインタ長は，4 バイトです。
- near 領域の関数は !addr16 で，far 領域の関数は !!addr20 で関数を呼び出します。
- CS レジスタを参照せずに関数ポインタ呼び出しをする命令がないため，関数ポインタ呼び出し時は，常に CS レジスタを設定します。
- near 領域の関数への関数ポインタは，CS レジスタに 0 を設定するコードを出力します。
- far ポインタの最上位 1 バイトは，不定とします。
- near ポインタ，または int から far ポインタへのキャスト，near ポインタから long へのキャストは，以下の動作となります。
  - 変数ポインタは，上位に 0xf を追加します（0 は例外で，0 拡張します）。
  - 関数ポインタは，0 拡張します。
- far ポインタの加減算は，下位 2 バイトで行い，上位は変化しません。
- ptrdiff_t は，常に int 型とします。
- far ポインタの等値演算は，下位 3 バイトで行います。

- far ポインタの関係演算は，下位 2 バイトで行います。同一オブジェクトを指していないポインタを比較する場合は，unsigned long にキャストする必要があります。また，-za オプション指定時は，下位 3 バイトで比較を行います。
- 文字列定数は，メモリ・モデルにより，far 領域，または near 領域に配置します。

メモリ・モデル	配置場所
スモール・モデル	near 領域
ミディアム・モデル	near 領域
ラージ・モデル	far 領域

- ラージ・モデルの場合，オート変数，引数，sreg 変数へのポインタは 4 バイトです。
- 同じ変数，関数において，モジュール間で，定義と宣言で near 領域配置指定，far 領域配置指定が異なる場合，以下のエラー・チェックを行います。(「記述例 2」参照)
  - 定義のあるモジュールの配置指定が far 領域で，宣言のあるモジュールの配置指定が near 領域の場合，宣言のあるモジュール内で，その変数，または関数を参照すると，リンク・エラーとなります。
  - ポインタ，配列，および関数宣言子（の任意の組み合わせ）の個数が 8 個までをエラー・チェックの対象とします。
  - エラー・チェックは，-g オプション指定時のみ行います。

[効果]

- __far 型修飾子を指定することによって，far 領域への配置，および参照を可能とします。
- __near 型修飾子を指定することによって，near 領域への配置，および参照を可能とします。
  near 領域に配置することで，短い命令で参照，および関数呼び出しが可能となります。

[方法]

- 関数，変数の宣言時に，__near/__far 型修飾子を追加します。

[使用例]

(1) 記述例 1

```
__near int i1 ;
__far  int i2 ;
__far  int *__near p1 ;
__far  int *__near *__far p2 ;
__far  int func1( ) ;
__far  int *__near func2 ( ) ;
__near int ( *__far fp1 ) ( ) ;
__far  int *__near ( *__near fp2 ) ( ) ;
__near int *__far ( *__near fp3 ) ( ) ;
__near int *__near ( *__far fp4 ) ( ) ;
```

- i1 は int 型で，near 領域に配置
- i2 は int 型で，far 領域に配置
- p1 は“far 領域にある int 型”を指す 4 バイト型の変数で，変数自身は near 領域に配置
- p2 は“near 領域にある，“far 領域にある int 型”を指す 4 バイト型”を指す 2 バイト型の変数で，変数自身は far 領域に配置
- func1 は「int 型」を返す関数で，関数自身は far 領域に配置
- func2 は「“far 領域にある int 型”を指す 4 バイト型」を返す関数で，関数自身は near 領域に配置
- fp1 は“「int 型」を返す near 領域にある関数”を指す 2 バイト型の変数で，変数自身は far 領域に配置
- fp2 は“「“far 領域にある int 型”を指す 4 バイト型」を返す near 領域にある関数”を指す 2 バイト型の変数で，変数自身は near 領域に配置
- fp3 は“「“near 領域にある int 型”を指す 2 バイト型」を返す far 領域にある関数”を指す 4 バイト型の変数で，変数自身は near 領域に配置
- fp4 は“「“near 領域にある int 型”を指す 2 バイト型」を返す near 領域にある関数”を指す 2 バイト型の変数で，変数自身は far 領域に配置

（2）記述例 2

- 同じ変数，関数において，モジュール間で，定義と宣言で near 領域配置指定，far 領域配置指定が異なる場合のエラーチェックの例を示します。
  - a.c

```
/* 定義 */
int     __near i1 ;
int     __far  i2 ;
int     __near *__near nnp1 ;
int     __near *__near nnp2 ;
int     __near *__far  fnp1 ;
int     __near *__near nnp3 ;
int     __far  *__near nfp1 ;

int     __far  *__near nffunc1( ){ }
int     __far  *__near nffunc2( ){ }
int     __far  *__far  fffunc1( ){ }
int     __near *__far  fnfunc1( ){ }
int     __far  *__far  fffunc2( ){ }
```

- b.c

```
/* extern 宣言 */
extern int     __far  i1 ;
extern int     __near i2 ;
extern int     __near *__near nnp1 ;
extern int     __near *__far  nnp2 ;
extern int     __near *__near fnp1 ;
extern int     __far  *__near nnp3 ;
extern int     __near *__near nfp1 ;
extern int     __far  *__near nffunc1( ) ;
extern int     __far  *__far  nffunc2( ) ;
extern int     __far  *__near fffunc1( ) ;
extern int     __far  *__far  fnfunc1( ) ;
extern int     __near *__far  fffunc2( ) ;

void main( void ) {
        i1 = 1 ;          /* OK */
        i2 = 1 ;          /* エラー */
        *nnp1 = 1 ;       /* OK */
        *nnp2 = 1 ;       /* OK */
        *fnp1 = 1 ;       /* エラー */
        *nnp3 = 1 ;       /* エラー */
        *nfp1 = 1 ;       /* エラー */
        nffunc1 ( ) ;     /* OK */
        nffunc2 ( ) ;     /* OK */
        fffunc1 ( ) ;     /* エラー */
        fnfunc1 ( ) ;     /* エラー */
        fffunc2 ( ) ;     /* エラー */
}
```

［制限］

- __far 型修飾子を指定しても，64K バイトをまたがってデータを配置することはできません。
  また，64K バイトをまたがったアクセスをすることもできません。
  関数は，64K バイトをまたがっても配置可能です。

[互換性]

(1) 他の C コンパイラから RL78,78K0R C コンパイラ

- 予約語 __near/__far を使用していなければ，修正する必要はありません。

(2) RL78,78K0R C コンパイラから他の C コンパイラ

- __near/__far 型修飾子を使用していなければ，修正する必要はありません。
- __near/__far 型修飾子を使用している場合は，#define により可能です。

[注意]

- 関係演算を下位 2 バイトで行う場合，64K バイト境界の領域の最後の 1 バイトにデータを配置することはできません。配置した場合，リンカ，またはコンパイラでエラーとなります。
  これは，配列の範囲を越えた部分を指すポインタの関係演算に対し，ANSI 準拠の動作[注]を行うためです。

  注 ANSI の関係演算子の制約
  式 P が配列オブジェクトの要素を指しており，式 Q が同じ配列オブジェクトの最後の要素を指している場合，ポインタ式 Q+1 は，P と比較してより大きいとします。

- far 領域へのポインタのサイズは 4 バイトですが，演算対象は下位 3 バイトのみであり，最上位 1 バイトは不定となります。
  例を示します。

```
union   tag {
        __far unsigned short    *ptr ;
        unsigned long           ldata ;
} un ;
```

  un.ptr に値を書き込んだ後に un.ldata を参照すると，最上位 1 バイトは不定となります。un.ldata の最上位 1 バイトが 0 であることを保証するためには，最初に共用体 un を 0 で初期化する必要があります。

- リンカは，以下のセグメント・タイプと再配置属性の組み合わせを持つセクションのデータ配置のチェックを行います。
  DSEG UNIT64KP
  DSEG PAGE64KP
  CSEG PAGE64KP
- #pragma section，またはリンク・ディレクティブ・ファイルでこれらの再配置属性を変更した場合，リンカでチェックは行いません。
- ミラー空間がないデバイスでは ROM データを near 領域に配置できないため，ROM データを指すポインタは強制的に far ポインタとなります（ワーニング W0071 を常に出力します）。さらに，スモールおよびミディアム・モデルにて「const *」を引数に持つ標準ライブラリ関数を呼ぶには，末尾に“_f”を付けた関数名で呼び出さな

ければなりません（ワーニング W0072 を常に出力します）。また，スモールおよびミディアム・モデルで ANSI 準拠ではなくなります。ANSI 準拠オプション -za 指定時は，ワーニング W0073 を出力します。

メモリ・モデル指定（-ms/-mm/-ml）

コンパイル時に，-ms/-mm/-ml オプションを指定することにより，関数，変数の配置場所をメモリ・モデルで指定することができます。

［機能］

- 関数，変数の配置場所を指定します。

メモリ・モデル	データ	関数
スモール・モデル	near 領域	near 領域
ミディアム・モデル	near 領域	far 領域
ラージ・モデル	far 領域	far 領域

- __near/__far 型修飾子が指定された場合は，__near/__far 型修飾子を優先します。
- スモール・モデル

  データ 64K バイト，コード 64K バイトの合計 128K バイトとなります。

  データ ROM は，0000H ～ 0FFFFH，または 10000H ～ 1FFFFH に配置し，FxxxxH にミラーリングされます。

  コードは，00000H ～ 0FFFFH に配置します。

  __far 型修飾子により CS レジスタの値が変更されている可能性があるので，関数ポインタ呼び出し時は，常に CS レジスタを設定します。
- ミディアム・モデル

  変数は near 領域に，関数は far 領域に配置します。つまり，データ 64K バイト，コード 1M バイトとなります。

  データ ROM は 000000H ～ 00FFFFH，または 010000H ～ 01FFFFH に配置し，FxxxxH にミラーリングされます。コードの配置に制限はありません。
- ラージ・モデル

  変数，関数ともに far 領域に配置します。つまり，データ 1M バイト，コード 1M バイトとなります。データ，コードの配置に制限はありません。

［方法］

- コンパイル時に，-ms/-mm/-ml オプションを指定します。

オプション	説明
-ms	スモール・モデル
-mm	ミディアム・モデル
-ml	ラージ・モデル

［使用例］

C ソースを以下に示します。

```
int     i ;
int     *p ;
void    func( void ) { }

void    ( *fp ) ( void ) ;

void main( void ) {
        int     r ;

        r = i ;         /* データアクセス */
        func( ) ;       /* 関数呼び出し */
        r = *p ;        /* データポインタ */
        fp( ) ;         /* 関数ポインタ */
}
```

コンパイラの出力オブジェクトは，以下のようになります。

（1）スモール・モデルの場合

```
movw    hl, !_i
call    !_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
movw    ax, !_fp
mov     CS, #00H        ; 0
call    ax
```

（2）ミディアム・モデルの場合

```
movw    hl, !_i
call    !!_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
mov     a, !_fp + 2
mov     CS, a
movw    ax, !_fp
call    ax
```

(3) ラージ・モデルの場合

```
mov     ES, #highw ( _i )
movw    hl, ES:!_i
call    !!_func
mov     ES, #highw ( _p )
mov     a, ES:!_p + 2
movw    de, ES:!_p
mov     ES, a
movw    ax, ES:[de]
movw    hl, ax
mov     ES, #highw ( _fp )
mov     a, ES:!_fp + 2
mov     CS, a
movw    ax, ES:!_fp
call    ax
```

[制限]

- ラージ・モデルを指定した場合でも，データを 64K バイトをまたがって配置することはできません。
- 異なるメモリ・モデルを指定したモジュール同士は，リンクすることはできません。
- far 領域に配置された初期値あり変数，初期値なし変数の 1 ロード・モジュール・ファイルあたりのサイズは，おのおの（64K - 1）バイト（注：-za 指定時は 64K バイト）です。
  この制限を回避するためには，「コンパイラ出力セクション名の変更（#pragma section）」の機能を利用して，特定のファイル中の初期値あり変数や初期値なし変数のセクション名を別の出力セクション名に変更することで，サイズを増やすことが可能です。
  ただし，スタートアップ・ルーチン，終端ルーチンの変更を行う必要があります（「コンパイラ出力セクション名の変更（#pragma section）」の ROM 化関連セクション名変更に伴うスタートアップ・ルーチンなどの変更例を参照してください）。
  なお，1 つの出力セクション名あたりの最大サイズは変わりません。
- -za オプションを指定しない場合は，データを 64K バイト境界の領域の最後の 1 バイトに配置することはできません（「near/far 領域指定（__near/__far）」の［注意］を参照してください）。

[注意]

- ミラー空間がないデバイスでは ROM データを near 領域に配置できないため，ROM データは far 域に配置します（ワーニング W0071 を常に出力します）。さらに，スモールおよびミディアム・モデルにて「const *」を引数に持つ標準ライブラリ関数を呼ぶには，末尾に“_f”を付けた関数名で呼び出さなければなりません（ワーニング W0072 を常に出力します）。また，スモールおよびミディアム・モデルで ANSI 準拠ではなくなります。ANSI 準拠オプション -za 指定時は，ワーニング W0073 を出力します。

ROM データ配置先指定（-rf/-rn）

ROM データを far 領域，または near 領域の任意の領域に配置することができます。

［機能］

- -rf オプション指定時は，ROM データを far 領域に配置します。
- -rn オプション指定時は，ROM データを near 領域に配置します。
- -rf オプション，-rn オプションを指定しない場合，ROM データの配置先はメモリ・モデルに依存します。
- ROM データの配置指定の優先順位は以下の順番となります。

（1）フラッシュ領域の先頭アドレスとミラー元領域のアドレスによる near/far 領域指定（「フラッシュ領域分岐テーブル，フラッシュ領域指定（-zt/-zz）」参照）

（2）__near，__far キーワード

（3）-rn，-rf オプション指定

（4）メモリ・モデル

- ROM データは，以下のデータを指します。
  - const 修飾型変数
  - 文字列リテラル
  - 集成体型自動変数の初期値
  - switch 文の分岐テーブル

［効果］

- ROM データを far 領域，または near 領域の任意の領域に配置することができます。

［方法］

- コンパイル時に，-rf/-rn オプションを指定します。

［制限］

- 同じ const 修飾型変数を異なるモジュールで参照する場合，ROM データの配置指定の優先順位に従い配置し，エラー・チェックを行います。エラー・チェック方法は「near/far 領域指定（__near/__far）」を参照してください。

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- -rf，-rn オプションで ROM データの配置を指定し再コンパイルすればソースの修正は必要ありません。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- 他の C 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］

- ミラー空間がないデバイスでは ROM データを near 領域に配置できないため，-rn オプションは無視され，ROM データは far 領域に配置されます（ワーニング W0071 を常に出力します）。

セルフ・プログラミングにおける RAM 配置先指定（-zx）

コードおよび ROM データを RAM 領域に配置することができます。

［機能］

- RL78 ファミリのセルフ・プログラミングに対応した機能です。
- -zx 指定時は，コードおよび ROM データを RAM に配置します。
- -zx 指定時は，メモリ・モデルにかかわらずコードに far 属性を付加します。
- -zx1 指定時は，ROM 配置用ランタイム・ライブラリを呼び出します。
- -zx2 指定時は，RAM 配置用ランタイム・ライブラリを呼び出します。
- ROM データは，以下のデータを指します。
  - const 修飾型変数
  - 文字列リテラル
  - 集成体型自動変数の初期値
  - switch 文の分岐テーブル

［効果］

- コードおよび ROM データを RAM 領域に配置することができます。

［方法］

- コンパイル時に，-zx オプションを指定します。

［制限］

- RL78 ファミリのセルフ・プログラミングに対応していないデバイスに対して本オプションを指定し，フラッシュ領域配置指定オプション -zf を指定していない時に，割り込み関数や RTOS 割り込みハンドラを定義した場合はエラーとします。
- -zx2 指定時に最適化オプション -ql を指定した場合，-ql のレベルは自動的に 1 となります。
- -zx 指定時に，callt 関数は定義できません。callt 関数を記述した場合はエラーとなります。
- セルフ・プログラミング・モード時は割り込みの仕様が変わるため，-zx 指定時に #pragma interrupt 指令や #pragma rtos_interrupt 指令を記述した場合はエラーとなります。-zx 指定時に割り込み関数や RTOS 割り込みハンドラを定義する場合は，__interrupt/__interrupt_brk/__rtos_interrupt 修飾子を使用してください。
- -zx 指定時には，関数は RAM 領域に配置されるため，ワーニングとなり，全ての関数は far 関数となります。
- RAM 容量の問題から，-zx 指定時でも標準ライブラリは ROM に配置されます。したがって，ROM が見えなくなる可能性があるセルフ・プログラミング・モード時は，標準ライブラリを呼び出さないでください。RAM に配置した関数からの標準ライブラリ呼び出しはユーザ責任となります。セルフ・プログラミング・モード時に標準ライブラリを呼び出した場合の動作は保証されません。

- RAM 容量の問題から，#pragma 指令を用いる乗算関数／除算関数／積和演算関数／ BCD 演算関数で用いるライブラリは，-zx 指定時でも ROM に配置されます。したがって，ROM が見えなくなる可能性があるセルフ・プログラミング・モード時は，これらの関数を呼び出さないでください。RAM に配置した関数からの，#pragma 指令を用いる乗算関数／除算関数／積和演算関数／ BCD 演算関数の呼び出しはユーザ責任となります。セルフ・プログラミング・モード時にこれらの関数を呼び出した場合の動作は保証されません。

［互換性］

（1）他の C コンパイラから RL78,78K0R C コンパイラ

- -zx オプションを指定し再コンパイルすればソースの修正は必要ありません。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- 他の C 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。

3. 2. 5 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することができます。しかし，拡張機能はRL78,78K0R 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 RL78,78K0R C コンパイラへの移植と，RL78,78K0R 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。

（1）他の C コンパイラから RL78,78K0R 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 コンパイラが予約語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。

（2）RL78,78K0R C コンパイラから他の C コンパイラ

- RL78,78K0R C コンパイラは，拡張機能として予約語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，予約語を削除するか，#ifdef で切り分けなければなりません。

例を以下に示します。

（a）予約語を無効にする（callf，sreg，norec なども同様）

```
#ifndef __K0R__
#define callt           /* callt 関数を通常の関数にします。*/
#endif
```

（b）他の型に変更する

```
#ifndef __K0R__
#define bit     char    /* bit 型変数を char 型変数にします。*/
#endif
```

3. 3　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。

- 戻 り 値（すべての関数で共通）
- 通常関数呼び出しインタフェース

3. 3. 1　戻 り 値

関数の戻り値は，レジスタ，またはキャリー・フラグに格納します。

戻り値の格納場所を以下に示します。

表 3—17　戻り値の格納場所

戻り値の型	格納方法
1 ビット	CY
1 バイト，2 バイト整数	BC
near ポインタ	BC
4 バイト整数	BC（下位），DE（上位）
far ポインタ	BC（下位），DE（上位）
浮動小数点数	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，DE に格納します。

3. 3. 2　通常関数呼び出しインタフェース

（1）引数の渡し方

- 関数呼び出し時，第 2 引数以降はスタックで関数定義側に渡します。
- 第 1 引数はレジスタ，またはスタックで関数定義側に渡します。

  第 1 引数の受け渡し場所を次に示します。

表 3—18　第 1 引数の渡し場所（関数呼び出し側）

型	格納場所
1 バイト・データ注 2 バイト・データ注	AX
near データ・ポインタ	AX
3 バイト・データ注 4 バイト・データ注	AX，BC
関数ポインタ， far データ・ポインタ	AX，BC
浮動小数点数	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／自動変数の格納場所

- レジスタ宣言，または -qv 指定により，関数の先頭で引数／自動変数をレジスタに割り当てます。それ以外の引数／自動変数は，スタックに積みます。
- 関数呼び出し側からの受け渡しがスタックである引数をレジスタに割り当てない場合は，受け渡し場所がそのまま割り当て場所になります。
- 引数／自動変数を割り当てるレジスタは HL です。ただし，スタック・フレームがある場合は，HL には割り当たりません。

  -qr 指定時は，さらに _@KREGxx にも割り当て可能となります。_@KREGxx については，「3. 4 saddr 領域のラベル一覧」を参照してください。
- 参照回数の多いものから順に，レジスタに割り当てます。

  レジスタ宣言や，-qv を指定しても，参照回数が少ない引数／自動変数は，レジスタに割り当てない場合があります。
- 引数／自動変数を割り当てるレジスタは，関数定義側で退避・復帰します。

（3）例

（a）例 1

C ソースを以下に示します。

```
void    func0 ( register int, int ) ;

void main ( void ) {
        func0 ( 0x1234, 0x5678 ) ;
}

void    func0 ( register int p1, int p2 ) {
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

- -qr 指定の場合

  出力コードは，以下のようになります。

```
_main :
; line  4 :     func0 ( 0x1234, 0x5678 ) ;
        movw    ax,  #05678H    ; 22136
        push    ax                      ; 第 2 引数以降はスタック渡し
        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                      ; 関数呼び出し時に積んだスタックを解放
; line  5 : }
        ret
; line  6 :
; line  7 : void    func0 ( register int p1, int p2 ) {
_func0 :
        push    hl
        movw    de, _@KREG14
        push    de                      ; レジスタ引数用 saddr 領域退避
        movw    de, _@KREG12
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    _@KREG14, ax            ; 第 1 引数 p1 を saddr に割り当てる
        push    ax                      ; 自動変数 a の領域確保
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, [hl+12]     ; p2    ; 引数 p2
        movw    _@KREG12, ax    ; r     ; 自動変数 r
; line 11 :     a = p1 ;
        movw    ax, _@KREG14    ; p1    ; 引数 p1
        movw    [hl], ax        ; a     ; 自動変数 a
; line 12 : }
        pop     ax                      ; 自動変数 a の領域解放
        pop     ax
        movw    _@KREG12, ax            ; レジスタ変数用 saddr 領域復帰
        pop     ax
        movw    _@KREG14, ax            ; レジスタ引数用 saddr 領域復帰
        pop     hl
        ret
```

（b）例 2

C ソースを以下に示します。

```
void    func1 ( int, register int ) ;

void main ( void ) {
        func1 ( 0x1234, 0x5678 ) ;
}

void    func1 ( int p1, register int p2 ) {
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

- -qr 指定の場合

出力コードは，以下のようになります。

```
_main :
; line  4 :     func1 ( 0x1234, 0x5678 ) ;
        movw    ax, #05678H     ; 22136
        push    ax                      ; 第 2 引数以降はスタック渡し
        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                      ; 関数呼び出し時に積んだスタックを解放
; line  5 : }
        ret
; line  6 :
; line  7 : void    func1 ( int p1, register int p2 ) {

_func0 :
        push    hl
        push    ax                      ; 第 1 引数 p1 をスタックに積む
        movw    de, _@KREG14
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    de, _@KREG12
        push    de                      ; レジスタ変数用 saddr 領域退避
        movw    ax, [sp+12]
        movw    _@KREG12, ax            ; 引数 p2 を saddr に割り当てる
        push    ax                      ; 自動変数 a の領域確保
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, _@KREG12    ; p2    ; 引数 p2
        movw    _@KREG14, ax    ; r     ; 自動変数 r
; line 11 :     a = p1 ;
        movw    ax, [hl+6]      ; p1    ; 引数 p1
        movw    [hl], ax        ; a     ; 自動変数 a
; line 12 : }
        pop     ax                      ; 自動変数 a の領域解放
        pop     ax
        movw    _@KREG12, ax            ; レジスタ引数用 saddr 領域復帰
        pop     ax
        movw    _@KREG14, ax            ; レジスタ変数用 saddr 領域復帰
        pop     ax                      ; 第 1 引数 p1 の領域解放
        pop     hl
        ret
```

3. 4　saddr 領域のラベル一覧

RL78,78K0R C コンパイラでは，次に示すラベル名によって 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，次のラベルと同じ名前を使用することはできません。

また，これらのラベル名は，ライブラリ中に定義されていますので，コンパイラがこれらのラベル名を使用している場合には，ライブラリをリンクしてください。

(1) レジスタ変数

ラベル名	アドレス
_@KREG00	0FFEB4H
_@KREG01	0FFEB5H
_@KREG02	0FFEB6H
_@KREG03	0FFEB7H
_@KREG04	0FFEB8H
_@KREG05	0FFEB9H
_@KREG06	0FFEBAH
_@KREG07	0FFEBBH
_@KREG08	0FFEBCH
_@KREG09	0FFEBDH
_@KREG10	0FFEBEH
_@KREG11	0FFEBFH
_@KREG12	0FFEC0H 注
_@KREG13	0FFEC1H 注
_@KREG14	0FFEC2H 注
_@KREG15	0FFEC3H 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。

(2) ワーク用

ラベル名	アドレス
_@NRARG0	0FFEC4H
_@NRARG1	0FFEC6H
_@NRARG2	0FFEC8H
_@NRARG3	0FFECAH
_@NRAT00	0FFECCH
_@NRAT01	0FFECDH
_@NRAT02	0FFECEH

ラベル名	アドレス
_@NRAT03	0FFECFH
_@NRAT04	0FFED0H
_@NRAT05	0FFED1H
_@NRAT06	0FFED2H
_@NRAT07	0FFED3H

（3）セグメント情報格納用

ラベル名	アドレス
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

（4）ランタイム・ライブラリの引数

ラベル名	アドレス
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

3.5　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，以下のとおりです。

- CSEG の再配置属性

CALLT0	指定セグメントを 80H ～ BFH 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（00000H ～ FFEFFH 内）。
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（C0H ～ EFFFEH 内）。

- DSEG の再配置属性

SADDRP	指定セグメントを saddr 領域内の FFE20H ～ FFEFFH 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（デフォルトは RAM 領域内）。

3.5.1　セグメント名一覧

（1）プログラム領域，データ領域

セクション名	セグメント・タイプ	再配置属性	説明
@@CODE	CSEG	BASE	コード部用セグメント（near 領域配置）
@@CODEL	CSEG		コード部用セグメント（far 領域配置）
@@CODER	CSEG		RAM 配置コード部用セグメント
@@LCODE	CSEG	BASE	ライブラリ・コード用セグメント（near 領域配置）
@@LCODEL	CSEG		ライブラリ・コード用セグメント（far 領域配置）
@@LCODER	CSEG		RAM 配置ライブラリ・コード用セグメント
@@CNST	CSEG	MIRRORP	ROM データ（near 領域配置）**注 1**
@@CNSTR	CSEG	MIRRORP（ミラー空間ありの場合）	RAM 配置 ROM データ用セグメント（near 領域配置）
		UNITP（ミラー空間なしの場合）	
@@CNSTL	CSEG	PAGE64KP	ROM データ（far 領域配置）**注 1**
@@CNSTLR	CSEG	PAGE64KP	RAM 配置 ROM データ用セグメント（far 領域配置）
@@R_INIT	CSEG	UNIT64KP	near 初期化データ用セグメント（初期値あり）
@@RLINIT	CSEG	UNIT64KP	far 初期化データ用セグメント（初期値あり）
@@R_INIS	CSEG	UNIT64KP	初期化データ用セグメント（初期値あり sreg 変数）

セクション名	セグメント・タイプ	再配置属性	説明
@@CALT	CSEG	CALLT0	callt 関数のテーブル用セグメント
@@VECT*nn*	CSEG	AT 00*mm*H	ベクタ・テーブル用セグメント 注2
@@BASE	CSEG	BASE	callt 関数・割り込み関数用セグメント
@@LBASE	CSEG	BASE	ライブラリ・callt 関数用セグメント
@@INIT	DSEG	BASEP	データ領域用セグメント（初期値あり，near 領域配置）
@@INITL	DSEG	UNIT64KP	データ領域用セグメント（初期値あり，far 領域配置）
@@DATA	DSEG	BASEP	データ領域用セグメント（初期値なし，near 領域配置）
@@DATAL	DSEG	UNIT64KP	データ領域用セグメント（初期値なし，far 領域配置）
@@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 1. ROM データは，以下のデータを指します。

- const 変数用セグメント
- switch-case 文用のテーブル参照
- 無名文字列定数
- 自動変数の初期値データ

2. 割り込みの種類により，*nn*，*mm* の値が変わります。

（2）フラッシュ・メモリ領域

セクション名	セグメント・タイプ	再配置属性	説明
@ECODE	CSEG	BASE	コード部用セグメント（near 領域配置）
@ECODEL	CSEG		コード部用セグメント（far 領域配置）
@ECODER	CSEG		RAM 配置コード部用セグメント
@LECODE	CSEG	BASE	ライブラリ・コード用セグメント（near 領域配置）
@LECODEL	CSEG		ライブラリ・コード用セグメント（far 領域配置）
@LECODER	CSEG		RAM 配置ライブラリ・コード用セグメント
@ECNST	CSEG	MIRRORP	ROM データ（near 領域配置） 注1
@ECNSTR	CSEG	MIRRORP（ミラー空間ありの場合）	RAM 配置 ROM データ用セグメント（near 領域配置）
		UNITP（ミラー空間なしの場合）	
@ECNSTL	CSEG	PAGE64KP	ROM データ（far 領域配置） 注1
@ECNSTLR	CSEG	PAGE64KP	RAM 配置 ROM データ用セグメント（far 領域配置）
@ER_INIT	CSEG	UNIT64KP	near 初期化データ用セグメント（初期値あり）
@ERLINIT	CSEG	UNIT64KP	far 初期化データ用セグメント（初期値あり）

セクション名	セグメント・タイプ	再配置属性	説明
@ER_INIS	CSEG	UNIT64KP	初期化データ用セグメント（初期値あり sreg 変数）
@EVECT*nn*	CSEG	AT *mmmm*H	ベクタ・テーブル用セグメント注2
@EXT*xx*	CSEG	AT *yyyy*H	フラッシュ領域分岐テーブル用セグメント注3
@EINIT	DSEG	BASEP	データ領域用セグメント（初期値あり，near 領域配置）
@EINITL	DSEG	UNIT64KP	データ領域用セグメント（初期値あり，far 領域配置）
@EDATA	DSEG	BASEP	データ領域用セグメント（初期値なし，near 領域配置）
@EDATAL	DSEG	UNIT64KP	データ領域用セグメント（初期値なし，far 領域配置）
@EINIS	DSEG	SADDRP	データ領域用セグメント（初期値あり sreg 変数）
@EDATS	DSEG	SADDRP	データ領域用セグメント（初期値なし sreg 変数）
@EBITS	BSEG		boolean 型変数，bit 型変数用セグメント
@ECALT	CSEG		ダミー・セグメント
@EBASE	CSEG	BASE	ダミー・セグメント

注 1. ROM データは，以下のデータを指します。

- const 変数用セグメント
- switch-case 文用のテーブル参照
- 無名文字列定数
- 自動変数の初期値データ

2. 割り込みの種類により，*nn*，*mmmm* の値が変わります。

3. フラッシュ領域関数の ID により，*xx*，*yyyy* の値が変わります。

3. 5. 2 セグメントの配置

セグメント・タイプ	配置先（デフォルト）
CSEG	ROM
BSEG	RAM の saddr 領域
DSEG	RAM

3. 5. 3 C ソース例

```
#pragma INTERRUPT       INTP0   inter   rb1     /* 割り込みベクタ */

void inter ( void ) ;                           /* 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 */
}
```

3. 5. 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。

詳細については，「第 4 章 アセンブラ言語仕様」を参照してください。

(1) スモール・モデルの場合

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cf104le sample.c -ms -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( F104LE )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH, 0xxxH, 00H, 00H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _i_cnst
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _main
```

```
            PUBLIC  _f_clt
            PUBLIC  ?f_clt
            PUBLIC  _@vect08

@@BITS      BSEG                        ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; const 変数用セグメント
_i_cnst :   DW      01H             ; 1

@@R_INIT    CSEG    UNIT64KP            ; 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    BASEP               ; 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    BASEP               ; 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    UNIT64KP            ; 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 関数のテーブル用セグメント
?f_clt :    DW      _f_clt

; line  1 : #pragma interrupt   INTP0   inter   rb1     /* 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 */

@@CODE      CSEG  BASE                  ; コード部用セグメント
```

```
_main :
        push    hl                      ;[INF] 1, 1
; line 13 :         int     i ;
; line 14 :         i = 100 ;
        movw    hl, #064H       ; 100   ;[INF] 3, 1
; line 15 : }
        pop   hl                        ;[INF] 1, 1
        ret                             ;[INF] 1, 6
; line 16 :
; line 17 : void    inter ( ) {                 /* 割り込み関数定義 */

@@BASE          CSEG    BASE            ; callt 関数・割り込み関数用セグメント
_inter :
        sel     RB1                     ;[INF] 2, 1 レジスタ・バンク 1 を選択
; line 18 : unsigned char   uc = 0;
        mov     l,#00H          ; 0     ;[INF] 2, 1
; line 19 : uc++;
        inc     l                       ;[INF] 1, 1
; line 20 : if (b_bit)
        bf      _b_bit,$L0005           ;[INF] 4, 5
; line 21 : b_bit = 0;
        clr1    _b_bit                  ;[INF] 3, 2
L0005:
; line 22 : }
        reti                            ;[INF] 2, 6
; line 23 :
; line 24 : __callt void    f_clt ( ) {         /* callt 関数定義 */
_f_clt :
; line 25 : }
         ret
@@VECT08        CSEG    AT      0008H   ; ベクタ・テーブル用セグメント
_@vect08 :
                DW      _inter
                END


; *** Code Information ***
;
; $FILE D:¥CA78K0R¥Vx.xx¥Smp78k0r¥cc78k0r¥sample.c
;
; $FUNC main(12)
;         void=(void)
;         CODE SIZE= 6 bytes, CLOCK_SIZE= 9 clocks, STACK_SIZE= 2 bytes
;
; $FUNC inter(17)
```

```
;         void=(void)
;         CODE SIZE= 14 bytes, CLOCK_SIZE= 16 clocks, STACK_SIZE= 0 bytes
;
; $FUNC f_clt(24)
;         void=(void)
;         CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes

; Target chip : R5F104LE
; Device file : Vx.xx
```

(2) ミディアム・モデルの場合

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cf104le sample.c -mm -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( F104LE )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH, 0xxxH, 00H, 04000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _i_cnst
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _main
            PUBLIC  _f_clt
            PUBLIC  ?f_clt
            PUBLIC  _@vect08

@@BITS      BSEG                        ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; const 変数用セグメント
_i_cnst :   DW      01H                 ; 1

@@R_INIT    CSEG    UNIT64KP            ; 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    BASEP               ; 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    BASEP               ; 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    UNIT64KP            ; 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 関数のテーブル用セグメント
?f_clt :    DW      _f_clt

; line  1 : #pragma interrupt   INTP0   inter   rb1     /* 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 */

@@CODEL     CSEG                        ; コード部用セグメント
_main :
         push    hl                         ;[INF] 1, 1
; line 13 :          int     i ;
; line 14 :          i = 100 ;
         movw    hl, #064H          ; 100   ;[INF] 3, 1
; line 15 : }
         pop     hl                         ;[INF] 1, 1
         ret                                ;[INF] 1, 6
; line 16 :
; line 17 : void    inter ( ) {                 /* 割り込み関数定義 */

@@BASE  CSEG    BASE                    ; callt 関数・割り込み関数用セグメント
_inter :
```

```
        sel     RB1                             ;[INF] 2, 1 レジスタ・バンク 1 を選択
; line 18 : unsigned char   uc = 0;
        mov     l,#00H  ; 0             ;[INF] 2, 1
; line 19 : uc++;
        inc     l                       ;[INF] 1, 1
; line 20 : if (b_bit)
        bf      _b_bit,$L0005           ;[INF] 4, 5
; line 21 : b_bit = 0;
        clr1    _b_bit                  ;[INF] 3, 2
L0005:
; line 22 : }
        reti
; line 23 :                               ;[INF] 2, 6
; line 24 : __callt void    f_clt ( ) {          /* callt 関数定義 */
_f_clt :
; line 25 : }                             ;[INF] 1, 6
        ret
@@VECT08 CSEG    AT      0008H          ; ベクタ・テーブル用セグメント
_@vect08 :
        DW      _inter
        END


; *** Code Information ***
;
; $FILE D:¥CA78K0R¥Vx.xx¥Smp78k0r¥cc78k0r¥sample.c
;
; $FUNC main(12)
;       void=(void)
;       CODE SIZE= 6 bytes, CLOCK_SIZE= 9 clocks, STACK_SIZE= 2 bytes
;
; $FUNC inter(17)
;       void=(void)
;       CODE SIZE= 14 bytes, CLOCK_SIZE= 16 clocks, STACK_SIZE= 0 bytes
;
; $FUNC f_clt(24)
;       void=(void)
;       CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes

; Target chip : R5F104LE
; Device file : Vx.xx
```

(3) ラージ・モデルの場合

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cf104le sample.c -ml -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( F104LE )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH, 0xxxH, 00H, 08000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _i_cnst
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _main
            PUBLIC  _f_clt
            PUBLIC  ?f_clt
            PUBLIC  _@vect08

@@BITS      BSEG                        ; boolean 型変数，bit 型変数用セグメント
_b_bit      DBIT

@@R_INIS    CSEG    UNIT64KP            ; 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 )

@@CNSTL     CSEG    PAGE64KP            ; const 変数用セグメント
_i_cnst :   DW      01H             ; 1

@@RLINIT    CSEG    UNIT64KP            ; 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

@@INITL     DSEG  UNIT64KP              ; データ領域用セグメント（初期値あり外部変数）
```

```
_l_init :   DS    ( 4 )

@@DATAL     DSEG  UNIT64KP              ; データ領域用セグメント（初期値なし外部変数）
_i_data :   DS    ( 2 )

@@CALT      CSEG    CALLT0              ; callt 関数のテーブル用セグメント
?f_clt :    DW      _f_clt

; line  1 : #pragma interrupt   INTP0   inter   rb1     /* 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 */

@@CODEL     CSEG                          ; コード部用セグメント
_main :
        push    hl                      ;[INF] 1, 1
; line 13 :int     i ;
; line 14 :i = 100 ;
        movw    hl, #064H     ; 100     ;[INF] 3, 1
; line 15 : }
        pop     hl                      ;[INF] 1, 1
        ret                             ;[INF] 1, 6
; line 16 :
; line 17 : void    inter ( ) {                  /* 割り込み関数定義 */

@@BASE      CSEG    BASE                ; callt 関数・割り込み関数用セグメント
_inter :
        sel     RB1                     ;[INF] 2, 1 レジスタ・バンク 1 を選択
; line 18 : unsigned char   uc = 0;
        mov     l,#00H          ; 0     ;[INF] 2, 1
; line 19 : uc++;
        inc     l                       ;[INF] 1, 1
; line 20 : if (b_bit)
        bf      _b_bit,$L0005           ;[INF] 4, 5
; line 21 : b_bit = 0;
        clr1    _b_bit                  ;[INF] 3, 2
L0005:
```

```
; line 22 : }
        reti                            ;[INF] 2, 6
; line 23 :
; line 24 : __callt void    f_clt ( ) {         /* callt 関数定義 */
_f_clt :
; line 25 : }
        ret
@@VECT08    CSEG    AT      0008H       ; ベクタ・テーブル用セグメント
_@vect08 :
            DW      _inter
            END



; *** Code Information ***
;
; $FILE D:¥CA78K0R¥Vx.xx¥Smp78k0r¥cc78k0r¥sample.c
;
; $FUNC main(12)
;       void=(void)
;       CODE SIZE= 6 bytes, CLOCK_SIZE= 9 clocks, STACK_SIZE= 2 bytes
;
; $FUNC inter(17)
;       void=(void)
;       CODE SIZE= 14 bytes, CLOCK_SIZE= 16 clocks, STACK_SIZE= 0 bytes
;
; $FUNC f_clt(24)
;       void=(void)
;       CODE SIZE= 1 bytes, CLOCK_SIZE= 6 clocks, STACK_SIZE= 0 bytes

; Target chip : R5F104LE
; Device file : Vx.xx
```

第４章　アセンブラ言語仕様

この章では，RL78,78K0R アセンブラがサポートするアセンブラ言語仕様について説明します。

4.1　ソースの記述方法

この節では，ソースの記述方法，式と演算子などについて説明します。

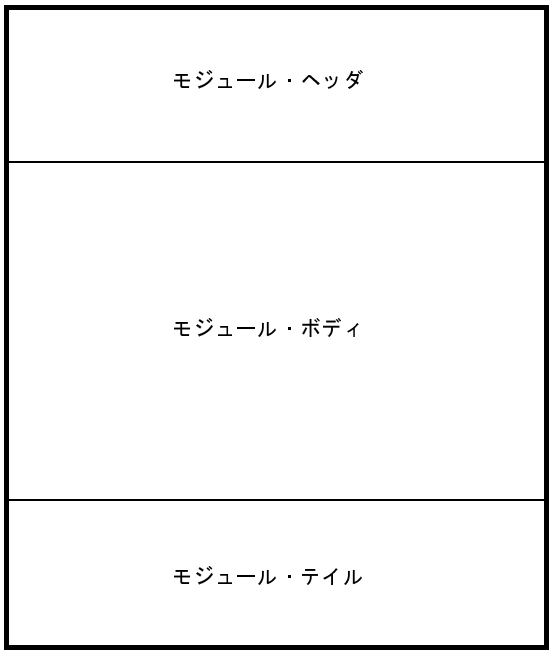
4.1.1　基本構成

１つのソースをいくつかのモジュールに分割して記述したとき，アセンブラの入力単位となる各モジュールをソース・モジュールと呼びます（プログラムが１つのモジュールからなるとき，ソースとソース・モジュールは同じ意味を持ちます）。

アセンブラの入力単位となるソース・モジュールは，大まかには次の３つの構成部分からなります。

- モジュール・ヘッダ（Module Header）
- モジュール・ボディ（Module Body）
- モジュール・テイル（Module Tail）

図 4―1　ソース・モジュールの構成



(1) モジュール・ヘッダ

次に，モジュール・ヘッダに記述できる制御命令を示します。これらの制御命令は，モジュール・ヘッダ以外には記述できません。

また，モジュール・ヘッダは省略することが可能です。

表 4—1 モジュール・ヘッダに記述できるもの

記述できるもの	説明
アセンブラ・オプションと同様の機能を持つ制御命令	- PROCESSOR - DEBUG/NODEBUG/DEBUGA/NODEBUGA - XREF/NOXREF - SYMLIST/NOSYMLIST - TITLE - FORMFEED/NOFORMFEED - WIDTH - LENGTH - TAB - KANJICODE
C コンパイラなどの上位プログラムが出力する特別な制御命令	- TOL_INF - DGS - DGL

(2) モジュール・ボディ

モジュール・ボディには，次のものは記述できません。

- アセンブラ・オプションと同様の機能を持つ制御命令

上記以外のすべての疑似命令，制御命令，インストラクションは，モジュール・ボディに記述可能です。

また，モジュール・ボディは，セグメントと呼ぶ単位に分割して記述します。

セグメントは，それぞれ対応する疑似命令で定義します。

- コード・セグメント

  CSEG 疑似命令で定義します。

- データ・セグメント

  DSEG 疑似命令で定義します。

- ビット・セグメント

  BSEG 疑似命令で定義します。

- アブソリュート・セグメント

  CSEG，DSEG，BSEG 疑似命令で，再配置属性に配置アドレス（AT 配置アドレス）を指示してセグメントを定義します。また，ORG 疑似命令で定義することもできます。

モジュール・ボディは，どのようなセグメントの組み合わせで構成してもかまいません。

ただし，データ・セグメントやビット・セグメントの定義は，コード・セグメントの定義よりも前で行うようにしてください。

（3）モジュール・テイル

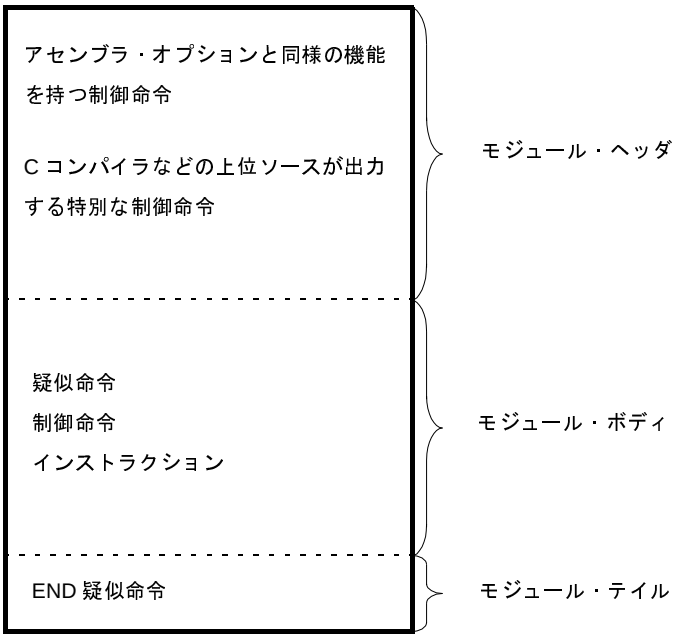
モジュール・テイルは，ソース・モジュールの終了を示すもので，END 疑似命令を記述します。

END 疑似命令のあとにコメント，空白，タブ，改行コード以外のものを記述すると，警告メッセージが出力され，それらは無視されます。

（4）ソースの全体構成

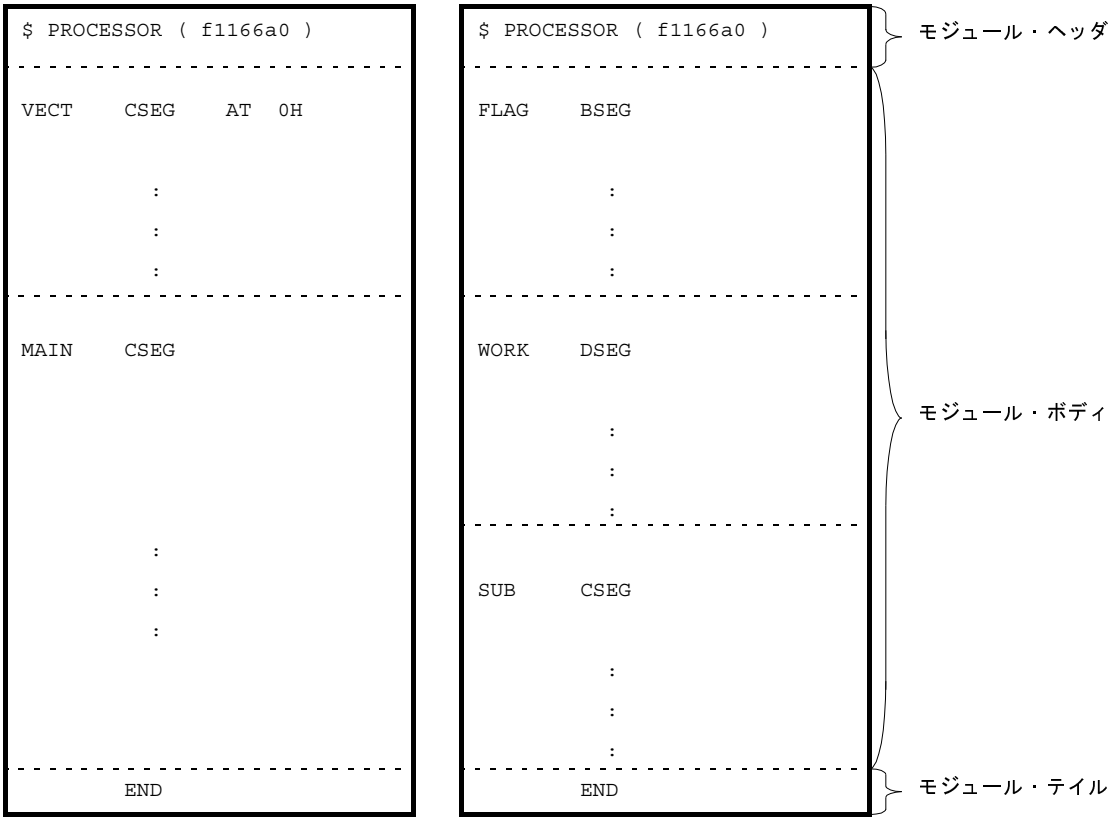
次に，ソース・モジュール（ソース）の全体構成を示します。

図 4―2　ソース・モジュールの全体構成



また，次に，ソース・モジュールの構成例を簡単に示します。

図 4—3 ソース・モジュールの構成例

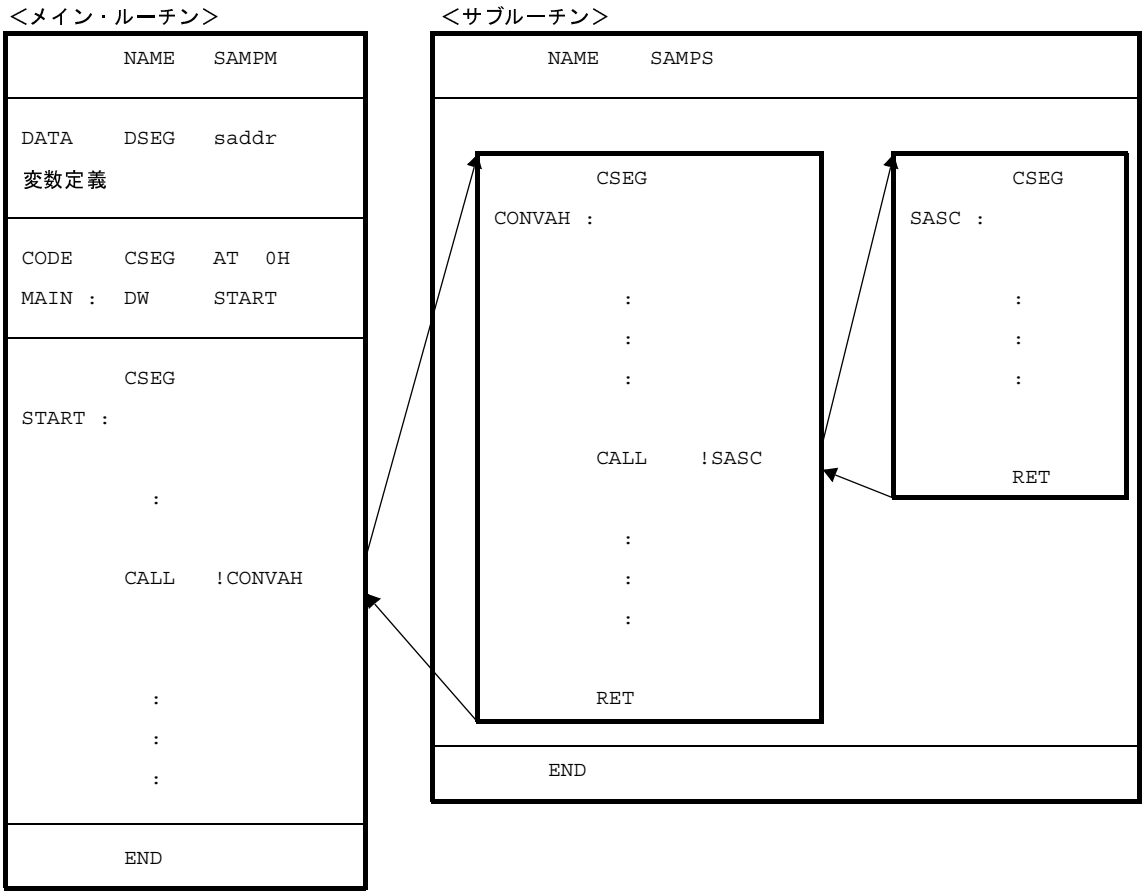


(5) 記述例

ここで，ソース・モジュール（ソース）の記述例をサンプル・プログラムとして示します。

次に，サンプル・プログラムの構成を簡単に示します。

図 4—4 サンプル・プログラムの構成



- メイン・ルーチン

```
        NAME    SAMPM                   ; (1)
; ********************************************
;       HEX -> ASCII Conversion Program
;               main-routine
; ********************************************

PUBLIC  MAIN, START                     ; (2)
EXTRN   CONVAH                          ; (3)
EXTRN   _@STBEG                         ; (4)   ←エラー


DATA    DSEG    AT      0FFE20H         ; (5)
HDTSA : DS      1
STASC : DS      2
```

```
CODE    CSEG    AT      0H              ; (6)
MAIN :  DW      START

        CSEG                            ; (7)
START :
        ; chip initialize
        MOVW    SP, #_@STBEG

        MOV     HDTSA, #1AH
        MOVW    HL, #LOWW ( HDTSA )     ; set hex 2-code data in HL registor

        CALL    !CONVAH                 ; convert ASCII <- HEX
                                        ; output BC-register <- ASCII code
        MOVW    DE, #LOWW ( STASC )     ; set DE <- store ASCII code table
        MOV     A, B
        MOV     [DE], A
        INCW    DE
        MOV     A, C
        MOV     [DE], A
        BR      $$
        END                             ; (8)
```

(1) モジュール名を宣言

(2) ほかのモジュールから参照されるシンボルを外部定義シンボルとして宣言

(3) ほかのモジュールで定義されているシンボルを外部参照シンボルとして宣言

(4) リンカの -s オプションで生成されるスタック解決用シンボルを外部参照シンボルとして宣言（リンクする際に -s オプションを指定しないとエラーになる）

(5) データ・セグメントの開始を宣言（saddr に配置する）

(6) コード・セグメントの開始を宣言（アブソリュート・セグメントとして 0H 番地から配置する）

(7) コード・セグメントの開始を宣言（アブソリュート・セグメントの終了を意味する）

(8) モジュールの終了を宣言

- サブルーチン

```
        NAME    SAMPS           ;  (9)
; ************************************************************
;       HEX -> ASCII Conversion Program
;       sub-routine
;
;   input condition     : ( HL )        <- hex 2 code
;   output condition    : BC-register   <- ASCII 2 code
; ************************************************************

PUBLIC  CONVAH                  ;  (10)

        CSEG                    ;  (11)
CONVAH :
        XOR     A, A
        ROL4    [HL]            ; hex upper code load  (12)
        CALL    !SASC
        MOV     B, A            ; store result

        XOR     A, A
        ROL4    [HL]            ; hex lower code load
        CALL    !SASC
        MOV     C, A            ; store result
        RET

; ************************************************************
;       subroutine   convert ASCII code
;
;   input       Acc ( lower 4bits )     <- hex code
;   output      Acc                     <- ASCII code
; ************************************************************

SASC :
        CMP     A, #0AH         ; check hex code > 9
        BC      $SASC1
        ADD     A, #07H         ; bias ( +7H )
SASC1 :
        ADD     A, #30H         ; bias ( +30H )
        RET
        END                     ;  (13)
```

(9) モジュール名を宣言

(10) ほかのモジュールから参照されるシンボルを外部定義シンボルとして宣言

（11）コード・セグメントの開始を宣言

（12）ROL4 命令は，RL78,78K0R ではサポートしていない 78K0 用の命令であるため，アセンブラ・オプション（-compati）の指定が必要です。
アセンブラ・オプション（-compati）については，「CubeSuite+ 統合開発環境 ユーザーズマニュアル RL78,78K0R ビルド編」を参照してください。

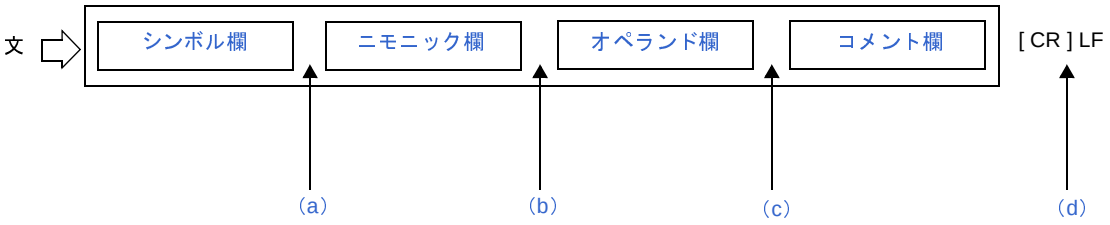
（13）モジュールの終了を宣言

4. 1. 2 記述方法

（1）構成

ソースは，文（ステートメント）で構成します。
1 つの文は，次に示す 4 つのフィールドで構成します。

図 4―5 文の構成フィールド



（a）シンボル欄とニモニック欄は，コロン（：），または 1 つ以上の空白（または TAB）で区切ります（コロンで区切るか空白で区切るかは，ニモニック欄で記述する命令により異なります）。

（b）ニモニック欄とオペランド欄は，1 つ以上の空白（または TAB）で区切ります。ニモニック欄に記述する命令によっては，オペランド欄が必要ない場合もあります。

（c）コメント欄を記述するときは，コメント欄の前にセミコロン（；）を記述します。

（d）各行の終わりは，LF で区切ります（LF の直前に CR が 1 つ存在してもかまいません）。

- 1 つの文は 1 行以内に記述します。1 行には最大 2048 文字（CR/LF を含む）まで記述することができます。このとき，TAB，および単独の CR は，それぞれ 1 文字として数えます。2049 文字以上記述した場合には，警告メッセージが出力され，2049 文字以降は無視されます。ただし，アセンブル・リストには 2049 文字以降も出力されます。
- 単独の CR は，アセンブル・リストには出力されません。
- 次のような行の記述が可能です。
  - 空行（文の記述のない行）
  - シンボル欄のみの行

- コメント欄のみの行

(2) 文字セット

ソース・ファイル中に記述可能な文字は，次の 3 つから構成されます。

- 言語文字
- 文字データ
- 注釈（コメント）用文字

(a) 言語文字

ソース上で命令を記述するために使用する文字です。

言語文字の文字セットには，英数字，および特殊文字があります。

表 4—2 英数字

名称		文字
数字		0 1 2 3 4 5 6 7 8 9
英字	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
	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

表 4—3 特殊文字

文字	名称	主な用途	
?	疑問符	英字相当文字	
@	単価記号	英字相当文字	
_	下線	英字相当文字	
	空白	各欄の区切り記号	区切り記号
HT (09H)	タブ・コード	空白相当文字	
,	コンマ	オペランドの区切り文字	
:	コロン	ラベル区切り記号	
;	セミコロン	コメント欄開始記号	
CR (0DH)	復帰コード	1 行の最終記号（アセンブラでは無視）	
LF (0AH)	改行コード	1 行の最終記号	
+	プラス	加算演算子，または正符号	アセンブラ演算子
-	マイナス	減算演算子，または負符号	
*	アスタリスク	乗算演算子	
/	スラッシュ	除算演算子	
.	ピリオド	ビット位置指定子	
( )	左右かっこ	演算順序	
< >	不等号	比較演算子	
=	等号	比較演算子	

文字	名称	主な用途
'	引用符	- 文字定数の開始・終了記号 - マクロの引数を１つにまとめる記号
$	ドル記号	- ロケーション・カウンタの値 - アセンブラ・オプションに相当する制御命令の開始記号 - 相対アドレッシング指定記号
&	アンパーサンド	コンカティネート記号（マクロ・ボディ内で使用）
#	シャープ	イミーディエト・アドレッシング指定記号
!	感嘆符	絶対アドレッシング指定記号
[]	大かっこ	インダイレクト・アドレッシング指定記号

（b）文字データ

文字データは，文字列定数，文字列，および制御命令部（TITLE，SUBTITLE，INCLUDE）を記述するために使用する文字です。

注意 1. 00H を除くすべての文字（漢字かなを含みます。ただし，OS によってコードは異なります）が記述可能です。00H を記述するとエラーとなり，それ以降，引用符（'）で閉じるまで無視されます。

2. 不正文字が入力された場合，アセンブラは，不正文字を“!”に置き換えてアセンブル・リストに出力します（CR（0DH）は，アセンブル・リストに出力されません）。

3. Windows では，1AH をファイルの末尾と判断するため，入力データとはなりません。

（c）注釈（コメント）用文字

コメントを記述するために使用する文字です。

注意 文字データの文字セットと同一です。ただし，00H が入力されてもエラーは出力されません。アセンブル・リストには“!”に置き換えて出力されます。

（3）シンボル欄

シンボル欄には，シンボルを記述します。シンボルとは，数値データやアドレスなどに付けた名前のことです。

シンボルを使用することにより，ソースの内容がわかりやすくなります。

（a）シンボルの種類

シンボルは，その使用目的，定義方法によって，次に示す種類に分けられます。

シンボルの種類	使用目的	定義方法
ネーム	ソース中で，数値データやアドレスとして使用	EQU，SET，DBIT 疑似命令等のシンボル欄に記述します。
ラベル	ソース中で，アドレス・データとして使用	シンボルのあとにコロン（：）を付けることにより定義します。

シンボルの種類	使用目的	定義方法
外部参照名	あるモジュールで定義されたシンボルをほかのモジュールで参照するときに使用	EXTRN，EXTBIT 疑似命令のオペランド欄に記述します。
セグメント名	リンク時に使用	CSEG，DSEG，BSEG，ORG 疑似命令のシンボル欄に定義します。
モジュール名	シンボリック・デバッグ時に使用	NAME 疑似命令のオペランド欄に記述します。
マクロ名	ソース中で，マクロ参照時に使用	MACRO 疑似命令のシンボル欄に記述します。

注意　シンボル欄に記述可能なシンボルは，ネーム，ラベル，セグメント名，マクロ名の 4 種類です。

（b）シンボル記述上の規則

シンボルは，次の規則に基づいて記述します。

- シンボルは，英数字，および英字相当文字（?，@，_）で構成します。
  ただし，先頭文字に数字（0 ～ 9）は使用できません。
- シンボルの長さは，1 ～ 256 文字です。
  認識最大文字数を越えた文字は無視されます。
- シンボルとして，予約語は使用できません。
  予約語については，「4.5　予約語」を参照してください。
- 同一シンボルを二度以上定義することはできません。
  ただし，SET 疑似命令で定義したネームは，SET 疑似命令で再定義することができます。
- アセンブラは，シンボルの大文字／小文字を区別します。
- シンボル欄にラベルを記述する場合は，ラベルの直後にコロン（:）を記述します。

正しいシンボルの例を以下に示します。

```
CODE01   CSEG            ;  “CODE01”はセグメント名
VAR01    EQU     10H     ;  “VAR01”はネーム
LAB01 :  DW      0       ;  “LAB01”はラベル
         NAME    SAMPLE  ;  “SAMPLE”はモジュール名
MAC1     MACRO           ;  “MAC1”はマクロ名
```

誤ったシンボルの例を以下に示します。

```
1ABC     EQU     3       ; 先頭文字に数字は使用できません。
LAB      MOV     A, R0   ;  “LAB”ラベルです。ニモニック欄とコロン（ : ）で区切ります。
FLAG :   EQU     10H     ; ネームにはコロン（ : ）が必要ありません。
```

長いシンボルの例を以下に示します。

```
A123456789B12 ～ Y123456789Z123456      EQU     70H
        257 文字
                                ; 認識最大文字数（256 文字）を越えた文字“6”は
                                ; 無視されます。
                                ; A123456789B12 ～ Y123456789Z12345
                                ; というシンボルが定義されていることになります。
```

シンボルのみからなる文の例を以下に示します。

```
ABCD :                          ; ABCD がラベルとして定義されます。
```

（c）シンボルに関する注意事項

??RAnnnn（nnnn = 0000 ～ FFFF）というシンボルは，マクロ・ボディ内のローカル・シンボルが展開されるごとに，アセンブラによって自動的に置き換えられるシンボルであるため，二重定義しないように注意してください。

また，セグメント定義疑似命令でセグメント名が指定されなかったときは，アセンブラがセグメント名を自動生成します。次に，そのセグメントを示します。

同名で定義するとエラーとなります。

セグメント名	疑似命令	再配置属性
?A0nnnnn（nnnnn = 00000 ～ FFFFF）	ORG 疑似命令	（なし）
?CSEG	CSEG 疑似命令	UNIT
?CSEGUP		UNITP
?CSEGT0		CALLT0
?CSEGFX		FIXED
?CSEGSI		SECUR_ID
?CSEGB		BASE
?CSEGP64		PAGE64KP
?CSEGU64		UNIT64KP
?CSEGMIP		MIRRORP
?CSEGOB0		OPT_BYTE
?DSEG	DSEG 疑似命令	UNIT
?DSEGUP		UNITP
?DSEGS		SADDR
?DSEGSP		SADDRP
?DSEGBP		BASEP
?DSEGP64		PAGE64KP
?DSEGU64		UNIT64KP
?BSEG	BSEG 疑似命令	UNIT

（d）シンボルの属性

ネーム，およびラベルは，値と属性を持ちます。

値とは，定義された数値データやアドレス・データの値そのものです。

セグメント名，モジュール名，およびマクロ名は，値を持ちません。

属性とは，次に示すシンボル属性のことです。

属性の種類	区分	値
NUMBER	- 数値定数を割り付けたネーム - EXTRN 疑似命令で定義されたシンボル - 定数	10 進表現 ： 0 ～ 1048575 16 進表現 ： 00000H ～ FFFFFH （符号なし）
ADDRESS	- ラベルとして定義されたシンボル - ラベルを EQU，SET 疑似命令で定義したネーム	10 進表現 ： 0 ～ 1048575 16 進表現 ： 0H ～ FFFFFH
BIT	- ビット値として定義されたネーム - BSEG 内のネーム - EXTBIT 疑似命令で定義されたシンボル	0H ～ FFFFFH
SFR	SFR を EQU 疑似命令で定義したネーム	SFR 領域
SFRP	SFR を EQU 疑似命令で定義したネーム	
CSEG	CSEG 疑似命令で定義されたセグメント名	値を持ちません
DSEG	DSEG 疑似命令で定義されたセグメント名	
BSEG	BSEG 疑似命令で定義されたセグメント名	
MODULE	NAME 疑似命令で定義されたモジュール名 （指定されなかった場合は，入力ソース・ファイル名のプライマリ・ネームから作成されます）	
MACRO	MACRO 疑似命令で定義されたマクロ名	

例

```
TEN      EQU     10H          ; ネーム“TEN”は NUMBER 属性と値 10H を持ちます。
         ORG     80H
START :  MOV     A, #10H      ; ラベル“START”は，ADDRESS 属性と値 80H を持ちます。
BIT1     EQU     0FFE20H.0    ; ネーム“BIT1”は，BIT 属性と値 0FFE20H.0 を持ちます。
```

（4）ニモニック欄

ニモニック欄には，インストラクションのニモニック，疑似命令，およびマクロ参照を記述します。

オペランドの必要なインストラクションや疑似命令の場合，ニモニック欄とオペランド欄を 1 つ以上の空白，または TAB で区切ります。

ただし，インストラクションの第 1 オペランドの先頭が“#”，“$”，“！”，“［”の場合には，ニモニックと第 1 オペランドの間に何もなくても，正常にアセンブルが行われます。

正しい例：

```
MOV     A, #0H
CALL    !CONVAH
RET
```

誤った例：

```
MOVA    #0H             ; ニモニック欄とオペランド欄の間に，空白がありません。
C ALL    !CONVAH        ; ニモニック中に空白があります。
ZZZ                     ; RL78,78K0R の命令には，“ZZZ”はありません。
```

(5) オペランド欄

オペランド欄には，インストラクションや疑似命令，およびマクロ参照の実行に必要なデータ（オペランド）を記述します。

各インストラクションや疑似命令により，オペランドを必要としないものや，複数のオペランドを必要とするものがあります。

2 個以上のオペランドを記述する場合には，各オペランドをコンマ（ , ）で区切ります。

オペランド欄に記述できるものは，次のものです。

- 定数（数値定数，文字列定数）
- 文字列
- レジスタ名
- 特殊文字（$ # ! [ ]）
- セグメント定義疑似命令の再配置属性
- シンボル
- 式
- ビット項

なお，各インストラクションや疑似命令により，要求するオペランドのサイズ，属性などが異なります。これらについては「4. 1. 16 オペランドの特性」を参照してください。

また，インストラクション・セットにおけるオペランドの表現形式と記述方法については，開発対象となる各デバイスのユーザーズ・マニュアルを参照してください。

以降に，オペランド欄に記述可能な各項目について説明します。

(a) 定数

定数は，それ自身で定まる値を持つもので，イミーディエト・データとも呼びます。

定数には，数値定数と文字列定数があります。

- 数値定数

数値定数として，2 進数，8 進数，10 進数，16 進数が記述可能です。

次に，各数値定数の表現方法を示します。

数値定数は，符号なしの 32 ビット・データとして処理されます。

値の範囲 0 ≦ n ≦ 0FFFFFFFFH

マイナスの値を記述するには，演算子のマイナス符号を使用します。

数値定数の種類	表記方法	表記例
2 進数	数値の最後に文字“B”，または“Y”を付加	1101B 1101Y
8 進数	数値の最後に文字“O”，または“Q”を付加	74O 74Q
10 進数	数値をそのまま記述，または最後に文字“D”，または“T”を付加	128 128D 128T
16 進数	- 数値の最後に文字“H”を付加 - 先頭文字が“A”，“B”，“C”，“D”，“E”，“F”で始まる場合には，その前に“0”を付加	8CH 0A6H

- 文字列定数

文字列定数は，「(2) 文字セット」で示した文字を引用符（’）で囲んだものです。

文字列定数は，アセンブルされた結果，パリティ・ビットを 0 とした 7 ビット ASCII コードに変換されます。

文字列の長さは 0 ～ 2 です。

引用符自体を文字列定数とする場合には，引用符を 2 個続けて記述します。

例

```
'ab'            ; 6162H
'A'             ; 0041H
'A'''           ; 4127H
' '             ; 0020H（空白 1 個）
```

（b）文字列

文字列は，「(2) 文字セット」で示した文字を引用符（’）で囲んだものです。

文字列は，DB，CALL 疑似命令や TITLE，SUBTITLE 制御命令のオペランドに使用します。

例

```
        CSEG
MAS1 :  DB      'YES'           ; 文字列“YES”で初期化します。
MAS2 :  DB      'NO'            ; 文字列“NO”で初期化します。
```

（c）レジスタ名

オペランド欄に記述可能なレジスタとして，次のものがあります。

- 汎用レジスタ
- 汎用レジスタ・ペア

- 特殊機能レジスタ

汎用レジスタや汎用レジスタ・ペアは，絶対名称（R0 ～ R7，RP0 ～ RP3）での記述のほかに，機能名称（X，A，B，C，D，E，H，L，AX，BC，DE，HL）での記述も可能です。

なお，インストラクションの種類により，オペランド欄に記述可能なレジスタ名が異なります。各レジスタの記述方法の詳細については，開発対象となる各デバイスのユーザーズ・マニュアルを参照してください。

(d) 特殊文字

次に，記述可能な特殊文字を示します。

特殊文字	機能
$	- このオペランドを待つインストラクションが割り当てられているロケーション・アドレス（複数バイト命令の場合は 1 バイト目）を示します。 - 分岐命令の相対アドレッシングを示します。
!	- 分岐命令の絶対アドレッシングを示します。 - MOV 命令の全メモリ空間指定可能な addr16 指定を示します。
#	- イミーディエト・データを示します。
[ ]	- インダイレクト・アドレッシングを示します。

例

```
アドレス      ソース
100             ADD     A, #10H
102     LOOP :  INC     A
103             BR      $$ - 1          ; オペランドの 2 番目の $ は，103H 番地を示します。
                                        ; “BR $ - 1”と記述しても同様に動作します。
105             BR      !$ + 100H       ; オペランドの 2 番目の $ は，105H 番地を示します。
                                        ; “BR $ + 100H”と記述しても同様に動作します。
```

(e) セグメント定義疑似命令の再配置属性

オペランド欄には，再配置属性を記述するとこができます。

再配置属性の詳細については，「4. 2. 2 セグメント定義疑似命令」を参照してください。

(f) シンボル

シンボルをオペランド欄に記述した場合は，そのシンボルに割り付けられたアドレス（または値）がオペランドの値になります。

例

```
VALUE   EQU     1234H
        MOV     AX, #VALUE      ; MOV     AX, #1234H と記述することができます。
```

（g）式

式は，定数，ロケーション・アドレスを示す $，ネーム，またはラベルを演算子で結合したものです。
インストラクションのオペランドとして数値表現可能なところに記述することができます。
式と演算子については，「4. 1. 3 式と演算子」を参照してください。

例

```
TEN     EQU     10H
        MOV     A, #TEN - 5H
```

この記述例では，“TEN - 5H”が式です。
この式は，ネームと数値定数が -（マイナス）演算子で結合されています。式の値は“0BH”です。
したがって，この記述は“MOV A, #0BH”と書き換えることが可能です。

（h）ビット項

ビット項は，ビット位置指定子によって得ることができます。
ビット項の詳細については，「4. 1. 14 ビット位置指定子」を参照してください。

例

```
        CLR1    A.5
        SET1    1 + 0FFE30H.3   ; オペランドの値は 0FFE31H.3 です。
        CLR1    0FFE40H.4 + 2   ; オペランドの値は 0FFE40H.6 です。
```

（6）コメント欄

コメント欄には，セミコロン（ ; ）のあとにコメント（注釈）を記述します。
コメント欄は，セミコロンからその行の改行コード，または EOF までです。
コメントを記述することにより，理解しやすいソースを作成できます。
コメント欄の記述は，機械語変換というアセンブル処理の対象とはならず，そのままアセンブル・リストに出力されます。
記述可能な文字は，「（2）文字セット」に示すものです。

例

```
        NAME    SAMPM
; ***************************************
;       HEX -> ASCII Conversion Program
;       main-routine
; ***************************************

        PUBLIC  MAIN, START
        EXTRN   CONVAH
        EXTRN   @STBEG

DATA    DSEG    saddr
HDTSA : DS      1
STASC : DS      2

CODE    CSEG    AT 0H
MAIN :  DW      START

        CSEG
START :
                                ; chip initialize
        MOVW    SP, #_@STBEG

        MOV     HDTSA, #1AH
        MOVW    HL, #HDTSA      ; set hex 2-code data in HL register

        CALL    !CONVAH         ; convert ASCII <- HEX
                                ; output BC-register <- ASCII code

        MOVW    DE, #STASC      ; set DE <- store ASCII code table
        MOV     A, B
        MOV     [DE], A
        INCW    DE
        MOV     A, C
        MOV     [DE], A
        BR      $$
        END
```

コメント欄のみの行

コメント欄のみの行

コメント欄にコメントが記述されている行

4.1.3 式と演算子

式とは，シンボル，定数，ロケーション・アドレスを示す$，ビット項，前述の4つに演算子を付加したもの，または演算子で結合したものです。

式を構成する演算子以外の要素を項といい，記述された左側から順に第1項，第2項，…と呼びます。

演算子には「表 4—4 演算子の種類」に示すものがあり，演算実行上の優先順位が「表 4—5 演算子の優先順位」のように決められています。

演算の順序を変更するには，かっこ“( )”を使用します。

例を示します。

```
MOV     A, #5 * ( SYM + 1 )
```

上記の例では，“5 * (SYM+1)”が式です。“5”が第1項，“SYM”が第2項，“1”が第3項です。“*”，“+”，“( )”が演算子です。

表 4—4 演算子の種類

演算子の種類	演算子
算術演算子	+，-，*，/，MOD，+符号，-符号
論理演算子	NOT，AND，OR，XOR
比較演算子	EQ（=），NE（< >），GT（>），GE（>=），LT（<），LE（<=）
シフト演算子	SHR，SHL
バイト分離演算子	HIGH，LOW
ワード分離演算子	HIGHW，LOWW，MIRHW，MIRLW
特殊演算子	DATAPOS，BITPOS，MASK
その他の演算子	( )

上記の演算子は，単項演算子，特殊単項演算子，2項演算子，N項演算子，その他の演算子に分けられます。

単項演算子	+符号，-符号，NOT，HIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW
特殊単項演算子	DATAPOS，BITPOS
2項演算子	+，-，*，/，MOD，AND，OR，XOR，EQ (=)，NE (<>)，GT (>)，GE (>=)，LT (<)，LE (<=)，SHR，SHL
N項演算子	MASK
その他の演算子	( )

表 4—5 演算子の優先順位

優先度	優先順位	演算子
高い	1	+ 符号，- 符号，NOT，HIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW，DATAPOS，BITPOS，MASK
	2	*，/，MOD，SHR，SHL
	3	+，-
	4	AND
	5	OR，XOR
低い	6	EQ (=)，NE (<>)，GT (>)，GE (>=)，LT (<)，LE (<=)

式の演算は，次の規則に従います。

- 演算の順序は，演算子の優先順位に従います。
  同一順位の場合は，左から右に演算されます。単項演算子の場合は，右から左に演算されます。
- かっこ“( )”の中の演算は，かっこの外の演算に先立って行われます。
- 単項演算子の多重演算が可能です。
  例を示します。
  1 = - - 1 == 1
  -1 = -+1 = -1
- 式の演算は，符号なし 32 ビットで行います。
  演算中に 32 ビットを越えてオーバフローした場合，オーバフローした値は無視されます。
- 定数が 32 ビットを越える場合には，エラーとなり，その値は 0 とみなされて計算されます。
- 除算では，小数部分を切り捨てます。
  除算がゼロの場合は，エラーとなり，結果は 0 となります。
- 負の値は，2 の補数形式となります。
- 外部参照記号のアセンブル時の評価値はゼロです（評価値はリンク時に決定されます）。
- オペランド欄に記述した式の演算結果は，命令の要求を満たす値でなければなりません。
  8 ビット長のオペランドを要求される命令で，リロケータブル，または外部参照の式を記述した場合は，下位 8 ビットの値からオブジェクトが生成され，リロケーション情報には 16 ビットで必要な情報が出力されます。そして，リンカにおいて，決定された値が 8 ビットの範囲に収まるかのチェックがされ，オーバフローすると，リンク時にエラーとなります。
  アブソリュートな式を記述した場合は，アセンブラ内で値が決定されるので，要求した範囲に収まるかのチェックが行われます。
  例えば，MOV 命令の場合，オペランドは 8 ビットなので，0H ～ 0FFH の範囲に入っていなければなりません。

（1）正しい例

```
MOV     A, #'2*' AND 0FH
MOV     A, #4 * 8 * 8 - 1
```

（2）誤った例

```
MOV     A, #'2*.
MOV     A, #4 * 8 * 8
```

（3）式評価の例

式	評価値
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0FFFFFFFFH
-1 > 1	00H（偽）
EXT 注 + 1	1

注　EXT　：外部参照記号

4. 1. 4　算術演算子

算術演算子には，次のものがあります。

演算子	概要
+	第１項と第２項の値の加算
-	第１項と第２項の値の減算
*	第１項と第２項の値の乗算
/	第１項と第２項の値で剰余算を行い，整数部を求める
MOD	第１項と第２項の値で剰余算を行い，余りを求める
+ 符号	項の値をそのまま返す
- 符号	項の値の２の補数を求める

+

第 1 項と第 2 項の値の加算を行います。

［機能］

第 1 項と第 2 項の値の和を返します。

［使用例］

```
        ORG     100H
START : BR      !$ + 6          ; (1)
```

（1）BR 命令により，“現在のロケーション・アドレス＋ 6 番地”へジャンプします。
つまり，“100H + 6H = 106H”へジャンプします。
したがって，“START:　BR　!106H”と記述することもできます。

-

第 1 項と第 2 項の値の減算を行います。

［機能］

第 1 項と第 2 項の値の差を返します。

［使用例］

```
        ORG     100H
BACK :  BR      BACK - 6H       ; (1)
```

(1) BR 命令により，「“BACK”に割り付けられたアドレス－ 6 番地」へジャンプします。
つまり，“100H - 6H = 0FAH”へジャンプします。
したがって，“BACK:　BR　!0FAH”と記述することもできます。

*

第 1 項と第 2 項の値の乗算を行います。

［機能］

第 1 項と第 2 項の値の積を返します。

［使用例］

```
TEN     EQU     10H
        MOV     A, #TEN * 3     ; (1)
```

（1）EQU 疑似命令により，ネーム“TEN”に 10H という値が定義されます。
“#”はイミーディエト・データを示します。
“TEN * 3”という式は“10H * 3”のことで，30H を返します。
したがって，“MOV A, #30H”と記述することもできます。

/

第 1 項と第 2 項の値で剰余算を行い，整数部を求めます。

［機能］

第 1 項の値を第 2 項の値で割り，その値の整数部を返します。

小数部は切り捨てられます。

除数（第 2 項）が 0 の場合は，エラーとなります。

［使用例］

```
MOV     A, #256 / 50            ; (1)
```

(1)“256 / 50 = 5　余り 6”となります。

よって，整数部の 5 を返します。

したがって，“MOV A, #5”と記述することもできます。

MOD

第 1 項と第 2 項の値で剰余算を行い，余りを求めます。

［機能］

第 1 項の値を第 2 項の値で割り，その値の余りを返します。

除数が 0 の場合は，エラーとなります。

MOD の前後には，空白が必要です。

［使用例］

```
MOV     A, #256  MOD  50        ; (1)
```

(1)“256 / 50 = 5　余り 6”となります。

よって，余りの 6 を返します。

したがって，“MOV A, #6”と記述することもできます。

+ 符号

項の値をそのまま返します。

［機能］

項の値をそのまま返します。

［使用例］

```
FIVE    EQU     +5      ; (1)
```

(1) 項の値“5”をそのまま返します。

EQU 疑似命令により，ネーム“FIVE”に 5 という値が定義されます。

- 符号

項の値の 2 の補数を求めます。

［機能］

項の値の 2 の補数をとった値を返します。

［使用例］

```
NO      EQU     -1      ; (1)
```

(1)“-1”は 1 の 2 の補数となります。

0000 0000 0000 0000 0000 0000 0000 0001 の 2 の補数は

1111 1111 1111 1111 1111 1111 1111 1111

となります。

よって，EQU 疑似命令により，ネーム“NO”に 0FFFFFFFFH が定義されます。

4. 1. 5　論理演算子

論理演算子には，次のものがあります。

演算子	概要
NOT	項のビットごとの論理否定を求める
AND	第１項の値と第２項の値のビットごとの論理積を求める
OR	第１項の値と第２項の値のビットごとの論理和を求める
XOR	第１項の値と第２項の値のビットごとの排他的論理和を求める

NOT

項のビットごとの論理否定を求めます。

［機能］

項のビットごとの論理否定をとり，その値を返します。

NOT と項との間には，空白が必要です。

［使用例］

```
MOVW    AX, #LOWW ( NOT 3H )    ; (1)
```

(1)“3H”の論理否定をとります。

よって，0FFFFFFFCH を返します。

したがって，“MOVW AX, #LOWW 0FFFFFFFCH”と記述することもできます。

NOT）	0000	0000	0000	0000	0000	0000	0000	0011
	1111	1111	1111	1111	1111	1111	1111	1100

AND

第 1 項の値と第 2 項の値のビットごとの論理積を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの論理積をとり，その値を返します。

AND の前後には，空白が必要です。

［使用例］

```
MOV     A, #6FAH  AND  0FH     ; (1)
```

(1)“6FAH”と“0FH”の論理積をとります。

よって，“0AH”を返します。

したがって，(1)は“MOV A, #0AH”と記述することもできます。

	0000	0000	0000	0000	0000	0110	1111	1010
AND）	0000	0000	0000	0000	0000	0000	0000	1111
	0000	0000	0000	0000	0000	0000	0000	1010

OR

第 1 項の値と第 2 項の値のビットごとの論理和を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの論理和をとり，その値を返します。

OR の前後には，空白が必要です。

［使用例］

```
MOV     A, #0AH  OR  1101B     ; (1)
```

(1)“0AH”と“1101B”の論理和をとります。

よって，“0FH”を返します。

したがって，(1)は“MOV A, #0FH”と記述することもできます。

	0000	0000	0000	0000	0000	0000	0000	1010
OR）	0000	0000	0000	0000	0000	0000	0000	1101
	0000	0000	0000	0000	0000	0000	0000	1111

XOR

第 1 項の値と第 2 項の値のビットごとの排他的論理和を求めます。

［機能］

第 1 項の値と第 2 項の値のビットごとの排他的論理和をとり，その値を返します。

XOR の前後には，空白が必要です。

［使用例］

```
MOV     A, #9AH  XOR  9DH      ; (1)
```

(1)“9AH”と“9DH”の排他的論理和をとります。

よって，“7H”を返します。

したがって，(1)は“MOV A, #7H”と記述することもできます。

	0000	0000	0000	0000	0000	0000	1001	1010
XOR）	0000	0000	0000	0000	0000	0000	1001	1101
	0000	0000	0000	0000	0000	0000	0000	0111

4. 1. 6　比較演算子

比較演算子には，次のものがあります。

演算子	概要
EQ（=）	第１項の値と第２項の値が等しいかどうか比較
NE（< >）	第１項の値と第２項の値が等しくないかどうか比較
GT（>）	第１項の値が第２項の値より大きいかどうか比較
GE（>=）	第１項の値が第２項の値より大きい，または等しいかどうか比較
LT（<）	第１項の値が第２項の値より小さいかどうか比較
LE（<=）	第１項の値が第２項の値より小さい，または等しいかどうか比較

EQ（=）

第 1 項の値と第 2 項の値が等しいかどうか比較します。

［機能］

第 1 項の値と第 2 項の値が等しいときに 0FFH（真），等しくないときに 00H（偽）を返します。

EQ の前後には，空白が必要です。

［使用例］

```
A1      EQU     12C4H
A2      EQU     12C0H

        MOV     A, #A1  EQ  ( A2 + 4H )         ; (1)
        MOV     X, #A1  EQ   A2                 ; (2)
```

(1)“A1 EQ (A2 + 4H)”は，“12C4H EQ (12C0H + 4H) ”となります。

このとき，第 1 項の値と第 2 項の値が等しいので，0FFH を返します。

(2)“A1 EQ A2”は，“12C4H EQ 12C0H”となります。

このとき，第 1 項の値と第 2 の値が等しくないので，00H を返します。

NE（< >）

第 1 項の値と第 2 項の値が等しくないかどうか比較します。

［機能］

第 1 項の値と第 2 項の値が等しくないときに 0FFH（真），等しいときに 00H（偽）を返します。

NE の前後には，空白が必要です。

［使用例］

```
A1      EQU     5678H
A2      EQU     5670H

        MOV     A, #A1  NE   A2                ;  (1)
        MOV     A, #A1  NE  ( A2 + 8H )        ;  (2)
```

（1）“A1 NE A2”は“5678H NE 5670H”となります。

このとき，第 1 項の値と第 2 項の値が等しくないので，0FFH を返します。

（2）“A1 NE (A2 + 8H) ”は“5678H NE (5670H + 8H) ”となります。

このとき，第 1 項の値と第 2 項の値が等しいので，00H を返します。

GT（>）

第 1 項の値が第 2 項の値より大きいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より大きいときに 0FFH（真），等しいか小さいときに 00H（偽）を返します。

GT の前後には，空白が必要です。

［使用例］

```
A1      EQU     1023H
A2      EQU     1013H

        MOV     A, #A1  GT   A2                ;  (1)
        MOV     X, #A1  GT  ( A2 + 10H )       ;  (2)
```

（1）“A1 GT A2”は“1023H GT 1013H”となります。

このとき，第 1 項の値が第 2 項の値より大きいので，0FFH を返します。

（2）“A1 GT (A2 + 10H) ”は“1023H GT (1013H + 10H) ”となります。

このとき，第 1 項の値が第 2 項の値と等しいので，00H を返します。

GE（>=）

第 1 項の値が第 2 項の値より大きい，または等しいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より大きいか，等しいときに 0FFH（真），小さいときに 00H（偽）を返します。
GE の前後には，空白が必要です。

［使用例］

```
A1      EQU     2037H
A2      EQU     2015H

        MOV     A, #A1  GE   A2                ;  (1)
        MOV     X, #A1  GE  ( A2 + 23H )       ;  (2)
```

(1)“A1 GE A2”は“2037H GE 2015H”となります。
このとき，第 1 項の値が第 2 項の値より大きいので，0FFH を返します。

(2)“A1 GE (A2 + 23H) ”は“2037H GE (2015H + 23H) ”となります。
このとき，第 1 項の値が第 2 項の値より小さいので，00H を返します。

LT（<）

第 1 項の値が第 2 項の値より小さいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より小さいときに 0FFH（真），等しいか大きいときに 00H（偽）を返します。

LT の前後には，空白が必要です。

［使用例］

```
A1      EQU     1000H
A2      EQU     1020H

        MOV     A, #A1  LT  A2              ; (1)
        MOV     X, # ( A1 + 20H )  LT  A2   ; (2)
```

(1)“A1 LT A2”は“1000H LT 1020H”となります。

このとき，第 1 項の値が第 2 の値より小さいので，0FFH を返します。

(2)“ (A1 + 20H) LT A2”は“ (1000H + 20H) LT 1020H”となります。

このとき，第 1 項の値と第 2 項の値が等しいので，00H を返します。

LE（<=）

第 1 項の値が第 2 項の値より小さい，または等しいかどうか比較します。

［機能］

第 1 項の値が第 2 項の値より小さいか等しいときに 0FFH（真），大きいときに 00H（偽）を返します。

LE の前後には，空白が必要です。

［使用例］

```
A1      EQU     103AH
A2      EQU     1040H

        MOV     A, #A1  LE  A2                  ; (1)
        MOV     X, # ( A1 + 7H )  LE  A2        ; (2)
```

(1)“A1 LE A2”は“103AH LE 1040H”となります。

このとき，第 1 項の値が第 2 の値より小さいので，0FFH を返します。

(2)“ (A1 + 7H) LE A2”は“ (103AH + 7H) LE 1040H”となります。

このとき，第 1 項の値が第 2 項の値より大きいので，00H を返します。

4. 1. 7　シフト演算子

シフト演算子には，次のものがあります。

演算子	概要
SHR	第１項の値を第２項で示す値分だけ右シフトした値を求める
SHL	第１項の値を第２項で示す値分だけ左シフトした値を求める

SHR

第 1 項の値を第 2 項で示す値分だけ右シフトした値を求めます。

［機能］

第 1 項の値を第 2 項で示す値（ビット数）分だけ右シフトし，その値を返します。

上位ビットには，シフトされたビット数だけ 0 が挿入されます。

SHR の前後には，空白が必要です。

シフト数が 0 の場合は，第 1 項の値がそのまま返されます。シフト数が 32 を越えた場合は，自動的に 0 が埋め込まれます。

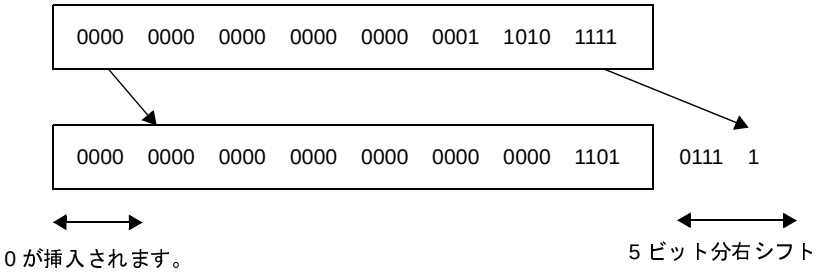
［使用例］

```
MOV     A, #01AFH  SHR  5       ; (1)
```

(1)“01AFH”を 5 ビット分右シフトします。

よって，“000DH”を返します。

したがって，“MOV A, #0DH”と記述することもできます。



SHL

第 1 項の値を第 2 項で示す値分だけ左シフトした値を求めます。

［機能］

第 1 項の値を第 2 項で示す値（ビット数）分だけ左シフトし，その値を返します。

下位ビットには，シフトされたビット数だけ 0 が挿入されます。

SHL の前後には，空白が必要です。

シフト数が 0 の場合は，第 1 項の値がそのまま返されます。シフト数が 32 を越えた場合は，自動的に 0 が埋め込まれます。

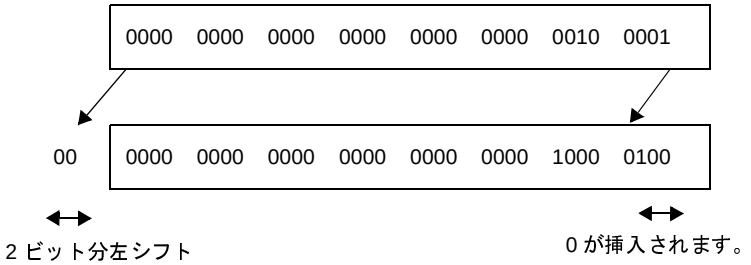
［使用例］

```
MOV     A, #21H  SHL  2        ; (1)
```

(1)“21H”を 2 ビット分左シフトします。

よって，“84H”を返します。

したがって，“MOV A, #84H”と記述することもできます。



4. 1. 8　バイト分離演算子

バイト分離演算子には，次のものがあります。

演算子	概要
HIGH	項の上位 8 ビットを求める
LOW	項の下位 8 ビットを求める

HIGH

項の上位 8 ビットを求めます。

[機能]

項の上位 8 ビットを返します。

HIGH と項との間には，空白が必要です。

[使用例]

```
MOV     A, #HIGH  1234H        ; (1)
```

(1) MOV 命令実行により，1234H の上位 8 ビット値 12H を返します。

したがって，(1) は“MOV A, #12H”と記述することもできます。

[備考]

SFR 名に対して HIGH 演算を行う場合は，次のように行います。

記述方法には，次の 2 つの方法があります。

```
HIGH SFR名
```

または

```
HIGH[ ]([ ]SFR名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄   ニーモニック欄   オペランド欄
             MOV            R0, #HIGH PM0
             MOV            R1, #HIGH PM1 + 1H      ; #( HIGH PM1 ) + 1 と同意
             MOV            R1, #HIGH ( PM1 + 1H )  ; SFR 名に HIGH, LOW, HIGHW, LOWW
                                                    ; 以外の演算子が施されているのでエラー
```

LOW

項の下位 8 ビットを求めます。

[機能]

項の下位 8 ビットを返します。

LOW と項との間には，空白が必要です。

[使用例]

```
MOV     A, #LOW  1234H          ; (1)
```

(1) MOV 命令実行により，1234H の下位 8 ビット値 34H を返します。

したがって，(1) は“MOV A, #34H”と記述することもできます。

[備考]

SFR 名に対して LOW 演算を行う場合は，以下のように行います。

記述方法には，次の 2 つの方法があります。

```
LOW SFR名
```

または

```
LOW[ ]([ ]SFR名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄   ニーモニック欄   オペランド欄
             MOV            R0, #LOW PM0
             MOV            R1, #LOW PM1 + 1H        ; #( LOW PM1 ) + 1 と同意
             MOV            R1, #LOW ( PM1 + 1H )    ; SFR 名に HIGH, LOW, HIGHW, LOWW
                                                     ; 以外の演算子が施されているのでエラー
```

4. 1. 9　ワード分離演算子

ワード分離演算子には，次のものがあります。

演算子	概要
HIGHW	項の上位 16 ビットを求める
LOWW	項の下位 16 ビットを求める
MIRHW	オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの項の上位 16 ビットを求める
MIRLW	オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの項の下位 16 ビットを求める

HIGHW

項の上位 16 ビットを求めます。

[機能]

項の上位 16 ビットを返します。

HIGHW と項との間には，空白が必要です。

[使用例]

```
MOVW    AX, #HIGHW      12345678H       ; (1)

MOV     ES, #HIGHW      LAB             ; (2)
MOVW    AX, ES:!LAB
```

(1) MOVW 命令実行により，12345678H の上位 16 ビット値 1234H を返します。
したがって，“MOVW AX, #1234H”と記述することもできます。

(2) MOV 命令実行により，ラベル LAB の上位アドレスを ES レジスタに設定します。

[備考]

SFR 名に対して HIGHW 演算を行う場合は，以下のように行います。

記述方法には，次の 2 つの方法があります。

```
HIGHW SFR名
```

または

```
HIGHW[ ]([ ]SFR名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄  ニーモニック欄  オペランド欄
            MOVW          RP0, #HIGHW PM0
            MOVW          RP1, #HIGHW PM1 + 1H      ; #( HIGHW PM1 ) + 1 と同意
            MOVW          RP1, #HIGHW ( PM1 + 1H )  ; SFR 名に HIGH, LOW, HIGHW, LOWW, MIRHW,
                                                    ; MIRLW 以外の演算子が施されているのでエラー
```

LOWW

項の下位 16 ビットを求めます。

[機能]

項の下位 16 ビットを返します。

LOWW と項との間には，空白が必要です。

[使用例]

```
MOVW    AX, #LOWW       12345678H       ; (1)
```

(1) MOVW 命令実行により，12345678H の下位 16 ビット値 5678H を返します。
したがって，“MOVW AX, #5678H”と記述することもできます。

[備考]

SFR 名に対して LOWW 演算を行う場合は，以下のように行います。

記述方法には，次の 2 つの方法があります。

```
LOWW SFR名
```

または

```
LOWW[ ]([ ]SFR名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄   ニーモニック欄   オペランド欄
             MOVW           RP0, #LOWW PM0
             MOVW           RP1, #LOWW PM1 + 1H      ; #( LOWW PM1 ) + 1 と同意
             MOVW           RP1, #LOWW ( PM1 + 1H )  ; SFR 名に HIGH, LOW, HIGHW, LOWW, MIRHW,
                                                     ; MIRLW 以外の演算子が施されているのでエラー
```

MIRHW

オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの項の上位 16 ビットを求めます。

［機能］

オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの 32 ビット中上位 16 ビット値を返します。

MIRHW と項との間には，空白が必要です。

［使用例］

```
MOVW    AX, #MIRHW      00001000H       ; (1)
```

(1) MOVW 命令実行により，00001000H の上位 16 ビット値 0000H をミラー先領域アドレスの上位 16 ビット値 000FH へ変換し，000FH が AX レジスタにロードされます。
オペランド値がミラー元領域の範囲外である場合は，HIGHW と同じ動作となります。

［備考］

SFR 名に対して MIRHW 演算を行う場合は，以下のように行います。

記述方法には，次の 2 つの方法があります。

```
MIRHW SFR 名
```

または

```
MIRHW[ ]([ ]SFR 名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄  ニーモニック欄  オペランド欄
            MOVW          RP0, #MIRHW PM0
            MOVW          RP1, #MIRHW PM1 + 1H      ; #( MIRHW PM1 ) + 1 と同意
            MOVW          RP1, #MIRHW ( PM1 + 1H )  ; SFR 名に HIGH, LOW, HIGHW, LOWW, MIRHW,
                                                    ; MIRLW 以外の演算子が施されているのでエラー
```

MIRLW

オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの項の下位 16 ビットを求めます。

[機能]

オペランド値をミラー元領域アドレスとした場合の，ミラー先領域アドレスの 32 ビット中下位 16 ビット値を返します。

MIRLW と項との間には，空白が必要です。

[使用例]

```
MOVW    AX, #MIRLW      00001000H       ; (1)
```

(1) 8 ビット CPU の場合，MOVW 命令実行により，00001000H の下位 16 ビット値 1000H をミラー先領域アドレスの下位 16 ビット値 9000H へ変換し，9000H が AX レジスタにロードされます。
16 ビット CPU の場合，MOVW 命令実行により，00001000H の下位 16 ビット値 1000H をミラー先領域アドレスの下位 16 ビット値 1000H へ変換し，1000H が AX レジスタにロードされます。
オペランド値がミラー元領域の範囲外である場合は，LOWW と同じ動作となります。

[備考]

SFR 名に対して MIRLW 演算を行う場合は，以下のように行います。

記述方法には，次の 2 つの方法があります。

```
MIRLW SFR名
```

または

```
MIRLW[ ]([ ]SFR名[ ])
```

演算結果は，アブソリュートな NUMBER 属性の項となります。

ただし，SFR 名に他の演算を施すことはできません。

例を以下に示します。

```
シンボル欄  ニーモニック欄  オペランド欄
            MOVW          RP0, #MIRLW PM0
            MOVW          RP1, #MIRLW PM1 + 1H      ; #( LOWW PM1 ) + 1 と同意
            MOVW          RP1, #MIRLW ( PM1 + 1H ) ; SFR 名に HIGH, LOW, HIGHW, LOWW, MIRHW,
                                                    ; MIRLW 以外の演算子が施されているのでエラー
```

4. 1. 10　特殊演算子

特殊演算子には，次のものがあります。

演算子	概要
DATAPOS	ビット・シンボルのアドレス部を求める
BITPOS	ビット・シンボルのビット部を求める
MASK	指定のビット位置に 1，その他を 0 にした 16 ビット値を求める

DATAPOS

ビット・シンボルのアドレス部を求めます。

［機能］

ビット・シンボルのアドレス部（バイト・アドレス）を返します。

［使用例］

```
SYM     EQU     0FE68H.6                ; (1)

        MOV     A, !DATAPOS  SYM        ; (2)
```

(1) EQU 疑似命令により，ネーム“SYM”に 0FE68H.6 という値が定義されます。

(2)“DATAPOS SYM”は“DATAPOS 0FE68H.6”ということで，0FE68H を返します。
したがって，“MOV A, !0FE68H”と記述することもできます。

BITPOS

ビット・シンボルのビット部を求めます。

［機能］

ビット・シンボルのビット部（ビット位置）を返します。

［使用例］

```
SYM     EQU     0FE68H.6                ;  (1)

        CLR1    [HL].BITPOS  SYM        ;  (2)
```

(1) EQU 疑似命令により，ネーム“SYM”に 0FE68H.6 という値が定義されます。

(2)“BITPOS.SYM”は“BITPOS 0FE68H.6”ということで，6 を返します。
CLR1 命令実行により，[HL].6 を 0 クリアします。

MASK

指定のビット位置に 1，その他を 0 にした 16 ビット値を求めます。

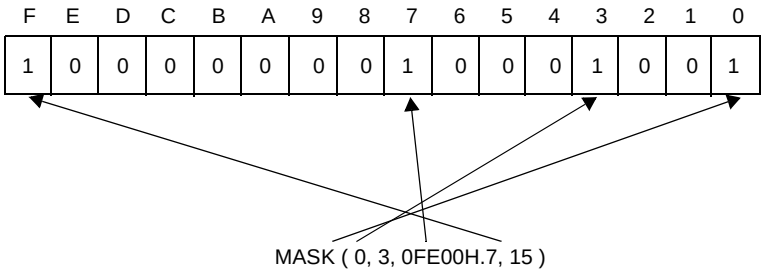
［機能］

指定のビット位置に 1，その他を 0 にした 16 ビット値を返します。

［使用例］

```
MOVW    AX, #MASK ( 0, 3, 0FE00H.7, 15 )    ; (1)
```

（1）MOVW 命令実行により，8089H を返します。



4. 1. 11　その他の演算子

その他の演算子には，次のものがあります。

演算子	概要
( )	( ) 内の演算を優先して行う

( )

( ) 内の演算を優先して行います。

[機能]

( ) 内の演算を ( ) 外の演算に先立って行います。

演算の優先順位を変更したいときに使用します。

( ) が多重になっている場合は，一番内側の ( ) 内の式から演算します。

[使用例]

```
MOV     A, # ( 4 + 3 ) * 2
```

```
(4+3)*2
 └──┘
  (1)
 └────┘
   (2)
```

(1)，(2) の順で演算を行い，14 という値を返します。

( ) がなければ

```
4+3*2
  └┘
  (1)
 └──┘
  (2)
```

(1)，(2) の順で演算を行い，10 という値を返します。

演算子の優先順位については，「表 4—5 演算子の優先順位」を参照してください。

4. 1. 12　演算の制限

式の演算は，項を演算子で結びつけて行います。項として記述できるものには，定数，$，ネーム，ラベルがあり，各項はリロケーション属性とシンボル属性を持ちます。

各項の持つリロケーション属性，シンボル属性の種類により，その項に対して演算可能な演算子が限られます。したがって，式を記述する場合には，式を構成する各項のリロケーション属性，シンボル属性に留意することが大切です。

(1) 演算とリロケーション属性

式を構成する各項は，リロケーション属性とシンボル属性を持ちます。

各項をリロケーション属性により分類すると，アブソリュート項，リロケータブル項，外部参照項の３種類に分けられます。

次に，演算におけるリロケーション属性の種類とその性質，およびそれに該当する項を示します。

表 4―6　リロケーション属性の種類

種類	性質	該当項
アブソリュート項	アセンブル時に値，定数が決定する項	- 定数 - アブソリュート・セグメント内のラベル - アブソリュート・セグメント内で定義したロケーション・アドレスを示す＄ - 定数，上記のラベル，上記の＄等，絶対値を定義したネーム
リロケータブル項	アセンブル時には値が決定しない項	- リロケータブル・セグメント内で定義したラベル - リロケータブル・セグメント内で定義したロケーション・アドレスを示す＄ - リロケータブルなシンボルで定義したネーム
外部参照項注	ほかのモジュールのシンボルを外部参照する項	- EXTRN 疑似命令で定義したラベル - EXTBIT 疑似命令で定義したネーム

注　外部参照項を演算対象にすることができる演算子は，“+”，“HIGH”，“LOW”，“HIGHW”，“LOWW”，“MIRHW”，“MIRLW”の７つです。ただし，１つの式に記述可能な外部参照項は，１つだけです。その場合，必ず演算子“+”で結合されていなければなりません。

演算可能な演算子と項の組み合わせをリロケーション属性により分類すると，次のようになります。

表 4―7　リロケーション属性による項と演算子の組み合わせ（リロケータブル項）

演算子の種類	項のリロケーション属性			
	X：ABS Y：ABS	X：ABS Y：REL	X：REL Y：ABS	X：REL Y：REL
X + Y	A	R	R	－
X - Y	A	－	R	A 注1

演算子の種類	項のリロケーション属性			
	X：ABS Y：ABS	X：ABS Y：REL	X：REL Y：ABS	X：REL Y：REL
X * Y	A	–	–	–
X / Y	A	–	–	–
X MOD Y	A	–	–	–
X SHL Y	A	–	–	–
X SHR Y	A	–	–	–
X EQ Y	A	–	–	A 注1
X LT Y	A	–	–	A 注1
X LE Y	A	–	–	A 注1
X GT Y	A	–	–	A 注1
X GE Y	A	–	–	A 注1
X NE Y	A	–	–	A 注1
X AND Y	A	–	–	–
X OR Y	A	–	–	–
X XOR Y	A	–	–	–
NOT X	A	A	–	–
+ X	A	A	R	R
- X	A	A	–	–
HIGH X	A	A	R 注2	R 注2
LOW X	A	A	R 注2	R 注2
HIGHW X	A	A	R 注2	R 注2
LOWW X	A	A	R 注2	R 注2
MIRHW X	A	A	R 注2	R 注2
MIRLW X	A	A	R 注2	R 注2
MASK (X)	A	A	–	–
DATAPOS X.Y	A	–	–	–
BITPOS X.Y	A	–	–	–
MASK (X.Y)	A	–	–	–
DATAPOS X	A	A	R	R
BITPOS X	A	A	A	A

ABS ： アブソリュート項

REL ： リロケータブル項

A ： 演算結果がアブソリュート項になります。

R ： 演算結果がリロケータブル項になります。

－ ： 演算不可

注 1. X，またはYがHIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW，DATAPOS演算を行ったリロケータブル項ではなく，X，Yがともに同一セグメントにある場合にかぎり，演算可能です。

2. X，またはYがHIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW，DATAPOS演算を行ったリロケータブル項でない場合にかぎり，演算可能です。

外部参照項を演算対象にすることができる演算子は，“+”，“HIGH”，“LOW”，“HIGHW”，“LOWW”，“MIRHW”，“MIRLW”の7つです。ただし，1つの式に記述可能な外部参照項は，1つだけです。

これらの演算子と外部参照項との実行可能な組み合わせをリロケーション属性により分類すると，次のようになります。

表 4—8 リロケーション属性による項と演算子の組み合わせ（外部参照項）

演算子の種類	項のリロケーション属性				
	X：ABS Y：EXT	X：EXT Y：ABS	X：REL Y：EXT	X：EXT Y：REL	X：EXT Y：EXT
X + Y	E	E	—	—	—
X - Y	—	E	—	—	—
+ X	A	E	R	E	E
HIGH X	A	E 注1	R 注2	E 注1	E 注1
LOW X	A	E 注1	R 注2	E 注1	E 注1
HIGHW X	A	E 注1	R 注2	E 注1	E 注1
LOWW X	A	E 注1	R 注2	E 注1	E 注1
MIRHW X	A	E 注1	R 注2	E 注1	E 注1
MIRLW X	A	E 注1	R 注2	E 注1	E 注1
MASK (X)	A	—	—	—	—
DATAPOS X.Y	—	—	—	—	—
BITPOS X.Y	—	—	—	—	—
MASK (X.Y)	—	—	—	—	—
DATAPOS X	A	E	R	E	E
BITPOS X	A	E	A	E	E

ABS ：アブソリュート項

EXT ：外部参照項

REL ：リロケータブル項

A ：演算結果がアブソリュート項になります。

E ：演算結果が外部参照項になります。

R ：演算結果がリロケータブル項になります。

ー ：演算不可

注 1. X，またはY が HIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW，DATAPOS，BITPOS 演算を行った外部参照項でない場合にかぎり，演算可能です。

2. X，またはY が HIGH，LOW，HIGHW，LOWW，MIRHW，MIRLW，DATAPOS 演算を行ったリロケータブル項でない場合にかぎり，演算可能です。

(2) 演算とシンボル属性

式を構成する各項は，リロケーション属性に加えてシンボル属性を持ちます。

各項をシンボル属性により分類すると，NUMBER 項，ADDRESS 項の 2 種類に分けられます。

演算におけるシンボル属性の種類とそれに該当する項を次に示します。

表 4—9 演算におけるシンボル属性の種類

シンボル属性の種類	該当項
NUMBER 項	- NUMBER 属性を持つシンボル - 定数
ADDRESS 項	- ADDRESS 属性を持つシンボル - ロケーション・カウンタを示す“$”

演算可能な演算子と項の組み合わせをシンボル属性により分類すると，次のようになります。

表 4—10 シンボル属性による項と演算子の組み合わせ

演算子の種類	項のシンボル属性			
	X：ADDRESS Y：ADDRESS	X：ADDRESS Y：NUMBER	X：NUMBER Y：ADDRESS	X：NUMBER Y：NUMBER
X + Y	A	A	A	N
X - Y	N	A	N	N
X * Y	N	N	N	N
X / Y	N	N	N	N
X MOD Y	N	N	N	N
X SHL Y	N	N	N	N
X SHR Y	N	N	N	N
X EQ Y	N	N	N	N
X LT Y	N	N	N	N
X LE Y	N	N	N	N
X GT Y	N	N	N	N
X GE Y	N	N	N	N
X NE Y	N	N	N	N
X AND Y	N	N	N	N
X OR Y	N	N	N	N
X XOR Y	N	N	N	N

演算子の種類	項のシンボル属性			
	X：ADDRESS Y：ADDRESS	X：ADDRESS Y：NUMBER	X：NUMBER Y：ADDRESS	X：NUMBER Y：NUMBER
NOT X	A	A	N	N
+ X	A	A	N	N
- X	A	A	N	N
HIGH X	A	A	N	N
LOW X	A	A	N	N
HIGHW X	A	A	N	N
LOWW X	A	A	N	N
MIRHW X	A	A	N	N
MIRLW X	A	A	N	N
DATAPOS X	A	A	N	N
MASK X	N	N	N	N

ADDRES ：ADDRESS 項
NUMBER ：NUMBER 項
A ：演算結果が ADDRESS 項
N ：演算結果が NUMBER 項
－ ：演算不可

（3）演算の制限についての確認方法

リロケーション属性，シンボル属性による演算の見方について，例を示します。

```
BR      $TABLE + 5H
```

ここで，“TABLE”はリロケータブルなコード・セグメント中で定義されたラベルであると仮定します。

（a）演算とリロケーション属性

“TABLE + 5H”は，“リロケータブル項＋アブソリュート項”となるので，この演算を「表 4—7 リロケーション属性による項と演算子の組み合わせ（リロケータブル項）」にあてはめます。

演算子の種類 … X + Y
項のリロケーション属性 … X：REL，Y：ABS

したがって，演算結果は“R”，すなわちリロケータブル項になることがわかります。

（b）[演算とシンボル属性]

“TABLE + 5H”は“ADDRESS 項＋ NUMBER 項”となるので，この演算を「表 4—10 シンボル属性による項と演算子の組み合わせ」にあてはめます。

演算子の種類 … X + Y
項のリロケーション属性 … X：ADDRESS，Y：NUMBER

したがって，演算結果は“A”，すなわち ADDRESS 項となることがわかります。

4. 1. 13　絶対式の定義

絶対式は，アセンブルの途中で，その式を評価したときに値が確定しているような式を指します。

以下のいずれかに属するものを絶対式と呼びます。

- 定数
- 定数同士に演算を施した式（定数式）
- 定数，または定数式で定義された EQU シンボル，または SET シンボル
- 上記に演算を施した式

備考　シンボルは後方参照のみ可能です。

4. 1. 14　ビット位置指定子

ビット位置指定子（ . ）を使用することにより，ビット・アクセスが可能になります。

(1) 記述形式

```
X[ ].[ ]Y
└──────┘
  ビット項
```

X（第 1 項）		Y（第 2 項）
汎用レジスタ	A	式（0 ～ 7）
制御レジスタ	PSW	式（0 ～ 7）
特殊機能レジスタ	sfr 注	式（0 ～ 7）
メモリ	[HL] 注	式（0 ～ 7）

注　具体的な記述については，各デバイスのユーザーズ・マニュアルを参照してください。

(2) 機能

- 第 1 項にバイト・アドレスを指定するもの，第 2 項にビット位置を指定するものを指定します。これにより，ビット・アクセスが可能になります。

(3) 説明

- ビット位置指定子を使用したものをビット項と呼びます。
- ビット位置指定子には演算子との優先順位はなく，左辺を第 1 項，右辺を第 2 項と認識します。
- 第 1 項には，次の制限があります。
  - NUMBER，ADDRESS 属性の式，8 ビット・アクセスが可能な SFR 名，またはレジスタ名（A）が記述可能です。
  - 第 1 項にアブソリュートな式を記述する場合は，0H ～ 0FFFFFH の範囲でなければなりません。
  - 外部参照シンボルを記述することができます。
- 第 2 項には，次の制限があります。

- 式の値は，0 ～ 7 の範囲です。範囲を越えた場合にはエラーとなります。
- アブソリュートな NUMBER 属性の式のみ記述可能です。
- 外部参照シンボルを記述することはできません。

(4) 演算とリロケーション属性

- リロケーション属性における第 1 項と第 2 項の組み合わせを次に示します。

項の組み合わせ X :	ABS	ABS	REL	REL	ABS	EXT	REL	EXT	EXT
項の組み合わせ Y :	ABS	REL	ABS	REL	EXT	ABS	EXT	REL	EXT
X.Y	A	—	R	—	—	E	—	—	—

ABS ： アブソリュート項
REL ： リロケータブル項
EXT ： 外部参照項
A ： 演算結果がアブソリュート項になります。
E ： 演算結果が外部参照項になります。
R ： 演算結果がリロケータブル項になります。
ー ： 演算不可

(5) ビット・シンボルの値

- EQU 疑似命令のオペランドにビット位置指定子を用いたビット項を記述しビット・シンボルを定義した場合，そのビット・シンボルが持つ値を次に示します。

オペランドの種類	シンボル値
A.bit 注 2	1.bit
PSW.bit 注 2	FFFFAH.bit
sfr 注 1.bit 注 2	FFFXXH.bit 注 3
式 .bit 注 2	XXXXXH.bit 注 4

注 1. 具体的な記述については，各デバイスのユーザーズ・マニュアルを参照してください。
2. bit = 0 ～ 7
3. FFFXXH は，sfr のアドレス
4. XXXXXH は，式の値

(6) 使用例

```
SET1    0FFE20H.3
SET1    A.5
CLR1    P1.2
SET1    1 + 0FFE30H.3    ; 0FFE31H.3 に等しい
SET1    0FFE40H.4 + 2    ; 0FFE40H.6 に等しい
```

4. 1. 15 識 別 子

識別子とは，シンボル，ラベル，マクロ名などに使用する名前です。

識別子は，次の規則に基づいて記述します。

- 識別子は，英数字，および英字相当文字（?，@，_）で構成します。
  ただし，先頭文字に数字（0 ～ 9）は使用できません。
- 識別子として，予約語は使用できません。
  予約語については，「4. 5 予 約 語」を参照してください。
- アセンブラは，識別子の大文字／小文字を区別します。

4. 1. 16 オペランドの特性

オペランドを必要とする命令（インストラクション，および疑似命令）は，その種類により要求するオペランド値のサイズ，範囲，シンボル属性などが異なります。

たとえば，“MOV r, #byte”というインストラクションの機能は，「レジスタ r に，byte で示される値を転送する」ものです。このとき，レジスタ r は 8 ビット長のレジスタであるため，転送されるデータ“byte”のサイズは，8 ビット以下でなければなりません。

もし，“MOV R0, #100H”と記述した場合には，第 2 オペランド“100H”のサイズが 8 ビット長を越えているため，アセンブル・エラーとなります。

このように，オペランドを記述する場合には，次のような注意が必要です。

- 値のサイズ，アドレス範囲がその命令のオペランドに適しているかどうか（数値やネーム，ラベル）
- シンボル属性がその命令のオペランドに適しているかどうか（ネーム，ラベル）

(1) オペランドの値のサイズとアドレス範囲

命令のオペランドとして記述可能な数値／ネーム／ラベルの値のサイズとアドレス範囲には条件があります。

インストラクションの場合は，各インストラクションのオペランドの表現形式により，疑似命令の場合には命令の種類により，記述可能なオペランドのサイズとアドレス範囲に条件があります。

これらの条件を次に示します。

表 4—11 インストラクションのオペランド値の範囲

オペランドの表現形式	値の範囲	
byte	8 ビット値 ： 0H ～ 0FFH	
word	word [B] word [C] word [BC]	(1) 数値定数，および NUMBER 属性のシンボルの場合 0H ～ FFFFH (2) ADDRESS 属性のシンボルの場合 以下のどちらかの領域内 - F0000H ～ FFFFFH - MAA=0 の時のミラー元領域（例：01000H ～ 0xxxxH），または MAA=1 の時のミラー元領域（例：11000H ～ 1xxxxH）注 1

オペランドの表現形式	値の範囲	
	ES:word [B] ES:word [C] ES:word [BC]	(1) 数値定数 および NUMBER 属性のシンボルの場合 0H ～ FFFFH (2) ADDRESS 属性のシンボルの場合 0H ～ FFFFFH
	上記以外	16 ビット値 ： 0H ～ FFFFH
saddr	FFE20H ～ FFF1FH 注 4	
saddrp	FFE20H ～ FFF1FH の偶数値 注 4	
sfr	FFF20H ～ FFFFFH：特殊機能レジスタ略号（SFR 略号），および数値定数，NUMBER 属性のシンボル 注 5	
sfrp	FFF20H ～ FFFFFH：特殊機能レジスタ略号（16 ビット操作可能な SFR 略号／偶数値のみ），および数値定数，NUMBER 属性のシンボル 注 5	
addr20	!!addr20	0H ～ FFFFFH
	$addr20	0H ～ FFFFFH，かつ分岐命令の次のアドレスから分岐先までが (-80H) ～ (+7FH) の範囲
	$!addr20	0H ～ FFFFFH，かつ分岐，コール命令の次のアドレスから分岐先までが (-8000H) ～ (+7FFFH) の範囲
addr16	!addr16 （BR，CALL 命令）	0H ～ FFFFH （数値定数，シンボルともに，指定可能な範囲は同じ）
	!addr16 注 2 （BR，CALL 以外の命令）	(1) 数値定数，および NUMBER 属性のシンボルの場合 注 3 0H ～ FFFFH (2) ADDRESS 属性のシンボルの場合 注 3 以下のどちらかの領域内 - F0000H ～ FFFFFH - MAA=0 のときのミラー元領域（例：01000H ～ 0xxxxH），または MAA=1 のときのミラー元領域（例：11000H ～ 1xxxxH） 注 1
	ES:!addr16	(1) 数値定数，および NUMBER 属性のシンボルの場合 注 3 0H ～ FFFFH (2) ADDRESS 属性のシンボルの場合 注 3 0H ～ FFFFFH
	!addr16.bit	(1) DBIT シンボル，SFBIT 属性，SABIT 属性のビット・シンボル，EQU 疑似命令で定義されたビット・シンボル（オペランドに ADDRESS 属性のシンボルを含む場合のみ） 以下のどちらかの領域内 - F0000H ～ FFFFFH - MAA=0 のときのミラー元領域（例：01000H ～ 0xxxxH），または MAA=1 のときのミラー元領域（例：11000H ～ 1xxxxH） 注 1 (2) 上記以外のビットシンボル 0H ～ FFFFH

オペランドの表現形式	値の範囲	
	ES:!addr16.bit	（1）DBIT シンボル，SFBIT 属性，SABIT 属性のビット・シンボル，EQU 疑似命令で定義されたビット・シンボル（オペランドに ADDRESS 属性のシンボルを含む場合のみ） 0H ～ FFFFFH （2）上記以外のビットシンボル 0H ～ FFFFH
addr5	0080H ～ 00BFH（CALLT 命令テーブル領域／偶数値のみ）	
bit	3 ビット値 ： 0 ～ 7	
n	2 ビット値 ： 0 ～ 3	

注 1. ミラー元領域のアドレス範囲は，デバイスによって異なります。詳細については，各デバイスのユーザーズ・マニュアルを参照してください。

2. sfr，2ndsfr をオペランドに記述したい場合は，!sfr，!2ndsfr の記述が可能です。これらは，!addr16 のオペランドとしてコードが出力されます。
2ndsfr は，“!”なしでも記述可能ですが，!addr16 のオペランドとしてコードが出力されます。

3. 16 ビット・データの場合は，偶数アドレスのみとなります。

4. 78K0 との互換性を保つため，数値定数，および NUMBER 属性のシンボルの場合のみ FE20H ～ FF1FH の範囲も記述可能となっています。

5. 数値定数，および NUMBER 属性のシンボルの場合は，指定アドレスの SFR の書き込み／読み出しのアクセス・チェックを行いません。

addr16，および word において，オペランドに記述されたシンボルの属性によって，そのオペランドの取り得る値の範囲は異なります。その理由について，次に示します。

なお，シンボルの属性については，「（d）シンボルの属性」を参照してください。

（a）!addr16（BR，CALL 以外の命令）

!addr16（BR，CALL 以外の命令）のオペランドで，数値定数，および NUMBER 属性のシンボルと ADDRESS 属性のシンボルで記述可能な範囲が異なっている理由について説明します。

例を以下に示します。

```
NUMBER0         EQU     0F100H                  ; (a)
NUMBER1         EQU     0F102H
NUMBER2         EQU     0F103H

D0              DSEG    AT  0FF100H
ADDRESS0 :      DS      1
ADDRESS1 :      DS      1
ADDRESS2 :      DS      1
```

```
                CSEG
                MOV     !NUMBER0, A             ; (b)
                MOV     !0F100H, A              ; (c)
                MOV     !ADDRESS0, A            ; (d)
```

（a）は，NUMBER 属性のシンボルであり，このシンボルを !addr16 のオペランドに記述した場合について説明します。

「MOV !addr16, A」の命令セットのオペランド !addr16 では，ダイレクト・アドレッシングが行われ，（b）では，「A レジスタの値を 0FF100H のアドレスへ転送する」という処理が行われます。（a）は，NUMBER 属性のシンボルであり，（c）と置き換えることが可能なので，!addr16 に記述した NUMBER 属性のシンボル NUMBER0，および数値 0F100H は，「0FF100H」のアドレスを指していることになります。

つまり，!addr16（BR，CALL 以外の命令）の「NUMBER 属性のシンボル」は「0H ～ FFFFH」の値を取ることができ，それは「F0000H ～ FFFFFH」のアドレスを指していることになります。

次に，ラベル ADDRESS0（ADDRESS 属性のシンボル）を使って，同様の処理を行う場合について説明します。

addr16 の範囲「0000H ～ FFFFH」に対して，（d）の ADDRESS0 のシンボル値は RAM 空間の「FxxxxH ～ FFFFFH」であるため，このままではエラーとなります。そこで，オペランドにラベル ADDRESS0（ADDRESS 属性のシンボル）を記述した場合は，オペランドの範囲を「F0000H ～ FFFFFH」とすることにより，記述の簡便化を図っています。

つまり，!addr16（BR，CALL 以外の命令）の「ADDRESS 属性のシンボル」は，「F0000H ～ FFFFFH」の値を取ることができ，そのままオペランドに記述することが可能です。

さらに，ROM 領域がミラー先領域にミラーされることに対して，!addr16 での対応が必要となります。例を以下に示します。

```
M0              CSEG    MIRRORP
ADDRESS0 :      DB      12H
ADDRESS1 :      DB      34H
ADDRESS2 :      DB      56H
                CSEG
                MOV     A, !ADDRESS0            ; (e)
```

M0 のセグメントは，ミラー元領域へ配置されます。M0 のセグメントは，MAA=0 のときは「01000H ～ 0xxxxH」，MAA=1 のときは「11000H ～ 1xxxxH」へ配置されます。そのため，（e）の ADDRESS0 のシンボル値は「01000H ～ 0xxxxH，または 11000H ～ 1xxxxH」の値となります。このことから，ミラーされるセグメント中のシンボルを参照する（e）のような記述を可能とするため，!addr16 の範囲は「01000H ～ 0xxxxH，または 11000H ～ 1xxxxH」となっています。

つまり，!addr16（BR，CALL 以外の命令）の「ADDRESS 属性のシンボル」は，「01000H ～ 0xxxxH，または 11000H ～ 1xxxxH」の値も，そのままオペランドに記述することが可能です。

（b）ES:!addr16

ES:!addr16 のオペランドで，数値定数，および NUMBER 属性のシンボルと ADDRESS 属性のシンボルで記述可能な範囲が異なっている理由について説明します。

例を以下に示します。

```
DATA            CSEG    AT      12345H
ADDRESS0 :      DB      12H
ADDRESS1 :      DB      34H
ADDRESS2 :      DB      56H


                CSEG
                MOV     ES, #HIGHW ADDRESS0     ; (f)
                MOV     A, ES:!ADDRESS0         ; (g)
```

（f），（g）で「ADDRESS0 にあるデータを A レジスタへ転送する」という処理を行う場合について説明します。

addr16 の範囲「0000H ～ FFFFH」に対して，（g）の ADDRESS0 のシンボル値は「12345H」であるため，このままではエラーとなります。

そこで，ADDRESS0 の範囲を「0H ～ FFFFFH」とすることにより，（g）の記述を可能として，記述の簡便化を図っています。

つまり，ES:!addr16 のオペランドの「ADDRESS 属性のシンボル」は，「0H ～ FFFFFH」の値も，そのままオペランドに記述することが可能です。

（c）!addr16.bit，ES:!addr16.bit

!addr16.bit，ES:!addr16.bit のオペランドで，DBIT シンボル，SFBIT 属性，SABIT 属性のビット・シンボル，EQU 疑似命令で定義されたビット・シンボル（オペランドに ADDRESS 属性のシンボルを含む場合のみ）とそれ以外のシンボルで記述可能な範囲が異なっている理由について説明します。

例を以下に示します。

```
                BSEG
DBITSYM0        DBIT                            ; (h)
DBITSYM1        DBIT
DBITSYM2        DBIT

BIT1_PM0        EQU     PM0.1                   ; (i)
BIT2_P0         EQU     P0.2                    ; (j)

                DSEG
ADDRESS0 :      DS      1
ADDRESS1 :      DS      1
ADDRESS2 :      DS      1

ADR_BIT0        EQU     ADDRESS0.0              ; (k)
```

```
ADR_BIT1        EQU     ADDRESS0.1
ADR_BIT2        EQU     ADDRESS0.2

                CSEG
                SET1    !DBITSYM0               ; (l)
                SET1    !BIT1_PM0               ; (m)
                SET1    !BIT2_P0                ; (n)
                SET1    !ADR_BIT0               ; (o)
```

（h）の DBIT シンボル，（i），（j）の SFBIT 属性，SABIT 属性のビット・シンボル，（k）の EQU 疑似命令で定義されたビット・シンボル（オペランドに ADDRESS 属性のシンボルを含む場合のみ）を（l）～（o）のように !addr16.bit のオペランドにそのまま記述できるようにするため，シンボルの属性によって取り得る値の範囲が異なります。

同様の理由により，ES:!addr16.bit についても，シンボルの属性によって取り得る値の範囲が異なります。

（d）word

word のオペランドで，数値定数，および NUMBER 属性のシンボルと ADDRESS 属性のシンボルで記述可能な範囲が異なっている理由について説明します。

例を以下に示します。

```
                DSEG
ADDRESS0 :      DS      1
ADDRESS1 :      DS      1
ADDRESS2 :      DS      1
                CSEG
                MOV     B, #0
                MOV     ADDRESS0[B], A          ; (p)
                MOV     C, #1
                MOV     ADDRESS0[C], A          ; (q)
                MOVW    BC, #2
                MOV     ADDRESS0[BC], AX        ; (r)
```

word をオペランドとする word[B]，wor[C]，word[BC] では，（p）～（r）のようにラベル（ADDRESS 属性のシンボル）を記述することが多いため，!addr16 と同様，ラベルでの記述を可能とすることで，記述の簡便化を図っています。

同様の理由により，ES:word[B]，ES:wor[C]，ES:word[BC] についても，記述の簡便化を図っています。

表 4—12　疑似命令のオペランド値の範囲

種類	疑似命令	値の範囲
セグメント定義	CSEG AT	0H ～ 0FFFFFH（SFR，2ndSFR を除く）
	DSEG AT	0H ～ 0FFFFFH（SFR，2ndSFR を除く）
	BSEG AT	0H ～ 0FFFFFH（SFR，2ndSFR を除く）
	ORG	0H ～ 0FFFFFH（SFR，2ndSFR を除く）
シンボル定義	EQU	20 ビット値　0H ～ FFFFFH
	SET	20 ビット値　0H ～ FFFFFH
メモリ初期化領域確保	DB	8 ビット値　0H ～ FFH
	DW	16 ビット値　0H ～ FFFFH
	DG	20 ビット値　0H ～ FFFFFH
	DS	16 ビット値　0H ～ FFFFH
分岐命令自動選択	BR/CALL	0H ～ FFFFFH

(2) 命令の要求するオペランドのサイズ

命令には，機械命令と疑似命令がありますが，オペランドとしてイミーディエト・データ，またはシンボルを要求する命令については，各命令により要求するオペランドのサイズが異なります。したがって，命令の要求するオペランドのサイズ以上のデータを記述すると，エラーとなります。

なお，式の演算は，符号なし，32 ビットで行います。評価結果が 0FFFFFFFFH（32 ビット）を越えた場合には警告メッセージが出力されます。

ただし，オペランドにリロケータブルなシンボル，または外部シンボルを記述した場合は，アセンブラ内では値が決定されないので，リンカにおいて値の決定と範囲のチェックが行われます。

(3) オペランドのシンボル属性，リロケーション属性

命令のオペランドとしてネーム，ラベル，$（ロケーション・カウンタを示す）を記述する場合は，それらの式の項としてのシンボル属性，リロケーション属性（「4. 1. 12　演算の制限」を参照），また，ネーム，ラベルの場合は，その参照方向の条件により，オペランドとして記述可能かどうかが異なります。

ネーム，ラベルの参照方向には，後方参照と前方参照があります。

- 後方参照：オペランドとして参照するネーム，ラベルがそれ以前の行で定義されています。
- 前方参照：オペランドとして参照するネーム，ラベルがそれ以降の行で定義されています。

例を以下に示します。

```
        NAME    TEST
        CSEG
L1 :                        <-- 後方参照
        BR      !L1         --> 前方参照
        BR      !L2         <--
L2 :
        END
```

次に，シンボル属性，リロケーション属性，ネーム，ラベルの参照方向の条件を示します。

表 4—13 オペランドとして記述可能なシンボルの性質

	シンボル属性	NUMBER		ADDRESS				NUMBER ADDRESS		sfr 予約語[注 1]	
	リロケーション属性	アブソリュート項		アブソリュート項		リロケータブル項		外部参照項			
	参照パターン	後方	前方	後方	前方	後方	前方	後方	前方	sfr	2ndsfr
記述形式	byte	○	○	○	○	○	○	○	○	×	×
	word	○	○	○	○	○	○	○	○	×	×
	saddr	○	○	○	○	○	○	○	○	○[注 3]	×
	saddrp	○	○	○	○	○	○	○	○	○[注 2,4]	×
	sfr	×	×	×	×	×	×	×	×	○[注 2,5]	×
	sfrp	×	×	×	×	×	×	×	×	○[注 2,6]	×
	addr20	○	○	○	○	○	○	○	○	×	×
	addr16	○	○	○	○	○	○	○	○	○[注 7]	○[注 7]
	addr5	○	○	○	○	○	○	○	○	×	×
	bit	○	○	×	×	×	×	×	×	×	×
	n	○	○	○	○	×	×	×	×	×	×

前方 ： 前方参照
後方 ： 後方参照
○ ： 記述可能
× ： エラー

注 1. EQU 疑似命令のオペランドに，sfr，sfrp（saddr と sfr がオーバーラップしていない領域の sfr）を指定し，定義されたシンボルは後方参照のみとし，前方参照は禁止します。

2. オペランドの組み合わせに，saddr/saddrp を sfr/sfrp に入れ替えた組み合わせが存在する命令に対し，saddr 領域の sfr 予約語を記述した場合は，saddr/saddrp としてコードが出力されます。

3. saddr 領域の sfr 予約語

4. saddr 領域の sfrp 予約語

5. 8 ビット・アクセス可能な sfr 予約語のみ

6. 16 ビット・アクセス可能な sfr 予約語のみ

7. BR，CALL 以外の命令のオペランド !addr16 でのみ，!sfr，!2ndsfr の指定が可能です。

表 4—14　疑似命令のオペランドとして記述可能なシンボルの性質

	シンボル属性		NUMBER		ADDRESS，SADDR						BIT					
	リロケーション属性		アブソリュート項		アブソリュート項		リロケータブル項		外部参照項		アブソリュート項		リロケータブル項		外部参照項	
	参照方向		後方	前方	後方	前方	後方	前方	後方	前方	後方	前方	後方	前方	後方	前方
疑似命令	ORG		○注1	—	—	—	—	—	—	—	—	—	—	—	—	—
	EQU 注2		○	—	○	—	○注3	—	—	—	○	—	○注3	—	—	—
	SET		○注1	—	—	—	—	—	—	—	—	—	—	—	—	—
	DB	サイズ	○注1	—	—	—	—	—	—	—	—	—	—	—	—	—
		初期値	○	○	○	○	○	○	○	○	—	—	—	—	—	—
	DW	サイズ	○注1	—	—	—	—	—	—	—	—	—	—	—	—	—
		初期値	○	○	○	○	○	○	○	○	—	—	—	—	—	—
	DG	サイズ	○注1	—	—	—	—	—	—	—	—	—	—	—	—	—
		初期値	○	○	○	○	○	○	○	○	—	—	—	—	—	—
	DS		○注4	—	—	—	—	—	—	—	—	—	—	—	—	—
	BR/CALL		○	○	○	○	○	○	○	○	—	—	—	—	—	—

前方　：前方参照

後方　：後方参照

○　：記述可能

ー　：記述不可能

注 1.　絶対式のみが記述可能です。

2.　次のパターンを含む式を記述するとエラーとなります。

- ADDRESS 属性ー ADDRESS 属性
- ADDRESS 属性　比較演算子　ADDRESS 属性
- HIGH　アブソリュートな ADDRESS 属性
- LOW　アブソリュートな ADDRESS 属性
- HIGHW　アブソリュートな ADDRESS 属性
- LOWW　アブソリュートな ADDRESS 属性
- MIRHW　アブソリュートな ADDRESS 属性
- MIRLW　アブソリュートな ADDRESS 属性
- DATAPOS　アブソリュートな ADDRESS 属性
- MASK　アブソリュートな ADDRESS 属性
- 上記の 10 つで，演算結果が最適化の影響を受ける可能性がある場合

3.　リロケータブルな項をオペランドに持つ HIGH/LOW/HIGHW/LOWW/MIRHW/MIRLW/DATAPOS/MASK 演算子によってできた項は許されません。

4.　「4. 2. 4　メモリ初期化，領域確保疑似命令」を参照してください。

4. 2 疑似命令

この章では，疑似命令について説明します。

疑似命令とは，RL78,78K0R アセンブラが一連の処理を行う際に必要な各種の指示を行うものです。

4. 2. 1 概　　要

インストラクションは，アセンブルの結果，オブジェクト・コード（機械語）に変換されますが，疑似命令は，原則としてオブジェクト・コードに変換されません。

疑似命令は，主に次の機能を持ちます。

- ソースの記述を容易にします。
- メモリの初期化や領域の確保を行います。
- アセンブラ，リンカがその処理を行うために必要となる情報を与えます。

次に，疑似命令の種類を示します。

表 4—15 疑似命令一覧

種類	疑似命令
セグメント定義疑似命令	CSEG，DSEG，BSEG，ORG
シンボル定義疑似命令	EQU，SET
メモリ初期化，領域確保疑似命令	DB，DW，DG，DS，DBIT
リンケージ疑似命令	EXTRN，EXTBIT，PUBLIC
オブジェクト・モジュール名宣言疑似命令	NAME
分岐命令自動選択疑似命令	BR，CALL
マクロ疑似命令	MACRO，LOCAL，REPT，IRP，EXITM，ENDM
アセンブル終了疑似命令	END

以降，各疑似命令について詳細な説明を行います。

説明の中で，[　] は大かっこの中が省略可能であることを … は同一の形式を繰り返すことを示します。

4. 2. 2　セグメント定義疑似命令

ソース・モジュールは，セグメント単位に分割して記述します。
この“セグメント”を定義するのが，セグメント定義疑似命令です。
セグメントには，次の 4 種類があります。

- コード・セグメント
- データ・セグメント
- ビット・セグメント
- アブソリュート・セグメント

セグメントの種類により，メモリのどの範囲に配置されるかが決まります。
次に，各セグメントの定義方法と配置されるメモリ・アドレスを示します。

表 4―16　セグメントの定義方法と配置されるメモリ・アドレス

セグメントの種類	定義方法	配置されるメモリ・アドレス
コード・セグメント	CSEG 疑似命令	内部，または外部の ROM アドレス内
データ・セグメント	DSEG 疑似命令	内部，または外部の RAM アドレス内
ビット・セグメント	BSEG 疑似命令	内部 RAM の saddr 領域内
アブソリュート・セグメント	CSEG，DSEG，BSEG 疑似命令で再配置属性に配置アドレス（AT 配置アドレス）を指示する	指定したアドレス

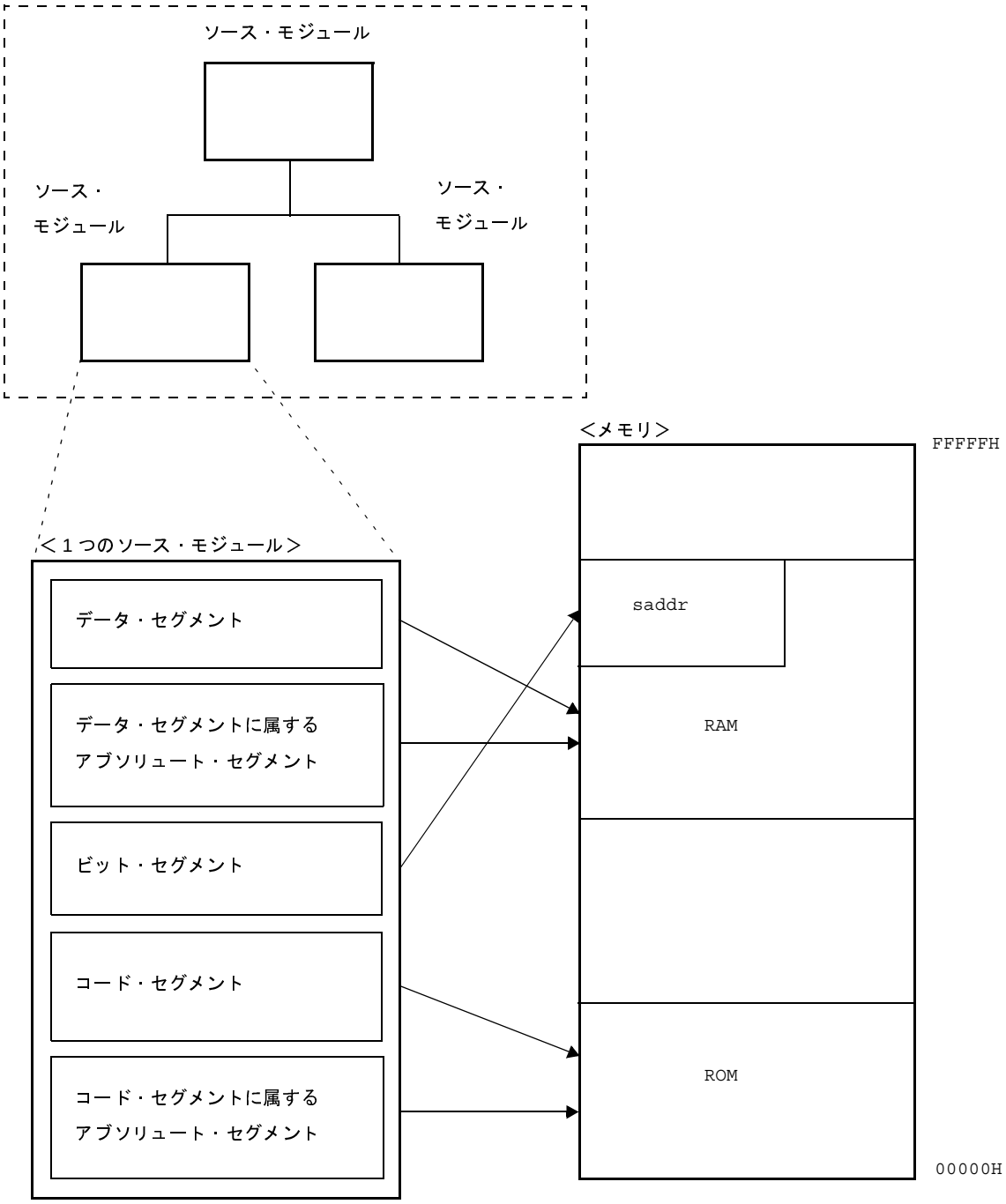
メモリの配置アドレスをユーザが決定したい場合には，アブソリュート・セグメントを記述します。スタック領域は，ユーザがデータ・セグメント内に領域を確保し，スタック・ポインタに設定する必要があります。
また，以下の領域には，セグメントを配置することはできません。

オプション・バイト領域	C0 ～ C2H（ユーザ・オプション・バイト），C3H（オンチップ・デバッグ・オプション・バイト）
セキュリティ ID を指定する場合	C4H ～ CDH 番地
オンチップ・デバッグ機能を使用する場合	02H ～ 03H，CE ～ D7H（オンチップ・デバッグ用） ユーザが -go オプションで指定するスタート・アドレスからプログラム・サイズ分の領域

オンチップ・デバッグ機能を使用する場合，オンチップ・デバッグ用のモニタ領域が配置できるように，リンク・ディレクティブにてオンチップ・デバッグ用の領域を空けてください。

セグメントの配置の例を次に示します。

図 4—6 セグメントのメモリ配置



セグメント定義疑似命令には，次のものがあります。

制御命令	概要
CSEG	アセンブラにコード・セグメントの開始を指示
DSEG	アセンブラにデータ・セグメントの開始を指示
BSEG	アセンブラにビット・セグメントの開始を指示
ORG	ロケーション・カウンタに，オペランドで指定した式の値を設定

CSEG

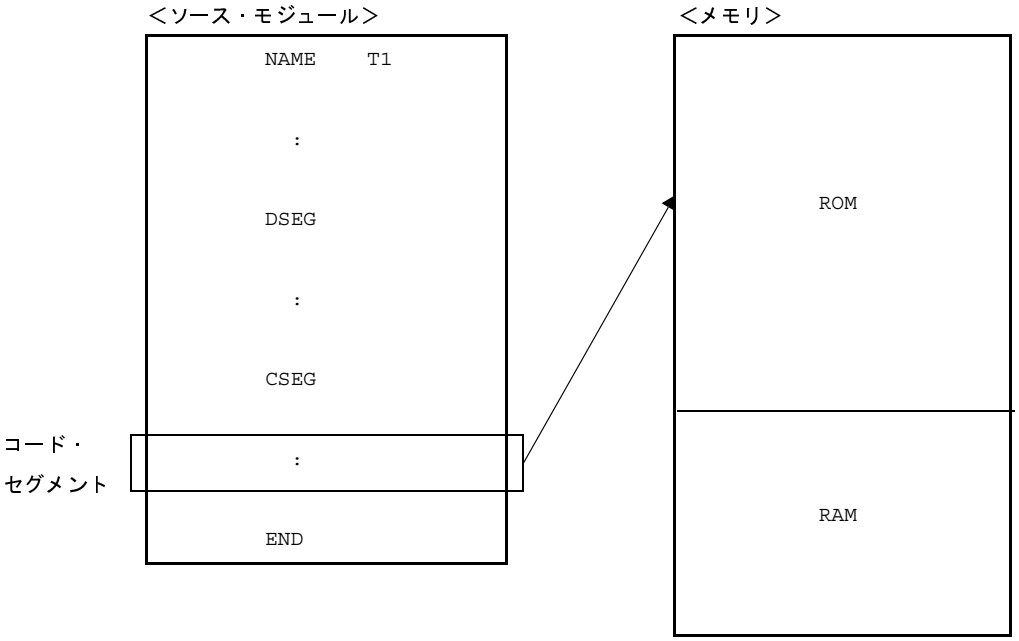
アセンブラにコード・セグメントの開始を指示します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セグメント名*]	CSEG	[*再配置属性*]	[; *コメント*]

[機能]

- CSEG 疑似命令は，アセンブラにコード・セグメントの開始を指示します。
- CSEG 疑似命令以降に記述した命令は，再びセグメント定義疑似命令（CSEG，DSEG，BSEG，ORG），または END 疑似命令が現れるまでコード・セグメントに属し，最終的に機械語に変換された時点で ROM アドレス内に配置されます。



[用途]

- CSEG 疑似命令で定義するコード・セグメントには，インストラクションや DB，DW 疑似命令等を記述します。ただし，そのセグメントを固定アドレスから配置する場合には，再配置属性欄に“AT 絶対式”を記述してください。
- サブルーチンなどの 1 つの機能を持つ単位の記述は，1 つのコード・セグメントとして定義します。
  その規模が比較的大きい場合や，そのサブルーチンに高い汎用性（ほかのプログラム開発にも流用できる）がある場合には，1 つのモジュールとして定義することをお勧めします。

[説明]

- コード・セグメントの開始アドレスは，ORG 疑似命令により指定できます。
  また，再配置属性欄に“AT 絶対式”を記述することによって，開始アドレスを指定することもできます。
- 再配置属性とは，セグメントの配置アドレスの範囲を限定するものです。
  次に，CSEG の再配置属性を示します。

表 4—17 CSEG の再配置属性

再配置属性	記述形式	説明
CALLT0	CALLT0	指定セグメントを 00080H ～ 000BFH 番地内で先頭が偶数番地になるように配置します。
FIXED	FIXED	指定セグメントを 000C0H ～ 0FFFFH 番地に配置します。
BASE	BASE	指定セグメントを 000C0H ～ 0FFFFH に配置します。
AT	AT 絶対式	指定セグメントを絶対番地に配置します（SFR，2ndSFR 領域を除く）。
UNIT	UNIT	指定セグメントを任意の位置へ配置します（メモリ領域名“ROM”内 000C0H ～ EFFFFH）。
UNITP	UNITP	指定セグメントを任意の位置へ，先頭が偶数番地になるように配置します（メモリ領域名“ROM”内 000C0H ～ EFFFFH）。
IXRAM	IXRAM	指定セグメントを任意の位置へ配置します（メモリ領域名“ROM”内 000C0H ～ EFFFFH）。
SECUR_ID	SECUR_ID	セキュリティ ID 指定専用の属性です。セキュリティ ID 以外は指定しないでください。 指定セグメントを 000C4H ～ 000CDH 番地へ配置します。
PAGE64KP	PAGE64KP	64K 境界にまたがらない，先頭が偶数番地となるように配置します。 異なるファイルの同名セグメントは結合しません。
UNIT64KP	UNIT64KP	64K 境界にまたがらない，先頭が偶数番地となるように配置します。 同名セグメントは結合します。
MIRRORP	MIRRORP	MAA=0 のときのミラー元領域（01000H ～ 0xxxxH），または MAA=1 のときのミラー元領域（11000H ～ 1xxxxH）のいずれかの領域に配置します。**注**
OPT_BYTE	OPT_BYTE	ユーザ・オプション・バイト，およびオンチップ・デバッグ指定専用の属性です。ユーザ・オプション・バイト，およびオンチップ・デバッグ以外は指定しないでください。 指定セグメントを 000C0H ～ 000C3H 番地へ配置します。

注 ミラー元領域のアドレス範囲は，デバイスによって異なります。

- 再配置属性を省略した場合，“UNIT”と解釈されます。

- 「表 4—17 CSEG の再配置属性」以外の再配置属性を指定した場合は，アセンブラはエラーを出力し，“UNIT”が指定されたものとみなします。また，各セグメントのサイズが領域のサイズを越えた場合には，エラーとなります。
- 再配置属性 AT で不当な絶対式を指定すると，アセンブラはエラーを出力し，絶対式の値を 0 とみなし，処理を続けます。
- CSEG 疑似命令のシンボル欄にセグメント名を記述することにより，そのコード・セグメントにネーム（名前）を付けることができます。セグメント名が省略されたコード・セグメントには，アセンブラが自動的にデフォルトのセグメント名を与えます。

次に，CSEG のデフォルト・セグメント名を示します。

再配置属性	デフォルト・セグメント名
CALLT0	?CSEGT0
FIXED	?CSEGFX
UNIT（または省略時）	?CSEG
UNITP	?CSEGUP
IXRAM	?CSEGIX
BASE	?CSEGB
SECUR_ID	?CSEGSI
PAGE64KP	?CSEGP64
UNIT64KP	?CSEGU64
MIRRORP	?CSEGMIP
OPT_BYTE	?CSEGOB0
AT	セグメント名省略不可

- C コンパイラが出力するデフォルト・セグメントのうち，下記のセグメントのサイズが 0 の場合，リンカで再配置属性を変更します。

セクション名	再配置属性	サイズ 0 の場合の再配置属性
@@CALT	CSEG CALLT0	CSEG UNIT
@@CNST	CSEG MIRRORP	CSEG UNIT

- 再配置属性が AT の場合，セグメント名を省略するとエラーとなります。
- 再配置属性が同一であれば，複数のコード・セグメントに同一のセグメント名を与えることができます（ただし，AT の場合は同名セグメントは許されません）。

これらは，アセンブラ内部で 1 つのセグメントとして処理されます。同名セグメントの再配置属性が異なる場合には，エラーとなります。したがって，再配置属性ごとの同名セグメントの数は 1 つです。

- コード・セグメントは分割記述が可能です。つまり，同一モジュール内に記述された同一再配置属性，同一セグメント名のコード・セグメントは，アセンブラ内部で連続したひとつのセグメントとして処理されます。

注意 1. 再配置属性が AT の場合は，分割記述はできません。

2. 再配置属性が CALLT0 のアドレスが偶数となるように，必要ならば 1 バイトの間隙をおいてください。

- 別モジュール間での同名セグメントは，UNIT，CALLT0，FIXED，UNITP，BASE，PAGE64KP，UNIT64KP，MIRRORP，SECUR_ID の場合のみ記述することができ，リンク時に連続した１つのセグメントとして結合されます。
- セグメント名は，シンボルとして参照できません。
- アセンブラの出力するセグメントの総数は，ORG 疑似命令によるセグメントをあわせて，別名セグメントが 256 個までです。同名セグメントは１つと数えます。
- セグメント名の最大認識文字数は，8 文字です。
- セグメント名の大文字，小文字は区別されます。
- OPT_BYTE で，ユーザ・オプション・バイト，およびオンチップ・デバッグを指定します。
  ユーザ・オプション・バイト指定機能を持つデバイスに対して，ユーザ・オプション・バイトを指定していない場合には，各番地に“?CSEGOB0”というデフォルト・セグメントが定義され，デバイス・ファイルから読み込んだ初期値が設定されます。

［使用例］

```
        NAME    SAMP1
C1      CSEG            ; (1)

C2      CSEG    CALLT0  ; (2)

        CSEG    FIXED   ; (3)

C1      CSEG    CALLT0  ; (4)   ←エラー

        CSEG            ; (5)
        END
```

(1) セグメント名が“C1”，再配置属性が“UNIT”と解釈されます。

(2) セグメント名が“C2”，再配置属性が“CALLT0”と解釈されます。

(3) セグメント名が“?CSEGFX”，再配置属性が“FIXED”と解釈されます。

(4) (1) でセグメント名“C1”は再配置属性“UNIT”として定義されているので，エラーとなります。

(5) セグメント名が“?CSEG”，再配置属性が“UNIT”と解釈されます。

DSEG

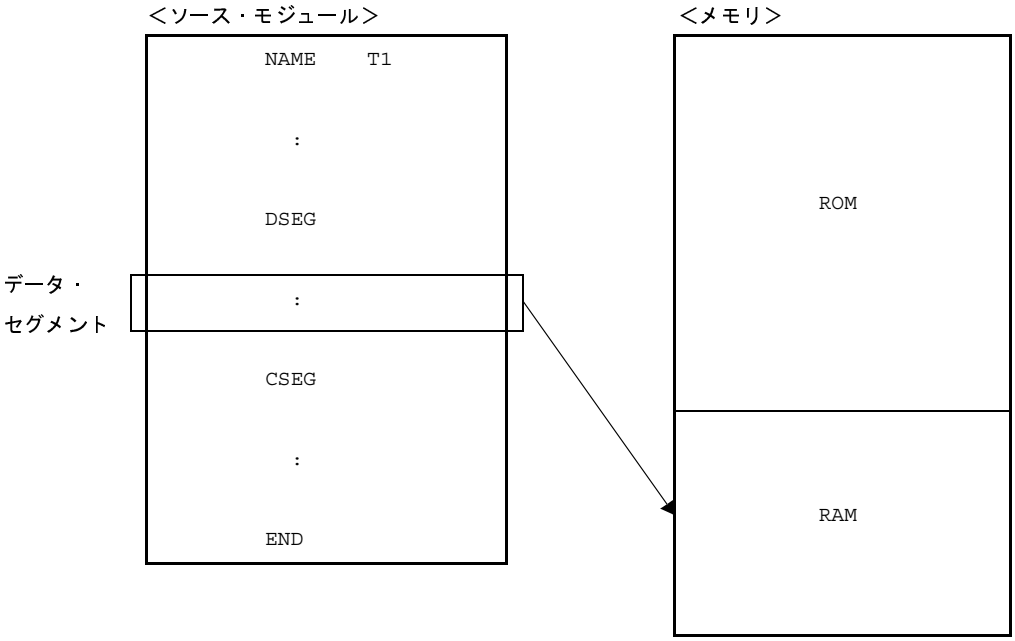
アセンブラにデータ・セグメントの開始を指示します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セグメント名*]	DSEG	[*再配置属性*]	[; *コメント*]

[機能]

- DSEG 疑似命令は，アセンブラにデータ・セグメントの開始を指示します。
- DSEG 疑似命令以降，再びセグメント定義疑似命令（CSEG，DSEG，BSEG，ORG），または END 疑似命令が現れるまでに DS 疑似命令により定義されたメモリ領域は，データ・セグメントに属し，最終的に，RAM アドレス内に確保されます。



[用途]

- DSEG 疑似命令で定義するデータ・セグメントには，主に DS 疑似命令を記述します。
  データ・セグメントは，RAM 内に配置されます。したがって，データ・セグメント内にインストラクションを記述することはできません。
- データ・セグメントでは，プログラムで使用する RAM の作業領域を DS 疑似命令により確保し，それぞれの作業領域のアドレスにラベルを付けます。プログラムを記述する場合，このラベルを利用します。
  データ・セグメントとして確保された領域は，RAM 上でほかの作業領域（スタック領域，ほかのモジュールで定義された作業領域など）と重複しないよう，リンカにより配置されます。

汎用レジスタ領域と重複する場合，リンカは警告メッセージを出力します。この警告メッセージの出力レベルは，警告メッセージ出力指定オプション（-w）により切り替えることができます。

-w の値	チェック対象
0	チェックしない
1	RB0
2	RB0 ～ RB3

[説明]

- データ・セグメントの開始アドレスは，ORG 疑似命令により指定できます。
  また，再配置属性欄に“AT 絶対式”を記述することによって，開始アドレスを指定することもできます。
- 再配置属性とは，データ・セグメントの配置アドレスの範囲を限定するものです。
  次に，DESG の再配置属性を示します。

表 4―18 DSEG の再配置属性

再配置属性	記述形式	説明
SADDR	SADDR	指定セグメントを saddr 領域に配置します（saddr 領域：FFE20H ～ FFEFFH）。
SADDRP	SADDRP	指定セグメントを saddr 領域に先頭が偶数番地となるように配置します（saddr 領域：FFE20H ～ FFEFFH）。
AT	AT 絶対式	指定セグメントを絶対番地に配置します（SFR，2ndSFR 領域を除く）。
UNIT	UNIT，または指定なし	指定セグメントを内部または外部の任意の位置へ配置します（メモリ領域名“RAM”内）。
UNITP	UNITP	指定セグメントを内部または外部の任意の位置へ，偶数番地から配置します（メモリ領域名“RAM”内）。
BASEP	BASEP	指定セグメントを内部 RAM 領域に先頭が偶数番地となるように配置します（ただし，saddr 領域（FxxxxH ～ FFEFFH）は含みません）。**注** ES 参照なしでアクセスしたいデータを配置するときに使用します。
PAGE64KP	PAGE64KP	メモリ領域名“RAM”内に，64K 境界にまたがらない，先頭が偶数番地となるように配置します。 異なるファイルの同名セグメントは結合しません。
UNIT64KP	UNIT64KP	メモリ領域名“RAM”内に，64K 境界にまたがらない，先頭が偶数番地となるように配置します。 同名セグメントは結合します。

注 xxxx に当てはまるアドレスは，デバイスに依存します。

- 78K0 用アセンブラでの再配置属性も記述可能で，“UNIT”と同様の配置を行います。

次に，78K0 用 DSEG の再配置属性を示します。

再配置属性	記述形式
IHRAM	IHRAM
LRAM	LRAM
DSPRAM	DSPRAM
IXRAM	IXRAM

- 再配置属性が省略された場合，“UNIT”と解釈されます。
- 「表 4—18 DSEG の再配置属性」以外の再配置属性が指定された場合には，アセンブラはエラーを出力し，“UNIT”が指定されたものとみなします。また，各セグメントのサイズが領域のサイズを越えた場合には，エラーとなります。
- 再配置属性 AT で不当な絶対式を指定すると，アセンブラはエラーを出力し，絶対式の値を 0 とみなし，処理を続けます。
- データ・セグメント中に機械語命令（BR 疑似命令も含む）は記述できません。記述した場合はエラーとなり，その行は無視されます。
- DSEG 疑似命令のシンボル欄にセグメント名を記述することにより，そのデータ・セグメントにネーム（名前）を付けることができます。セグメント名が省略されたデータ・セグメントには，アセンブラが自動的にデフォルトのセグメント名を与えます。

次に，DSEG のデフォルト・セグメント名を示します。

再配置属性	デフォルト・セグメント名
SADDR	?DSEGS
SADDRP	?DSEGSP
UNIT（または省略時）	?DSEG
UNITP	?DSEGUP
IHRAM	?DSEGIH
LRAM	?DSEGL
DSPRAM	?DSEGDSP
IXRAM	?DSEGIX
BASEP	?DSEGBP
PAGE64KP	?DSEGP64
UNIT64KP	?DSEGU64
AT	セグメント名省略不可

- C コンパイラが出力するデフォルト・セグメントのうち，下記のセグメントのサイズが 0 の場合，リンカで再配置属性を変更します。

セクション名	再配置属性	サイズ 0 の場合の再配置属性
@@INIS	DSEG SADDRP	DSEG UNITP

セクション名	再配置属性	サイズ 0 の場合の再配置属性
@@DATS	DSEG SADDRP	DSEG UNITP
@EINIS	DSEG SADDRP	DSEG UNITP
@EDATS	DSEG SADDRP	DSEG UNITP

- 再配置属性が同一であれば，複数のデータ・セグメントに同一のセグメント名を与えることができます（ただし，AT の場合は同名セグメントは許されません）。
  これらは，アセンブラ内部で 1 つのセグメントとして処理されます。
- データ・セグメントは分割記述が可能です。つまり，同一モジュール内に記述された同一再配置属性，同一セグメント名のデータ・セグメントは，アセンブラ内部で連続したひとつのセグメントとして処理されます。

注意 1. 再配置属性が AT の場合は，分割記述はできません。

2. 再配置属性が SADDR の場合は，DESG 疑似命令を記述した直後のアドレスが偶数となるように，必要ならば 1 バイトの間隙をおいてください。

- 再配置属性が SADDRP の場合，DSEG 疑似命令を記述した直後のアドレスが 2 の倍数になるように配置されます。
- 同名セグメントの再配置属性が異なる場合には，エラーとなります。したがって，再配置属性ごとの同名セグメントの数は 1 つです。
- 別モジュール間での同名セグメントは，UNIT，UNITP，SADDR，SADDRP，LRAM，IHRAM，DSPRAM，IXRAM，BASEP，PAGE64KP，UNIT64KP の場合のみ記述することができ，リンク時に連続した 1 つのセグメントとして結合されます。
- セグメント名はシンボルとして参照できません。
- アセンブラの出力するセグメントの総数は，ORG 疑似命令によるセグメントをあわせて，別名セグメントが 255 個までです。同名セグメントは 1 つと数えます。
- セグメント名の最大認識文字数は，8 文字です。
- セグメント名の大文字，小文字は区別されます。

［使用例］

```
        NAME    SAMP1
        DSEG                    ; (1)
WORK1 : DS      2
WORK2 : DS      1
        CSEG
        MOV     A, !WORK2       ; (2)
        MOV     A, WORK2        ; (3)   ←エラー
        MOVW    DE, #WORK1      ; (4)
        MOVW    AX, WORK1       ; (5)   ←エラー
        END
```

（1）DSEG 疑似命令により，データ・セグメントの開始を定義します。
再配置属性が省略されたので“UNIT”と解釈されます。デフォルトのセグメント名は“?DSEG”です。

（2）この記述は，“MOV A, !addr16”に該当します。

（3）この記述は，“MOV A, saddr”に該当します。
リロケータブルなラベル“WORK2”は“saddr”としては記述できません。したがって，（3）の記述はエラーです。

（4）この記述は，“MOVW rp, #word”に該当します。

（5）この記述は，“MOVW AX, saddrp”に該当します。
リロケータブルなラベル“WORK1”は“saddrp”としては記述できません。したがって，（5）の記述はエラーです。

BSEG

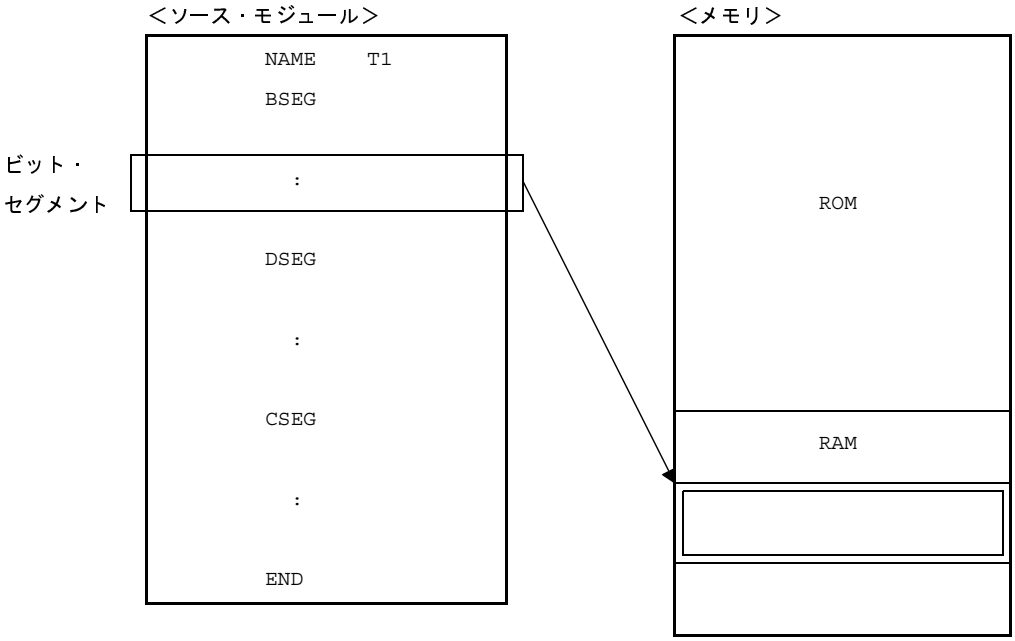
アセンブラにビット・セグメントの開始を指示します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セグメント名*]	BSEG	[*再配置属性*]	[; *コメント*]

[機能]

- BSEG 疑似命令は，アセンブラにビット・セグメントの開始を指示します。
- ビット・セグメントは，ソース・モジュール中で使用する RAM アドレスの定義を行うセグメントです。
- BSEG 疑似命令以降，再びセグメント定義疑似命令（CSEG，DSEG，BSEG），または END 疑似命令が現れるまでに DBIT 疑似命令により定義されたメモリ領域は，ビット・セグメントに属します。



[用途]

- BSEG 疑似命令で定義するビット・セグメントには，DBIT 疑似命令を記述します。
- ビット・セグメント内にインストラクションを記述することはできません。

[説明]

- ビット・セグメントの開始アドレスは，再配置属性欄に“AT 絶対式”を記述することによって指定することができます。

- 再配置属性とは，ビット・セグメントの配置アドレスの範囲を限定するものです。
  次に，BSEG の再配置属性を示します。

表 4―19 BSEG の再配置属性

再配置属性	記述形式	説明
AT	AT 絶対式	指定セグメントの先頭を絶対番地の 0 ビット目に配置します。ビット単位で指定することはできません（00000H ～ FFFFFH）（SFR，2ndSFR を除く）。
UNIT	UNIT，または指定なし	指定セグメントを任意の位置へ配置します（FFE20H ～ FFEFFH）。

- 再配置属性を省略した場合，“UNIT”と解釈されます。
- 上記の表以外の再配置属性が指定された場合には，アセンブラはエラーを出力し，“UNIT”が指定されたものとみなします。また，各セグメントのサイズが領域のサイズを越えた場合には，エラーとなります。
- アセンブラ，リンカでは，ビット・セグメント内のロケーション・カウンタを“0xxxxx.b”の形式で表示します（バイト・アドレスは 16 進 5 桁，ビット位置は 16 進 1 桁（0 ～ 7））。

(1) アブソリュート

バイト・アドレス	ビット位置							
	0	1	2	3	4	5	6	7
0FFE20H	0FFE20H.0	0FFE20H.1	0FFE20H.2	0FFE20H.3	0FFE20H.4	0FFE20H.5	0FFE20H.6	0FFE20H.7
0FFE21H	0FFE21H.0	0FFE21H.1	0FFE21H.2	0FFE21H.3	0FFE21H.4	0FFE21H.5	0FFE21H.6	0FFE21H.7

(2) リロケータブル

バイト・アドレス	ビット位置							
	0	1	2	3	4	5	6	7
0H	0H.0	0H.1	0H.2	0H.3	0H.4	0H.5	0H.6	0H.7
1H	1H.0	1H.1	1H.2	1H.3	1H.4	1H.5	1H.6	1H.7

備考 リロケータブルなビット・セグメント中でのバイト・アドレスは，セグメントの先頭からのバイト単位のオフセットを指定します。
なお，オブジェクト・コンバータが出力するシンボル・テーブルでは，ビットの定義を行う領域の先頭からのビット・オフセットで表示，出力されます。

シンボル値	ビット・オフセット
0FFE20H.0	0000
0FFE20H.1	0001
0FFE20H.2	0002
:	:

シンボル値	ビット・オフセット
0FFE20H.7	0007
0FFE21H.0	0008
0FFE21H.1	0009
:	:
0FFE80H.0	0300
:	:

- 再配置属性 AT で不当な絶対式を指定すると，アセンブラはエラーを出力し，絶対式の値を 0 とみなし，処理を続けます。
- BSEG 疑似命令のシンボル欄にセグメント名を記述することにより，そのビット・セグメントにネーム（名前）を付けることができます。セグメント名が省略されたビット・セグメントには，アセンブラが自動的にデフォルトのセグメント名を与えます。

  次に，BSEG のデフォルト・セグメント名を示します。

再配置属性	デフォルト・セグメント名
UNIT（または省略時）	?BSEG
AT	セグメント名省略不可

- C コンパイラが出力するデフォルト・セグメントのうち，下記のセグメントのサイズが 0 の場合，リンカで再配置属性を変更します。

セクション名	再配置属性	サイズ 0 の場合の再配置属性
@@BITS	BSEG UNIT（SADDR 領域内）	BSEG UNIT（RAM 領域内）

- 再配置属性が UNIT であれば，複数のデータ・セグメントに同一のセグメント名を与えることができます（ただし，AT の場合に同名セグメントは許されません）。

  これらは，アセンブラ内部で 1 つのセグメントとして処理されます。したがって，再配置属性ごとの同名セグメントの数は 1 つです。
- 同じセグメント名のビット・セグメント同士は，同一の再配置属性 UNIT（AT の場合は同名セグメントは禁止）でなければなりません。
- 同一モジュール内に記述された同一セグメント名の再配置属性が UNIT ではないときは，エラーとなり，その行は無視されます。
- 別モジュール間での同名セグメントは，リンク時に連続した 1 つのセグメントとして結合されます。結合は，ビット単位で行われます。
- セグメント名は，シンボルとして参照できません。
- ビット・セグメントは，リンカにより 0H ～ FFFFFH に配置されます。
- ビット・セグメント内でラベルを記述することはできません。
- ビット・セグメント内で記述できる命令は，DBIT 疑似命令，EQU，SET，PUBLIC，EXTBIT，EXTRN，MACRO，REPT，IRP，ENDM 疑似命令，およびマクロ定義とマクロ参照のみです。これ以外のものが記述された場合には，エラーとなります。

- アセンブラの出力するセグメントの総数は，ORG 疑似命令によるセグメントをあわせて，別名セグメントが 256 個までです。同名セグメントは 1 つと数えます。
- セグメント名の最大認識文字数は，8 文字です。
- セグメント名の大文字，小文字は区別されます。

［使用例］

```
        NAME    SAMP1

FLAG    EQU     0FFE20H
FLAG0   EQU     FLAG.0          ; (1)
FLAG1   EQU     FLAG.1          ; (2)

        BSEG                    ; (3)
FLAG2   DBIT

        CSEG
        SET1    FLAG0           ; (4)
        SET1    FLAG2           ; (5)
        END
```

(1) バイト・アドレス境界を意識して，ビット・アドレス（0FFE20H のビット 0）を定義しています。

(2) バイト・アドレス境界を意識して，ビット・アドレス（0FFE20H のビット 1）を定義しています。

(3) BSEG 疑似命令により，ビット・セグメントを定義します。再配置属性が省略されているので，アセンブラは，再配置属性が“UNIT”，セグメント名が“?BSEG”と解釈します。
ビット・セグメント内では，DBIT 疑似命令により，ビット作業領域を 1 ビットごとに定義します。ビット・セグメントは，モジュール・ボディのはじめの方に記述します。
ビット・セグメント内で定義したビット・アドレス FLAG2 は，バイト・アドレス境界を意識しないで配置されます。

(4) この記述は，“SET1 FLAG.0”と書き換えられます。ここで，FLAG は，バイト・アドレスを示します。

(5) この記述では，バイト・アドレスが意識されません。

ORG

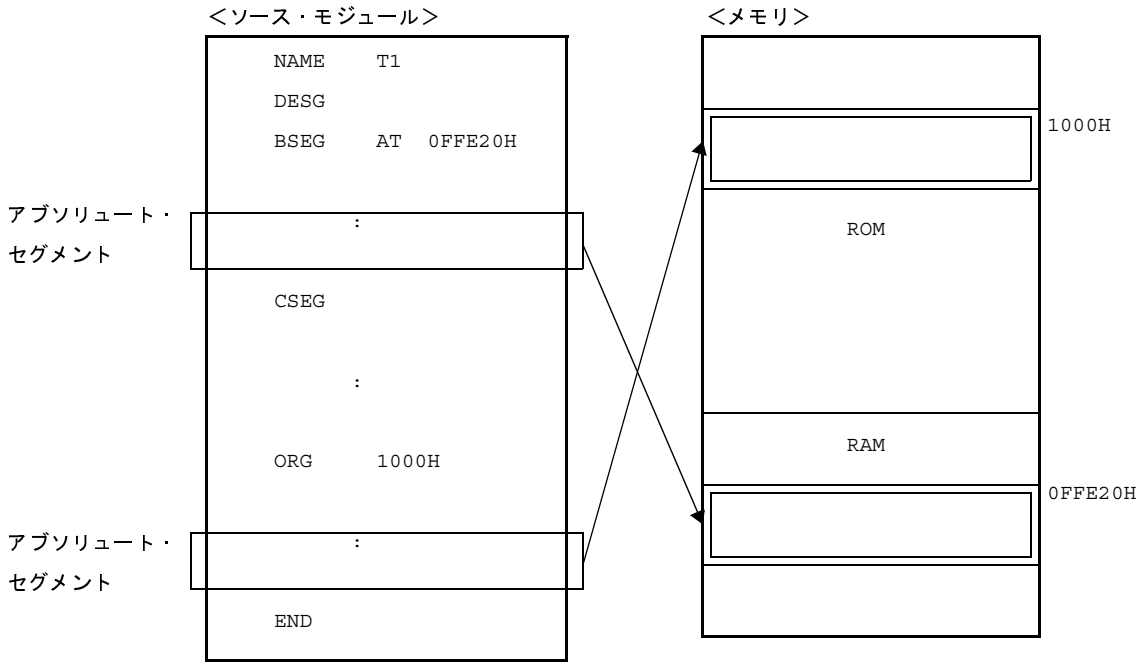
ロケーション・カウンタに，オペランドで指定した式の値を設定します。

［記述形式］

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*セグメント名*]	ORG	[*絶対式*]	[; *コメント*]

［機能］

- ロケーション・カウンタに，オペランドで指定した式の値を設定します。
- ORG 疑似命令以降，再びセグメント定義疑似命令（CSEG，DSEG，BSEG，ORG），または END 疑似命令が現れるまでに記述された命令や，確保されたメモリ領域は，アブソリュート・セグメントに属し，オペランドで指定したアドレスから配置されます。



［用途］

- コード・セグメント，データ・セグメントを特定のアドレスから配置させる場合に，ORG 疑似命令を指定します。

［説明］

- ORG 疑似命令により定義されたアブソリュート・セグメントは，その直前に CSEG，DSEG 疑似命令により定義されたコード・セグメント，またはデータ・セグメントに属します。
  データ・セグメントに属するアブソリュート・セグメント内では，インストラクションを記述することはできません。また，ビット・セグメントに属するアブソリュート・セグメントを記述することはできません。
- ORG 疑似命令により定義されたコード，データ・セグメントは，再配置属性 AT のコード，データ・セグメントとして解釈されます。
- ORG 疑似命令のシンボル欄に，セグメント名を記述することにより，そのアブソリュート・セグメントにネームを付けることができます。
  セグメント名の最大認識文字数は 8 文字です。
- ORG 疑似命令によって定義されたモジュール内の同名セグメントの取り扱いは，CSEG 疑似命令，および DESG 疑似命令の AT 属性のセグメントと同一とします。
- ORG 疑似命令によって定義された別モジュール間の同名セグメントの取り扱いは，CSEG 疑似命令，および DESG 疑似命令の AT 属性のセグメントと同一とします。
- セグメント名が省略されたアブソリュート・セグメントには，アセンブラが自動的に“?A0nnnnn”というセグメント名を与えます。nnnnn は指定されたセグメントの先頭アドレスで，00000 ～ FFFFF（16 進 5 桁）が入ります。
- ORG 疑似命令以前に CSEG，DSEG 疑似命令の記述がない場合，そのアブソリュート・セグメントは，コード・セグメント中のアブソリュート・セグメントと解釈されます。
- ORG 疑似命令のオペランドとして，ネーム／ラベルを記述する場合，そのネーム／ラベルは，ソース・モジュール中ですでに定義されたアブソリュート項でなければなりません。
- 絶対式として不当なものを記述した場合，または絶対式の評価値が 00000H ～ FFEFFH を越える場合は，エラーとなり，アセンブラは絶対式の値を 00000H とみなして，処理を続けます。
- オペランドの絶対式は，符号なし 32 ビットで評価されます。
- セグメント名は，シンボルとして参照できません。
- アセンブラの出力するセグメントの総数は，セグメント定義疑似命令によるセグメントをあわせて，別名セグメントが 256 個までです。同名セグメントは 1 つと数えます。
- セグメント名の最大認識文字数は，8 文字です。
- セグメント名の大文字，小文字は区別されます。

［使用例］

```
        NAME    SAMP1

        DSEG
        ORG     0FFE20H         ; (1)
SADR1 : DS      1
SADR2 : DS      1
SADR3 : DS      2

MAIN0   ORG     100H
        MOV     A, SADR1        ; (2)   ←エラー

        CSEG                    ; (3)
MAIN1   ORG     1000H           ; (4)
        MOV     A, SADR2
        MOVW    AX, SADR3
        END
```

(1) データ・セグメントに属するアブソリュート・セグメントを定義します。
このアブソリュート・セグメントは，ショート・ダイレクト・アドレッシング領域の先頭アドレス FFE20H 番地から配置されます。セグメント名の指定を省略しているので，アセンブラが自動的に“?A0FFE20”というセグメント名を与えます。

(2) データ・セグメントに属するアブソリュート・セグメント内では，インストラクションの記述はできないため，エラーとなります。

(3) コード・セグメントの開始を宣言します。

(4) このアブソリュート・セグメントは，1000H 番地から配置されます。

4. 2. 3　シンボル定義疑似命令

シンボル定義疑似命令は，ソース・モジュールを記述する際に使用するデータにネーム（名前）を割り付けます。これにより，データ値の意味がはっきりし，ソース・モジュールの内容がわかりやすくなります。

シンボル定義疑似命令は，ソース・モジュール中で使用するネームの値をアセンブラに知らせるものです。

シンボル定義疑似命令には，次のものがあります。

制御命令	概要
EQU	オペランドで指定した式の値と属性を持つ数値データをネームとして定義
SET	オペランドで指定した式の値と属性を持つ変数をネームとして定義

EQU

オペランドで指定した式の値と属性を持つ数値データをネームとして定義します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
ネーム	EQU	*式*	[; *コメント*]

[機能]

- オペランドで指定した式の値と属性（シンボル属性，およびリロケーション属性）を持つネーム（名前）を定義します。

[用途]

- ソース・モジュールの中で使用する数値データをネームとして定義し，命令のオペランドに数値データの代わりに記述します。
  特に，ソース・モジュールの中で頻繁に使用する数値データは，ネームとして定義しておくことをお勧めします。何らかの理由により，ソース・モジュール中のあるデータ値を変更しなければならないとき，ネームとして定義しておけば，そのネームのオペランド値を変更するだけで済みます。

[説明]

- EQU 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
- EQU 疑似命令で定義したシンボルは，SET，およびラベルとして再定義することはできません。また，SET で定義したシンボルやラベルも EQU 疑似命令で再定義することはできません。
- EQU 疑似命令のオペランドにネーム／ラベルを記述する場合は，すでにソース・モジュール中で定義されているネーム／ラベルを使用します。
  オペランドとして外部参照項を記述することはできません。
  SFR，SFR ビット名称は，記述可能です。
- リロケータブルな項をオペランドに持つ HIGH/LOW/HIGHW/LOWW/MIRHW/MIRLW/DATAPOS/BITPOS 演算子によってできた項を含む式を記述することはできません。
- オペランドに次のパターンを含む式を記述すると，エラーとなります。

（1）ADDRESS 属性の式 1 － ADDRESS 属性の式 2

上記で，次の（a），（b）があてはまる場合。

（a）ADDRESS 属性の式 1 中のラベル 1 と ADDRESS 属性の式 2 のラベルの間に，その場でオブジェクト・コードのバイト数が決定できない BR 疑似命令が記述されている場合。

（b）ラベル 1 とラベル 2 が別セグメントであり，属するセグメントの先頭からラベルまでの間に，その場でオブジェクト・コードのバイト数が決定できない BR 疑似命令が記述されている場合。

（2）ADDRESS 属性の式 1　比較演算子　ADDRESS 属性の式 2

（3）HIGH　アブソリュートな ADDRESS 属性の式

（4）LOW　アブソリュートな ADDRESS 属性の式

（5）HIGHW　アブソリュートな ADDRESS 属性の式

（6）LOWW　アブソリュートな ADDRESS 属性の式

（7）MIRHW　アブソリュートな ADDRESS 属性の式

（8）MIRLW　アブソリュートな ADDRESS 属性の式

（9）DATAPOS　アブソリュートな ADDRESS 属性の式

（10）BITPOS　アブソリュートな ADDRESS 属性の式

（11）上記（3）～（10）の中で，次の（a）があてはまる場合。

（a）ADDRESS 属性の式中のラベルと，属するセグメントの先頭の間にその場でオブジェクト・コードのバイト数が決定できない BR 疑似命令が記述されている場合。

- オペランドの記述形式にエラーがある場合，アセンブラはエラーを出力し，解析可能なかぎりの値をネームの値として登録します。
- EQU 疑似命令により定義したネームは，同一のソース・モジュール中では再定義することはできません。
- EQU 疑似命令でビット値を定義したネームは，値としてアドレスとビット位置を持ちます。
- EQU 疑似命令のオペランドとして記述可能なビット値とその参照可能範囲を次に示します。

オペランドの種類	シンボル値	参照可能範囲
A.bit 注 1	1.bit	同一モジュール内でのみ参照可能
PSW.bit 注 1	0FFFFAH.bit	
sfr 注 2.bit 注 1	0FFFXXH 注 3.bit	
2ndsfr 注 2.bit 注 1	0FXXXXH 注 4.bit	
saddr.bit 注 1	0FFXXXH 注 5.bit	別モジュールから参照可能
式 .bit 注 1	0XXXXXH 注 6.bit	

注 1.　bit = 0 ～ 7

2. 具体的な記述については，各デバイスのユーザーズ・マニュアルを参照してください。
3. 0FFFXXH ： sfr のアドレス
4. 0FXXXXH ： 2ndsfr 領域
5. 0FXXXXH ： saddr 領域（0FFE20H ～ 0FFF1FH）
6. 0XXXXXH ： 0H ～ 0FFFFFH

［使用例］

```
        NAME    SAMP1

WORK1   EQU     0FFE20H         ; (1)
WORK10  EQU     WORK1.0         ; (2)
P02     EQU     P0.2            ; (3)
A4      EQU     A.4             ; (4)
PSW5    EQU     PSW.5           ; (5)

        SET1    WORK10          ; (6)
        SET1    P02             ; (7)
        SET1    A4              ; (8)
        SET1    PSW5            ; (9)
        END
```

（1）ネーム“WORK1”は，値“0FFE20H”と，シンボル属性“NUMBER”，リロケーション属性“ABSOLUTE”を持ちます。

（2）“saddr.bit”にあたるビット値“WORK1.0”に，ネーム“WORK10”を割り当てます。
オペランドに記述されている“WORK1”は，（1）で値“0FFE20H”と定義済みです。

（3）“sfr.bit”にあたるビット値“P0.2”に，ネーム“P02”を割り当てます。

（4）“A.bit”にあたるビット値“A.4”に，ネーム“A4”を割り当てます。

（5）“PSW.bit”にあたるビット値“PSW.5”に，ネーム“PSW5”を割り当てます。

（6）この記述は，“SET1 saddr.bit”に該当します。

（7）この記述は，“SET1 sfr.bit”に該当します。

（8）この記述は，“SET1 A.bit”に該当します。

（9）この記述は，“SET1 PSW.bit”に該当します。

なお，（4），（5）のように“A.bit”，“PSW.bit”を定義したネームは，そのモジュール内でのみ参照することができます。

“sfr.bit”，“saddr.bit”，“式 .bit”を定義したネームは，外部定義シンボルとして別のモジュールからも参照することができます（「4. 2. 5　リンケージ疑似命令」を参照）。

使用例のソース・モジュールをアセンブルすると，次のようなアセンブル・リストが生成されます。

```
        Assemble list

ALNO    STNO    ADRS    OBJECT      M I SOURCE STATEMENT

   1       1                                NAME    SAMP
   2       2
   3       3            ( FFE20 )       WORK1   EQU     0FFE20H         ; (1)
   4       4            ( FFE20.0 )     WORK10  EQU     WORK1.0         ; (2)
   5       5            ( FFF00.2 )     P02     EQU     P0.2            ; (3)
   6       6            ( 00001.4 )     A4      EQU     A.4             ; (4)
   7       7            ( FFFFA.5 )     PSW5    EQU     PSW.5           ; (5)
   8       8
   9       9   00000    710220                  SET1    WORK10          ; (6)
  10      10   00003    712200                  SET1    P02             ; (7)
  11      11   00006    71CA                    SET1    A4              ; (8)
  12      12   00008    715AFA                  SET1    PSW5            ; (9)
  13      13                                   END
```

ビット値をネームとして定義している（2）～（5）の行には，アセンブル・リストのオブジェクト・コード欄に定義されたネームの持つビット・アドレスの値が表示されています。

SET

オペランドで指定した式の値と属性を持つ変数をネームとして定義します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
ネーム	SET	*絶対式*	[; *コメント*]

[機能]

- オペランドで指定した式の値と属性（シンボル属性，およびリロケーション属性）を持つネーム（名前）を定義します。
- SET 疑似命令で定義したネームの値と属性は，同一モジュール内において SET 疑似命令により再定義できます。
  SET 疑似命令により定義したネームの値と属性は，再び同じネームを再定義するまで有効です。

[用途]

- ソース・モジュールの中で使用する変数をネームとして定義し，命令のオペランドに数値データ（変数）の代わりに記述します。
  ソース・モジュールの中で，ネームの値を変更したい場合には，再度 SET 疑似命令で同じネームに異なる数値データを定義できます。

[説明]

- オペランドには，絶対式を記述します。
- SET 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
  ただし，SET 疑似命令でネームを定義したネームを前方参照することはできません。
- SET 疑似命令でネームを定義した文にエラーがあると，アセンブラはエラーを出力し，解析可能なかぎりの値をネームの値として登録します。
- EQU 疑似命令で定義したシンボルを SET 疑似命令で再定義することはできません。
  また，SET 疑似命令で定義したシンボルを EQU 疑似命令で再定義することもできません。
- ビット・シンボルは定義できません。

［使用例］

```
        NAME    SAMP1
COUNT   SET     10H             ; (1)

        CSEG
        MOV     B, #COUNT       ; (2)
LOOP :
        DEC     B
        BNZ     $LOOP

COUNT   SET     20H             ; (3)

        MOV     B, #COUNT       ; (4)
        END
```

(1) ネーム“COUNT”は，値“10H”と，シンボル属性“NUMBER”，リロケーション属性“ABSOLUTE”を持ちます。これらは，(3) の記述の直前まで有効です。

(2) レジスタ B には，“COUNT”の値 10H が転送されます。

(3) ネーム“COUNT”の値を“20H”に変更します。

(4) レジスタ B には，“COUNT”の値 20H が転送されます。

4. 2. 4　メモリ初期化，領域確保疑似命令

メモリ初期化疑似命令は，プログラムで使用する定数データを定義します。

定義したデータの値は，オブジェクト・コードとして生成されます。

領域確保疑似命令は，プログラムで使用するメモリの領域を確保します。

メモリ初期化，領域確保疑似命令には，次のものがあります。

制御命令	概要
DB	バイト領域を初期化
DW	ワード領域を初期化
DG	20 ビット領域を 32 ビット（4 バイト）単位でを初期化
DS	オペランドで指定したバイト数分のメモリ領域を確保
DBIT	ビット・セグメント中で 1 ビットのメモリ領域を確保

DB

バイト領域を初期化します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	DB	(*サイズ*)	[; *コメント*]
		または	
[*ラベル* :]	DB	*初期値* [, …]	[; *コメント*]

[機能]

- バイト領域を初期化します。
  初期化するバイト数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリをバイト単位で初期化します。

[用途]

- プログラムで使用する式や文字列を定義するときに，DB 疑似命令を使用します。

[説明]

- オペランドがカッコ“(”，“)”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは，指定されたバイト数分の領域を“00H”で初期化します。

(b) サイズには，絶対式を記述します。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

(2) 初期値指定の場合

(a) 式

式の値は 8 ビットのデータとして確保されます。したがって，式の値は 0H ～ 0FFH の間でなければなりません。8 ビットを越えた場合，下位 8 ビットがデータとして確保され，エラーが出力されます。

（b）文字列

文字列が記述された場合，1 文字に対して，それぞれ 8 ビット ASCII コードが確保されます。

- DB 疑似命令は，ビット・セグメント内では記述することはできません。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式を記述することができます。

［使用例］

```
                NAME    SAMP1
                CSEG
WORK1 :         DB      ( 1 )                   ; (1)
WORK2 :         DB      ( 2 )                   ; (1)
                CSEG
MASSAG :        DB      'ABCDEF'                ; (2)
DATA1 :         DB      0AH, 0BH, 0CH           ; (3)
DATA2 :         DB      ( 3 + 1 )               ; (4)
DATA3 :         DB      'AB' + 1                ; (5)   ←エラー
                END
```

（1）サイズを指定しているので，それぞれのバイト領域を値“00H”で初期化します。

（2）6 バイトの領域を文字列“ABCDEF”で初期化します。

（3）3 バイトの領域を 0AH，0BH，0CH で初期化します。

（4）4 バイトの領域を 00H で初期化します。

（5）“AB”+1 の値は，4143H (4142H + 1) で 0H ～ 0FFH の範囲を越えています。
したがって，この記述はエラーとなります。

DW

ワード領域を初期化します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	DW	(*サイズ*)	[; *コメント*]
		または	
[*ラベル* :]	DW	*初期値* [, …]	[; *コメント*]

[機能]

- ワード領域を初期化します。
  初期化するワード数は，サイズとして指定することができます。
- オペランドで指定された初期値で，メモリをワード（2 バイト）単位に初期化します。

[用途]

- プログラムで使用するアドレスやデータなどの 16 ビットの定数を定義するときに，DW 疑似命令を使用します。

[説明]

- オペランドがカッコ“ (”，“) ”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

（1）サイズ指定の場合

（a）オペランドにサイズを記述した場合，アセンブラは指定されたワード数分の領域を“00H”で初期化します。

（b）サイズには，絶対式を記述します。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

（2）初期値指定の場合

（a）定数

16 ビット以下の定数です。

（b）式

式の値は，16 ビット・データとして確保されます。

文字列は，初期値として記述できません。

- DW 疑似命令は，ビット・セグメント内では記述できません。
- 初期値の上位 2 桁がメモリの上位アドレスに，下位 2 桁がメモリの下位アドレスに確保されます。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式が記述することができます。

［使用例］

```
                NAME    SAMP1
                CSEG
WORK1 :         DW      ( 10 )          ; (1)
WORK2 :         DW      ( 128 )         ; (1)
                CSEG
                ORG     10H
                DW      MAIN            ; (2)
                DW      SUB1            ; (2)

                CSEG
MAIN :
                CSEG
SUB1 :

DATA :          DW      1234H, 5678H    ; (3)
                END
```

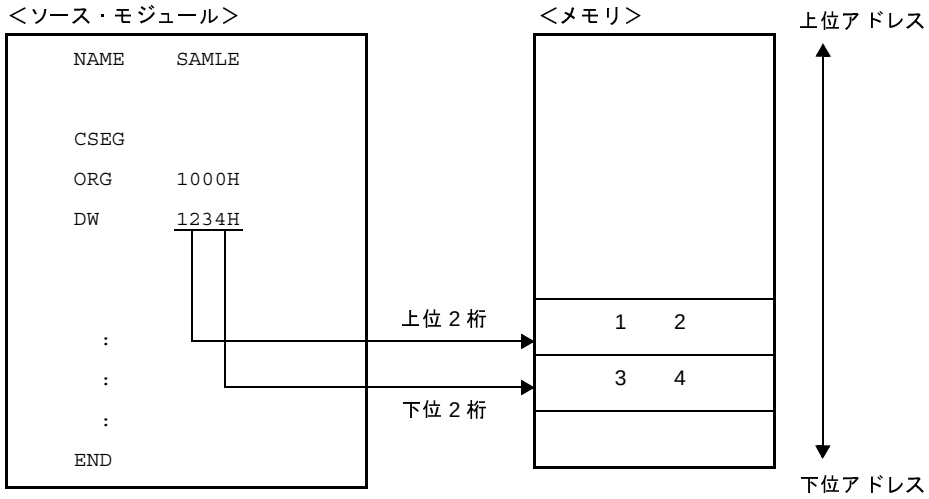
（1）サイズを指定しているので，それぞれのワード領域を値“00H”で初期化します。

（2）ベクタ・エントリ・アドレスを DW 疑似命令で定義します。

（3）2 ワードの領域を“34127856”の値で初期化します。

注意　ワード値は，上位 2 桁でメモリの上位アドレスを下位 2 桁でメモリの下位アドレスを初期化します。

【例】


＜ソース・モジュール＞
NAME SAMLE
CSEG
ORG 1000H
DW 1234H
:
:
:
END
上位 2 桁
下位 2 桁
＜メモリ＞
1 2
3 4
上位アドレス
下位アドレス

DG

20 ビット領域を 32 ビット（4 バイト）単位で初期化します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	DG	(*サイズ*)	[; *コメント*]
		または	
[*ラベル* :]	DG	*初期値* [, …]	[; *コメント*]

[機能]

- 20 ビット領域を 32 ビット（4 バイト）単位でを初期化します。オペランドとして，初期値，あるいはサイズを指定することができます。
- オペランドで指定された初期値で，メモリを 4 バイト単位に初期化します。

[用途]

- プログラムで使用するアドレスやデータなどの 20 ビットの定数を定義するときに，DG 疑似命令を使用します。

[説明]

- オペランドがカッコ“ (”，“) ”で囲まれている場合はサイズ指定とみなされ，そうでない場合は初期値とみなされます。

(1) サイズ指定の場合

(a) オペランドにサイズを記述した場合，アセンブラは指定された数 × 4 バイト分の領域を“00H”で初期化します。

(b) サイズには，絶対式を記述します。サイズの記述が不正な場合，エラーが出力され，初期化は行われません。

(2) 初期値指定の場合

(a) 定数

20 ビット以下の定数です。

（b）式

式の値は，20 ビット・データとして確保されます。

文字列は，初期値として記述できません。

- DG 疑似命令は，ビット・セグメント内では記述できません。
- 初期値の最上位 1 バイトがメモリの HIGH WORD アドレスに，最下位 1 バイトがメモリの下位アドレスに，最下位 2 バイト中上位 1 バイトがメモリの上位アドレスに確保されます。
- 初期値は，1 行の範囲であれば複数指定することができます。
- 初期値として，リロケータブルなシンボルや外部参照シンボルを含んだ式を記述することができます。

［使用例］

```
                NAME    SAMP1

DATA1 :         DG      12345H, 56789H          ; (1)
DATA2 :         DG      ( 10 )                  ; (2)
                END
```

（1）4 バイトの領域を“4523010089670500”の値で初期化します。

（2）40 バイト（10 × 4 バイト）の領域が“00H”で初期化されます。

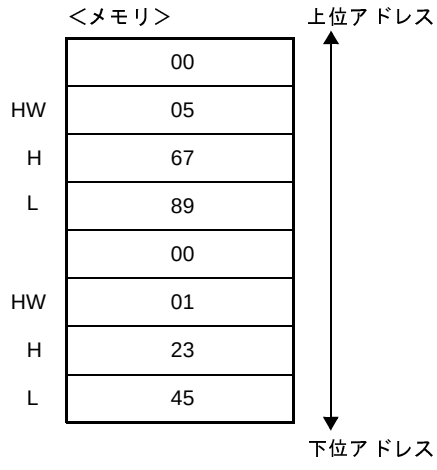
注意　20 ビット値は，最上位 1 バイトでメモリの HIGH WORD アドレスを最下位 1 バイトで，メモリの下位アドレスを最下位 2 バイト中上位 1 バイトで上位アドレスを初期化します。

【例】

＜ソース・モジュール＞

```
NAME    SAMP1
        CSEG
DATA1 : DG      12345H, 56789H
          :
        END
```

＜メモリ＞



HW : HIGH WORD

H : HIGH

L : LOW

DS

オペランドで指定したバイト数分のメモリ領域を確保します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	DS	*絶対式*	[; *コメント*]

[機能]

- オペランドで指定したバイト数分のメモリ領域を確保します。

[用途]

- DS 疑似命令は，主にプログラムで使用するメモリ（RAM）の領域を確保するときに使用します。
  ラベルがある場合は，確保したメモリ領域の先頭アドレスの値をそのラベルに割り付けます。ソース・モジュールでは，このラベルを使用してメモリを操作する記述をします。

[説明]

- 確保する領域の内容は，不定です。
- 絶対式は，符号なし 16 ビットで評価されます。
- オペランドの値が 0 のときは，領域は確保されません。
- DS 疑似命令は，ビット・セグメント内では記述することはできません。
- DS 疑似命令のシンボルは後方参照のみです。
- オペランドに記述できるのは，絶対式を拡張した次のものです。
  - 定数
  - 定数に演算を施した式（定数式）
  - 定数，または定数式で定義された EQU シンボル，または SET シンボル
  - ADDRESS 属性の式 1 － ADDRESS 属性の式 2
    “ADDRESS 属性の式 1”中のラベル 1 と“ADDRESS 属性の式 2”中のラベル 2 は，リロケータブルな場合には，同一セグメント中で定義されていなければなりません。
    ただし，以下の場合にはエラーとなります。
    - ラベル 1 とラベル 2 が同一セグメントで，2 つのラベルの間にその場でオブジェクト・コードのバイト数が決定できない BR 疑似命令が記述されている場合
    - ラベル 1 とラベル 2 が別のセグメントで，属するセグメントの先頭からラベルまでの間に，その場でオブジェクト・コードのバイト数が決定できない BR 疑似命令が記述されている場合
  - 上記の 4 つの式に演算を施した式
- オペランドに記述することのできないものを次に示します。
  - 外部参照シンボル

- ADDRESS 属性の式 1 － ADDRESS 属性の式 2 を EQU で定義したシンボル
- ADDRESS 属性の式 1 － ADDRESS 属性の式 2 の形で式 1，2 のいずれかにロケーション・カウンタ（$）が記述された場合
- ADDRESS 属性の式に HIGH/LOW/DATAPOS/BITPOS を施した式を EQU で定義したシンボル

［使用例］

```
                NAME    SAMPLE
                DSEG
TABLE1 :        DS      10              ; (1)
WORK1 :         DS      2               ; (2)
WORK2 :         DS      1               ; (3)
                CSEG
                MOVW    HL, #TABLE1
                MOV     A, !WORK2
                MOVW    BC, #WORK1
                END
```

(1) 10 バイトの作業領域を確保しますが，領域の内容は不定です。ラベル“TABLE1”を先頭アドレスに割り付けます。

(2) 1 バイトの作業領域を確保します。

(3) 2 バイトの作業領域を確保します。

DBIT

ビット・セグメント中で 1 ビットのメモリ領域を確保します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ネーム*]	DBIT	*なし*	[; *コメント*]

[機能]

- ビット・セグメント中で 1 ビットのメモリ領域を確保します。

[用途]

- DBIT 疑似命令は，ビット・セグメント中で，ビット領域を確保するために使用します。

[説明]

- DBIT 疑似命令は，ビット・セグメント中でのみ記述します。
- 確保した領域の内容は，不定です。
- シンボル欄にネームを記述した場合，そのネームは値として，アドレスとビット位置を持ちます。
- 定義したネームは，saddr.bit，addr16.bit，ES:addr16.bit を要求される箇所に記述することができます。

[使用例]

```
        NAME    SAMPLE
        BSEG
BIT1    DBIT            ;  (1)
BIT2    DBIT            ;  (1)
BIT3    DBIT            ;  (1)

        CSEG
        SET1    BIT1    ;  (2)

        CLR1    BIT2    ;  (3)
        END
```

(1) 1 ビットごとの領域を確保し，それぞれのアドレスとビット位置を値として持つネーム（BIT1，BIT2，BIT3）を定義します。

（2）この記述は，“SET1 saddr.bit”に該当します。

“saddr.bit”として，（1）で確保したビット領域のネーム BIT1 を記述します。

（3）この記述は，“CLR1 saddr.bit”に該当します。

“saddr.bit”として，ネーム BIT2 を記述します。

4. 2. 5 リンケージ疑似命令

リンケージ疑似命令は，ほかのモジュールで定義されているシンボルを参照する場合に，その関連性を明白にさせるためのものです。

1 つのプログラムがモジュール 1 とモジュール 2 に分けて作成されている場合を考えます。モジュール 1 中において，モジュール 2 中で定義されているシンボルを参照したい場合，お互いのモジュールで何の宣言もなくそのシンボルを使うわけにはいきません。このため，「使いたい」，「使ってもよい」の表示をそれぞれのモジュールで行う必要があります。

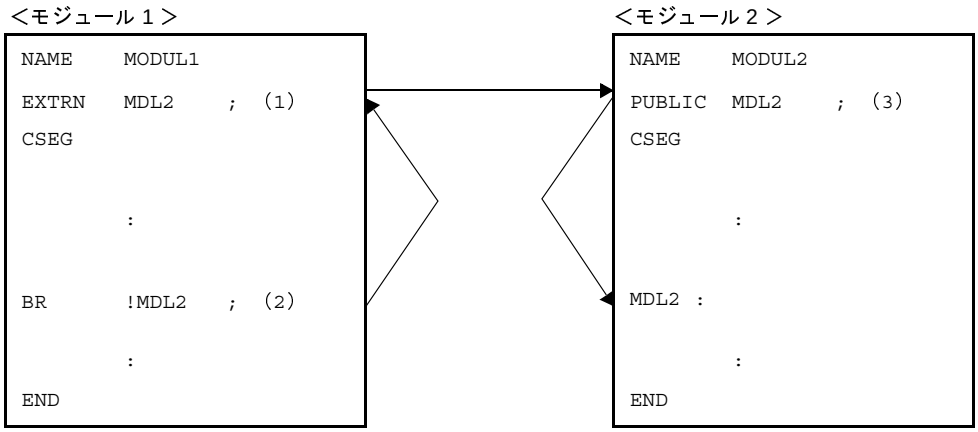
モジュール 1 では，「ほかのモジュール中で定義されているシンボルを参照したい」というシンボルの外部参照宣言をします。一方，モジュール 2 では，「そのシンボルは，ほかのシンボルで参照してもよい」というシンボルの外部定義宣言をします。

外部参照と外部定義という 2 つの宣言が有効に行われて，はじめてそのシンボルを参照することができます。

この相互関係を成立させるのが，リンケージ疑似命令であり，次の命令があります。

- シンボルの外部参照宣言を行うもの：EXTRN，および EXTBIT 疑似命令
- シンボルの外部定義宣言を行うもの：PUBLIC 疑似命令

図 4―7 2 つのモジュール間のシンボルの関係



上記のモジュール 1 では，モジュール 2 の中で定義しているシンボル“MDL2”を（2）で参照しているため，（1）で EXTRN 疑似命令により外部参照宣言を行っています。

モジュール 2 では，モジュール 1 から参照されるシンボル“MDL2”を（3）で，PUBLIC 疑似命令により外部定義宣言を行っています。

この外部参照，外部定義シンボルが正しく対応しているかどうかは，リンカによりチェックされます。

リンケージ疑似命令には，次のものがあります。

制御命令	概要
EXTRN	本モジュールで参照するほかのモジュールのビット・シンボル以外のシンボルを宣言
EXTBIT	本モジュールで参照するほかのモジュールのビット・シンボルを宣言
PUBLIC	オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言

EXTRN

本モジュールで参照するほかのモジュールのビット・シンボル以外のシンボルを宣言します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	EXTRN	*シンボル名* [, …]	[; *コメント*]
		または	
[*ラベル* :]	EXTRN	BASE (*シンボル名* [, …])	[; *コメント*]

[機能]

- 本モジュールで参照するほかのモジュールのシンボル（ビット・シンボルを除く）を宣言します。

[用途]

- ほかのモジュールの中で定義されているシンボルを参照する場合には，必ずそのシンボルを EXTRN 疑似命令で外部参照宣言します。
- オペランドの記述形式により，以下の違いがあります。

BASE (*シンボル名* [, …])	64K バイト内の（0H ～ 0FFFF 内）領域のシンボルとして参照可能となります。
再配置属性なし	リンカが配置後，PUBLIC されたシンボルの領域にあわせて処理を行い，参照可能となります。

[説明]

- EXTRN 疑似命令は，ソース・プログラムのどこに記述してもかまいません（「4. 1. 1 基本構成」を参照してください）。
- オペランドには，コンマ（ , ）で区切って最大 20 個のシンボルを指定することができます。
- ビット値を持つシンボルを参照する場合は，EXTBIT 疑似命令で外部参照宣言をします。
- EXTRN 疑似命令で宣言されたシンボルは，ほかのモジュールで PUBLIC 疑似命令で宣言されていなければなりません。
- EXTRN 疑似命令で宣言されたシンボルをモジュール中で参照しなくても，エラーにはなりません。
- EXTRN 疑似命令のオペランドとして，マクロ名を記述することはできません（マクロ名については，「4. 4 マクロ」を参照してください）。
- シンボルは，全モジュール中で一度だけ EXTRN 宣言できます。2 回目以降の宣言に対しては，ワーニングが出力されます。
- すでに宣言されたシンボルは，EXTRN 疑似命令のオペランドに記述することはできません。逆に，EXTRN 宣言したシンボルも，ほかの疑似命令により再定義，宣言することはできません。

- 64K バイト内の（0H ～ 0FFFFH 内）領域を EXTRN 疑似命令で定義したシンボルで参照することができます。
“BASE (シンボル名)”の記述形式で宣言されたシンボル名は，64K バイト内へ参照可能です。

［使用例］

- モジュール 1

```
        NAME    SAMP1
        EXTRN   SYM1, SYM2, BASE ( SYM3 )   ; (1)
        CSEG
S1 :    DW      SYM1                        ; (2)
        MOV     A, SYM2                     ; (3)
        BR      !SYM3                       ; (4)
        END
```

- モジュール 2

```
        NAME    SAMP2
        PUBLIC  SYM1, SYM2, SYM3            ; (5)

        CSEG
SYM1    EQU     0FFH                        ; (6)
DATA1   DSEG    SADDR
SYM2 :  DB      012H                        ; (7)
C1      CSEG    BASE
SYM3 :  MOV     A, #20H                     ; (8)
        END
```

（1）（2）と（3）と（4）で参照するシンボル“SYM1”，“SYM2”，“SYM3”の外部参照宣言を行います。
オペランド欄には，複数のシンボルを記述することができます。

（2）シンボル“SYM1”を参照します。

（3）シンボル“SYM2”を参照します。saddr 領域を参照するコードを出力します。

（4）シンボル“SYM3”を参照します。64K バイト内（0H ～ 0FFFFH 内）の領域を参照するコードを出力します。

（5）シンボル“SYM1”，“SYM2”，“SYM3”を外部定義宣言します。

（6）シンボル“SYM1”を定義します。

（7）シンボル“SYM2”を定義します。

（8）シンボル“SYM3”を定義します。

EXTBIT

本モジュールで参照するほかのモジュールのビット・シンボルを宣言します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	EXTBIT	*ビット・シンボル名* [, …]	[; *コメント*]

[機能]

- 本モジュールで参照するほかのモジュールのビット・シンボルを宣言します。

[用途]

- ほかのモジュールの中で定義されているビット値を持つシンボルを参照する場合には，必ずそのシンボルを EXTBIT 疑似命令で外部参照宣言します。

[説明]

- EXTBIT 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
- オペランドには，コンマ（ , ）で区切って最大 20 個のシンボルを指定することができます。
- EXTBIT 疑似命令で宣言されたシンボルは，ほかのモジュールで PUBLIC 疑似命令で宣言されていなければなりません。
- シンボルは，1 モジュール中で一度だけ EXTBIT 宣言することができます。2 回目以降の宣言に対しては，ワーニングが出力されます。
- EXBIT 疑似命令で宣言されたシンボルをモジュール中で参照しなくても，エラーにはなりません。

[使用例]

- モジュール 1

```
NAME    SAMP1
EXTBIT  FLAG1, FLAG2    ;  (1)
CSEG
SET1    FLAG1           ;  (2)
CLR1    FLAG2           ;  (3)
        END
```

- モジュール 2

```
        NAME    SAMP2
        PUBLIC  FLAG1, FLAG2    ; (4)
        BSEG
FLAG1   DBIT                    ; (5)
FLAG2   DBIT                    ; (6)
        CSEG
        NOP
        END
```

(1) 参照するシンボル“FLAG1”，“FLAG2”の外部参照宣言を行います。
オペランド欄には，複数のシンボルを記述することができます。

(2) シンボル“FLAG1”を参照します。
この記述は，“SET1 saddr.bit”に該当します。

(3) シンボル“FLAG2”を参照します。
この記述は，“CLR1 saddr.bit”に該当します。

(4) シンボル“FLAG1”，“FLAG2”を定義します。

(5) シンボル“FLAG1”を SADDR 領域のビット・シンボルとして定義します。

(6) シンボル“FLAG2”を SADDR 領域のビット・シンボルとして定義します。

PUBLIC

オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	PUBLIC	*シンボル名* [, …]	[; *コメント*]

[機能]

- オペランドに記述したシンボルをほかのモジュールから参照できるよう宣言します。

[用途]

- ほかのモジュールから参照されるシンボル（ビット・シンボルを含む）を定義している場合には，必ず，そのシンボルを PUBLIC 疑似命令で外部定義宣言します。

[説明]

- PUBLIC 疑似命令は，ソース・プログラムのどこに記述してもかまいません。
- オペランドには，コンマ（ , ）で区切って最大 20 個のシンボルを指定することができます。
- オペランドに記述するシンボルは，同一モジュール内で定義していなければなりません。
- シンボルは，全モジュール中で一度だけ PUBLIC 宣言することができます。2 回目以降の宣言は，無視されます。
- 各ビット領域にあるビットシンボルは，PUBLIC 宣言することが可能です。
- 次のシンボルは，オペランドとして記述することはできません。

（1）SET 疑似命令で定義したネーム

（2）同一モジュール内で EXTRN，EXTBIT 疑似命令で定義したシンボル

（3）セグメント名

（4）モジュール名

（5）マクロ名

（6）モジュール内で定義されていないシンボル

（7）SFBIT 属性を持つオペランドを EQU 疑似命令で定義したシンボル

（8）sfr，2ndSFR を EQU 疑似命令で定義したシンボル（ただし，sfr 領域と saddr 領域のオーバーラップしている箇所は除きます。）

［使用例］

- モジュール 1

```
        NAME    SAMP1
        PUBLIC  A1, A2          ; (1)
        EXTRN   B1
        EXTBIT  C1

A1      EQU     10H
A2      EQU     0FFE20H.1

        CSEG
        BR      B1
        SET1    C1
        END
```

- モジュール 2

```
        NAME    SAMP2
        PUBLIC  B1              ; (2)
        EXTRN   A1
        CSEG
B1 :
        MOV     C, #LOW ( A1 )
        END
```

- モジュール 3

```
        NAME    SAMP3
        PUBLIC  C1              ; (3)
        EXTBIT  A2
C1      EQU     0FFE21H.0
        CSEG
        CLR1    A2
        END
```

（1）シンボル A1，A2 が，ほかのモジュールから参照されるシンボルであることを宣言します。

（2）シンボル B1 が，ほかのモジュールから参照されるシンボルであることを宣言します。

（3）シンボル C1 が，ほかのモジュールから参照されるシンボルであることを宣言します。

4. 2. 6　オブジェクト・モジュール名宣言疑似命令

オブジェクト・モジュール名宣言疑似命令は，アセンブラで生成するオブジェクト・モジュールにモジュール名を与えます。

オブジェクト・モジュール名宣言疑似命令には，次のものがあります。

制御命令	概要
NAME	オペランドに記述したオブジェクト・モジュール名をアセンブラの出力するオブジェクト・モジュールに与える

NAME

オペランドに記述したオブジェクト・モジュール名をアセンブラの出力するオブジェクト・モジュールに与えます。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	NAME	*オブジェクト・モジュール名*	[; *コメント*]

[機能]

- オペランドに記述したオブジェクト・モジュール名をアセンブラの出力するオブジェクト・モジュールに与えます。

[用途]

- モジュール名は，デバッガによるシンボリック・デバッグ時に必要となります。

[説明]

- NAME 疑似命令は，ソース中プログラムのどこに記述してもかまいません。
- モジュール名の規則については，「(3) シンボル欄」を参照してください。
- モジュール名として指定できる文字は，OS でファイル名として許す文字から“ (”（28H），“) ”（29H）と 2 バイト文字を除いた文字とします。
- モジュール名をその他の疑似命令，インストラクションのオペランドとして記述することはできません。
- NAME 疑似命令を省略すると，ソース・モジュール・ファイルのプライマリ・ネーム（先頭から 256 文字）がモジュール名になります。なお，プライマリ・ネームは，大文字に変換されて取り出されます。
  複数個指定した場合は，ワーニングが出力され，2 回目以降の宣言は無視されます。
- オペランド欄のモジュール名は，256 文字以内で指定してください。
- シンボル名の大文字，小文字は区別されます。

[使用例]

```
        NAME    SAMPLE  ;  (1)
        DSEG
BIT1 :  DBIT

        CSEG
        MOV     A, B
        END
```

（1）モジュール名を SAMPLE として宣言します。

4. 2. 7　分岐命令自動選択疑似命令

無条件分岐命令において，分岐先アドレスをオペランドとして直接記述するものには，“BR　$addr20”，“BR !addr16”，“BR　$!addr20”，“BR　!!addr20”などがあります。

これらの命令は，命令のバイト数が異なるため，メモリ効率のよいプログラムを作成するためには，ユーザが分岐先の範囲に応じて，どのオペランドが適しているかを選択して使用する必要があります。

そこで，RL78,78K0R アセンブラが自動的に分岐先の範囲に応じて，2 バイト，3 バイト，または 4 バイトの分岐命令を選択する疑似命令を設けました。これを分岐命令自動選択疑似命令と呼びます。

分岐命令自動選択疑似命令には，次のものがあります。

制御命令	概要
BR	オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 2，3，4 バイトの BR 分岐命令を選択し，該当するオブジェクト・コードを生成
CALL	オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 3 バイトから 4 バイトの CALL 分岐命令を選択し，該当するオブジェクト・コードを生成

BR

オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 2 バイトから 4 バイトの BR 分岐命令を選択し，該当するオブジェクト・コードを生成します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	BR	*式*	[; *コメント*]

[機能]

- オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 2 バイトから 4 バイトの BR 分岐命令を選択し，該当するオブジェクト・コードを生成します。

[用途]

- 次の分岐命令のうち，分岐先範囲を判断して，可能であれば命令バイト数の少ない命令を自動的に選択し，出力します。2 バイトの分岐命令で記述できるかどうか，はっきりしない分岐命令については，BR 疑似命令を使用します。

分岐命令	説明
“BR　$addr20”（2 バイト）	分岐先が BR 疑似命令の次のアドレス -80H から +7FH の範囲内で使用可能
“BR　!addr16”（3 バイト）	分岐先が 64K バイト内の場合，使用可能
“BR　$!addr20”（3 バイト）	相対距離を計算し，-8000H から +7FFFH の範囲内で使用可能
“BR　!!addr20”（4 バイト）	上記以外の場合に使用

なお，オペランド（分岐先）が，疑似命令とは異なるリロケータブル・セグメント内で，BASE 領域外に割り当てられる場合は，4 バイト命令に置き換えて出力します。
また，疑似命令とオペランド（分岐先）が異なるセグメントで，BASE 領域外に割り当てられ，かつ，別タイプの場合，オペランドがアブソリュート・セグメント内にあった場合でも 4 バイト命令に置き換えられます。
疑似命令と分岐先が別セグメントで BASE 領域内にある場合には，3 バイト命令（BR　!addr16）に置き換えられます。

備考　別タイプとは，BR 疑似命令がアブソリュート・セグメント内であれば，リロケータブルな別セグメント，BR 疑似命令がリロケータブル・セグメントであれば，アブソリュート・セグメントを示すことです。

- 2 バイトから 4 バイトのどの分岐命令を記述するべきかが明確に判断できる場合は，該当するインストラクションを記述するようにしてください。これにより，BR 疑似命令を記述する場合に比べ，アセンブル時間を短縮することができます。

[説明]

- BR 疑似命令は，コード・セグメント内でのみ使用可能です。
- BR 疑似命令のオペランドには，直接ジャンプ先を記述します。式の先頭に，現在のロケーション・カウンタを示す“$”を記述することはできません。
- 最適化の対象となるためには，次のような条件があります。
  - 式中のラベル，または前方参照シンボルが 1 個以下。
  - ADDRESS 属性の EQU シンボルが記述されていない。
  - ADDRESS 属性の式－ ADDRESS 属性の式を EQU 定義したシンボルが記述されていない。
  - ADDRESS 属性の式に HIGH/LOW/HIGHW/LOWW/MIRHW/MIRLW/DATAPOS/BITPOS を施した式が記述されていない。

これらの条件が満たされていない場合には，4 バイト命令となります。

ただし，これらの条件を満たしている場合でも，分岐先のアドレスが 10000H 地付近の場合で，前方参照と後方参照が混在していると 4 バイト命令になる場合があります。

[使用例]

```
アドレス                NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  BR      L1                      ; (1)
00052H                  BR      L2                      ; (2)
00055H                  BR      L3                      ; (3)

0007DH          L1 :
0FFFFH          L2 :
10000H          L3 :

                C2      CSEG    AT      20050H
20050H                  BR      L4                      ; (4)
27FFFH          L4 :
                        END
```

(1) この BR 疑似命令は，分岐先との相対距離が -80H から +7FH の範囲内なので，2 バイトの分岐命令（BR $addr20）が生成されます。

(2) この BR 疑似命令は，分岐先が 64K 以内なので，3 バイトの分岐命令（BR !addr16）に置き換えられます。

(3) この BR 疑似命令は，4 バイトの分岐命令（BR !!addr20）に置き換えられます。

(4) この BR 疑似命令は，分岐先との相対距離が -8000H から +7FFFH の範囲内なので，3 バイトの分岐命令（BR $!addr20）に置き換えられます。

CALL

オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 3 バイトから 4 バイトの CALL 分岐命令を選択し，該当するオブジェクト・コードを生成します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	CALL	*式*	[; *コメント*]

[機能]

- オペランドで指定された式の値の範囲に応じて，アセンブラが自動的に 3 バイトから 4 バイトの CALL 分岐命令を選択し，該当するオブジェクト・コードを生成します。

[用途]

- 次の分岐命令のうち，分岐先範囲を判断して，可能であれば命令バイト数の少ない命令を自動的に選択し，出力します。3 バイトの分岐命令で記述できるかどうか，はっきりしない分岐命令については，CALL 疑似命令を使用します。

分岐命令	説明
“CALL !addr16”（3 バイト）	分岐先が 64K バイト内の場合，使用可能
“CALL $!addr20”（3 バイト）	相対距離を計算し，-8000H から +7FFFH の範囲内で使用可能
“CALL !!addr20”（4 バイト）	上記以外の場合に使用

なお，オペランド（分岐先）が，疑似命令とは異なるリロケータブル・セグメント内で，BASE 領域外に割り当てられる場合は，4 バイト命令に置き換えて出力します。
また，疑似命令とオペランド（分岐先）が異なるセグメントで，BASE 領域外に割り当てられ，かつ，別タイプ注の場合，オペランドがアブソリュート・セグメント内にあった場合でも 4 バイト命令に置き換えられます。
疑似命令と分岐先が別セグメントで BASE 領域内にある場合には，3 バイト命令（CALL !addr16）に置き換えられます。

注 別タイプとは，CALL 疑似命令がアブソリュート・セグメント内であれば，リロケータブルな別セグメント，CALL 疑似命令がリロケータブル・セグメントであれば，アブソリュート・セグメントを示すことです。

- 3 バイト，または 4 バイトのどのコール命令を記述するべきかが明確に判断できる場合は，該当するインストラクションを記述するようにしてください。これにより，CALL 疑似命令を記述する場合に比べ，アセンブル時間を短縮することができます。

[説明]

- CALL 疑似命令は，コード・セグメント内でのみ使用可能です。
- CALL 疑似命令のオペランドには，直接コール先を記述します。
- 最適化の対象となるためには，次のような条件があります。
  - 式中のラベル，または前方参照シンボルが 1 個以下。
  - ADDRESS 属性の EQU シンボルが記述されていない。
  - ADDRESS 属性の式ー ADDRESS 属性の式を EQU 定義したシンボルが記述されていない。
  - ADDRESS 属性の式に HIGH/LOW/HIGHW/LOWW/MIRHW/MIRLW/DATAPOS/BITPOS を施した式が記述されていない。

これらの条件が満たされていない場合には，4 バイト命令となります。

ただし，これらの条件を満たしている場合でも，分岐先のアドレスが 10000H 番地付近の場合で，前方参照と後方参照が混在していると 4 バイト命令になる場合があります。

[使用例]

```
アドレス                NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  CALL    L1                      ;  (1)
00053H                  CALL    L2                      ;  (2)
08052H          L1 :
0FFFFH          L2 :

                C2      CSEG    AT      20050H
20050H                  CALL    L3                      ;  (3)
27FFFH          L3 :
                        END
```

(1) この CALL 疑似命令は，分岐先が 64K 以内なので，3 バイトの分岐命令（CALL !addr16）に置き換えられます。

(2) この CALL 疑似命令は，4 バイトの分岐命令（CALL !!addr20）に置き換えられます。

(3) この CALL 疑似命令は，分岐先との相対距離が -8000H から +7FFFH の範囲内なので，3 バイトの分岐命令（CALL $!addr20）に置き換えられます。

4. 2. 8　マクロ疑似命令

ソースを記述する場合，使用頻度の高い一連の命令群をそのつど記述するのは面倒です。また，記述ミス増加の原因ともなります。

マクロ疑似命令により，マクロ機能を使用することにより，同じような一連の命令群を何回も記述する必要がなくなり，コーディングの効率を上げることができます。

マクロの基本的な機能は，一連の文の置き換えにあります。

マクロ疑似命令には，次のものがあります。

制御命令	概要
MACRO	MACRO 疑似命令と ENDM 疑似命令の間に記述された一連の文に対し，シンボル欄で指定したマクロ名を付け，マクロを定義
LOCAL	オペランド欄で指定されたシンボル名は，そのマクロ・ボディ内でのみ有効なローカル・シンボルであることを宣言
REPT	REPT 疑似命令と ENDM 疑似命令の間に記述された一連の文をオペランド欄で指定した式の値分だけ，繰り返し展開
IRP	IRP 疑似命令と ENDM 疑似命令の間にある一連の文をオペランドで指定された実引数で仮引数を置き換えながら，実引数の数だけ繰り返し展開
EXITM	MACRO 疑似命令で定義されたマクロ・ボディの展開，および REPT-ENDM，IRP-ENDM による繰り返しを強制的に終了
ENDM	マクロの機能として定義される一連のステートメントを終了

MACRO

MACRO 疑似命令と ENDM 疑似命令の間に記述された一連の文に対し，シンボル欄で指定したマクロ名を付け，マクロを定義します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
マクロ名	MACRO	[*仮引数* [, …]]	[; *コメント*]
	:		
	マクロ・ボディ		
	:		
	ENDM		[; *コメント*]

[機能]

- MACRO 疑似命令と ENDM 疑似命令の間に記述された一連の文（マクロ・ボディと呼びます）に対し，シンボル欄で指定したマクロ名を付け，マクロの定義を行います。

[用途]

- ソース中で，使用頻度の高い一連の文をマクロ定義しておきます。その定義以降では，定義されたマクロ名を記述するだけ（「(2) マクロの参照」を参照）で，そのマクロ名に対応するマクロ・ボディが展開されます。

[説明]

- MACRO 疑似命令には，対応する ENDM 疑似命令がなければなりません。
- シンボル欄に記述するマクロ名の規則については，「(3) シンボル欄」を参照してください。
- マクロを参照する場合は，ニモニック欄に定義済みのマクロ名を記述します。
- オペランド欄に記述する仮引数の規則については，シンボル記述上の規則と同じです。
- 1 つのマクロ疑似命令で指定できる仮引数は，16 個までです。
- 仮引数は，マクロ・ボディ内でのみ有効です。
- 仮引数として予約語を記述すると，エラーとなります。ただし，ユーザ定義シンボルを記述した場合には，仮引数としての認識が優先されます。
- 仮引数と実引数の個数は同じでなければなりません。
- マクロ・ボディ内で定義したネーム / ラベルを LOCAL 疑似命令で宣言すれば，そのネーム／ラベルは 1 回のマクロ展開でのみ有効になります。
- マクロのネスティング（マクロ・ボディ内でほかのマクロを参照すること）は，REPT，IRP あわせて，最大 8 レベルまでです。
- 1 つのモジュール内でのマクロ定義の最大数には，特に制限はありません。メモリが使えるかぎり定義することができます。

- クロスリファレンス・リストには，仮引数の定義行，参照行，シンボル名は出力されません。
- マクロ・ボディ中に，2 つ以上のセグメントを定義することはできません。定義した場合は，エラーが出力されます。

［使用例］

```
        NAME    SAMPLE

ADMAC   MACRO   PARA1, PARA2    ; (1)
        MOV     A, #PARA1
        ADD     A, #PARA2
        ENDM                    ; (2)

        ADMAC   10H, 20H        ; (3)
        END
```

（1）マクロ名“ADMAC”，2 つの仮引数“PARA1”，“PARA2”を指定したマクロ定義をしています。

（2）マクロ定義の終わりを示します。

（3）マクロ ADMAC を参照しています。

LOCAL

オペランド欄で指定されたシンボル名は，そのマクロ・ボディ内でのみ有効なローカル・シンボルであることを宣言します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
なし	LOCAL	*シンボル名* [, …]	[; *コメント*]

[機能]

- オペランド欄で指定されたシンボル名は，そのマクロ・ボディ内でのみ有効なローカル・シンボルであることを宣言します。

[用途]

- マクロ・ボディ内でシンボルを定義しているマクロを 2 回以上参照すると，シンボルは二重定義エラーとなります。LOCAL 疑似命令を使用することにより，シンボルを定義しているマクロを複数回，参照することができます。

[説明]

- オペランド欄に記述するシンボル名の規則については，「(3) シンボル欄」を参照してください。
- ローカル宣言されたシンボルは，展開されるごとに“??RAnnnn”（n = 0000 ～ FFFF）というシンボルに置き換えられます。置き換え後の“??RAnnnn”というシンボルは，グローバル・シンボルと同じ扱いとなり，シンボル・テーブルに登録され，“??RAnnnn”というシンボル名で参照することができます。
- マクロ・ボディ内でシンボルを定義し，そのマクロを 2 回以上参照すると，ソース・モジュール中でそのシンボルを 2 回以上定義することになってしまいます。このため，そのシンボルは，マクロ内でのみ有効なローカル・シンボルであると宣言します。
- LOCAL 疑似命令は，マクロ定義内でのみ使用できます。
- LOCAL 疑似命令は，オペランド欄で指定したシンボルを使用する前に記述しなければなりません（マクロ・ボディの先頭で記述してください）。
- 1 つのモジュール内で LOCAL 疑似命令により定義するシンボル名は，すべて別名でなければいけません（各マクロ内で使用するローカル・シンボル名に同一名は使えません）。
- オペランド欄で指定できるローカル・シンボル数は，1 行以内であればいくつでも定義することができます。ただし，マクロ・ボディ内での最大数は 64 個です。65 個以上のローカル・シンボルが宣言された場合はエラーが出力され，そのマクロ定義は空のマクロ・ボディとして登録されます。参照された場合は，何も展開されません。
- ローカル・シンボルを定義しているマクロは，ネストさせることができません。
- LOCAL 疑似命令で定義したシンボルをマクロ外から参照することはできません。

- シンボルとして予約語を記述することはできません。ただし，ユーザ定義シンボルと同じシンボルを記述した場合には，LOCAL シンボルとしての機能が優先されます。
- LOCAL 疑似命令のオペランドで宣言したシンボルは，クロスリファレンス・リスト，シンボル・テーブル・リストには出力されません。
- LOCAL 疑似命令の行は，展開時に出力されません。
- マクロ定義の仮引数と同名のシンボルをそのマクロ定義の中で LOCAL 宣言した場合，エラーとなります。

［使用例］

```
        NAME    SAMPLE

        ; マクロの定義
MAC1    MACRO
        LOCAL   LLAB            ; (1)
LLAB :                          ;
        BR      $LLAB           ; (2)
        ENDM                    ;

        ; ソース本文
REF1 :  MAC1                    ; (3)

??RA0000 :
        BR      $??RA0000       ; (2)

        BR      !LLAB           ; (4)     ←エラー

REF2 :  MAC1                    ; (5)

??RA0001 :
        BR      $??RA0001       ; (2)
        END
```

(1) シンボル名 LLAB をローカル・シンボルとして定義しています。

(2) マクロ MAC1 内でローカル・シンボル LLAB を参照しています。

(3) マクロ MAC1 を参照しています。

(4) マクロ MAC1 の定義外でローカル・シンボル LLAB を参照しています。
この記述は，エラーになります。

(5) マクロ MAC1 を参照しています。

使用例のアセンブル・リストを次に示します。

```
        Assemble list

ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT


  1       1                                     NAME    SAMPLE
  2       2                     M       MAC1    MACRO
  3       3                     M               LOCAL   LLAB            ;  (1)
  4       4                     M       LLAB :
  5       5                     M               BR      $LLAB           ;  (2)
  6       6                     M               ENDM
  7       7
  8       8     000000                  REF1 :  MAC1                    ;  (3)
          9                     #1      ;
 10             000000         #1      ??RA0000 :
 11             000000  14FE    #1              BR      $??RA0000       ;  (2)
  9      12
 10      13     000002  2C0000                  BR      !LLAB           ;  (4)
***  ERROR E2407 , STNO  13 ( 0 ) Undefined symbol reference 'LLAB'
***  ERROR E2303 , STNO  13 ( 13 ) Illegal expression
 11      14
 12      15     000005                  REF2 :  MAC1                    ;  (5)
 16                            #1      ;
 17             000005         #1      ??RA0001 :
 18             000005  14FE    #1              BR      $??RA0001       ;  (2)
 13      19
 14      20                                    END
```

REPT

REPT 疑似命令と ENDM 疑似命令の間に記述された一連の文をオペランド欄で指定した式の値分だけ，アセンブラが繰り返し展開します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	REPT	*絶対式*	[; *コメント*]
	:		
	ENDM		[; *コメント*]

[機能]

- REPT 疑似命令と ENDM 疑似命令の間に記述された一連の文（REPT-ENDM ブロックと呼びます）をオペランド欄で指定した式の値分だけ，アセンブラが繰り返し展開します。

[用途]

- ソース中で一連の文を連続して繰り返し記述する場合に，REPT，ENDM 疑似命令を使用します。

[説明]

- REPT 疑似命令に対応する ENDM 疑似命令がなければ，エラーとなります。
- REPT-ENDM ブロック内では，マクロ参照，REPT，IRP をあわせたネスト・レベルの最大数 8 までネスティングすることができます。
- REPT-ENDM のブロックの途中で EXITM が現れると，展開を中止します。
- REPT-ENDM のブロック内に，アセンブル制御命令を記述することができます。
- REPT-ENDM のブロック内に，マクロ定義を記述することはできません。
- オペランド欄に記述する絶対式は，符号なし 16 ビットで評価されます。
  絶対式が 0 の場合には，何も展開されません。

［使用例］

```
NAME    SAMP1
CSEG
    ; REPT-ENDM ブロック
REPT    3               ;  (1)
        INC     B
        DEC     C
            ; ソース本文
ENDM                    ;  (2)
END
```

（1）REPT-ENDM ブロックを 3 回連続して展開するよう，指示しています。

（2）REPT-ENDM ブロックの終了を示します。

アセンブルすると，REPT-ENDM ブロックは次のように展開されます。

```
NAME    SAMP1
CSEG
REPT    3
        INC     B
        DEC     C
ENDM
        INC     B
        DEC     C
        INC     B
        DEC     C
        INC     B
        DEC     C
END
```

（1），（2）で定義された REPT-ENDM ブロックが，3 回展開されています。
アセンブル・リスト上には，ソース・モジュールの REPT 疑似命令による定義分（1），（2）は，表示されません。

IRP

IRP 疑似命令と ENDM 疑似命令の間にある一連の文をオペランドで指定された実引数（左から順）で仮引数を置き換えながら，実引数の数だけ繰り返し展開します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	IRP	*仮引数* ,< [*実引数* [, …]] >	[; *コメント*]
	:		
	ENDM		[; *コメント*]

[機能]

- IRP 疑似命令と ENDM 疑似命令の間にある一連の文（IRP-ENDM ブロックと呼びます）をオペランドで指定された実引数（左から順）で仮引数を置き換えながら，実引数の数だけ繰り返し展開します。

[用途]

- ソース中で，一部分だけ変数となる一連の文を連続して繰り返し記述したい場合に，IRP-ENDM 疑似命令を使用します。

[説明]

- IRP 疑似命令には対応する ENDM 疑似命令がなければなりません。
- 実引数は，16 個まで記述することができます。
- IRP-ENDM ブロック内では，マクロ参照，REPT，IRP をあわせたネスト・レベルの最大数 8 までネスティングすることができます。
- IRP-ENDM ブロックの途中に EXITM を記述すると，そこで展開を中止します。
- IRP-ENDM ブロックで，マクロを定義することはできません。
- IRP-ENDM ブロック内に，アセンブル制御命令を記述することができます。

［使用例］

```
NAME    SAMP1
CSEG

IRP     PARA, <0AH, 0BH, 0CH>           ; (1)
    ; IRP-ENDM ブロック
ADD     A, #PARA
MOV     [DE], A
ENDM                                    ; (2)
    ; ソース本文
END
```

(1) 仮引数が PARA，実引数が 0AH，0BH，0CH の 3 個です。
仮引数“PARA”を実引数“0AH”，“0BH”，“0CH”に置き換えながら，IRP-ENDM ブロックを実引数の数 3 回分展開することを指示します。

(2) IRP-ENDM ブロックの終了を示します。

アセンブルすると，IRP-ENDM ブロックは次のように展開されます。

```
NAME    SAMP1
CSEG
    ; IRP-ENDM ブロック
ADD     A, #0AH         ; (3)
MOV     [DE], A
ADD     A, #0BH         ; (4)
MOV     [DE], A
ADD     A, #0CH         ; (5)
MOV     [DE], A
    ; ソース本文
END
```

(1)，(2) で定義された IRP-ENDM ブロックが，実引数の数 3 回分展開されています。

(3) PARA が 0AH に置き換えられました。

(4) PARA が 0BH に置き換えられました。

(5) PARA が 0CH に置き換えられました。

EXITM

MACRO 疑似命令で定義されたマクロ・ボディの展開，および REPT-ENDM，IRP-ENDM による繰り返しを強制的に終了させます。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	EXITM	*なし*	[; *コメント*]

[機能]

- MACRO 疑似命令で定義されたマクロ・ボディの展開，および REPT-ENDM，IRP-ENDM による繰り返しを強制的に終了させます。

[用途]

- この機能は，主に MACRO 疑似命令で定義したマクロ・ボディ中で，条件付きアセンブル（「4. 3. 7　条件付きアセンブル制御命令」を参照）機能を用いている場合に使用します。
- マクロ・ボディ中で，条件付きアセンブル機能を組み合わせて使用している場合，EXITM 疑似命令で強制的にマクロを抜けないと，アセンブルされてはならない部分がアセンブルされてしまう場合があります。このようなときに，EXITM 疑似命令を使用します。

[説明]

- マクロ・ボディ中に EXITM 疑似命令を記述した場合，マクロ・ボディとしては，ENDM 疑似命令までが登録されます。
- EXITM 疑似命令は，マクロ展開時にのみマクロの終了を指示します。
- オペランド欄に何かの記述がある場合には，エラーが出力されますが，EXITM 疑似命令の処理は行われます。
- EXITM 疑似命令が現れると，アセンブルは，IF/_IF/ELSE/ELSEIF/_ELSEIF/ENDIF のネスティング・レベルをそのマクロ・ボディに入ったときのネスティング・レベルまで強制的に戻します。
- マクロ・ボディ中に記述されたインクルード制御命令を展開したときに，インクルード・ファイル中の EXITM 疑似命令が現れた場合は，その EXITM 疑似命令を有効とし，そのレベルのマクロ展開を中止します。

［使用例］

```
        NAME    SAMP1
MAC1    MACRO                           ;  (1)
          ; マクロボディ
        NOT1    CY
$       IF ( SW1 )                      ;  (2)    ←IF ブロック
                BT      A.1, $L1
                EXITM                   ;  (3)
$       ELSE                            ;  (4)    ←ELSE ブロック
                MOV1    CY, A.1
                MOV     A, #0
$       ENDIF                           ;  (5)
$       IF ( SW2 )                      ;  (6)    ←IF ブロック
                BR      [HL]
$       ELSE                            ;  (7)    ←ELSE ブロック
                BR      [DE]
$       ENDIF                           ;  (8)
          ; ソース本文
        ENDM                            ;  (9)

        CSEG
$       SET ( SW1 )                     ;  (10)
        MAC1                            ;  (11)   ←マクロ参照
L1 :    NOP
        END
```

(1) マクロ MAC1 は，マクロ・ボディ内で条件付きアセンブル機能（(2)，(4) - (8)）を使用しています。

(2) 条件付きアセンブルの IF ブロックを定義します。
スイッチ名 SW1 が真（非 0）の場合，IF ブロックがアセンブルされます。

(3)(4) 以降のマクロ・ボディの展開を強制的に終了します。
この (3) の記述がないと，マクロが展開されたとき，アセンブルは (6) 以降のアセンブル処理に移ります。

(4) 条件付きアセンブルの ELSE ブロックを定義します。
スイッチ名 SW1 が偽（0）の場合，ELSE ブロックがアセンブルされます。

(5) 条件付きアセンブルの終了を示します。

(6) 再び，条件付きアセンブルの IF ブロックを定義します。
スイッチ名 SW2 が真（非 0）の場合，これに続く IF ブロックがアセンブルされます。

（7）条件付きアセンブルの ELSE ブロックを定義します。

スイッチ名 SW2 が偽（0）の場合，ELSE ブロックがアセンブルされます。

（8）（6），（7）の条件付きアセンブルの終了を示します。

（9）マクロ・ボディの終了を示します。

（10）SET 制御命令で，スイッチ名 SW1 に真の値（非 0）を与え，条件付きアセンブルの条件を設定します。

（11）マクロ MAC1 を参照しています。

備考　使用例には，条件付きアセンブル制御命令を記述してあります。詳細については，「4. 3. 7　条件付きアセンブル制御命令」を参照してください。マクロ・ボディ，マクロ展開については，「4. 4　マ ク ロ」を参照してください。

使用例のアセンブル・リストを次に示します。

```
          NAME    SAMP1
MAC1      MACRO                   ; (1)
            :
          ENDM                    ; (9)
          CSEG
$         SET ( SW1 )             ; (10)
          MAC1                    ; (11)
            ; マクロ展開部
          NOT1    CY
$         IF ( SW1 )
                  BT      A.1, $L1
            ; ソース本文
L1 :      NOP
          END
```

（11）のマクロ参照により，マクロ MAC1 のマクロ・ボディが展開されています。

（10）でスイッチ名 SW1 に真の値を設定しているため，マクロ・ボディ内の最初の IF ブロックがアセンブルされます。IF ブロックの最後に EXITM 疑似命令があるため，それ以降のマクロ・ボディは展開されていません。

ENDM

マクロの機能として定義される一連のステートメントの終了をアセンブラに指示します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
なし	ENDM	*なし*	[; *コメント*]

[機能]

- マクロの機能として定義される一連のステートメントの終了をアセンブラに指示します。

[用途]

- MACRO 疑似命令，REPT 疑似命令，および IRP 疑似命令に続く一連のマクロ・ステートメントの最後には，必ず ENDM 疑似命令を記述します。

[説明]

- MACRO 疑似命令と ENDM 疑似命令の間に記述された一連のマクロ・ステートメントがマクロ・ボディとなります。
- REPT 疑似命令と ENDM 疑似命令の間に記述された一連のステートメントが，REPT-ENDM ブロックとなります。
- IRP 疑似命令と ENDM 疑似命令の間に記述された一連のステートメントが，IRP-ENDM ブロックとなります。

[使用例]

(1) MACRO-ENDM

```
        NAME    SAMP1
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
          :
        END
```

（2）REPT-ENDM

```
NAME    SAMP2
CSEG
  :
REPT    3
        INC     B
        DEC     C
ENDM
  :
END
```

（3）IRP-ENDM

```
NAME    SAMP3
CSEG
  :
IRP     PARA, <1, 2, 3>
        ADD     A, #PARA
        MOV     [DE], A
ENDM
  :
END
```

4. 2. 9　アセンブル終了疑似命令

アセンブル終了疑似命令は，アセンブラにソース・モジュールの終了を指示します。ソース・モジュールの最後には，必ずアセンブル終了疑似命令を記述します。

アセンブラは，アセンブル終了疑似命令までをソース・モジュールとして処理します。したがって，REPT ブロックや IRP ブロックで，ENDM より前にアセンブル終了疑似命令があると，REPT ブロックと IRP ブロックは無効になります。

アセンブル終了疑似命令には，次のものがあります。

制御命令	概要
END	ソース・モジュールの終了を宣言

END

ソース・モジュールの終了を宣言します。

[記述形式]

シンボル欄	ニモニック欄	オペランド欄	コメント欄
なし	END	*なし*	[; *コメント*]

[機能]

- ソース・モジュールの終了をアセンブラに宣言します。

[用途]

- END 疑似命令は，ソース・モジュールの最後に必ず記述します。

[説明]

- アセンブラは，END 疑似命令が現れるまでソース・モジュールをアセンブルします。したがって，ソース・モジュールの最後には，END 疑似命令が必要です。
- END 疑似命令のあとにも，必ず改行コード（LF）を入力してください。
- END 疑似命令のあとに，空白，タブ，改行コード，コメント以外のステートメントがある場合には，ワーニングが出力されます。

[使用例]

```
NAME    SAMPLE
DSEG
  :
CSEG
  :
END                     ; (1)
```

(1) ソース・モジュールの最後には，必ず END 疑似命令を記述します。

4. 3　制御命令

この章では，制御命令について説明します。

制御命令とは，アセンブラの動作に対し細かい指示を与えるものです。

4. 3. 1　概　　要

制御命令は，アセンブラの動作に対し細かい指示を与えるもので，ソース中に記述します。

制御命令は，オブジェクト・コード生成の対象とはなりません。

次に，制御命令の種類を示します。

表 4—20　制御命令一覧

制御命令の種類	制御命令
アセンブル対象品種指定制御命令	PROCESSOR
デバッグ情報出力制御命令	DEBUG，NODEBUG，DEBUGA，NODEBUGA
クロスリファレンス・リスト出力指定制御命令	XREF，NOXREF，SYMLIST，NOSYMLIST
インクルード制御命令	INCLUDE
アセンブル・リスト制御命令	EJECT，LIST，NOLIST，GEN，NOGEN，COND，NOCOND，TITLE，SUBTITLE，FORMFEED，NOFORMFEED，WIDTH，LENGTH，TAB
条件付きアセンブル制御命令	IF，_IF，ELSEIF，_ELSEIF，ELSE，ENDIF，SET，RESET
漢字コード制御命令	KANJICODE
RAM 領域配置指定制御命令	RAM_ALLOCATE
その他の制御命令	TOL_INF，DGS，DGL

制御命令は，疑似命令と同様に，ソース中に記述します。

また，「表 4—20　制御命令一覧」で示した制御命令のうち，次に示すものは，アセンブラを起動するときにアセンブラ・オプションとしてコマンド行でも指定することができます。

表 4—21　制御命令とアセンブラ・オプション

制御命令	アセンブラ・オプション
PROCESSOR	-c
DEBUG/NODEBUG	-g/-ng
DEBUGA/NODEBUGA	-ga/-nga
XREF/NOXREF	-kx/-nkx
SYMLIST/NOSYMLIST	-ks/-nks
TITLE	-lh
FORMFEED/NOFORMFEED	-lf/-nlf
WIDTH	-lw

制御命令	アセンブラ・オプション
LENGTH	-ll
TAB	-lt
KANJICODE	-zs/-ze/-zn

4. 3. 2　アセンブル対象品種指定制御命令

アセンブル対象品種指定制御命令は，ソース・モジュール・ファイル中でアセンブル対象品種を指定します。

アセンブル対象品種指定制御命令には，次のものがあります。

制御命令	概要
PROCESSOR	ソース・モジュール・ファイル中でアセンブル対象品種を指定

PROCESSOR

ソース・モジュール・ファイル中でアセンブル対象品種を指定します。

[記述形式]

```
[ ]$[ ]PROCESSOR[ ]([ ] 品種名 [ ])
[ ]$[ ]PC[ ]([ ] 品種名 [ ])                    ; 短縮形
```

[機能]

- PROCESSOR 制御命令は，ソース・モジュール・ファイル中でアセンブル対象品種を指定します。

[用途]

- アセンブル対象品種指定は，ソース・モジュール・ファイル，またはコマンド・ラインのどちらかで必ず指定しなければなりません。
- ソース・モジュール・ファイル中でアセンブル対象品種指定の記述がない場合，アセンブルのたびにアセンブル対象品種を指定しなければなりません。したがって，ソース・モジュール・ファイル中でアセンブル対象品種を指定しておくことにより，アセンブラ起動時の手間を省くことができます。

[説明]

- PROCESSOR 制御命令は，モジュール・ヘッダ部にのみ記述することができます。その他に記述した場合，アセンブラはアボートします。
- 指定可能な品種名ついては，各デバイスのユーザーズ・マニュアル，または「CubeSuite+ 対応機能一覧」を参照してください。
- 指定した品種名がアセンブル対象品種と異なる場合，アセンブラはアボートします。
- PROCESSOR 制御命令は，複数指定することはできません。
- アセンブル対象品種指定は，コマンド・ライン上でアセンブラ・オプション（-c）によっても指定できます。ソース・モジュール・ファイル中とコマンド・ラインでの指定が異なる場合，アセンブラはワーニングを出力し，コマンド・ラインでの指定を優先します。
- アセンブラ・オプション（-c）を指定した場合でも，PROCESSOR 制御命令に対する文法チェックは行われます。
- ソース・モジュール・ファイル中，コマンド・ラインのどちらも指定されていない場合，アセンブラはアボートします。

［使用例］

```
$       PROCESSOR ( f1166a0 )
$       DEBUG
$       XREF

        NAME    TEST
          :
        CSEG
```

4. 3. 3　デバッグ情報出力制御命令

デバッグ情報出力制御命令は，ソース・モジュール・ファイル中で，オブジェクト・モジュール・ファイルに対してデバッグ情報の出力を指定することができます。

デバッグ情報出力制御命令には，次のものがあります。

制御命令	概要
DEBUG	オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加
NODEBUG	オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加しない
DEBUGA	オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加
NODEBUGA	オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加しない

DEBUG

オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加します。

[記述形式]

```
[ ]$[ ]DEBUG            ; 省略時解釈
[ ]$[ ]DG               ; 短縮形
```

[機能]

- DEBUG 制御命令は，オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加します。
- ローカル・シンボル情報とは，モジュール名，PUBLIC，EXTRN，EXTBIT シンボル以外のシンボルのことを示します。

[用途]

- DEBUG 制御命令は，ローカル・シンボルも含め，シンボリック・デバッグを行うときに使用します。

[説明]

- DEBUG 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- DEBUG/NODEBUG 制御命令を省略した場合，アセンブラは DEBUG 制御命令が指定されたと解釈して処理を行います。
- DEBUG/NODEBUG 制御命令が複数回記述された場合は，後者が優先されます。
- ローカル・シンボル情報の付加は，コマンド・ライン上でアセンブラ・オプション（-g/-ng）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-ng）を指定した場合でも，DEBUG/NODEBUG 制御命令に対する文法チェックは行われます。

NODEBUG

オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加しません。

[記述形式]

```
[ ]$[ ]NODEBUG
[ ]$[ ]NODG             ; 短縮形
```

[機能]

- NODEBUG 制御命令は，オブジェクト・モジュール・ファイル中にローカル・シンボル情報を付加しません。なお，この場合にも，セグメント名はオブジェクト・モジュール・ファイルに出力されます。
- ローカル・シンボル情報とは，モジュール名，PUBLIC，EXTRN，EXTBIT シンボル以外のシンボルのことを示します。

[用途]

- NODEBUG 制御命令は，次の 3 種類の場合に使用します。
  - グローバル・シンボルのみのシンボリック・デバッグ
  - シンボルなしでのデバッグ
  - オブジェクトのみを必要とするとき（PROM による評価時など）

[説明]

- NODEBUG 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- DEBUG/NODEBUG 制御命令を省略した場合，アセンブラは DEBUG 制御命令が指定されたと解釈して処理を行います。
- DEBUG/NODEBUG 制御命令が複数回記述された場合は，後者が優先されます。
- ローカル・シンボル情報の付加は，コマンド・ライン上でアセンブラ・オプション（-g/-ng）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-ng）を指定した場合でも，DEBUG/NODEBUG 制御命令に対する文法チェックは行われます。

DEBUGA

オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加します。

[記述形式]

```
[ ]$[ ]DEBUGA          ; 省略時解釈
```

[機能]

- DEBUGA 制御命令は，オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加します。

[用途]

- DEBUGA 制御命令は，アセンブラのソース・レベルで，デバッグするときに使用します。なお，ソース・レベルでのデバッグには，“統合デバッガ”が必要です。

[説明]

- DEBUGA 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- DEBUGA/NODEBUGA 制御命令を省略した場合，アセンブラは DEBUGA 制御命令が指定されたと解釈して，処理を行います。
- DEBUGA/NODEBUGA 制御命令が複数回記述された場合は，後者が優先されます。
- アセンブラ・ソース・デバッグ情報の付加は，コマンド・ライン上でアセンブラ・オプション（-ga/-nga）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-nga）を指定した場合でも，DEBUGA/NODEBUGA 制御命令に対する文法チェックは行われます。
- C コンパイラでデバッグ情報を出力して，コンパイルした場合，その出力アセンブル・ソースをアセンブルするときには，デバッグ情報出力制御命令を記述しないでください。アセンブル時に必要な制御命令は，C コンパイラが，アセンブラ・ソース中に制御文として出力します。

NODEBUGA

オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加しません。

[記述形式]

```
[ ]$[ ]NODEBUGA
```

[機能]

- NODEBUGA 制御命令は，オブジェクト・モジュール・ファイル中にアセンブラ・ソース・デバッグ情報を付加しません。

[用途]

- NODEBUGA 制御命令は，次の 2 種類の場合に使用します。
  - アセンブラ・ソース以外でのデバッグ
  - オブジェクトのみを必要とするとき（PROM による評価時など）

[説明]

- NODEBUGA 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- DEBUGA/NODEBUGA 制御命令を省略した場合，アセンブラは DEBUGA 制御命令が指定されたと解釈して，処理を行います。
- DEBUGA/NODEBUGA 制御命令が複数回記述された場合は，後者が優先されます。
- アセンブラ・ソース・デバッグ情報の付加は，コマンド・ライン上でアセンブラ・オプション（-ga/-nga）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-nga）を指定した場合でも，DEBUGA/NODEBUGA 制御命令に対する文法チェックは行われます。
- C コンパイラでデバッグ情報を出力して，コンパイルした場合，その出力アセンブル・ソースをアセンブルするときには，デバッグ情報出力制御命令を記述しないでください。アセンブル時に必要な制御命令は，C コンパイラが，アセンブラ・ソース中に制御文として出力します。

4. 3. 4　クロスリファレンス・リスト出力指定制御命令

クロスリファレンス・リスト出力指定制御命令は，ソース・モジュール・ファイル中でクロスリファレンス・リストの出力指定を行うことができます。

クロスリファレンス・リスト出力指定制御命令には，次のものがあります。

制御命令	概要
XREF	アセンブル・リスト・ファイルにクロスリファレンス・リストを出力
NOXREF	アセンブル・リスト・ファイルにクロスリファレンス・リストを出力しない
SYMLIST	リスト・ファイルにシンボル・リストを出力
NOSYMLIST	リスト・ファイルにシンボル・リストを出力しない

XREF

アセンブル・リスト・ファイルにクロスリファレンス・リストを出力します。

[記述形式]

```
[ ]$[ ]XREF
[ ]$[ ]XR               ; 短縮形
```

[機能]

- XREF 制御命令は，アセンブル・リスト・ファイルにクロスリファレンス・リストを出力することを指示します。

[用途]

- ソース・モジュール・ファイルで定義された各シンボルがソース・モジュール中のどこでどれだけ参照されているか，また，アセンブル・リストの何行目の記述で，そのシンボルを参照しているのか，などの情報を知りたいときに，クロスリファレンス・リストを出力します。
- アセンブルのたびに，クロスリファレンス・リスト出力指定を行うような場合には，ソース・モジュール・ファイル中にこれらを記述することにより，アセンブラ起動時の手間を省くことができます。

[説明]

- XREF 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- XREF/NOXREF 制御命令が複数指定された場合には，後者が優先されます。
- クロスリファレンス・リスト指定は，コマンド・ライン上でアセンブラ・オプション（-kx/-nkx）によって指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，XREF/NOXREF 制御命令に対する文法チェックは行われます。

NOXREF

アセンブル・リスト・ファイルにクロスリファレンス・リストを出力しません。

[記述形式]

```
[ ]$[ ]NOXREF          ; 省略時解釈
[ ]$[ ]NOXR            ; 短縮形
```

[機能]

- NOXREF 制御命令は，アセンブル・リスト・ファイルにクロスリファレンス・リストを出力しないことを指示します。

[用途]

- ソース・モジュール・ファイルで定義された各シンボルがソース・モジュール中のどこでどれだけ参照されているか，また，アセンブル・リストの何行目の記述で，そのシンボルを参照しているのか，などの情報を知りたいときに，クロスリファレンス・リストを出力します。
- アセンブルのたびに，クロスリファレンス・リスト出力指定を行うような場合には，ソース・モジュール・ファイル中にこれらを記述することにより，アセンブラ起動時の手間を省くことができます。

[説明]

- NOXREF 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- XREF/NOXREF 制御命令が複数指定された場合には，後者が優先されます。
- クロスリファレンス・リスト指定は，コマンド・ライン上でアセンブラ・オプション（-kx/-nkx）によって指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，XREF/NOXREF 制御命令に対する文法チェックは行われます。

SYMLIST

リスト・ファイルにシンボル・リストを出力します。

[記述形式]

```
[ ]$[ ]SYMLIST
```

[機能]

- SYMLIST 制御命令は，リスト・ファイルにシンボル・リストを出力することを指示します。

[用途]

- シンボル・リストを出力したい場合に使用します。

[説明]

- SYMLIST 制御命令は，モジュール・ヘッダ部のみに記述できます。
- SYMLIST/NOSYMLIST 制御命令が複数指定された場合には，後者が優先されます。
- シンボル・リストの出力は，コマンド・ライン上でアセンブラ・オプション（-ks/-nks）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，SYMLIST/NOSYMLIST 制御命令に対する文法チェックは行われます。

NOSYMLIST

リスト・ファイルにシンボル・リストを出力しません。

[記述形式]

```
[ ]$[ ]NOSYMLIST        ; 省略時解釈
```

[機能]

- NOSYMLIST 制御命令は，リスト・ファイルにシンボル・リストを出力しないことを指示します。

[用途]

- シンボル・リストを出力したい場合に使用します。

[説明]

- NOSYMLIST 制御命令は，モジュール・ヘッダ部のみに記述できます。
- SYMLIST/NOSYMLIST 制御命令が複数指定された場合には，後者が優先されます。
- シンボル・リストの出力は，コマンド・ライン上でアセンブラ・オプション（-ks/-nks）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，SYMLIST/NOSYMLIST 制御命令に対する文法チェックは行われます。

4. 3. 5　インクルード制御命令

インクルード制御命令は，ソース・モジュール中でほかのソース・モジュール・ファイルを引用する場合に使用します。

インクルード制御命令を有効に使用することにより，ソースの記述の手間を軽減することができます。

インクルード制御命令には，次のものがあります。

制御命令	概要
INCLUDE	ほかのソース・モジュール・ファイルの一連のステートメントを引用

INCLUDE

ほかのソース・モジュール・ファイルの一連のステートメントを引用します。

[記述形式]

```
[ ]$[ ]INCLUDE[ ]([ ] ファイル名 [ ])
[ ]$[ ]IC[ ]([ ] ファイル名 [ ])              ; 短縮形
```

[機能]

- 指定されたファイルの内容を指定された行以降に挿入展開し，アセンブルします。

[用途]

- 複数のソース・モジュール中で共通に記述する比較的大きな一連のステートメントを１つのファイル（インクルード・ファイル）としてまとめておきます。
  各ソース・モジュール中で，その一連のステートメントを引用する必要があるとき，INCLUDE 制御命令により，必要とするインクルード・ファイル名を指定します。
  これにより，ソース・モジュールの記述作業を軽減することができます。

[説明]

- INCLUDE 制御命令は，通常のソースにのみ記述することができます。
- アセンブラ・オプション（-I）で，インクルード・ファイルのパス名やドライブ名を指定することができます。
- インクルード・ファイルの読み込みパスのサーチの順番は，次のとおりです。

（1）インクルード・ファイルがパス名なしで指定された場合

（a）ソース・ファイルのあるパス

（b）アセンブラ・オプション（-I）で指定されたパス

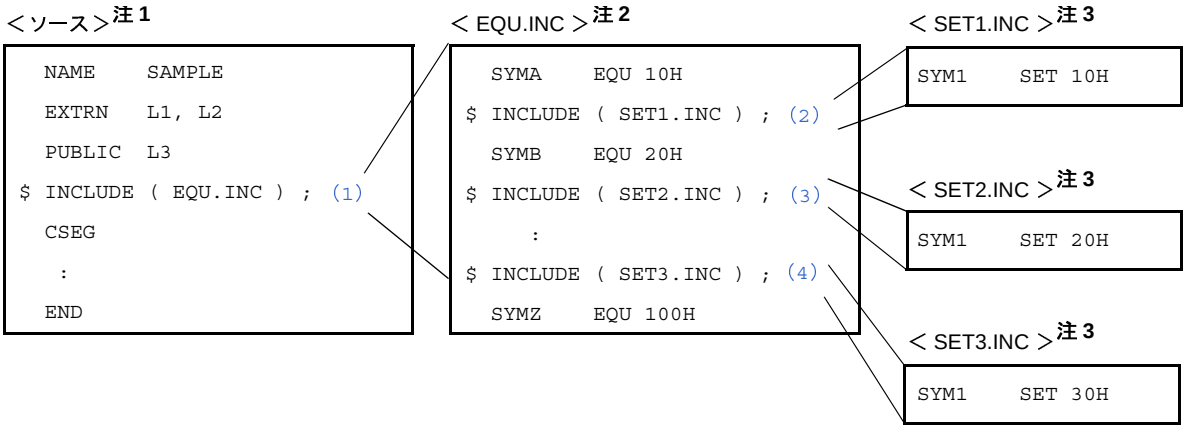
（c）環境変数 INC78K0R で指定されたパス

（2）インクルード・ファイルがパス名付きで指定された場合

ドライブ名，またはバックスラッシュ（￥）から始まるパス名付きで指定した場合には，インクルード・ファイル名に付いているパス，相対パス（先頭に¥がない）付きで指定された場合には，インクルード・ファイル名の前に（1）の順でパス名を付加します。

- インクルード・ファイルは，7 重までネスティングすることが可能です。つまり，ネスト・レベルの最大数は 8 です（ネスティングとは，インクルード・ファイル中で，別のインクルード・ファイルを指定することです）。
- インクルード・ファイルに，END 疑似命令の記述は必要ありません。
- インクルード・ファイルがオープンできない場合，アセンブラはアボートします。
- インクルード・ファイル中は，“IF，または _IF ～ ENDIF”の対応がとれた状態で，閉じなければなりません。もし，インクルード・ファイルの展開の入口の IF レベルと，展開終了直後の IF レベルの対応がとれていない場合，アセンブラはエラーを出力し，レベルを強制的に入口でのレベルに戻してアセンブルを続けます。
- インクルード・ファイル中でマクロを定義するときは，そのマクロ定義はインクルード・ファイル中で閉じていなければなりません。
  突然 ENDM 疑似命令が現れた場合には，エラーが出力され，その ENDM 疑似命令は無視されます。また，マクロ定義疑似命令があるのにそのインクルード・ファイル中に ENDM 疑似命令がない場合には，エラーが出力され，ENDM 疑似命令があるものとして処理されます。
- インクルード・ファイル中に，2 つ以上のセグメントを定義することはできません。定義した場合は，エラーが出力されます。

［使用例］



(1) インクルード・ファイルとして，“EQU.INC”を指定しています。

(2) インクルード・ファイルとして，“SET1.INC”を指定しています。

(3) インクルード・ファイルとして，“SET2.INC”を指定しています。

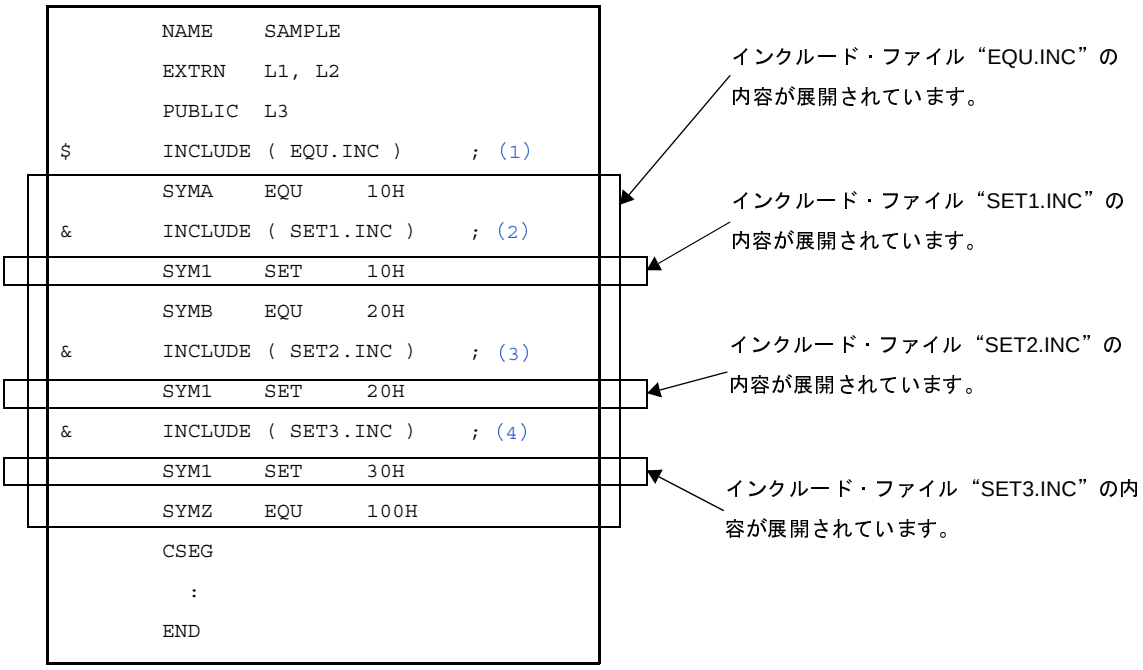
(4) インクルード・ファイルとして，“SET3.INC”を指定しています。

注 1. ソース・ファイル中には，$IC を複数指定することができます。また，同じインクルード・ファイルを複数指定することもできます。

2. EQU.INC には，$IC を複数指定することができます。

3. SET1.INC，SET2.INC，および SET3.INC 中には，$IC を指定することはできません。

このソースがアセンブルされると，インクルード・ファイルの内容が次のように展開されます。



4. 3. 6　アセンブル・リスト制御命令

アセンブル・リスト制御命令は，アセンブラの出力するアセンブル・リストに対して，改ページ，リスト出力をしない部分，サブタイトル・メッセージ出力などを指示するものです。

アセンブル・リスト制御命令には，次のものがあります。

制御命令	概要
EJECT	アセンブル・リストの改ページを指示
LIST	アセンブル・リストの出力開始位置を指示
NOLIST	アセンブル・リストの出力中止位置を指示
GEN	マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力
NOGEN	マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力しない
COND	条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力
NOCOND	条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力しない
TITLE	アセンブル・リスト，シンボル・テーブル・リスト，およびクロスリファレンス・リストのヘッダのタイトル欄に文字列を印字
SUBTITLE	アセンブル・リストのヘッダのサブタイトル欄に文字列を印字
FORMFEED	リスト・ファイルの最後にフォーム・フィードを出力
NOFORMFEED	リスト・ファイルの最後にフォーム・フィードを出力しない
WIDTH	リスト・ファイルの１行の最大文字数を指示
LENGTH	リスト・ファイルの１ページの行数を指示
TAB	リスト・ファイルのタブの展開文字数を指示

EJECT

アセンブル・リストの改ページを指示します。

[記述形式]

```
[ ]$[ ]EJECT
[ ]$[ ]EJ               ; 短縮形
```

[省略時解釈]

- EJECT 制御命令は，指定していないものとします。

[機能]

- EJECT 制御命令は，アセンブル・リストの改ページをアセンブラに指示します。

[用途]

- ソース・モジュール中で改ページを行いたい箇所に記述します。

[説明]

- EJECT 制御命令は，通常のソースのみに記述することができます。
- EJECT 制御命令自身のイメージを出力したあとに，リストを改ページします。
- コマンド・ラインでアセンブラ・オプション（-np），（-ll0）の指定がある場合や，制御命令の指定によりリスト出力禁止状態の場合，EJECT 制御命令は無効です。
- EJECT 制御命令のあとに不正な記述があった場合，アセンブラはエラーを出力します。

[使用例]

```
          :
        MOV     [DE+], A
        BR      $$
$       EJECT                   ; (1)
          :
        CSEG
          :
        END
```

（1）EJECT 制御命令により改ページを行います。

使用例のアセンブル・リストを次に示します。

```
          :
         MOV      [DE+], A
         BR       $$
$        EJECT                    ;  (1)
---------------------------------------------- 改ページ ------------------------
          :
         CSEG
          :
         END
```

LIST

アセンブル・リストの出力開始位置をアセンブラに指示します。

[記述形式]

```
[ ]$[ ]LIST             ; 省略時解釈
[ ]$[ ]LI               ; 短縮形
```

[機能]

- LIST 制御命令は，アセンブル・リストの出力開始位置をアセンブラに指示します。

[用途]

- LIST 制御命令は，NOLIST 制御命令で指定したアセンブル・リスト出力中止の状態を再びアセンブル・リスト出力の状態にする場合に使用します。
  NOLIST 制御命令と LIST 制御命令を組み合わせて使用することにより，出力するアセンブル・リストの量や印字内容を制御することができます。

[説明]

- LIST 制御命令は，通常のソースにのみに記述することができます。
- NOLIST 制御命令以降，LIST 制御命令の指定があると，LIST 制御命令以降のステートメントは再びアセンブル・リストに出力されます。記述した LIST/NOLIST 制御命令自身もアセンブル・リストに出力されます。
- LIST/NOLIST 制御命令を省略した場合には，すべてのステートメントがアセンブル・リストに出力されます。

[使用例]

```
        NAME    SAMP1
$       NOLIST          ; (1)
DATA1   EQU     10H     ; アセンブル・リストに出力されません。
DATA2   EQU     11H     ; アセンブル・リストに出力されません。
          :             ; アセンブル・リストに出力されません。
DATAX   EQU     20H     ; アセンブル・リストに出力されません。
DATAY   EQU     20H     ; アセンブル・リストに出力されません。
$       LIST            ; (2)
        CSEG
          :
        END
```

（1）NOLIST 制御命令を指定しているので，これ以降，（2）の LIST 制御命令までのステートメントは，アセンブル・リストに出力されません。
NOLIST 制御命令自身は出力されます。

（2）LIST 制御命令を指定しているので，これ以降のステートメントは，再びアセンブル・リストに出力されます。
LIST 制御命令自身は出力されます。

NOLIST

アセンブル・リストの出力中止位置をアセンブラに指示します。

[記述形式]

```
[ ]$[ ]NOLIST
[ ]$[ ]NOLI             ; 短縮形
```

[機能]

- NOLIST 制御命令は，アセンブル・リストの出力中止位置をアセンブラに指示します。
  NOLIST 制御命令を指定したあと，次に LIST 制御命令が現れるまでのステートメントは，アセンブルされますがアセンブル・リストには出力されません。

[用途]

- NOLIST 制御命令は，リストの出力量を制限するために使用します。
- LIST 制御命令は，NOLIST 制御命令で指定したアセンブル・リスト出力中止の状態を再びアセンブル・リスト出力の状態にする場合に使用します。
  NOLIST 制御命令と LIST 制御命令を組み合わせて使用することにより，出力するアセンブル・リストの量や印字内容を制御することができます。

[説明]

- NOLIST 制御命令は，通常のソースにのみに記述することができます。
- NOLIST 制御命令は，アセンブル・リストの出力を中止するもので，アセンブルを中止するものではありません。
- NOLIST 制御命令以降，LIST 制御命令の指定があると，LIST 制御命令以降のステートメントは再びアセンブル・リストに出力されます。記述した LIST/NOLIST 制御命令自身もアセンブル・リストに出力されます。
- LIST/NOLIST 制御命令を省略した場合には，すべてのステートメントがアセンブル・リストに出力されます。

［使用例］

```
        NAME    SAMP1
$       NOLIST          ; (1)
DATA1   EQU     10H     ; アセンブル・リストに出力されません。
DATA2   EQU     11H     ; アセンブル・リストに出力されません。
          :             ; アセンブル・リストに出力されません。
DATAX   EQU     20H     ; アセンブル・リストに出力されません。
DATAY   EQU     20H     ; アセンブル・リストに出力されません。
$       LIST            ; (2)
        CSEG
          :
        END
```

(1) NOLIST 制御命令を指定しているので，これ以降，(2) の LIST 制御命令までのステートメントは，アセンブル・リストに出力されません。
NOLIST 制御命令自身は出力されます。

(2) LIST 制御命令を指定しているので，これ以降のステートメントは，再びアセンブル・リストに出力されます。
LIST 制御命令自身は出力されます。

GEN

マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力します。

[記述形式]

```
[ ]$[ ]GEN              ; 省略時解釈
```

[機能]

- GEN 制御命令は，マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力します。

[用途]

- アセンブル・リストの出力量を制限するために使用します。

[説明]

- GEN 制御命令は，通常のソースのみに記述することができます。
- GEN/NOGEN 制御命令を省略した場合，マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力します。
- GEN/NOGEN 制御命令自身のイメージが出力されたあとに，リスト制御が行われます。
- NOGEN 制御命令のあと，GEN 制御命令が指定された場合には，再びマクロ展開部の出力が開始されます。

[使用例]

```
        NAME    SAMP
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        END
```

使用例のアセンブル・リストを次に示します。

```
        NAME    SAMP1
$       NOGEN                           ;  (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        MOV     A, #10H                 ; マクロ展開部は出力されません。
        AUD     A, #20H                 ; マクロ展開部は出力されません。
        END
```

（1）NOGEN 制御命令が指定されているので，マクロ展開部はリストに出力されません。

NOGEN

マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力しません。

[記述形式]

```
[ ]$[ ]NOGEN
```

[機能]

- NOGEN 制御命令はマクロ定義部，および参照行はアセンブル・リストに出力しますが，マクロ展開部は出力されません。

[用途]

- アセンブル・リストの出力量を制限するために使用します。

[説明]

- NOGEN 制御命令は，通常のソースのみに記述することができます。
- GEN/NOGEN 制御命令を省略した場合，マクロ定義部，参照行，およびマクロ展開部をアセンブル・リストに出力します。
- GEN/NOGEN 制御命令自身のイメージが出力されたあとに，リスト制御が行われます。
- NOGEN 制御命令でリスト出力中断後もアセンブルは続けられ，アセンブル・リスト上のステートメント・ナンバ（STNO）の値がカウント・アップされます。
- NOGEN 制御命令のあと，GEN 制御命令が指定された場合には，再びマクロ展開部の出力が開始されます。

[使用例]

```
        NAME    SAMP
$       NOGEN                           ; (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        END
```

使用例のアセンブル・リストを次に示します。

```
        NAME    SAMP1
$       NOGEN                           ;  (1)
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A,  #PARA2
        ENDM
        CSEG
        ADMAC   10H, 20H
        MOV     A, #10H                 ; マクロ展開部は出力されません。
        AUD     A, #20H                 ; マクロ展開部は出力されません。
        END
```

（1）NOGEN 制御命令が指定されているので，マクロ展開部はリストに出力されません。

COND

条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力します。

[記述形式]

```
[ ]$[ ]COND             ; 省略時解釈
```

[機能]

- COND 制御命令は，条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力します。

[用途]

- アセンブル・リストの出力量を制限するために使用します。

[説明]

- COND 制御命令は，通常のソースにのみに記述することができます。
- COND/NOCOND 制御命令を省略した場合，条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力します。
- COND/NOCOND 制御命令自身のイメージが出力されたあとに，リスト制御が行われます。
- NOCOND 制御命令のあと，COND 制御命令が指定された場合には，再び条件不成立部分，および IF/_IF/ELSEIF/_ELSEIF/ELSE/ENDIF が記述されている行の出力が開始されます。

[使用例]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                      ; アセンブルしてもリストには，出力されません。
                MOV     A, #1H
$       ELSE                            ; アセンブルしてもリストには，出力されません。
                MOV     A, #0H          ; アセンブルしてもリストには，出力されません。
$       ENDIF                           ; アセンブルしてもリストには，出力されません。
        END
```

NOCOND

条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力しません。

[記述形式]

```
[ ]$[ ]NOCOND
```

[機能]

- NOCOND 制御命令は，条件アセンブルの条件成立部分のみをアセンブル・リストに出力し，条件不成立部分，および IF/_IF/ELSEIF/_ELSEIF/ELSE/ENDIF が記述されている行は，出力されません。

[用途]

- アセンブル・リストの出力量を制限するために使用します。

[説明]

- NOCOND 制御命令は，通常のソースにのみに記述することができます。
- COND/NOCOND 制御命令を省略した場合，条件アセンブルの条件成立部分，および不成立部分をアセンブル・リストへ出力します。
- COND/NOCOND 制御命令自身のイメージが出力されたあとに，リスト制御が行われます。
- NOCOND 制御命令で，リスト出力中断後も ALNO，STNO がカウント・アップされます。
- NOCOND 制御命令のあと，COND 制御命令が指定された場合には，再び条件不成立部分，および IF/_IF/ELSEIF/_ELSEIF/ELSE/ENDIF が記述されている行の出力が開始されます。

[使用例]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                      ; アセンブルしてもリストには，出力されません。
                MOV     A, #1H
$       ELSE                            ; アセンブルしてもリストには，出力されません。
                MOV     A, #0H          ; アセンブルしてもリストには，出力されません。
$       ENDIF                           ; アセンブルしてもリストには，出力されません。
        END
```

TITLE

アセンブル・リスト，シンボル・テーブル・リスト，およびクロスリファレンス・リストのヘッダのタイトル欄に文字列を印字します。

[記述形式]

```
[ ]$[ ]TITLE[ ]([ ]'タイトル・ストリング'[ ])
[ ]$[ ]TT[ ]([ ]'タイトル・ストリング'[ ])          ; 短縮形
```

[省略時解釈]

- TITLE 制御命令は指定されていないものとし，アセンブル・リストのヘッダのタイトル欄は空白となります。

[機能]

- TITLE 制御命令は，アセンブル・リスト，シンボル・テーブル・リスト，およびクロスリファレンス・リストの各ページのヘッダのタイトル欄に，印字する文字列を指定します。

[用途]

- タイトルを各ページに印字することにより，リストの内容が一目でわかります。
- アセンブルのたびにタイトル指定を行うような場合，ソース・モジュール・ファイル中にこれらを記述することにより，アセンブラ起動時の手間を省くことができます。

[説明]

- TITLE 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- TITLE 制御命令が複数指定された場合には，あとで指定したものが有効となります。
- タイトル・ストリングは，60 文字以内です。61 文字以上の場合には，先頭の 60 文字を有効とします。
  ただし，アセンブル・リスト・ファイルの 1 行の文字数指定（X とします）が 119 文字以下の場合，“X - 60”文字以内とします。
- タイトル・ストリングに引用符（'）を記述する場合には，2 個続けて記述します。
- 文字数が 0 の場合，タイトル欄は空白になります。
- タイトル・ストリングに「(2) 文字セット」にない不当文字が記述された場合は，“！”に置き換えてタイトル欄に出力されます。
- タイトル指定は，コマンド・ライン上でアセンブラ・オプション（-lh）によって指定することができます。

［使用例］

```
$       PROCESSOR ( f1166a0 )
$       TITLE ( 'THIS IS TITLE' )
        NAME    SAMPLE
        CSEG
        MOV     A, B
        END
```

使用例のアセンブル・リストを次に示します（行数は 72 行と指定しています）。

```
78K0R Assembler Vx.xx    THIS IS TITLE    Date:xx xxx xx  Page:1


Command :       -ll72 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn


         Assemble list


ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT


 1       1                              $       PROCESSOR ( f1166a0 )
 2       2                              $       TITLE ( 'THIS IS TITLE' )
 3       3                                      NAME    SAMPLE
 4       4       ----                           CSEG
 5       5       00000   63                     MOV     A, B
 6       6                                      END


Segment information :


ADRS    LEN     NAME


00000   00001H  ?CSEG


Target chip : uPD78F1166_A0
Device file : Vx.xx

Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

SUBTITLE

アセンブル・リストのヘッダのサブタイトル欄に文字列を印字します。

[記述形式]

```
[ ]$[ ]SUBTITLE[ ]([ ]'タイトル・ストリング'[ ])
[ ]$[ ]ST[ ]([ ]'タイトル・ストリング'[ ])          ; 短縮形
```

[省略時解釈]

- SUBTITLE 制御命令は指定されていないものとし，アセンブル・リストのヘッダのサブタイトル欄は空白となります。

[機能]

- アセンブル・リストの各ページ・ヘッダのサブタイトル欄に印字する文字列を指定します。

[用途]

- サブタイトルを各ページに印字することにより，アセンブル・リストの内容をわかりやすくします。
  サブタイトルは，各ページごとに印字する文字列を変更することができます。

[説明]

- SUBTITLE 制御命令は，通常のソースにのみに記述することができます。
- 指定可能な文字列は，72 文字までです。
  73 文字以上記述した場合，先頭の 72 文字が有効です。なお，全角文字は 2 文字，タブは 1 文字として数えます。
- SUBTITLE 制御命令は指定した文字列をその次のページからサブタイトル部に印字します。ただし，ページの先頭行に指定した場合には，そのページから印字します。
- SUBTITLE 制御命令を指定しない場合は，サブタイトル部は空白です。
- 文字列に引用符（'）を記述する場合には，2 個続けて記述してください。
- 文字数が 0 の場合，サブタイトル欄は空白となります。
- 指定された文字列の中に「(2) 文字セット」にない不当文字が記述された場合には，“！”に置き換えてサブタイトル欄に出力します。CR（0DH）を記述した場合は，エラーとなり，リスト上には何も出力されません。00H を記述すると，それ以降，引用符（'）で閉じるまで出力されません。

［使用例］

```
        NAME    SAMP
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 1' )       ; (1)
$       EJECT                                   ; (2)
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 2' )       ; (3)
$       EJECT                                   ; (4)
$       SUBTITLE ( 'THIS IS SUBTITLE 3' )       ; (5)
        END
```

(1) 文字列“THIS IS SUBTITLE 1”を指定します。

(2) 改ページ指示です。

(3) 文字列“THIS IS SUBTITLE 2”を指定します。

(4) 改ページ指示です。

(5) 文字列“THIS IS SUBTITLE 3”を指定します。

使用例のアセンブル・リストを次に示します（行数は 80 行です）。

```
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:1

Command :       -cf1166a0 -ll80 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn
        Assemble list

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 1     1                        NAME SAMP
 2     2    -----                    CSEG
 3     3                     $   SUBTITLE ( 'THIS IS SUBTITLE 1' )   ; (1)
 4     4                     $   EJECT                               ; (2)
------------------------------------------- 改ページ -------------------------
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:2

THIS IS SUBTITLE 1
```

```
ALNO   STNO   ADRS   OBJECT   M I SOURCE STATEMENT

 5      5     -----                  CSEG
 6      6                      $     SUBTITLE ( 'THIS IS SUBTITLE 2' )    ;  (3)
 7      7                      $     EJECT                                ;  (4)
------------------------------------------- 改ページ -------------------------
78K0R Assembler Vx.xx                            Date:xx xxx xx  Page:3

THIS IS SUBTITLE 2

ALNO   STNO   ADRS   OBJECT   M I SOURCE STATEMENT

 8      8                      $     SUBTITLE ( 'THIS IS SUBTITLE 3' )    ;  (5)
 9      9                            END

Segment informations :

ADRS   LEN      NAME

00000 00000H   ?CSEG

Target chip : uPD78F1166_A0
Device file : Vx.xx
Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

FORMFEED

リスト・ファイルの最後にフォーム・フィードを出力します。

[記述形式]

```
[ ]$[ ]FORMFEED
```

[機能]

- FORMFEED 制御命令は，リスト・ファイルの最後にフォーム・フィードを出力することを指示します。

[用途]

- アセンブル・リスト・ファイルの内容を印字したあとで，改ページしておきたい場合に使用します。

[説明]

- FORMFEED 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- アセンブル・リストをプリント・アウトするとき，プリント・アウトが終了しても印字が最終ページの途中だと，最後のページが出てこない場合があります。
  このような場合，FORMFEED 制御命令，またはアセンブラ・オプション（-lf）を使用して，アセンブル・リストの最後に FORMFEED コードを付けてください。
  なお，多くの場合，ファイルの終了により FORMFEED コードが送られるので，最後に FORMFEED コードがあると不要な白紙が 1 ページ送られてしまいます。これを防止するために，NOFORMFEED 制御命令，またはアセンブラ・オプション（-nlf）がデフォルトで設定されます。
- FORMFEED/NOFORMFEED 制御命令を複数指定した場合は，最後に指定した命令が有効となります。
- フォーム・フィードの出力は，コマンド・ライン上でアセンブラ・オプション（-lf/-nlf）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，FORMFEED/NOFORMFEED 制御命令に対する文法チェックは行われます。

NOFORMFEED

リスト・ファイルの最後にフォーム・フィードを出力しません。

[記述形式]

```
[ ]$[ ]NOFORMFEED      ; 省略時解釈
```

[機能]

- NOFORMFEED 制御命令は，リスト・ファイルの最後にフォーム・フィードを出力しないことを指示します。

[用途]

- アセンブル・リスト・ファイルの内容を印字したあとで，改ページしておきたい場合に使用します。

[説明]

- NOFORMFEED 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- アセンブル・リストをプリント・アウトするとき，プリント・アウトが終了しても印字が最終ページの途中だと，最後のページが出てこない場合があります。
  このような場合，FORMFEED 制御命令，またはアセンブラ・オプション（-lf）を使用して，アセンブル・リストの最後に FORMFEED コードを付けてください。
  なお，多くの場合，ファイルの終了により FORMFEED コードが送られるので，最後に FORMFEED コードがあると不要な白紙が 1 ページ送られてしまいます。これを防止するために，NOFORMFEED 制御命令，またはアセンブラ・オプション（-nlf）がデフォルトで設定されます。
- FORMFEED/NOFORMFEED 制御命令を複数指定した場合は，最後に指定した命令が有効となります。
- フォーム・フィードの出力は，コマンド・ライン上でアセンブラ・オプション（-lf/-nlf）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，FORMFEED/NOFORMFEED 制御命令に対する文法チェックは行われます。

WIDTH

リスト・ファイルの 1 行の最大文字数を指示します。

[記述形式]

```
[ ]$[ ]WIDTH[ ]([ ] 文字数 [ ])
```

[省略時解釈]

$WIDTH (132)

[機能]

- WIDTH 制御命令は，リスト・ファイルの 1 行の最大文字数を指示します。
  なお，文字数の指定範囲は，72 ～ 260 です。

[用途]

- 各種リスト・ファイルの 1 行の文字数を変更したい場合に使用します。

[説明]

- WIDTH 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- WIDTH 制御命令が複数指定された場合は，最後に指定した命令が有効となります。
- 行文字数は，コマンド・ライン上でアセンブラ・オプション（-lw）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，WIDTH 制御命令に対する文法チェックは行われます。

LENGTH

リスト・ファイルの 1 ページの行数を指示します。

[記述形式]

```
[ ]$[ ]LENGTH[ ]([ ] 行数 [ ])
```

[省略時解釈]

$LENGTH (66)

[機能]

- LENGTH 制御命令は，リスト・ファイルの 1 ページの行数を指示します。
  なお，行数の指定範囲は，0，および 20 ～ 32767 です。

[用途]

- アセンブル・リスト・ファイルの 1 ページの行数を変更したい場合に使用します。

[説明]

- LENGTH 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- LENGTH 制御命令が複数指定された場合は，最後に指定した命令が有効となります。
- 行数は，コマンド・ライン上でアセンブラ・オプション（-ll）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，LENGTH 制御命令に対する文法チェックは行われます。

TAB

リスト・ファイルのタブの展開文字数を指示します。

[記述形式]

```
[ ]$[ ]TAB[ ]([ ]展開文字数[ ])
```

[省略時解釈]

$TAB (8)

[機能]

- TAB 制御命令は，リスト・ファイルのタブの展開文字数を指示します。
  なお，展開文字数の指定範囲は，0 ～ 8 です。
- TAB 制御命令は，ソース・モジュール中の HT（Horizontal Tabulation）コードを各種リスト上でいくつかのブランク（空白）に置き換えて出力する（タビュレーション処理）ための基本となる文字数を指定します。

[用途]

- TAB 制御命令で，各種リストの 1 行の文字数を少なく指定した場合に，HT コードによるブランクを少なくし文字数を節約します。

[説明]

- TAB 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- TAB 制御命令が複数指定された場合は，最後に指定した命令が有効となります。
- タブの展開文字数は，コマンド・ライン上でアセンブラ・オプション（-lt）によっても指定することができます。
- ソース・モジュール・ファイル中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- アセンブラ・オプション（-np）を指定した場合でも，TAB 制御命令に対する文法チェックは行われます。

4. 3. 7　条件付きアセンブル制御命令

条件付きアセンブル制御命令は，ソース・モジュール中のある一連のステートメントをアセンブルの対象とする／しないを条件付きアセンブルのスイッチ設定により選択するものです。

条件付きアセンブル制御命令を有効に使用すると，ソース・モジュールをほとんど変更せずに，不必要なステートメントを除いたアセンブルを行うことができます。

条件付きアセンブル制御命令には，次のものがあります。

制御命令	概要
IF	アセンブル対象とするソース・ステートメントを限定するための条件を設定
_IF	
ELSEIF	
_ELSEIF	
ELSE	
ENDIF	
SET	IF/ELSEIF 制御命令で指定するスイッチ名に値を与える
RESET	

IF

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]IF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
        :
[ ]$[ ]ELSEIF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，IF 制御命令から ENDIF 制御命令までです。
- IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

- 例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF           ; (2)
          :
        END
```

(1) スイッチ名“SW1”の値が真の場合，text1 の部分がアセンブルされます。
スイッチ名“SW1”の値が偽の場合，text1 の部分はアセンブルされません。
スイッチ名“SW1”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。

(2) 条件付きアセンブル範囲の終了を示します。

- 例 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) スイッチ名“SW1”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”の値が真の場合，text1 の部分をアセンブルし，text2 の部分はアセンブルされません。

(2) (1) のスイッチ名“SW1”の値が偽の場合，text1 の部分はアセンブルされず，text2 の部分がアセンブルされます。

(3) 条件付きアセンブル範囲の終了を示します。

- 例 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) スイッチ名“SW1”，“SW2”，“SW3”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”または“SW2”の値が真の場合，text1 の部分がアセンブルされ，text2，text3，text4 の部分はアセンブルされません。
スイッチ名“SW1”と“SW2”の値がともに偽の場合，text1 の部分はアセンブルされず，(2) 以降の条件付きアセンブルが行われます。

(2) (1) のスイッチ名“SW1”と“SW2”の値がともに偽で，かつスイッチ名“SW3”の値が真の場合，text2 の部分がアセンブルされ，text1，text3，text4 の部分はアセンブルされません。

（3）（1）のスイッチ名“SW1”，“SW2”と，（2）のスイッチ名“SW3”の値がともに偽で，かつスイッチ名“SW4”の値が真の場合，text3 の部分がアセンブルされ，text1，text2，text4 の部分はアセンブルされません。

（4）（1），（2），（3）のスイッチ名“SW1”，“SW2”，“SW3”，“SW4”の値がすべて偽の場合，text4 の部分がアセンブルされ，text1，text2，text3 の部分はアセンブルされません。

（5）条件付きアセンブル範囲の終了を示します。

_IF

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]_IF 条件式
        :
[ ]$[ ]_ELSEIF 条件式
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，_IF 制御命令から ENDIF 制御命令までです。
- _IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

**(1) シンボル名“SYMA”の値は，text0 の部分で EQU，または SET 疑似命令により，定義されています。
シンボル名“SYMA”の値が真（非 0）の場合，text1 の部分がアセンブルされ，text2 はアセンブルされません。**

(2) シンボル名“SYMA”の値が偽（0）で SYMB と SYMC が同じ値をもつ場合，text2 がアセンブルされます。

（3）条件付きアセンブル範囲の終了を示します。

ELSEIF

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]IF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
        :
[ ]$[ ]ELSEIF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，IF/_IF 制御命令から ENDIF 制御命令までです。
- IF/_IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

（1）スイッチ名“SW1”，“SW2”，“SW3”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”または“SW2”の値が真の場合，text1 の部分がアセンブルされ，text2，text3，text4 の部分はアセンブルされません。
スイッチ名“SW1”と“SW2”の値がともに偽の場合，text1 の部分はアセンブルされず，（2）以降の条件付きアセンブルが行われます。

（2）（1）のスイッチ名“SW1”と“SW2”の値がともに偽で，かつスイッチ名“SW3”の値が真の場合，text2 の部分がアセンブルされ，text1，text3，text4 の部分はアセンブルされません。

（3）（1）のスイッチ名“SW1”，“SW2”と，（2）のスイッチ名“SW3”の値がともに偽で，かつスイッチ名“SW4”の値が真の場合，text3 の部分がアセンブルされ，text1，text2，text4 の部分はアセンブルされません。

（4）（1），（2），（3）のスイッチ名“SW1”，“SW2”，“SW3”，“SW4”の値がすべて偽の場合，text4 の部分がアセンブルされ，text1，text2，text3 の部分はアセンブルされません。

（5）条件付きアセンブル範囲の終了を示します。

_ELSEIF

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]_IF 条件式
        :
[ ]$[ ]_ELSEIF 条件式
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，IF/_IF 制御命令から ENDIF 制御命令までです。
- IF/_IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。
- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

**(1) シンボル名“SYMA”の値は，text0 の部分で EQU，または SET 疑似命令により，定義されています。
シンボル名“SYMA”の値が真（非 0）の場合，text1 の部分がアセンブルされ，text2 はアセンブルされません。**

(2) シンボル名“SYMA”の値が偽（0）で SYMB と SYMC が同じ値をもつ場合，text2 がアセンブルされます。

（3）条件付きアセンブル範囲の終了を示します。

ELSE

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]IF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
または[ ]$[ ]_IF 条件式
        :
[ ]$[ ]ELSEIF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
または[ ]$[ ]_ELSEIF 条件式
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，IF/_IF 制御命令から ENDIF 制御命令までです。
- IF/_IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。

- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

- 例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) スイッチ名“SW1”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”の値が真の場合，text1 の部分をアセンブルし，text2 の部分はアセンブルされません。

(2) (1) のスイッチ名“SW1”の値が偽の場合，text1 の部分はアセンブルされず，text2 の部分がアセンブルされます。

(3) 条件付きアセンブル範囲の終了を示します。

- 例 2

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) スイッチ名“SW1”，“SW2”，“SW3”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”または“SW2”の値が真の場合，text1 の部分がアセンブルされ，text2，text3，text4 の部分はアセンブルされません。
スイッチ名“SW1”と“SW2”の値がともに偽の場合，text1 の部分はアセンブルされず，(2) 以降の条件付きアセンブルが行われます。

(2) (1) のスイッチ名“SW1”と“SW2”の値がともに偽で，かつスイッチ名“SW3”の値が真の場合，text2 の部分がアセンブルされ，text1，text3，text4 の部分はアセンブルされません。

(3) (1) のスイッチ名“SW1”，“SW2”と，(2) のスイッチ名“SW3”の値がともに偽で，かつスイッチ名“SW4”の値が真の場合，text3 の部分がアセンブルされ，text1，text2，text4 の部分はアセンブルされません。

(4) (1)，(2)，(3) のスイッチ名“SW1”，“SW2”，“SW3”，“SW4”の値がすべて偽の場合，text4 の部分がアセンブルされ，text1，text2，text3 の部分はアセンブルされません。

(5) 条件付きアセンブル範囲の終了を示します。

ENDIF

アセンブル対象とするソース・ステートメントを限定するための条件を設定します。

[記述形式]

```
[ ]$[ ]IF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
または[ ]$[ ]_IF 条件式
        :
[ ]$[ ]ELSEIF[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
または[ ]$[ ]_ELSEIF 条件式
        :
[ ]$[ ]ELSE
        :
[ ]$[ ]ENDIF
```

[機能]

- アセンブル対象とするソース・ステートメントを限定するための条件を設定します。
  条件付きアセンブルの対象となるソース・ステートメントは，IF/_IF 制御命令から ENDIF 制御命令までです。
- IF/_IF 制御命令は，指定したスイッチ名，あるいは条件式の評価値が真（00H 以外）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  スイッチ名，あるいは条件式の評価値が偽（00H）の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントは，アセンブルされません。
- ELSEIF/_ELSEIF 制御命令は，それ以前に記述してあるすべての条件付きアセンブル制御命令の条件が不成立（評価値が偽）の場合にのみ，条件判定が行われます。
  ELSEIF/_ELSEIF 制御命令で指定するスイッチ名，あるいは条件式の評価値が真の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントがアセンブルされます。そのあとのアセンブル処理は，ENDIF 制御命令の次のステートメントに移ります。
  ELSEIF/ELSEIF の評価値が偽の場合，それ以降，次の条件付きアセンブル制御命令（ELSEIF/_ELSEIF/ELSE/ENDIF）が現れるまでのソース・ステートメントはアセンブルされません。
- ELSE 制御命令については，それ以前に記述したすべての IF/_IF/ELSEIF/_ELSEIF 制御命令の条件が不成立（スイッチ名の値が偽）の場合，ELSE 制御命令以降 ENDIF 制御命令が現れるまでのソース・ステートメントがアセンブルされます。
- ENDIF 制御命令は，条件付きアセンブルの対象となるソース・ステートメントの終了をアセンブラに指示します。

[用途]

- ソース・モジュールを大幅に変更することなく，アセンブル対象となるソース・ステートメントを変更することができます。

- ソース・モジュール中に，プログラム開発中にのみ必要となるデバッグ文などを記述した場合，そのデバッグ文を機械語に変換する／しないを条件付きアセンブルのスイッチ設定により選択することができます。

[説明]

- スイッチ名による条件判断を行う場合には，IF/ELSEIF 制御命令を使用し，条件式による条件判断を行う場合には，_IF/_ELSEIF 制御命令を使用します。
  両方を組み合わせて使用することもできます。つまり，1 つの IF，または _IF と ENDIF の対の中に，ELSEIF/_ELSEIF を組み合わせて使用することができます。
- 条件式には，絶対式を記述します。
- スイッチ名記述上の規則は，シンボル記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（：）で区切ります。
  ただし，1 つのモジュール内で使用できるスイッチ名は，最大 5 つです。
- IF/ELSEIF 制御命令で複数のスイッチ名を指定した場合，そのいずれか 1 つの値が真であれば，条件成立します。
- IF/ELSEIF 制御命令で指定するスイッチ名の値は，SET/RESET 制御命令により設定します。
  したがって，IF/ELSEIF 制御命令で指定するスイッチの値が，前もってソース・モジュール中で SET/RESET 制御命令により設定されていない場合は，RESET されたものとみなされます。
- スイッチ名，または条件式に不適当な記述がある場合，アセンブラはエラーを出力し，条件判断を偽とします。
- この制御命令を記述する場合には，IF，または _IF と ENDIF を対応させてください。
- マクロ・ボディ中に本制御命令が記述され，本体の途中で EXITM の処理を行って，そのレベルのマクロを抜けた場合，IF レベルは，アセンブラによってマクロ・ボディの入口の IF レベルまで強制的に戻されます。この場合，エラーになりません。
- 1 つの IF-ENDIF 制御命令の間に，別の IF-ENDIF 制御命令を記述することをネスティングと呼びます（8 レベルまでのネスティングが可能です）。
- 条件付きアセンブルで，アセンブルをしないステートメントは，オブジェクト・コードは生成されませんが，アセンブル・リストにはそのまま出力されます。出力したくない場合は，$NOCOND 制御命令を使用します。

[使用例]

- 例 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF           ; (2)
          :
        END
```

(1) スイッチ名“SW1”の値が真の場合，text1 の部分がアセンブルされます。
スイッチ名“SW1”の値が偽の場合，text1 の部分はアセンブルされません。
スイッチ名“SW1”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。

(2) 条件付きアセンブル範囲の終了を示します。

- 例 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) スイッチ名“SW1”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”の値が真の場合，text1 の部分をアセンブルし，text2 の部分はアセンブルされません。

(2) (1) のスイッチ名“SW1”の値が偽の場合，text1 の部分はアセンブルされず，text2 の部分がアセンブルされます。

(3) 条件付きアセンブル範囲の終了を示します。

- 例 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) スイッチ名“SW1”，“SW2”，“SW3”の値は，text0 の部分で SET/RESET 制御命令により真／偽に設定されています。
スイッチ名“SW1”または“SW2”の値が真の場合，text1 の部分がアセンブルされ，text2，text3，text4 の部分はアセンブルされません。
スイッチ名“SW1”と“SW2”の値がともに偽の場合，text1 の部分はアセンブルされず，(2) 以降の条件付きアセンブルが行われます。

(2) (1) のスイッチ名“SW1”と“SW2”の値がともに偽で，かつスイッチ名“SW3”の値が真の場合，text2 の部分がアセンブルされ，text1，text3，text4 の部分はアセンブルされません。

(3) (1) のスイッチ名“SW1”，“SW2”と，(2) のスイッチ名“SW3”の値がともに偽で，かつスイッチ名“SW4”の値が真の場合，text3 の部分がアセンブルされ，text1，text2，text4 の部分はアセンブルされません。

(4) (1)，(2)，(3) のスイッチ名“SW1”，“SW2”，“SW3”，“SW4”の値がすべて偽の場合，text4 の部分がアセンブルされ，text1，text2，text3 の部分はアセンブルされません。

(5) 条件付きアセンブル範囲の終了を示します。

- 例 4

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

(1) シンボル名“SYMA”の値は，text0 の部分で EQU，または SET 疑似命令により，定義されています。
シンボル名“SYMA”の値が真（非 0）の場合，text1 の部分がアセンブルされ，text2 はアセンブルされません。

(2) シンボル名“SYMA”の値が偽（0）で SYMB と SYMC が同じ値をもつ場合，text2 がアセンブルされます。

(3) 条件付きアセンブル範囲の終了を示します。

SET

IF/ELSEIF 制御命令で指定するスイッチ名に値を与えます。

[記述形式]

```
[ ]$[ ]SET[ ]([ ]スイッチ名[[ ]:[ ]スイッチ名] … [ ])
```

[機能]

- SET 制御命令は，IF/ELSEIF 制御命令で指定するスイッチ名に値を与えます。
- SET 制御命令で指定したスイッチ名に，真の値（0FFH）を与えます。

[用途]

- IF/ELSEIF 制御命令で指定するスイッチ名に真の値（0FFH）を与えたいときは，SET 制御命令を記述します。

[説明]

- SET 制御命令では，スイッチ名を記述します。
  スイッチ名の記述上の規則は，シンボルの記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- スイッチ名は，予約語，スイッチ名以外のユーザ定義シンボルと重複してもかまいません。
- SET 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（:）で区切ります。ただし，1 つのモジュール内で使用できるスイッチ名は，最大 1000 個です。
- 一度 SET したスイッチ名を RESET することができます。また，一度 RESET したスイッチ名を SET することができます。
- IF/ELSEIF 制御命令で指定するスイッチ名は，その制御命令を記述する以前のソース・モジュール中で，SET/RESET 制御命令により，少なくとも 1 回は定義しなければなりません。
- スイッチ名は，クロスリファレンス・リストには出力されません。

［使用例］

```
$       SET ( SW1 )             ; (1)
          :
$       IF ( SW1 )              ; (2)
                text1
$       ENDIF                   ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )              ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE                    ; (7)
                text4
$       ENDIF                   ; (8)
          :
        END
```

(1) スイッチ名“SW1”に真の値（0FFH）を与えます。

(2) (1) でスイッチ名“SW1”に真の値を与えていますので，text1 の部分がアセンブルされます。

(3) (2) から始まる条件付きアセンブル範囲の終了を示します。

(4) スイッチ名“SW1”と“SW2”に偽の値（00H）を与えます。

(5) スイッチ名“SW1”には (4) で偽の値を与えていますので，text2 の部分はアセンブルされません。

(6) スイッチ名“SW2”にも (4) で偽の値を与えていますので，text3 の部分もアセンブルされません。

(7) (5)，(6) のスイッチ名“SW1”，“SW2”の値がすべて偽のため，text4 の部分がアセンブルされます。

(8) (5) から始まる条件付きアセンブル範囲の終了を示します。

RESET

IF/ELSEIF 制御命令で指定するスイッチ名に値を与えます。

[記述形式]

```
[ ]$[ ]RESET[ ]([ ] スイッチ名 [[ ]:[ ] スイッチ名 ] … [ ])
```

[機能]

- RESET 制御命令は，IF/ELSEIF 制御命令で指定するスイッチ名に値を与えます。
- RESET 制御命令で指定したスイッチ名に，偽の値（00H）を与えます。

[用途]

- IF/ELSEIF 制御命令で指定するスイッチ名に偽の値（00H）を与えたいときは，RESET 制御命令を記述します。

[説明]

- RESET 制御命令では，スイッチ名を記述します。
  スイッチ名の記述上の規則は，シンボルの記述上の規則（「(3) シンボル欄」を参照してください）と同じです。
  ただし，最大認識文字数は，常に 31 文字です。
- スイッチ名は，予約語，スイッチ名以外のユーザ定義シンボルと重複してもかまいません。
- RESET 制御命令で複数のスイッチ名を指定する場合は，各スイッチ名をコロン（:）で区切ります。ただし，1 つのモジュール内で使用できるスイッチ名は，最大 1000 個です。
- 一度 SET したスイッチ名を RESET することができます。また，一度 RESET したスイッチ名を SET することができます。
- IF/ELSEIF 制御命令で指定するスイッチ名は，その制御命令を記述する以前のソース・モジュール中で，SET/RESET 制御命令により，少なくとも 1 回は定義しなければなりません。
- スイッチ名は，クロスリファレンス・リストには出力されません。

［使用例］

```
$       SET ( SW1 )             ; (1)
          :
$       IF ( SW1 )              ; (2)
                text1
$       ENDIF                   ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )              ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE                    ; (7)
                text4
$       ENDIF                   ; (8)
          :
        END
```

(1) スイッチ名“SW1”に真の値（0FFH）を与えます。

(2) (1) でスイッチ名“SW1”に真の値を与えていますので，text1 の部分がアセンブルされます。

(3) (2) から始まる条件付きアセンブル範囲の終了を示します。

(4) スイッチ名“SW1”と“SW2”に偽の値（00H）を与えます。

(5) スイッチ名“SW1”には (4) で偽の値を与えていますので，text2 の部分はアセンブルされません。

(6) スイッチ名“SW2”にも (4) で偽の値を与えていますので，text3 の部分もアセンブルされません。

(7) (5)，(6) のスイッチ名“SW1”，“SW2”の値がすべて偽のため，text4 の部分がアセンブルされます。

(8) (5) から始まる条件付きアセンブル範囲の終了を示します。

4. 3. 8　漢字コード制御命令

コメントに記述された漢字をどの漢字コードで解釈するのかを指定する制御命令です。

漢字コード制御命令には，次のものがあります。

制御命令	概要
KANJICODE	コメントに記述された漢字を指定された漢字コードとして解釈

KANJICODE

コメントに記述された漢字を指定された漢字コードとして解釈します。

[記述形式]

```
[ ]$[ ]KANJICODE[ ] 漢字コード
```

[省略時解釈]

$KANJICODE SJIS

[機能]

- コメントに記述された漢字を指定された漢字コードとして解釈します。
- 漢字コードは，SJIS/EUC/NONE を記述することができます。

  SJIS　　：シフト JIS コードとして解釈します。

  EUC　　：EUC コードとして解釈します。

  NONE　　：漢字として解釈しません。

[用途]

- コメント行の漢字の，漢字コードの解釈を指定するときに使用します。

[説明]

- KANJICODE 制御命令は，モジュール・ヘッダ部のみに記述することができます。
- KANJICODE 制御命令が，モジュール・ヘッダ部に複数回記述された場合は，その中でもっとも後者に記述された命令が優先されます。
- 漢字コード指定は，コマンド・ライン上でアセンブラ・オプション（-zs/-ze/-zn）によって指定することができます。
- ソース・モジュール中とコマンド・ライン上で異なる指定が行われた場合，コマンド・ライン上の指定が優先されます。
- コマンド・ライン上にオプションが指定された場合にも，KANJICODE 制御命令に対する文法チェックは行われます。

4. 3. 9　RAM 領域配置指定制御命令

指定されたセグメント名を持つセグメントを，メモリ領域名“RAM”内に配置する制御命令です。

RAM 領域配置指定制御命令には，次のものがあります。

制御命令	概要
RAM_ALLOCATE	指定されたセグメント名を持つセグメントをメモリ領域名“RAM”内に配置

RAM_ALLOCATE

指定されたセグメント名を持つセグメントを，メモリ領域名“RAM”内に配置します。

[記述形式]

```
[ ]$[ ]RAM_ALLOCATE[ ]([ ] セグメント名 [ ][, … ])[ ][; コメント]
```

[省略時解釈]

- メモリ領域名“ROM”内に配置します。

[機能]

- 指定されたセグメント名を持つセグメントを，メモリ領域名“RAM”内に配置します。

[説明]

- RAM_ALLOCATE 制御命令が，ソース・モジュール・ファイル中に複数回記述された場合は，その中でもっとも後者に記述された命令が優先されます。
- 指定できるセグメントは CSEG のみです。CSEG 以外が指定された場合には，ワーニングを出力し無視します。
- 1 つの RAM 領域配置指定制御命令で複数のセグメントを指定する場合は，各セグメント名を“,”で区切ります。ただし，1 つのモジュール内で指定できるセグメントの数は，256 個までです。

4. 3. 10　その他の制御命令

次に示す制御命令は，C コンパイラなどの上位プログラムが出力する特別な制御命令です。

- $TOL_INF
- $DGS
- $DGL

4.4 マクロ

この章では，マクロ機能の使い方について説明します。
プログラムの中で一連の命令群を何回も記述する場合に使用すると，便利な機能です。

4.4.1 概　　要

ソースの中で一連の命令群を何回も記述する場合，マクロ機能を使用すると便利です。

マクロ機能とは，MACRO，ENDM 疑似命令により，マクロ・ボディとして定義された一連の命令群をマクロ参照している箇所に展開することです。

マクロは，ソースの記述性を向上させるために使用するもので，サブルーチンとは異なります。

マクロとサブルーチンには，それぞれ次のような特長があります。それぞれ目的に応じて有効に使用してください。

(1) サブルーチン

- プログラム中で何回も必要となる処理を１つのサブルーチンとして記述します。サブルーチンは，アセンブラにより一度だけ機械語に変換されます。
- サブルーチンの参照には，サブルーチン・コール命令（一般にはその前後に引数設定の命令が必要）を記述するだけで済みます。
  したがって，サブルーチンを活用することにより，プログラムのメモリを効率よく使用することができます。
- プログラム中の一連のまとまった処理をサブルーチン化することにより，プログラムの構造化を図ることができます（プログラムを構造化することにより，プログラム全体の構造が分かりやすくなり，プログラムの設計が容易になります）。

(2) マクロ

- マクロの基本的な機能は，命令群の置き換えです。
  MACRO，ENDM 疑似命令によりマクロ・ボディとして定義された一連の命令群が，マクロ参照時にその場所に展開されます。アセンブラは，マクロ参照を検出するとマクロ・ボディを展開し，マクロ・ボディの仮引数を参照時の実引数に置き換えながら，命令群を機械語に変換します。
- マクロは，引数を記述することができます。
  たとえば，処理手順は同じであるがオペランドに記述するデータだけが異なる命令群がある場合，そのデータに仮引数を割り当ててマクロを定義します。マクロ参照時には，マクロ名と実引数を記述することにより，記述の一部分だけが異なる種々の命令群に対処することができます。

サブルーチン化の手法が，メモリ・サイズの削減やプログラムの構造化を図るために用いられるのに対し，マクロは，コーディングの効率を向上させるために用いられます。

4. 4. 2　マクロの利用

(1) マクロの定義

マクロの定義は，MACRO，ENDM 疑似命令により行います。

(a) 記述形式

シンボル欄	ニモニック欄	オペランド欄	コメント欄
マクロ名	MACRO	[*仮引数* [, …]]	[; *コメント*]
	:		
	ENDM		[; *コメント*]

(b) 機能

MACRO 疑似命令と ENDM 疑似命令の間に記述された一連の文（マクロ・ボディと呼びます）に対し，シンボル欄で指定したマクロ名を取り付け，マクロの定義を行います。

(c) 使用例

```
ADMAC   MACRO   PARA1, PARA2
        MOV     A, #PARA1
        ADD     A, #PARA2
        ENDM
```

上記の例は，PARA1 と PARA2 の 2 数を加算して，結果を A レジスタに格納する簡単なマクロ定義で，ADMAC というマクロ名が付けられています。PARA1，PARA2 が仮引数です。

詳細については，「4. 2. 8　マクロ疑似命令」を参照してください。

(2) マクロの参照

マクロの参照を行う場合は，すでにマクロ定義されているマクロ名をソースのニモニック欄に記述します。

(a) 記述形式

シンボル欄	ニモニック欄	オペランド欄	コメント欄
[*ラベル* :]	*マクロ名*	[*実引数* [, …]]	[; *コメント*]

(b) 機能

指定したマクロ名に割り付けられたマクロ・ボディを参照します。

(c) 用途

マクロ・ボディを参照するときに，この形式の記述を使用します。

（d）説明

- マクロ名は，参照以前に定義されていなければなりません。
- 実引数はコンマ（，）で区切って，1 行以内であれば最大 16 個まで記述することができます。
- 実引数の文字列中に，空白を記述することはできません。
- 実引数にコンマ（，），セミコロン（；），空白，TAB を記述する場合には，それらを含む文字列をシングル・クォート（’）で囲ってください。
- 仮引数から実引数への置き換えは，それぞれの記述順に対応して，左から順に行われます。仮引数と実引数の数が一致しない場合は，ワーニングが出力されます。

（e）使用例

```
        NAME    SAMPLE
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        CSEG
          :
        ADMAC   10H, 20H
          :
        END
```

すでに定義されているマクロ名“ADMAC”を参照しています。

10H，20H は実引数です。

（3）マクロの展開

アセンブラは，マクロを次のように処理します。

- マクロの参照を見つけると，それに対応するマクロ・ボディをマクロ名の位置に展開します。
- 展開したマクロ・ボディのステートメントをほかのステートメントと同様にアセンブルします。

(4) 使用例

「(2) マクロの参照」で参照されたマクロがアセンブルされ，展開されると，次のようになります。

```
        NAME    SAMPLE

        ; マクロ定義
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        ; ソース本文
        CSEG
          :

        ; マクロの展開
        ADMAC   10H, 20H                ; (a)
        MOV     A, #10H
        ADD     A, #20H
        ; ソース本文
          :
        END
```

(a) マクロの参照により，マクロ・ボディが展開されます。
マクロ・ボディ内の仮引数は，実引数に置き換えられます。

4. 4. 3 マクロ内のシンボル

マクロ内で定義するシンボルには，グローバル・シンボルとローカル・シンボルの 2 種類があります。

(1) グローバル・シンボル

- ソース内のすべてのステートメントから参照することができます。
  したがって，そのシンボルを定義しているマクロを 2 回以上参照し，一連のステートメントが展開されると，シンボルは二重定義エラーとなります。
- LOCAL 疑似命令で定義されていないシンボルは，グローバル・シンボルです。

(2) ローカル・シンボル

- ローカル・シンボルは，LOCAL 疑似命令で定義します(「4. 2. 8 マクロ疑似命令」を参照してください)。
- ローカル・シンボルは，LOCAL 疑似命令で LOCAL 宣言されたマクロ内でのみ参照することができます。
- マクロ外からローカル・シンボルを参照することはできません。

使用例を以下に示します。

```
        NAME    SAMPLE
          ; マクロの定義
MAC1    MACRO
        LOCAL   LLAB            ; (a)
LLAB :                          ; (b)
          :
GLAB :                          ; (c)
        BR      LLAB            ; (d)
        BR      GLAB            ; (e)
        ENDM
          :
          ; ソース本文
REF1 :  MAC1                    ; (f) ←マクロの参照
          :
        BR      LLAB            ; (g) ←エラー

REF2 :  MAC1                    ; (h) ←マクロの参照
          :
GLAB :                          ; (i) ←エラー
          :
        END
```

(a) ラベル LLAB をローカル・シンボルとして宣言しています。

(b) ラベル LLAB をローカル・シンボルとして定義しています。

(c) ラベル GLAB をグローバル・シンボルとして定義しています。

(d) マクロ MAC1 の定義内で，ローカル・シンボル LLAB を参照しています。

(e) マクロ MAC1 の定義内で，グローバル・シンボル GLAB を参照しています。

(f) マクロ MAC1 を参照しています。

(g) マクロ MAC1 の定義外で，ローカル・シンボル LLAB を参照しています。
この記述は，アセンブル時にエラーとなります。

(h) マクロ MAC1 を参照しています。
同一のマクロが 2 回参照されています。

(i) ラベル GLAB をグローバル・シンボルとして定義しています。
同一のラベルが 2 回定義されています。
この記述は，アセンブル時にエラーとなります。

使用例のアセンブル・リストを次に示します。

```
        NAME    SAMPLE
          :
REF1 :  MAC1
          ; マクロの展開
??RA0000 :
          :
GLAB :                          ←エラー
        BR      ??RA0000
        BR      GLAB
          ; ソース本文
          :
        BR      !LLAB           ←エラー
        BR      !GLAB
          :
REF2 :  MAC1
          ; マクロの展開
??RA0001 :
          :
GLAB :                          ←エラー
        BR      ??RA0001
        BR      GLAB
          ; ソース本文
          :
        END
```

グローバル・シンボル GLAB が，マクロ MAC1 内で定義されています。

マクロ MAC1 が 2 回参照されており，一連のステートメントが展開された結果，グローバル・シンボル GLAB は二重定義エラーとなります。

4. 4. 4 マクロ・オペレータ

マクロ・オペレータには，“&”と“ ' ”の 2 種類があります。

(1) &（コンカティネート）

- コンカティネート記号は，マクロ・ボディ内で文字列と文字列を連結します。
  マクロ展開時には，コンカティネート記号の左右の文字列を連結し，コンカティネート自身は消滅します。
- コンカティネート記号は，マクロ定義時にシンボル中の“&”の前後を仮引数，あるいは LOCAL シンボルとして認識することが可能であり，マクロ展開時にシンボル中の“&”の前後の仮引数，あるいは LOCAL シンボルを評価してシンボル中に連結することができます。
- 引用符で囲まれた文字列中の“&”は，単なるデータとして扱われます。
- “&”を 2 つ続けると，1 つの“&”として扱われます。

使用例を以下に示します。

(a) マクロ定義

```
MAC     MACRO   P
LAB&P :                 ←仮引数の P が認識される
        D&B     10H
        DB      'P'
        DB      P
        DB      '&P'
        ENDM
```

(b) マクロ参照

```
        MAC     1H
LAB1H :
        DB      10H     ←D と B が連結されて DB となる
        DB      'P'
        DB      1H
        DB      '&P'    ←引用符中の“&”は単なるデータとして扱われる
```

(2) '(シングル・クォート)

- マクロ参照行，および IRP の実引数の先頭，あるいは，区切り文字のあとに，文字列をシングル・クォートで囲んで記述すると，その文字列がそのまま 1 つの実引数とみなされます。
  実引数に渡されるときには，シングル・クォートが外されて渡されます。
- マクロ・ボディ中にシングル・クォートで囲まれた文字列がある場合には，単なるデータとして扱われます。
- シングル・クォートで囲まれた中にシングル・クォートを使用する場合には，“''”を 2 つ続けて記述します。

使用例を以下に示します。

```
        NAME    SAMP
MAC1    MACRO   P
        IRP     Q, <P>
                MOV     A, #Q
        ENDM
ENDM


        MAC1    '10, 20, 30'
```

このソースをアセンブルすると，MAC1 は次のように展開されます。

```
IRP     Q, <10, 20, 30>
        MOV A, #Q
ENDM
        MOV     A, #10          ; IRP の展開
        MOV     A, #20          ; IRP の展開
        MOV     A, #30          ; IRP の展開
```

4.5 予約語

予約語には，機械語命令，疑似命令，制御命令，演算子，レジスタ名，および sfr シンボルがあります。
予約語は，アセンブラがあらかじめ予約している文字列で，所定の目的以外には転用することができません。
ソースの各欄に記述可能な予約語の種類と予約語一覧を次に示します。

表 4—22 予約語の種類

種類	説明
シンボル欄	すべての予約語が記述不可
ニモニック欄	機械語命令，および疑似命令のみ記述可能
オペランド欄	演算子，sfr シンボル，およびレジスタ名のみ記述可能
コメント欄	すべての予約語が記述可能

表 4—23 予約語一覧

種類	予約語
演算子	AND，BITPOS，DATAPOS，EQ（=），GE（>=），GT（>），HIGH，HIGHW，LE（<=），LOW，LOWW，LT（<），MASK，MIRHW，MIRLW，MOD，NE（< >），NOT，OR，SHL，SHR，XOR
疑似命令	AT，BASE，BASEP，BR，BSEG，CALL，CALLT0，CSEG，DB，DBIT，DG，DS，DSEG，DSPRAM，DW，END，ENDM，ENDS，EQU，EXITM，EXTBIT，EXTRN，FIXED，IHRAM，IRP，IXRAM，LOCAL，LRAM，MACRO，MIRRORP，NAME，OPT_BYTE，ORG，PAGE64KP，PUBLIC，REPT，SADDR，SADDRP，SECUR_ID，SET，UNIT，UNIT64KP，UNITP
制御命令	COND，NOCOND，DEBUG，NODEBUG，DEBUGA [DG]，NODEBUGA [NODG]，EJECT [EJ]，FORMFEED，NOFORMFEED，GEN，NOGEN，IF，_IF，ELSEIF，_ELSEIF，ELSE，ENDIF，INCLUDE [IC]，KANJICODE，LENGTH，LIST [LI]，NOLIST [NOLI]，PROCESSOR [PC]，SET，RESET，SUBTITLE [ST]，SYMLIST，NOSYMLIST，TAB，TITLE [TT]，WIDTH，XREF [XR]，NOXREF [NOXR]
その他	DGL，DGS，SFR，SFRP，TOL_INF

備考 制御命令の [ ] 内は，短縮形を表します。

なお，sfr 一覧については，各デバイスのユーザーズ・マニュアルを参照してください。
割り込み要求ソース一覧，機械語命令，レジスタ名一覧については，各デバイスのユーザーズ・マニュアルを参照してください。

4.6 インストラクション

この節では，RL78 ファミリ，78K0R マイクロコントローラ製品の持つ各種命令機能を説明します。

注意 各命令の詳細な動作，および機械語（命令コード）については，「RL78 ファミリ ユーザーズ・マニュアル ソフトウェア編」／「78K0R マイクロコントローラ ユーザーズ・マニュアル 命令編」を参照してください。また，各デバイス製品のユーザーズ・マニュアルを参照してください。

4.6.1 78K0 マイクロコントローラとの違い

この項では，アセンブラ・ユーザにおける 78K0 マイクロコントローラとの違いについてを説明します。

- パイプライン化により全命令の処理クロック数が短くなっています。既存のプログラムは再評価が必要です。
- 命令コード・マップは，すべて変更になっています。再度アセンブラにて再アセンブルしてください。この際，コード・サイズ増加が予想されますが，新規命令が追加されていますので，新規命令と組み換えることで以前よりコード・サイズを小さくできる場合もあります。
- メモリ空間が 64 K バイトから 1 M バイトに変わり，積まれるスタック量が増えています。アセンブラのプログラム内で，スタック・ポインタ中の RAM 内を操作している場合はアドレス変更が必要です。スタック・サイズは多重 CALL，多重割り込みの深さによって，若干多めに設定してください。
- CALLT テーブルのアドレスが 0040H-007FH から 0080H-00BFH に変更になっています。CALLT テーブルのアドレスを変更してください。
- 78K0 マイクロコントローラのバンク切り替えを使用したプログラムにおいては，アセンブラ・プログラムを作り変える必要があります。
- 拡張 RAM を使用している場合はアドレスが変更になっています。アドレスを変更してください。
- 拡張 RAM からの命令実行をしている場合は，メモリ空間アドレスが変わったことにより BR !addr16 を使用している場合は BR !!addr20 に，CALL !addr16 を使用している場合は CALL !!addr20 に変更してください。
- IMS, IXS レジスタ（メモリ空間を設定するレジスタ）はありません。外部メモリが使用されていない場合は，それらのレジスタを使用したプログラムを削除してください。外部メモリを使用している場合は MM/MEM レジスタ（外部メモリ設定レジスタ）の仕様が変更となっていますので，各製品のユーザーズ・マニュアルを参照し，設定を変更してください。
- 次の命令は削除され，置き換えのコードが出力されますので，コード・サイズが大きくなります。命令をそのまま使っていても，-compati オプションの指定でアセンブラをかけることにより自動的に置き換えは行なわれます。

命令	オペランド	備考
DIVUW	C	置き換え命令は，シフトしながら除算を行いますので，実行時間が長くなります。 シフト命令が追加されていますので，命令の変更を推奨します。
ROR4	[HL]	置き換え命令は，実行時間が長くなります。 シフト命令が追加されていますので，命令の変更を推奨します。
ROL4	[HL]	置き換え命令は，実行時間が長くなります。 シフト命令が追加されていますので，命令の変更を推奨します。
ADJBA	なし	置き換え命令は，実行時間が長くなります。代わりの追加命令はありません。

命令	オペランド	備考
ADJBS	なし	置き換え命令は，実行時間が長くなります。代わりの追加命令はありません。
CALLF	!addr11	3 バイト命令の CALL !addr16 に自動的に変更されます。 そのまま使用しても問題ありません。
DBNZ	B, $addr16 C, $addr16 saddr, $addr16	2 命令に分割され，DEC B / DEC C/ DEC saddr と BNZ $addr20 になります。 そのまま使用しても問題ありません。

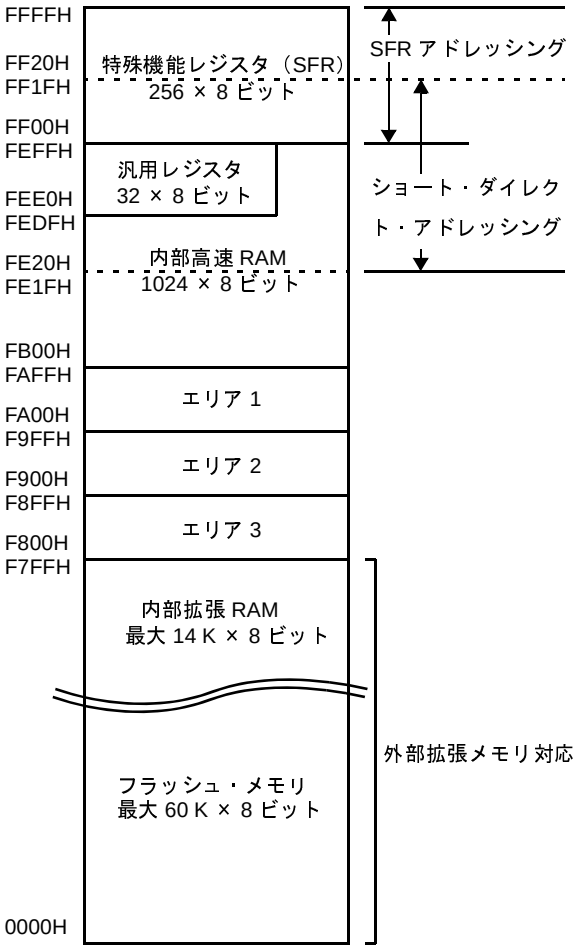
4. 6. 2　メモリ空間

（1）メモリ空間

78K0 マイクロコントローラのメモリ空間は 64 K バイトのみでしたが，RL78 ファミリ，78K0R マイクロコントローラのメモリ空間は 1M バイトに拡張されています。

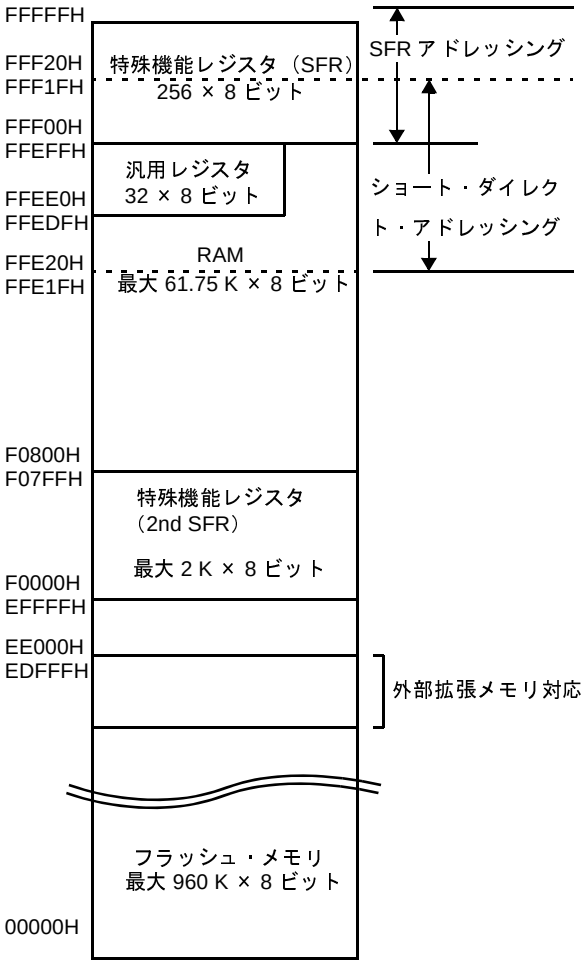
図 4—8　78K0 マイクロコントローラと RL78 ファミリ，78K0R マイクロコントローラのメモリ・マップ

【78K0 マイクロコントローラ】



- プログラム・メモリ空間は最大 60 K バイト
- 内部高速 RAM は最大 1 K バイト（スタック可能）
  内部拡張 RAM は最大 14 K バイト（フェッチ可能）
- エリア 1，エリア 2，エリア 3 は，F800H-FAFFH 固定
- 外部拡張メモリにも対応

【RL78 ファミリ，78K0R マイクロコントローラ】



- プログラム・メモリ空間は最大 960 K バイト
- RAM 空間として最大 61.75 K バイト（スタック可能，フェッチ可能）
- 2nd SFR と名称が変更になり，F0000H-F07FFH の最大 2 K バイト
- 外部拡張メモリにも対応
  外部拡張メモリ空間は製品実装フラッシュ・メモリ上から EDFFFH までに配置

（2）内部プログラム・メモリ空間

RL78 ファミリ，78K0R マイクロコントローラでは，00000H-EFFFFH がプログラム・メモリ空間のアドレスとなります。

内部 ROM（フラッシュ・メモリ）容量の最大値については各製品のユーザーズ・マニュアルを参照してください。

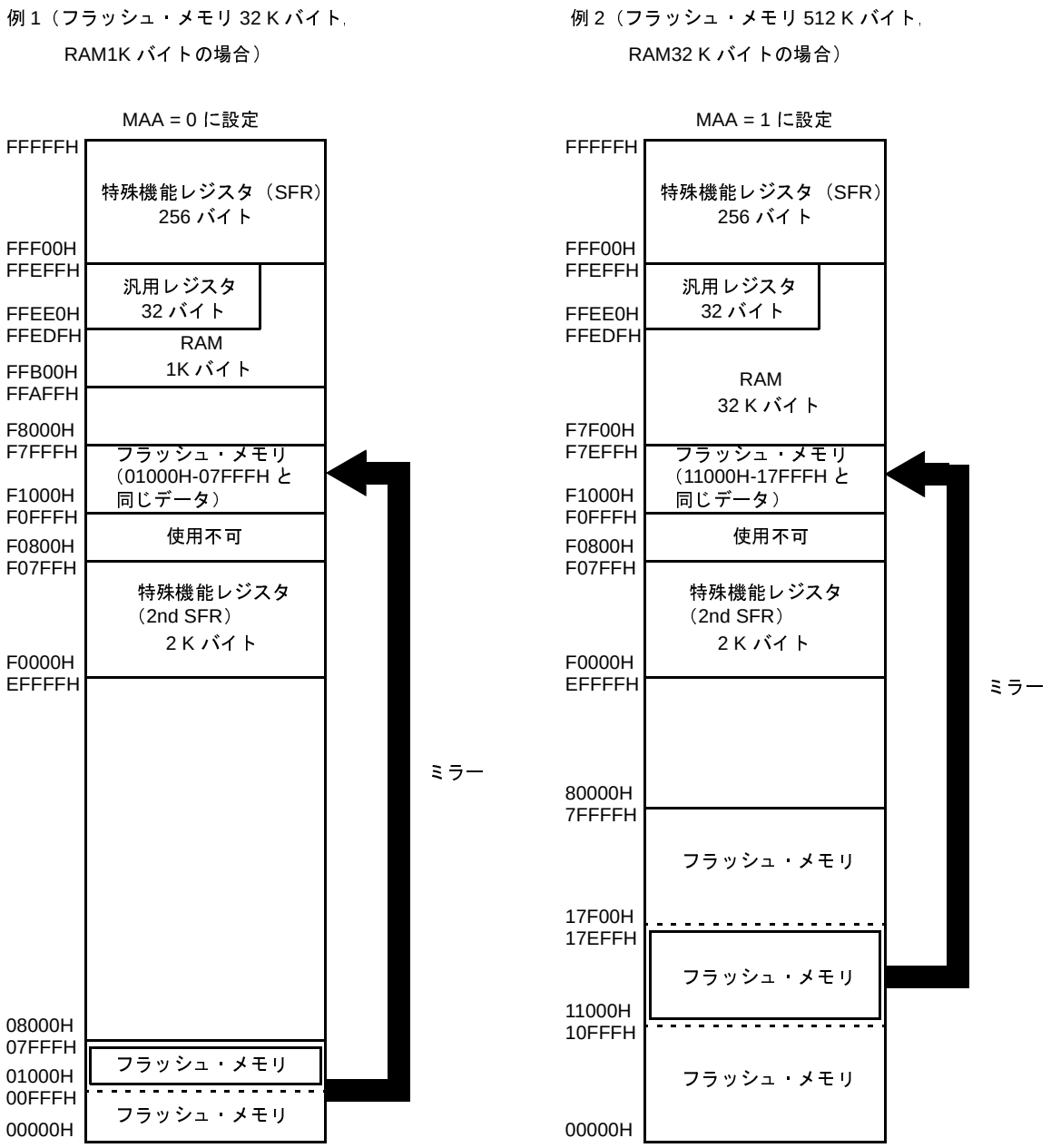
（a）ミラー領域

RL78 ファミリ，78K0R マイクロコントローラでは，00000H-0FFFFH（MAA = 0 のとき），または 10000H-1FFFFH（MAA = 1 のとき）のデータ・フラッシュ・エリアを F0000H-FFFFFH へミラーさせています。F0000H-FFFFFH のデータを読み込むことにより，短いコードでデータ・フラッシュ内容の読み出しを行うことができます。ただし，SFR，拡張 SFR（2nd SFR），RAM，使用不可領域にはミラーされません。

ミラー領域は読み出しのみ可能で，命令フェッチはできません。

次に例を示します。製品によって異なりますので，各製品のユーザーズ・マニュアルを参照してください。

図 4—9　ミラー領域の例



備考　MAA：プロセッサ・モード・コントロール・レジスタ（PMC）のビット 0（詳細は，「(a) プロセッサ・モード・コントロール・レジスタ（PMC）」参照）

(b) ベクタ・テーブル領域

RL78 ファミリ，78K0R マイクロコントローラの製品は，0000H-007FH の 128 バイトの領域がベクタ・テーブル領域として予約されています。割り込み本数は最大 61 本＋ RESET ベクタ＋オンチップ・ディバク・ベクタ＋ソフトウエア・ブレイク・ベクタとなります。また，ベクタ・コードを 2 バイトしか持たないため，割り込みの飛び先アドレスは 00000H-0FFFFH の 64 K アドレスとなります。0040H-007FH のアドレスは，78K0 マイクロコントローラでは CALLT テーブルですが，RL78 ファミリ，78K0R マイクロコントローラではベクタ・アドレスに変更になっています。

（c）CALLT 命令テーブル領域

RL78 ファミリ，78K0R マイクロコントローラの製品は，0080H-00BFH の 64 バイトの領域は CALLT 命令テーブル領域として予約されています。

78K0 マイクロコントローラでは 1 バイト・コール命令ですが，RL78 ファミリ，78K0R マイクロコントローラの製品では 2 バイト・コール命令となります。またアドレスも変更になっています。

また，アドレス・コードを 2 バイトしか持たないため，割り込みの飛び先アドレスは 00000H-0FFFFH の 64 K アドレスとなります。

（3）内部データ・メモリ（内部 RAM）空間

78K0 マイクロコントローラの製品では内部高速 RAM と内部拡張 RAM を持ち，内部高速 RAM はスタック可能，内部拡張 RAM はフェッチ可能と使い分けています。RL78 ファミリ，78K0R マイクロコントローラでは RAM 領域を 1 つにまとめ，同一領域でスタックとフェッチを可能としました。

アドレスは，上限を FFEFFH で固定し，製品搭載の RAM サイズにあわせ下に伸ばしていきます。最大サイズは 61.75 K バイトとします。下限アドレスは，各製品のユーザーズ・マニュアルを参照してください。

FFEE0H-FFEFFH の汎用レジスタ領域，saddr 空間は 78K0 マイクロコントローラと同じアドレスになります。スタック領域は搭載 RAM 内のどこでも指定可能です。

（4）特殊機能レジスタ（SFR：Special Function Register）領域

SFR は，汎用レジスタとは異なり，それぞれ特別な機能を持つレジスタです。

SFR 空間は，FFF00H-FFFFFH の領域に割り付けられています。

SFR については 78K0 マイクロコントローラと同じ仕様となりますが，78K0 マイクロコントローラでアドレス固定であったレジスタについては変更があります。詳細については各製品のユーザーズ・マニュアルを参照してください。

（5）拡張特殊機能レジスタ（2nd SFR：2nd Special Function Register）領域

拡張 SFR（2nd SFR）は，汎用レジスタとは異なり，それぞれ特別な機能を持つレジスタです。

拡張 SFR 空間は，F0000H-F07FFH の領域です。SFR 領域（FFF00H-FFFFFH）以外の SFR が割り付けられています。ただし，拡張 SFR 領域のアクセス命令は SFR 領域より 1 バイト長くなります。

（6）外部メモリ空間

メモリ拡張モード・レジスタの設定によりアクセスが可能な外部メモリ空間です。メモリ空間はフラッシュ・メモリの上から EDFFFH の空間となります。

外部端子として，セパレート・バス・モード時はアドレス A19-A0，D7-D0 の 28 本，マルチプレクスト・バス・モード時は A19-A8，AD7-AD0 の 20 本を持ちます。

外部メモリ使用時の端子設定については，各製品のユーザーズ・マニュアルのポート機能の章を参照してください。

注意　外部メモリ空間でのフェッチを行う場合は，フラッシュまたは RAM メモリ上の分岐命令（CALL, BR）で開始し，外部メモリ空間でのリターン命令（RET, RETB, RETI）で終了してください。フラッシュ・メモリと外部メモリ空間は連続した空間にありますが，連続したプログラムは実行できません。

4. 6. 3 レジスタ

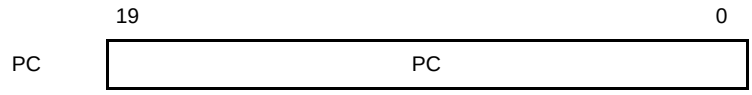
(1) 制御レジスタ

プログラム・シーケンス，ステータス，スタック・メモリの制御など専用の機能を持ったレジスタです。制御レジスタには，プログラム・カウンタ，プログラム・ステータス・ワード，スタック・ポインタがあります。

(a) プログラム・カウンタ（PC）

プログラム・カウンタは，次に実行するプログラムのアドレス情報を保持する 20 ビット・レジスタです。

図 4—10 プログラム・カウンタの構成



(b) プログラム・ステータス・ワード（PSW）

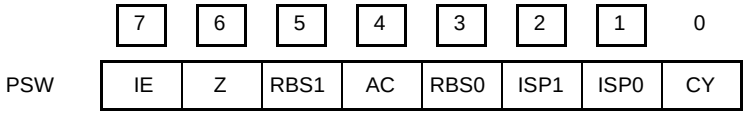
プログラム・ステータス・ワードは，命令の実行によってセット，リセットされる各種フラグで構成される 8 ビット・レジスタです。

割り込みレベルが 4 レベル対応の製品では，ビット 2 に ISP1 フラグが追加されます。

プログラム・ステータス・ワードの内容は，割り込み要求発生時および PUSH PSW 命令の実行時に自動的にスタックされ，RETB，RETI 命令および POP PSW 命令の実行時に自動的に復帰されます。

リセット信号の発生により，06H になります。

図 4—11 プログラム・ステータス・ワードの構成



- 割り込み許可フラグ（IE）

  CPU の割り込み要求受け付け動作を制御するフラグです。

  IE = 0 のときは割り込み禁止（DI）状態となり，ノンマスカブル割り込み以外の割り込みはすべて禁止されます。

  IE = 1 のときは割り込み許可（EI）状態となります。このときの割り込み要求の受け付けは，各割り込み要因に対する割り込みマスク・フラグおよび優先順位指定フラグにより制御されます。

  このフラグは，DI 命令実行または割り込みの受け付けでリセット（0）され，EI 命令実行によりセット（1）されます。

- ゼロ・フラグ（Z）

  演算結果がゼロのときセット（1）され，それ以外のときにリセット（0）されるフラグです。

- レジスタ・バンク選択フラグ（RBS0，RBS1）

  4 個のレジスタ・バンクのうちの 1 つを選択する 2 ビットのフラグです。

SBL RBn 命令の実行によって選択されたレジスタ・バンクを示す 2 ビットの情報が格納されています。

- 補助キャリー・フラグ（AC）

演算結果が，ビット 3 からキャリーがあったとき，またはビット 3 へのボローがあったときセット（1）され，それ以外のときリセット（0）されるフラグです。

- インサービス・プライオリティ・フラグ（ISP0, ISP1）

受け付け可能なマスカブル・ベクタ割り込みの優先順位レベルを管理するフラグです。優先順位指定フラグ・レジスタ（PR）で ISP0, ISP1 の値より低位に指定されたベクタ割り込み要求は受け付け禁止となります。なお，実際に割り込み要求が受け付けられるかどうかは，割り込み許可フラグ（IE）の状態により制御されます。

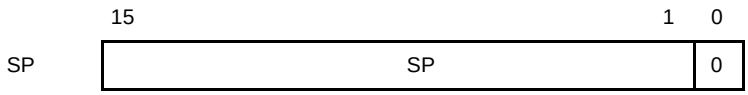
- キャリー・フラグ（CY）

加減算命令実行時のオーバフロー，アンダフローを記憶するフラグです。また，ローテート命令実行時はシフト・アウトされた値を記憶し，ビット演算命令実行時には，ビット・アキュームレータとして機能します。

（c）スタック・ポインタ（SP）

メモリのスタック領域の先頭アドレスを保持する 16 ビットのレジスタです。スタック領域としては内部 RAM 領域を設定可能です。

図 4—12 スタック・ポインタの構成



スタック・メモリへの書き込み（退避）動作に先立ってデクリメントされ，スタック・メモリからの読み取り（復帰）動作のあとインクリメントされます。

SP の内容は $\overline{\text{RESET}}$ 入力により，不定になりますので，必ず命令実行前にイニシャライズしてください。また SP のアドレスは必ず偶数アドレスを指定してください。奇数アドレスを指定した場合は，最下位ビットは 0 に固定されます。

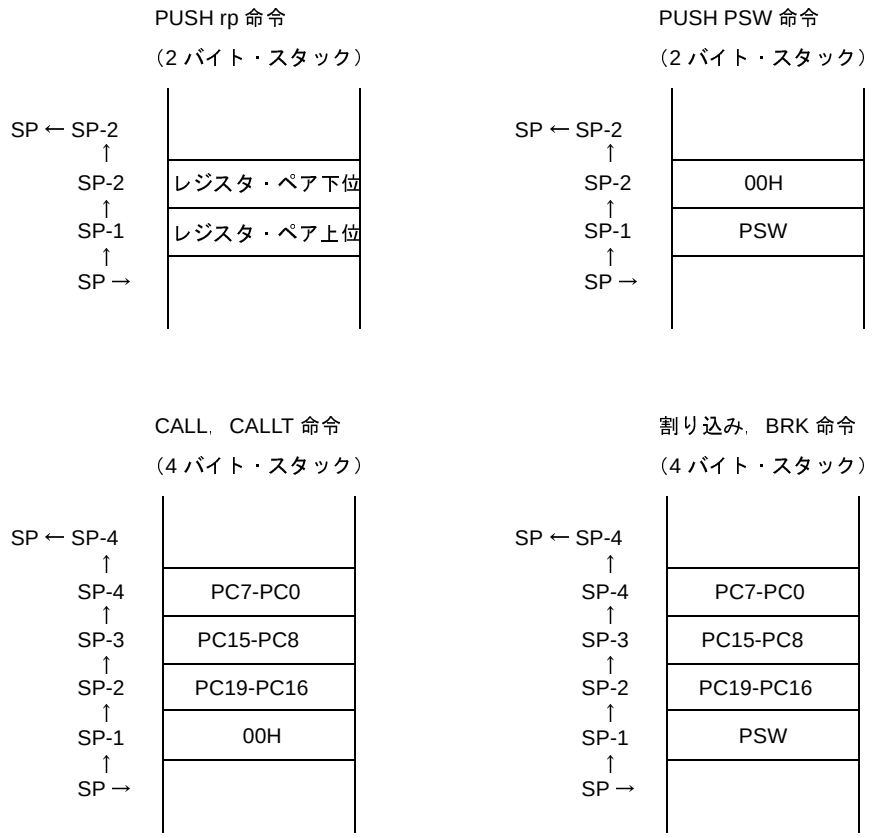
RL78 ファミリ，78K0R マイクロコントローラの製品では，メモリ空間が拡張されていますので，CALL，割り込み時のスタック・アドレスは 1 バイト増え，さらにスタックする RAM が 16 ビット幅のため 2 バイト／4 バイトのスタック・サイズになっています（「表 4—24 78K0 マイクロコントローラと 78K0R マイクロコントローラのスタック・サイズの違い」参照）。

表 4—24 78K0 マイクロコントローラと 78K0R マイクロコントローラのスタック・サイズの違い

退避命令	復帰命令	78K0 マイクロコントローラのスタック・サイズ	RL78 ファミリ，78K0R マイクロコントローラのスタック・サイズ
PUSH rp	POP rp	2 バイト	2 バイト
PUSH PSW	POP PSW	1 バイト	2 バイト
CALL, CALLT	RET	2 バイト	4 バイト
割り込み	RETI	3 バイト	4 バイト
BRK	RETB	3 バイト	4 バイト

RL78 ファミリ，78K0R マイクロコントローラでは，各スタック動作によって退避されるデータは次の図のようになります。

図 4—13 スタック・メモリへ退避されるデータ



スタック・ポインタは内部 RAM のみ指定可能です。ただしアドレスは F0000H-FFFFFH の空間を指定可能ですので，内部 RAM の空間を越えないようにしてください。内部 RAM 空間以外を指定した場合，書き込みは無視され，読み出すと不定値が読み出されます。

（2）汎用レジスタ

内蔵される汎用レジスタは，RAM の FFEE0H-FFEFFH に割り当てられており，8 ビット・レジスタ 8 個（X，A，C，B，E，D，L，H）を 1 バンクとして 4 つのバンクで構成されています。命令実行時に使用するバンクは CPU 制御命令（SEL RBn）によって設定します。

各レジスタは，それぞれ 8 ビット・レジスタとして使用できるほか，2 個の 8 ビット・レジスタをペアとして 16 ビット・レジスタとしても使用できます。

またプログラム記述では，機能名称（X，A，C，B，E，D，L，H，AX，BC，DE，HL）のほか，絶対名称（R0-R7，RP0-RP3）ででも記述できます。

注意 汎用レジスタ（FFEE0H-FFEFFH）の空間は，命令フェッチやスタック領域としての使用を禁止します。

表 4—25 汎用レジスター覧（78K0 マイクロコントローラと同様）

バンク名	レジスタ				絶対アドレス
	機能名称		絶対名称		
	16 ビット処理	8 ビット処理	16 ビット処理	8 ビット処理	
BANK0	HL	H	RP3	R7	FFEFFH
		L		R6	FFEFEH
	DE	D	RP2	R5	FFEFDH
		E		R4	FFEFCH
	BC	B	RP1	R3	FFEFBH
		C		R2	FFEFAH
	AX	A	RP0	R1	FFEF9H
		X		R0	FFEF8H
BANK1	HL	H	RP3	R7	FFEF7H
		L		R6	FFEF6H
	DE	D	RP2	R5	FFEF5H
		E		R4	FFEF4H
	BC	B	RP1	R3	FFEF3H
		C		R2	FFEF2H
	AX	A	RP0	R1	FFEF1H
		X		R0	FFEF0H

バンク名	レジスタ				絶対アドレス
	機能名称		絶対名称		
	16 ビット処理	8 ビット処理	16 ビット処理	8 ビット処理	
BANK2	HL	H	RP3	R7	FFEEFH
		L		R6	FFEEEH
	DE	D	RP2	R5	FFEEDH
		E		R4	FFEECH
	BC	B	RP1	R3	FFEEBH
		C		R2	FFEEAH
	AX	A	RP0	R1	FFEE9H
		X		R0	FFEF8H
BANK3	HL	H	RP3	R7	FFEE7H
		L		R6	FFEE6H
	DE	D	RP2	R5	FFEE5H
		E		R4	FFEE4H
	BC	B	RP1	R3	FFEE3H
		C		R2	FFEE2H
	AX	A	RP0	R1	FFEE1H
		X		R0	FFEE0H

(3) ES，CS レジスタ

RL78 ファミリ，78K0R マイクロコントローラでは ES，CS レジスタが追加されています。ES レジスタにてデータ・アクセス，CS レジスタにて分岐命令実行時の上位アドレスを指定できます。使用方法は，ES レジスタについては「(2) 処理データ・アドレスに対するアドレッシング」，CS レジスタについては「(1) 命令アドレスのアドレッシング」を参照してください。

ES のリセット後の初期値は 0FH，CS のリセット後の初期値は 00H です。

図 4—14　ES，CS レジスタの構成

	7	6	5	4	3	2	1	0
ES	0	0	0	0	ES3	ES2	ES1	ES0

	7	6	5	4	3	2	1	0
CS	0	0	0	0	CS3	CP2	CP1	CP0

（4）特殊機能レジスタ（SFR）

RL78 ファミリ，78K0R マイクロコントローラでのアドレス固定の SFR を示します。

表 4—26 固定 SFR レジスター覧

アドレス	レジスタ名称
FFFF8H	SPL
FFFF9H	SPH
FFFFAH	PSW
FFFFBH	Reserve
FFFFCH	CS
FFFFDH	ES
FFFFEH	PMC
FFFFFH	MEM

（a）プロセッサ・モード・コントロール・レジスタ（PMC）

プロセッサのモードを制御する 8 ビット・レジスタです。詳細は「(2) 内部プログラム・メモリ空間」を参照してください。

リセット時の初期値は 00H になります。

図 4—15 プロセッサ・モード・コントロール・レジスタの構成

アドレス：FFFFEH　リセット時：00H　R/W

略号	7	6	5	4	3	2	1	0
PMC	0	0	0	0	0	0	0	MAA

MAA	F0000H-FFFFFH 注へミラーするフラッシュ・メモリ空間選択
0	00000H-0FFFFH を F0000H-FFFFFH へミラー
1	10000H-1FFFFH を F0000H-FFFFFH へミラー

注　F0000H-FFFFFH の中には SFR, RAM 領域があり，この重なった領域は SFR, RAM が優先されます。

注意 1. PMC の設定は，初期設定で 1 度だけ行ってください。初期設定以外での PMC の書き替えは禁止です。

2. PMC の設定後，1 命令以上空けてミラー領域にアクセスしてください。

4. 6. 4 アドレッシング

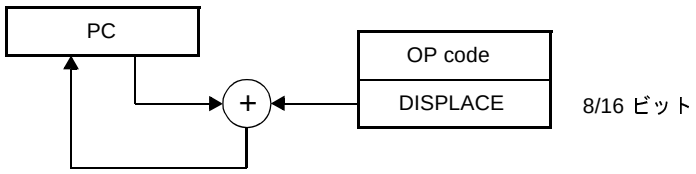
アドレッシングは，処理データ・アドレスに対するアドレッシング，プログラム・アドレスに対するアドレッシング，の 2 種類から構成されています。次にそれぞれのアドレッシング・モードを示します。

(1) 命令アドレスのアドレッシング

(a) レラティブ・アドレッシング

プログラム・カウンタ（PC）の値（次に続く命令の先頭アドレス）に対し，命令語に含まれるディスプレースメント値（符号付きの補数データ：-128 ～ +127 または -32768 ～ +32767）を加算した結果をプログラム・カウンタ（PC）に格納し分岐先プログラム・アドレスを指定するアドレッシングです。レラティブ・アドレッシングは分岐命令のみに適用されます。

図 4—16 レラティブ・アドレッシングの概略



(b) イミーディエト・アドレッシング

命令語中のイミーディエト・データをプログラム・カウンタに格納し，分岐先プログラム・アドレスを指定するアドレッシングです。

イミーディエト・アドレッシングには 20 ビットのアドレスを指定する CALL !!addr20 / BR !!addr20 と，16 ビットのアドレスを指定する CALL !addr16 / BR !addr16 があります。16 ビット・アドレスを指定する場合は上位 4 ビットには 0000 が入ります。

図 4—17 CALL !!addr20/BR !!addr20 の例

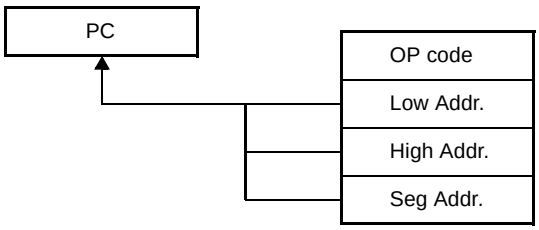
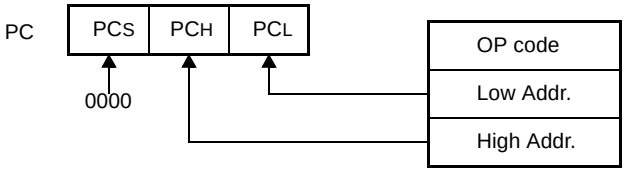


図 4—18 CALL !addr16/BR !addr16 の例

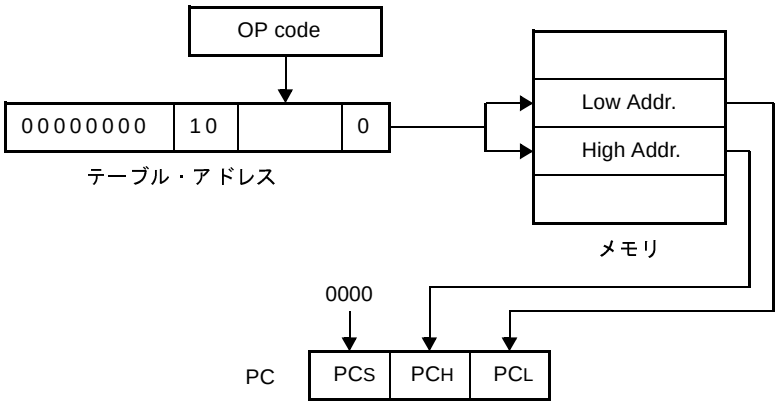


(c) テーブル・インダイレクト・アドレッシング

命令語中の 5 ビット・イミーディエト・データにより CALLT テーブル領域（0080H-00BFH）内のテーブル・アドレスを指定し，その内容とそれに続くアドレスの内容を 16 ビット・データとしてプログラム・カウンタ（PC）に格納し，プログラム・アドレスを指定するアドレッシングです。テーブル・インダイレクト・アドレッシングは CALLT 命令にのみ適用されます。

RL78 ファミリ，78K0R マイクロコントローラでは，00000H-0FFFFH の 64 K バイト空間のみ分岐可能です。

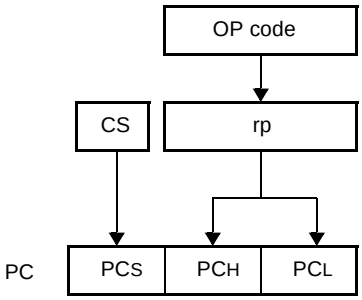
図 4—19 テーブル・インダイレクト・アドレッシングの概略



(d) レジスタ・ダイレクト・アドレッシング

命令語で指定されるカレント・レジスタ・バンク内の汎用レジスタ・ペア（AX/BC/DE/HL）と CS レジスタの内容を 20 ビット・データとしてプログラム・カウンタ（PC）に格納し，プログラム・アドレスを指定するアドレッシングです。レジスタ・ダイレクト・アドレッシングは CALL AX / BC / DE / HL と BR AX 命令にのみ適用されます。

図 4—20 レジスタ・ダイレクト・アドレッシングの概略



(2) 処理データ・アドレスに対するアドレッシング

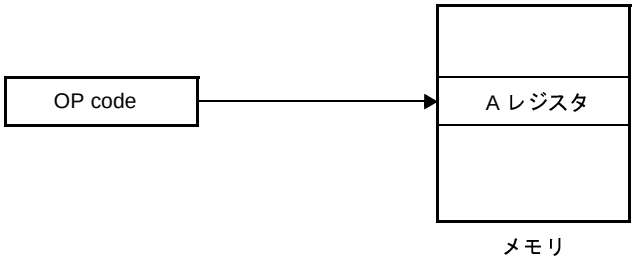
(a) インプライド・アドレッシング

アキュムレータなどの特別な機能を与えられたレジスタをアクセスする命令は，命令語中にはレジスタ指定フィールドを持たず命令語で直接指定します。

命令により自動的に使用できるため特定のオペランド形式を持ちません。

インプライド・アドレッシングは MULU X のみに適用されます。

図 4—21 インプライド・アドレッシングの概略



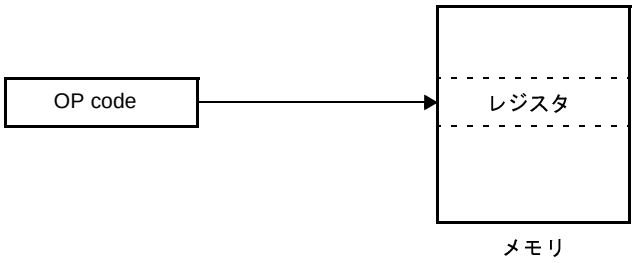
(b) レジスタ・アドレッシング

汎用レジスタをオペランドとしてアクセスするアドレッシングです。8 ビット・レジスタを指定する場合は命令語の 3 ビット，16 ビット・レジスタを指定する場合は命令語の 2 ビットによりレジスタが選択されます。

オペランド形式を以下に示します。

表現形式	記述方法
r	X，A，C，B，E，D，L，H
rp	AX，BC，DE，HL

図 4—22 レジスタ・アドレッシングの概略



(c) ダイレクト・アドレッシング

命令語中のイミーディエト・データがオペランド・アドレスとなり，対象となるアドレスを直接指定するアドレッシングです。

オペランド形式を以下に示します。

表現形式	記述方法
ADDR16	ラベルまたは 16 ビット・イミーディエト・データ （F0000H-FFFFFH 空間のみ指定可能）
ES:ADDR16	ラベルまたは 16 ビット・イミーディエト・データ （ES レジスタにて上位 4 ビット・アドレス指定）

図 4—23 ADDR16 の例

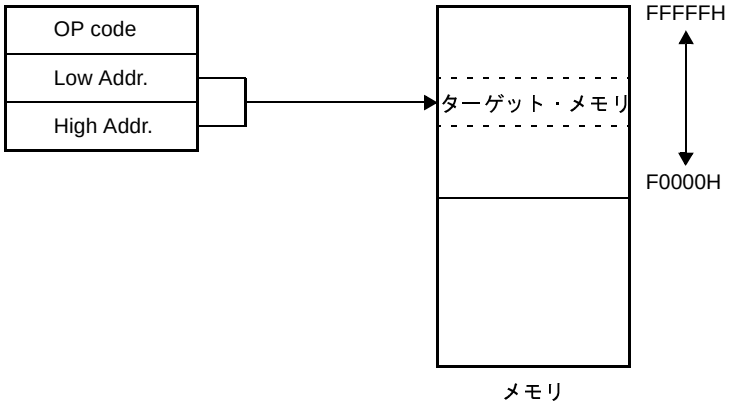
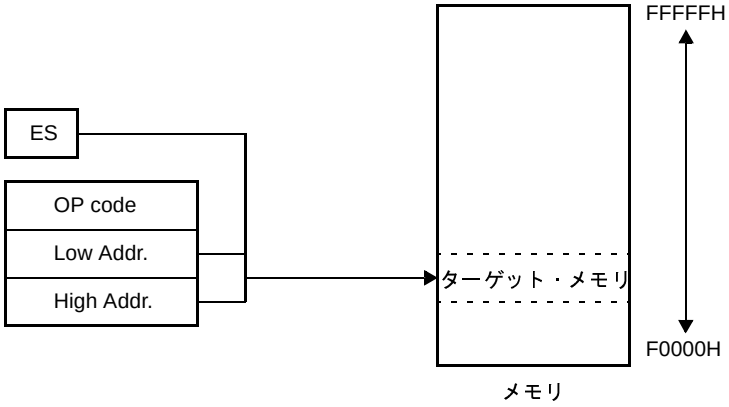


図 4—24 ES:ADDR16 の例



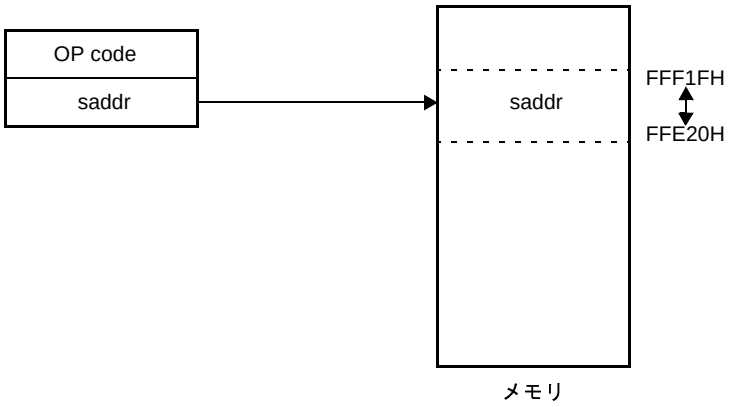
（d）ショート・ダイレクト・アドレッシング

命令語中の 8 ビット・データで対象となるアドレスを直接指定するアドレッシングです。このアドレッシングが適用されるのは FFE20H-FFF1FH の空間に限られます。

オペランド形式を以下に示します。

表現形式	記述方法
SADDR	ラベルまたは FFE20H-FFF1FH のイミーディエト・データまたは 0FE20H-0FF1FH のイミーディエト・データ （FFE20H-FFF1FH 空間のみ指定可能）
SADDRP	ラベルまたは FFE20H-FFF1FH のイミーディエト・データまたは 0FE20H-0FF1FH のイミーディエト・データ（偶数アドレスのみ） （FFE20H-FFF1FH 空間のみ指定可能）

図 4—25 ショート・ダイレクト・アドレッシングの概略



備考 SADDR，SADDRP は，（実アドレスの上位 4 ビット・アドレスを省略した）16 ビットのイミーディエト・データで FE20H-FF1FH の値を記述することができます。また，20 ビットのイミーディエト・データで FFE20H-FFF1FH の値を記述することもできます。
ただし，どちらの形式で書いても，メモリは FFE20H-FFF1FH 空間のアドレスが指定されます。

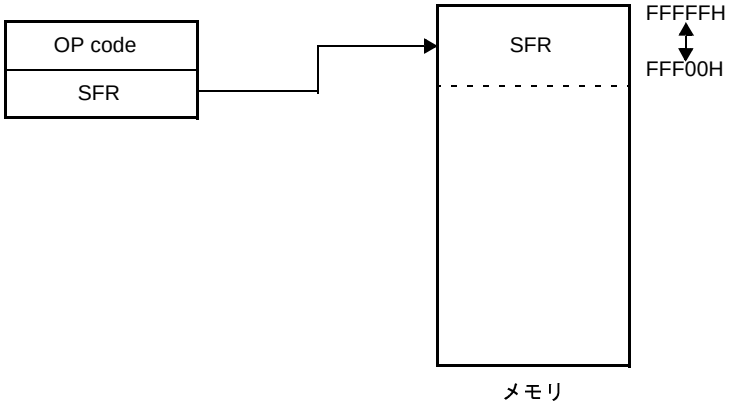
（e）SFR アドレッシング

命令語中の 8 ビット・データで対象となる SFR アドレスを直接指定するアドレッシングです。このアドレッシングが適用されるのは FFF00H-FFFFFH の空間に限られます。

オペランド形式を以下に示します。

表現形式	記述方法
SFR	SFR レジスタ名
SFRP	16 ビット操作可能な SFR レジスタ名（偶数アドレスのみ）

図 4—26 SFR アドレッシングの概略



(f) レジスタ・インダイレクト・アドレッシング

命令語で指定されたレジスタ・ペアの内容がオペランド・アドレスになり，対象となるアドレスを指定するアドレッシングです。

オペランド形式を以下に示します。

表現形式	記述方法
—	[DE]，[HL] （F0000H-FFFFFH 空間のみ指定可能）
—	ES:[DE]，ES:[HL] （ES レジスタにて上位 4 ビット・アドレス指定）

図 4—27 [DE]，[HL] の例

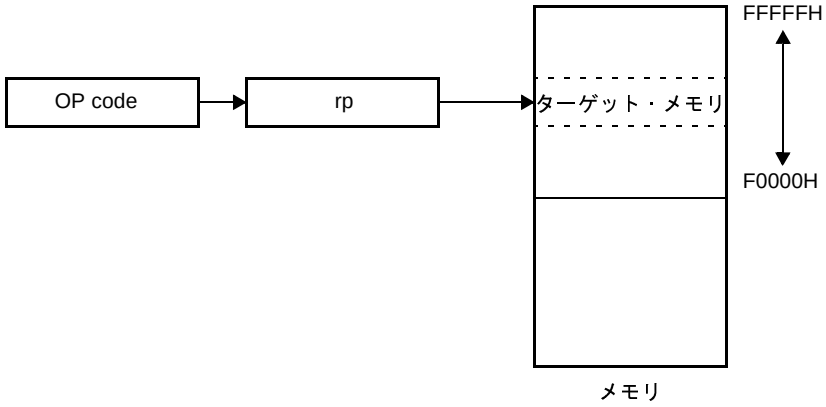
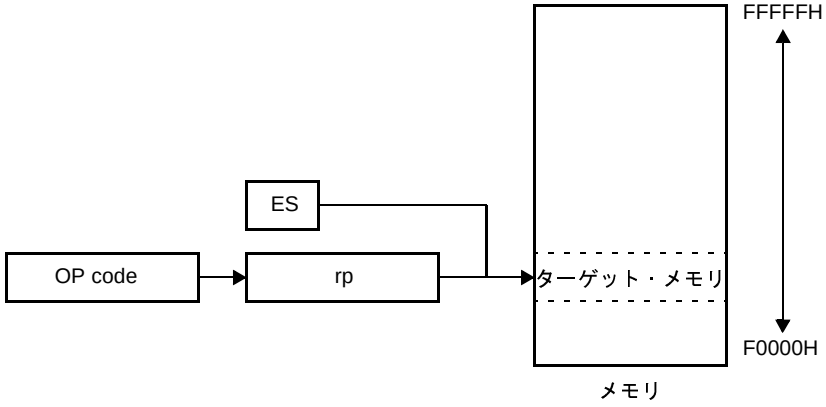


図 4—28 ES:[DE]，ES:[HL] の例



(g) ベースト・アドレッシング

命令語で指定されるレジスタ・ペアの内容をベース・アドレスとし，8 ビット・イミーディエト・データまたは 16 ビット・イミーディエト・データをオフセット・データとしてベース・アドレスに加算した結果で，対象となるアドレスを指定するアドレッシングです。

オペランド形式を以下に示します。

表現形式	記述方法
—	[HL ＋ byte]，[DE + byte]，[SP + byte] （F0000H-FFFFFH 空間のみ指定可能）
—	word[B]，word[C] （F0000H-FFFFFH 空間のみ指定可能）
—	word[BC] （F0000H-FFFFFH 空間のみ指定可能）
—	ES:[HL ＋ byte]，ES:[DE + byte] （ES レジスタにて上位 4 ビット・アドレス指定）
—	ES:word[B]，ES:word[C] （ES レジスタにて上位 4 ビット・アドレス指定）
—	ES:word[BC] （ES レジスタにて上位 4 ビット・アドレス指定）

図 4—29 [SP+byte] の例

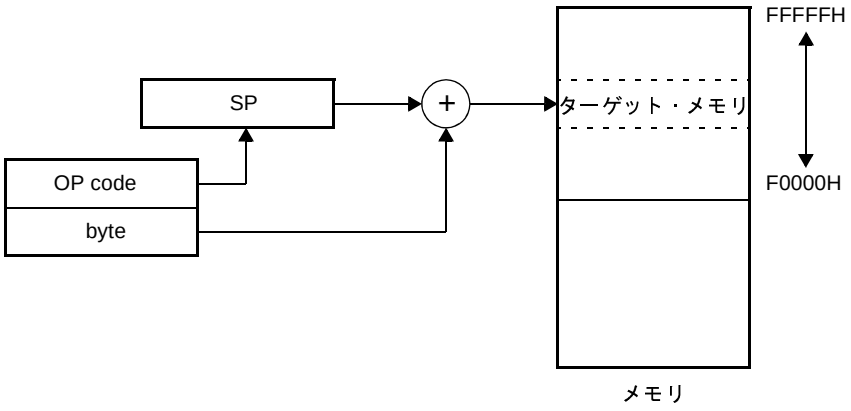


図 4—30 [HL+byte]，[DE+byte] の例

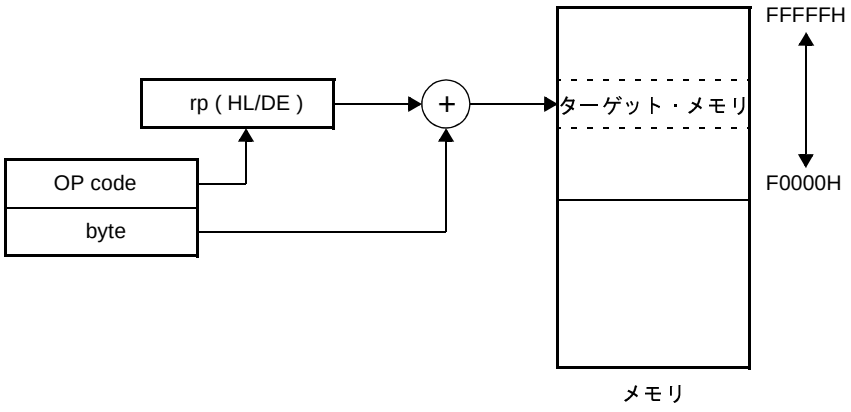


図 4—31 word[B]，word[C] の例

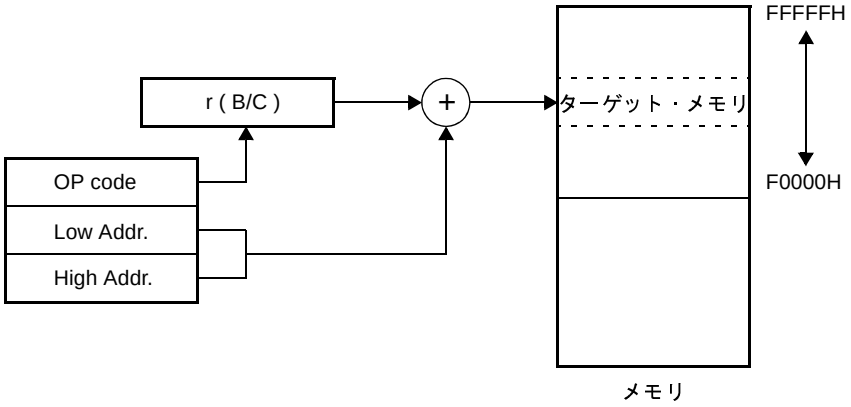


図 4—32 word[BC] の例

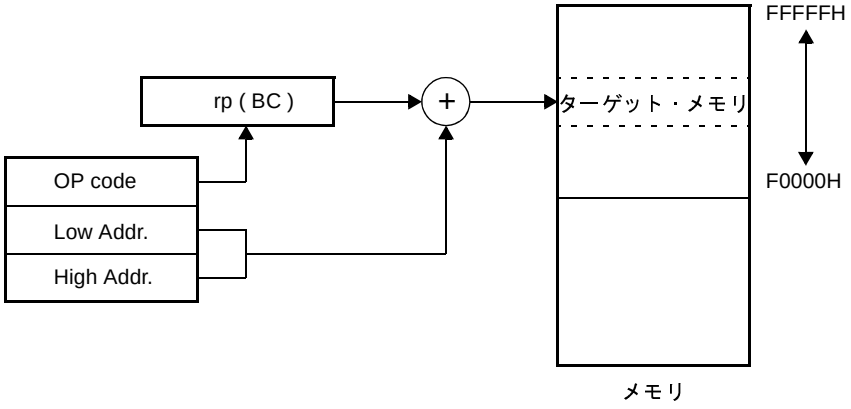


図 4—33 ES:[HL+byte]，ES:[DE+byte] の例

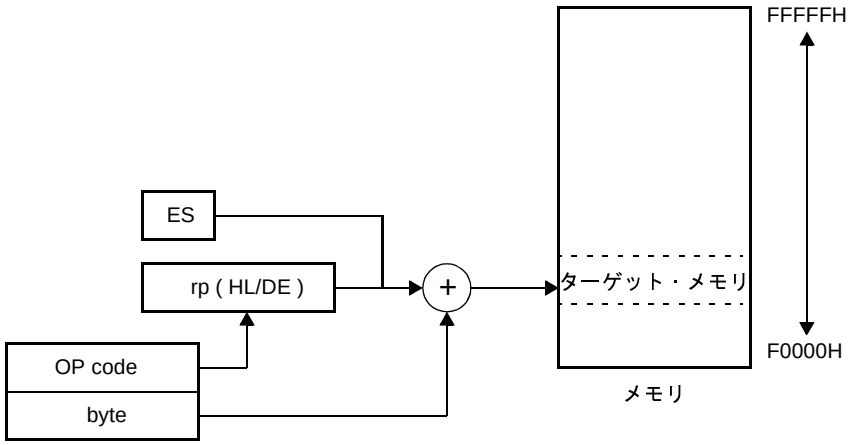


図 4—34 ES:word[B]，ES:word[C] の例

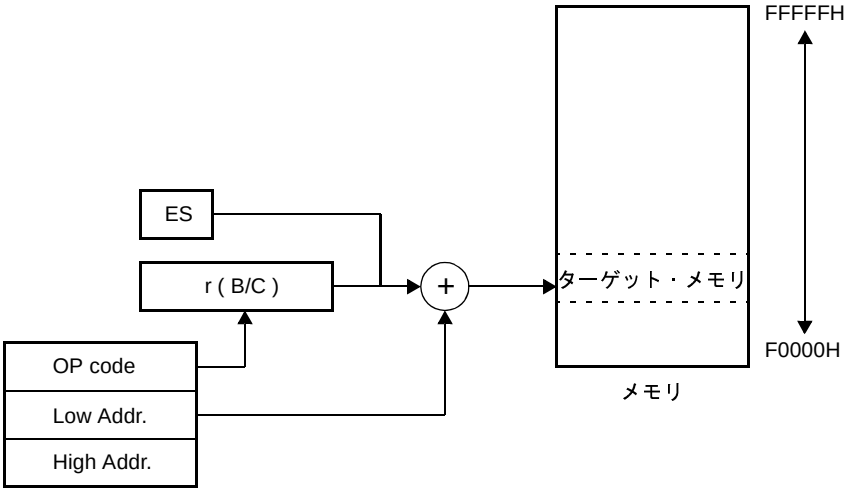
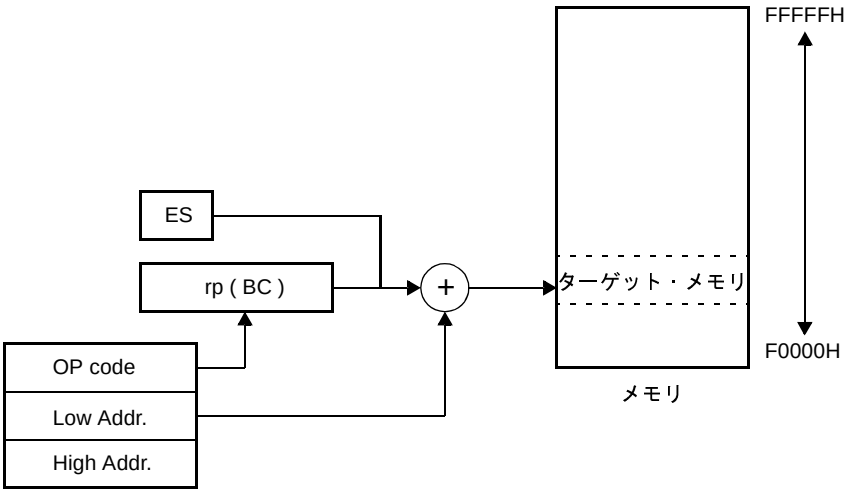


図 4—35 ES:word[BC] の例



（h）ベースト・インデクスト・アドレッシング

命令語で指定されるレジスタ・ペアの内容をベース・アドレスとし，同様に命令語で指定される B レジスタまたは C レジスタの内容をオフセット・アドレスとしてベース・アドレスに加算した結果で，対象となるアドレスを指定するアドレッシングです。

オペランド形式を以下に示します。

表現形式	記述方法
—	[HL ＋ B]，[HL ＋ C] （F0000H-FFFFFH 空間のみ指定可能）
—	ES:[HL ＋ B]，ES:[HL ＋ C] （ES レジスタにて上位 4 ビット・アドレス指定）

図 4—36 [HL+B]，[HL+C] の例

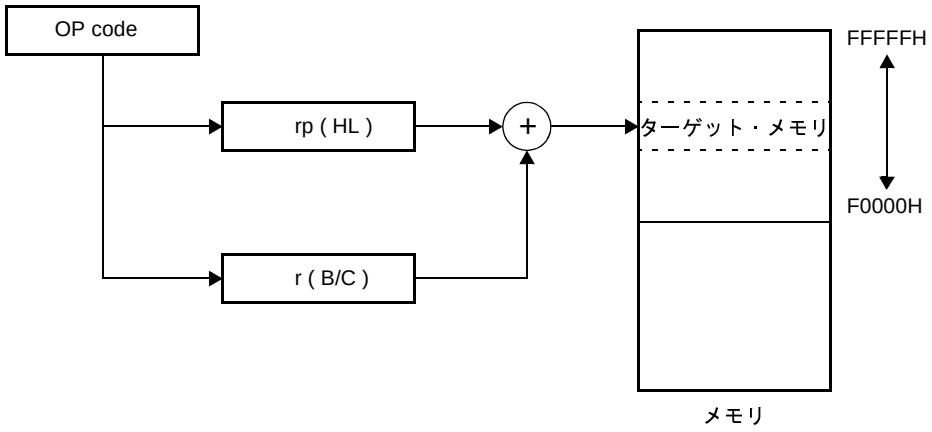
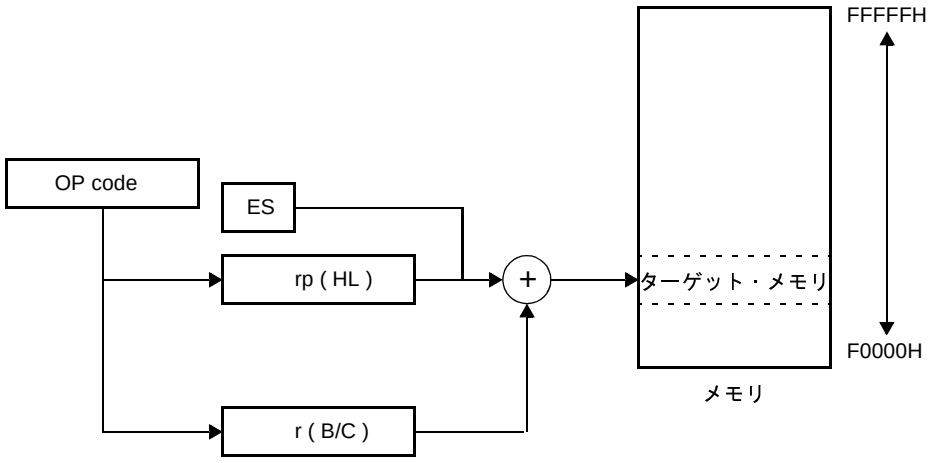


図 4—37 ES:[HL+B]，ES:[HL+C] の例



(i) スタック・アドレッシング

スタック・ポインタ（SP）の内容によりスタック領域を間接的に指定するアドレッシングです。PUSH，POP，サブルーチン・コール，リターン命令の実行時，および割り込み要求発生によるレジスタの退避／復帰時に自動的に用いられます。

スタック・アクセスは内部 RAM のみに用いられます。

オペランド形式を以下に示します。

表現形式	記述方法
—	PUSH AX/BC/DE/HL POP AX/BC/DE/HL CALL/CALLT RET BRK RETB （割り込み要求発生） RETI

4. 6. 5　命令セット

この項では，RL78 ファミリ，78K0R マイクロコントローラの命令セットを一覧表にして示します。
RL78 ファミリ，78K0R マイクロコントローラ製品の命令は，すべて共通です。

(1) オペランドの表現形式と記述方法

各命令のオペランド欄には，その命令のオペランド表現形式に対する記述方法に従ってオペランドを記述しています（詳細は，アセンブラ仕様によります）。記述方法の中で複数個あるものは，それらの要素の 1 つを選択します。大文字で書かれた英字および #，!，!!，$，$!，[ ]，ES: の記号はキーワードであり，そのまま記述します。記号の説明は，次のとおりです。

#	イミーディエト・データ指定
!	16 ビット絶対アドレス指定
!!	20 ビット絶対アドレス指定
$	8 ビット相対アドレス指定
$!	16 ビット相対アドレス指定
[ ]	間接アドレス指定
ES	拡張アドレス指定

イミーディエト・データのときは，適当な数値またはラベルを記述します。ラベルで記述する際も #，!，!!，$，$!，[ ]，ES: 記号は必ず記述してください。

また，オペランドのレジスタの記述形式 r，rp には，機能名称（X，A，C など），絶対名称（次の表の中のカッコ内の名称，R0，R1，R2 など）のいずれの形式でも記述可能です。

表 4―27　オペランドの表現形式と記述方法

表現形式	記述方法
r	X (R0)，A (R1)，C (R2)，B (R3)，E (R4)，D (R5)，L (R6)，H (R7)
rp	AX (RP0)，BC (RP1)，DE (RP2)，HL (RP3)
sfr	特殊機能レジスタ略号（SFR 略号）
sfrp	特殊機能レジスタ略号（16 ビット操作可能な SFR 略号。偶数アドレスのみ注）
saddr	FFE20H-FFF1FH：イミーディエト・データまたはラベル
saddrp	FFE20H-FFF1FH：イミーディエト・データまたはラベル（偶数アドレスのみ注）
addr20	00000H-FFFFFH：イミーディエト・データまたはラベル
addr16	0000H-0FFFFH：イミーディエト・データまたはラベル（16 ビット・データ時は偶数アドレスのみ注）
addr5	0080H-00BFH：イミーディエト・データまたはラベル（偶数アドレスのみ）
word	16 ビット・イミーディエト・データまたはラベル
byte	8 ビット・イミーディエト・データまたはラベル
bit	3 ビット・イミーディエト・データまたはラベル
RBn	RB0-RB3

注 奇数アドレスを指定した場合はビット 0 が“0”になります。

(2) オペレーション欄の記号

各命令のオペレーション欄には，その命令実行時の動作を次の記号を用いて表します。

表 4—28 オペレーション欄の記号

記号	機能
A	A レジスタ：8 ビット・アキュムレータ
X	X レジスタ
B	B レジスタ
C	C レジスタ
D	D レジスタ
E	E レジスタ
H	H レジスタ
L	L レジスタ
ES	ES レジスタ
CS	CS レジスタ
AX	AX レジスタ・ペア：16 ビット・アキュムレータ
BC	BC レジスタ・ペア
DE	DE レジスタ・ペア
HL	HL レジスタ・ペア
PC	プログラム・カウンタ
SP	スタック・ポインタ
PSW	プログラム・ステータス・ワード
CY	キャリー・フラグ
AC	補助キャリー・フラグ
Z	ゼロ・フラグ
RBS	レジスタ・バンク選択フラグ
IE	割り込み要求許可フラグ
( )	( ) 内のアドレスまたはレジスタの内容で示されるメモリの内容
XH，XL XS，XH，XL	16 ビット・レジスタの場合は XH = 上位 8 ビット，XL = 下位 8 ビット 20 ビット・レジスタの場合は XS（ビット 19-16），XH（ビット 15-8），XL（ビット 7-0）
∧	論理積（AND）
∨	論理和（OR）
∀	排他的論理和（exclusive OR）
—	反転データ
addr16	16 ビット・イミーディエト・データ
addr20	20 ビット・イミーディエト・データ

記号	機能
jdisp8	符号付き 8 ビット・データ（ディスプレースメント値）
jdisp16	符号付き 16 ビット・データ（ディスプレースメント値）

（3）フラグ欄の記号

各命令のフラグ欄には，その命令実行時のフラグの変化を下記の記号を用いて表します。

表 4—29 フラグ欄の記号

記号	フラグ変化
（ブランク）	変化なし
0	0 にクリアされる
1	1 にセットされる
×	結果にしたがってセット／リセットされる
R	以前に退避した値がリストアされる

（4）PREFIX 命令

ES: で示される命令は，PREFIX 命令コードを頭に付けることで，アクセスできるデータ領域を F0000H-FFFFFH の 64 K バイト空間から，ES レジスタの値を付加した 00000H-FFFFFH の 1 M バイト空間に拡張します。PREFIX 命令コードは対象となる命令の先頭に付けることで，PREFIX 命令コード直後の 1 命令だけを ES レジスタの値を付加したアドレスとして実行します。

表 4—30 PREFIX 命令コードの使用例

命令	命令コード				
	1	2	3	4	5
MOV !addr16, #byte	CFH	!addr16		#byte	—
MOV ES:!addr16, #byte	11H	CFH	!addr16		#byte
MOV A, [HL]	8BH	—	—	—	—
MOV A, ES:[HL]	11H	8BH	—	—	—

注意 ES レジスタの値は，PREFIX 命令を実行するまでに MOV ES，A などで事前に設定しておいてください。

(5) オペレーション一覧

(a) 8 ビット・データ転送命令

表 4—31 オペレーション一覧（8 ビット・データ転送命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注 1	注 2		Z	AC	CY
MOV	r, #byte	2	1	—	r ← byte			
	saddr, #byte	3	1	—	(saddr) ← byte			
	sfr, #byte	3	1	—	sfr ← byte			
	!addr16, #byte	4	1	—	(addr16) ← byte			
	A, r 注3	1	1	—	A ← r			
	r, A 注3	1	1	—	r ← A			
	A, saddr	2	1	—	A ← (saddr)			
	saddr, A	2	1	—	(saddr) ← A			
	A, sfr	2	1	—	A ← sfr			
	sfr, A	2	1	—	sfr ← A			
	A, !addr16	3	1	4	A ← (addr16)			
	!addr16, A	3	1	—	(addr16) ← A			
	PSW, #byte	3	3	—	PSW ← byte	×	×	×
	A, PSW	2	1	—	A ← PSW			
	PSW, A	2	3	—	PSW ← A	×	×	×
	ES, #byte	2	1	—	ES ← byte			
	ES, saddr	3	1	—	ES ← (saddr)			
	A, ES	2	1	—	A ← ES			
	ES, A	2	1	—	ES ← A			
	CS, #byte	3	1	—	CS ← byte			
	A, CS	2	1	—	A ← CS			
	CS, A	2	1	—	CS ← A			
	A, [DE]	1	1	4	A ← (DE)			
	[DE], A	1	1	—	(DE) ← A			
	[DE+byte], #byte	3	1	—	(DE + byte) ← byte			
	A, [DE+byte]	2	1	4	A ← (DE + byte)			
	[DE+byte], A	2	1	—	(DE + byte) ← A			
	A, [HL]	1	1	4	A ← (HL)			
	[HL], A	1	1	—	(HL) ← A			
	[HL+byte], #byte	3	1	—	(HL + byte) ← byte			
	A, [HL+byte]	2	1	4	A ← (HL + byte)			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ 注4		
			注1	注2		Z	AC	CY
	[HL+byte], A	2	1	—	(HL + byte) ← A			
	A, [HL+B]	2	1	4	A ← (HL + B)			
	[HL+B], A	2	1	—	(HL + B) ← A			
	A, [HL+C]	2	1	4	A ← (HL + C)			
	[HL+C], A	2	1	—	(HL + C) ← A			
	word[B], #byte	4	1	—	(B + word) ← byte			
	A, word[B]	3	1	4	A ← (B + word)			
	word[B], A	3	1	—	(B + word) ← A			
	word[C], #byte	4	1	—	(C + word) ← byte			
	A, word[C]	3	1	4	A ← (C + word)			
	word[C], A	3	1	—	(C + word) ← A			
	word[BC], #byte	4	1	—	(BC + word) ← byte			
	A, word[BC]	3	1	4	A ← (BC + word)			
	word[BC], A	3	1	—	(BC + word) ← A			
	[SP+byte], #byte	3	1	—	(SP + byte) ← byte			
	A, [SP+byte]	2	1	—	A ← (SP + byte)			
	[SP+byte], A	2	1	—	(SP + byte) ← A			
	B, saddr	2	1	—	B ← (saddr)			
	B, !addr16	3	1	4	B ← (addr16)			
	C, saddr	2	1	—	C ← (saddr)			
	C, !addr16	3	1	4	C ← (addr16)			
	X, saddr	2	1	—	X ← (saddr)			
	X, !addr16	3	1	4	X ← (addr16)			
	ES:!addr16, #byte	5	2	—	(ES, addr16) ← byte			
	A, ES:!addr16	4	2	5	A ← (ES, addr16)			
	ES:!addr16, A	4	2	—	(ES, addr16) ← A			
	A, ES:[DE]	2	2	5	A ← (ES, DE)			
	ES:[DE], A	2	2	—	(ES, DE) ← A			
	ES:[DE+byte], #byte	4	2	—	((ES, DE) + byte) ← byte			
	A, ES:[DE+byte]	3	2	5	A ← ((ES, DE) + byte)			
	ES:[DE+byte], A	3	2	—	((ES, DE) + byte) ← A			
	A, ES:[HL]	2	2	5	A ← (ES, HL)			
	ES:[HL], A	2	2	—	(ES, HL) ← A			
	ES:[HL+byte], #byte	4	2	—	((ES, HL) + byte) ← byte			
	A, ES:[HL+byte]	3	2	5	A ← ((ES, HL) + byte)			
	ES:[HL+byte], A	3	2	—	((ES, HL) + byte) ← A			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ 注4		
			注1	注2		Z	AC	CY
	A, ES:[HL+B]	3	2	5	A ← ((ES, HL) + B)			
	ES:[HL+B], A	3	2	—	((ES, HL) + B) ← A			
	A, ES:[HL+C]	3	2	5	A ← ((ES, HL) + C)			
	ES:[HL+C], A	3	2	—	((ES, HL) + C) ← A			
	ES:word[B], #byte	5	2	—	((ES, B) + word) ← byte			
	A, ES:word[B]	4	2	5	A ← ((ES, B) + word)			
	ES:word[B], A	4	2	—	((ES, B) + word) ← A			
	ES:word[C], #byte	5	2	—	((ES, C) + word) ← byte			
	A, ES:word[C]	4	2	5	A ← ((ES, C) + word)			
	ES:word[C], A	4	2	—	((ES, C) + word) ← A			
	ES:word[BC], #byte	5	2	—	((ES, BC) + word) ← byte			
	A, ES:word[BC]	4	2	5	A ← ((ES, BC) + word)			
	ES:word[BC], A	4	2	—	((ES, BC) + word) ← A			
	B, ES:!addr16	4	2	5	B ← (ES, addr16)			
	C, ES:!addr16	4	2	5	C ← (ES, addr16)			
	X, ES:!addr16	4	2	5	X ← (ES, addr16)			
XCH	A, r 注3	1 (r = X) 2 (r = X 以外)	1	—	A ←→ r			
	A, saddr	3	2	—	A ←→ (saddr)			
	A, sfr	3	2	—	A ←→ sfr			
	A, !addr16	4	2	—	A ←→ (addr16)			
	A, [DE]	2	2	—	A ←→ (DE)			
	A, [DE+byte]	3	2	—	A ←→ (DE + byte)			
	A, [HL]	2	2	—	A ←→ (HL)			
	A, [HL+byte]	3	2	—	A ←→ (HL + byte)			
	A, [HL+B]	2	2	—	A ←→ (HL + B)			
	A, [HL+C]	2	2	—	A ←→ (HL + C)			
	A, ES:!addr16	5	3	—	A ←→ (ES, addr16)			
	A, ES:[DE]	3	3	—	A ←→ (ES, DE)			
	A, ES:[DE+byte]	4	3	—	A ←→ ((ES, DE) + byte)			
	A, ES:[HL]	3	3	—	A ←→ (ES, HL)			
	A, ES:[HL+byte]	4	3	—	A ←→ ((ES, HL) + byte)			
	A, ES:[HL+B]	3	3	—	A ←→ ((ES, HL) + B)			
	A, ES:[HL+C]	3	3	—	A ←→ ((ES, HL) + C)			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ 注4		
			注1	注2		Z	AC	CY
ONEB	A	1	1	—	A ← 01H			
	X	1	1	—	X ← 01H			
	B	1	1	—	B ← 01H			
	C	1	1	—	C ← 01H			
	saddr	2	1	—	(saddr) ← 01H			
	!addr16	3	1	—	(addr16) ← 01H			
	ES:!addr16	4	2	—	(ES, addr16) ← 01H			
CLRB	A	1	1	—	A ← 00H			
	X	1	1	—	X ← 00H			
	B	1	1	—	B ← 00H			
	C	1	1	—	C ← 00H			
	saddr	2	1	—	(saddr) ← 00H			
	!addr16	3	1	—	(addr16) ← 00H			
	ES:!addr16	4	2	—	(ES,addr16) ← 00H			
MOVS	[HL+byte], X	3	1	—	(HL + byte) ← X	×		×
	ES:[HL+byte], X	4	2	—	(ES, HL + byte) ← X	×		×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. r = A を除く。

4. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（b）16 ビット・データ転送命令

表 4—32 オペレーション一覧（16 ビット・データ転送命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注 1	注 2		Z	AC	CY
MOVW	rp, #word	3	1	—	rp ← word			
	saddrp, #word	4	1	—	(saddrp) ← word			
	sfrp, #word	4	1	—	sfrp ← word			
	AX, saddrp	2	1	—	AX ← (saddrp)			
	saddrp, AX	2	1	—	(saddrp) ← AX			
	AX, sfrp	2	1	—	AX ← sfrp			
	sfrp, AX	2	1	—	sfrp ← AX			
	AX, rp 注3	1	1	—	AX ← rp			
	rp, AX 注3	1	1	—	rp ← AX			
	AX, !addr16	3	1	4	AX ← (addr16)			
	!addr16, AX	3	1	—	(addr16) ← AX			
	AX, [DE]	1	1	4	AX ← (DE)			
	[DE], AX	1	1	—	(DE) ← AX			
	AX, [DE+byte]	2	1	4	AX ← (DE + byte)			
	[DE+byte], AX	2	1	—	(DE + byte) ← AX			
	AX, [HL]	1	1	4	AX ← (HL)			
	[HL], AX	1	1	—	(HL) ← AX			
	AX, [HL+byte]	2	1	4	AX ← (HL + byte)			
	[HL+byte], AX	2	1	—	(HL + byte) ← AX			
	AX, word[B]	3	1	4	AX ← (B + word)			
	word[B], AX	3	1	—	(B + word) ← AX			
	AX, word[C]	3	1	4	AX ← (C + word)			
	word[C], AX	3	1	—	(C + word) ← AX			
	AX, word[BC]	3	1	4	AX ← (BC + word)			
	word[BC], AX	3	1	—	(BC + word) ← AX			
	AX, [SP+byte]	2	1	—	AX ← (SP + byte)			
	[SP+byte], AX	2	1	—	(SP + byte) ← AX			
	BC, saddrp	2	1	—	BC ← (saddrp)			
	BC, !addr16	3	1	4	BC ← (addr16)			
	DE, saddrp	2	1	—	DE ← (saddrp)			
	DE, !addr16	3	1	4	DE ← (addr16)			
	HL, saddrp	2	1	—	HL ← (saddrp)			
	HL, !addr16	3	1	4	HL ← (addr16)			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
	AX, ES:!addr16	4	2	5	AX ← (ES, addr16)			
	ES:!addr16, AX	4	2	–	(ES, addr16) ← AX			
	AX, ES:[DE]	2	2	5	AX ← (ES, DE)			
	ES:[DE], AX	2	2	–	(ES, DE) ← AX			
	AX, ES:[DE+byte]	3	2	5	AX ← ((ES, DE) + byte)			
	ES:[DE+byte], AX	3	2	–	((ES, DE) + byte) ← AX			
	AX, ES:[HL]	2	2	5	AX ← (ES, HL)			
	ES:[HL], AX	2	2	–	(ES, HL) ← AX			
	AX, ES:[HL+byte]	3	2	5	AX ← ((ES, HL) + byte)			
	ES:[HL+byte], AX	3	2	–	((ES, HL) + byte) ← AX			
	AX, ES:word[B]	4	2	5	AX ← ((ES, B) + word)			
	ES:word[B], AX	4	2	–	((ES, B) + word) ← AX			
	AX, ES:word[C]	4	2	5	AX ← ((ES, C) + word)			
	ES:word[C], AX	4	2	–	((ES, C) + word) ← AX			
	AX, ES:word[BC]	4	2	5	AX ← ((ES, BC) + word)			
	ES:word[BC], AX	4	2	–	((ES, BC) + word) ← AX			
	BC, ES:!addr16	4	2	5	BC ← (ES, addr16)			
	DE, ES:!addr16	4	2	5	DE ← (ES, addr16)			
	HL, ES:!addr16	4	2	5	HL ← (ES, addr16)			
XCHW	AX, rp 注3	1	1	–	AX ←→ rp			
ONEW	AX	1	1	–	AX ← 0001H			
	BC	1	1	–	BC ← 0001H			
CLRW	AX	1	1	–	AX ← 0000H			
	BC	1	1	–	BC ← 0000H			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. rp = AX を除く。

4. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄　：変化なし
0　　：0 にクリアされる
1　　：1 にセットされる
×　　：結果にしたがってセット／リセットされる
R　　：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。
3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(c) 8 ビット演算命令

表 4—33 オペレーション一覧（8 ビット演算命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注 4		
			注 1	注 2		Z	AC	CY
ADD	A, #byte	2	1	—	A, CY ← A + byte	×	×	×
	saddr, #byte	3	2	—	(saddr), CY ← (saddr) + byte	×	×	×
	A, r 注 3	2	1	—	A, CY ← A + r	×	×	×
	r, A	2	1	—	r, CY ← r + A	×	×	×
	A, saddr	2	1	—	A, CY ← A + (saddr)	×	×	×
	A, !addr16	3	1	4	A, CY ← A + (addr16)	×	×	×
	A, [HL]	1	1	4	A, CY ← A + (HL)	×	×	×
	A, [HL+byte]	2	1	4	A, CY ← A + (HL + byte)	×	×	×
	A, [HL+B]	2	1	4	A, CY ← A + (HL + B)	×	×	×
	A, [HL+C]	2	1	4	A, CY ← A + (HL + C)	×	×	×
	A, ES:!addr16	4	2	5	A, CY ← A + (ES, addr16)	×	×	×
	A, ES:[HL]	2	2	5	A,CY ← A + (ES, HL)	×	×	×
	A, ES:[HL+byte]	3	2	5	A,CY ← A + ((ES, HL) + byte)	×	×	×
	A, ES:[HL+B]	3	2	5	A,CY ← A + ((ES, HL) + B)	×	×	×
	A, ES:[HL+C]	3	2	5	A,CY ← A + ((ES, HL) + C)	×	×	×
ADDC	A, #byte	2	1	—	A, CY ← A + byte + CY	×	×	×
	saddr, #byte	3	2	—	(saddr), CY ← (saddr) + byte + CY	×	×	×
	A, r 注 3	2	1	—	A, CY ← A + r + CY	×	×	×
	r, A	2	1	—	r, CY ← r + A + CY	×	×	×
	A, saddr	2	1	—	A, CY ← A + (saddr) + CY	×	×	×
	A, !addr16	3	1	4	A, CY ← A + (addr16) + CY	×	×	×
	A, [HL]	1	1	4	A, CY ← A + (HL) + CY	×	×	×
	A, [HL+byte]	2	1	4	A, CY ← A + (HL + byte) + CY	×	×	×
	A, [HL+B]	2	1	4	A, CY ← A + (HL + B) + CY	×	×	×
	A, [HL+C]	2	1	4	A, CY ← A + (HL + C) + CY	×	×	×
	A, ES:!addr16	4	2	5	A, CY ← A + (ES, addr16) + CY	×	×	×
	A, ES:[HL]	2	2	5	A, CY ← A + (ES, HL) + CY	×	×	×
	A, ES:[HL+byte]	3	2	5	A, CY ← A + ((ES, HL) + byte) + CY	×	×	×

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
	A, ES:[HL+B]	3	2	5	A, CY ← A + ((ES, HL) + B) + CY	×	×	×
	A, ES:[HL+C]	3	2	5	A, CY ← A + ((ES, HL) + C) + CY	×	×	×
SUB	A, #byte	2	1	—	A, CY ← A - byte	×	×	×
	saddr, #byte	3	2	—	(saddr), CY ← (saddr) - byte	×	×	×
	A, r 注3	2	1	—	A, CY ← A - r	×	×	×
	r, A	2	1	—	r, CY ← r - A	×	×	×
	A, saddr	2	1	—	A, CY ← A - (saddr)	×	×	×
	A, !addr16	3	1	4	A, CY ← A - (addr16)	×	×	×
	A, [HL]	1	1	4	A, CY ← A - (HL)	×	×	×
	A, [HL+byte]	2	1	4	A, CY ← A - (HL + byte)	×	×	×
	A, [HL+B]	2	1	4	A, CY ← A - (HL + B)	×	×	×
	A, [HL+C]	2	1	4	A, CY ← A - (HL + C)	×	×	×
	A, ES:!addr16	4	2	5	A, CY ← A - (ES:addr16)	×	×	×
	A, ES:[HL]	2	2	5	A, CY ← A - (ES:HL)	×	×	×
	A, ES:[HL+byte]	3	2	5	A, CY ← A - ((ES:HL) + byte)	×	×	×
	A, ES:[HL+B]	3	2	5	A, CY ← A - ((ES:HL) + B)	×	×	×
	A, ES:[HL+C]	3	2	5	A, CY ← A - ((ES:HL) + C)	×	×	×
SUBC	A, #byte	2	1	—	A, CY ← A - byte - CY	×	×	×
	saddr, #byte	3	2	—	(saddr), CY ← (saddr) - byte - CY	×	×	×
	A, r 注3	2	1	—	A, CY ← A - r - CY	×	×	×
	r, A	2	1	—	r, CY ← r - A - CY	×	×	×
	A, saddr	2	1	—	A, CY ← A - (saddr) - CY	×	×	×
	A, !addr16	3	1	4	A, CY ← A - (addr16) - CY	×	×	×
	A, [HL]	1	1	4	A, CY ← A - (HL) - CY	×	×	×
	A, [HL+byte]	2	1	4	A, CY ← A - (HL + byte) - CY	×	×	×
	A, [HL+B]	2	1	4	A, CY ← A - (HL + B) - CY	×	×	×
	A, [HL+C]	2	1	4	A, CY ← A - (HL + C) - CY	×	×	×
	A, ES:!addr16	4	2	5	A, CY ← A - (ES:addr16) - CY	×	×	×
	A, ES:[HL]	2	2	5	A, CY ← A - (ES:HL) - CY	×	×	×
	A, ES:[HL+byte]	3	2	5	A, CY ← A - ((ES:HL) + byte) - CY	×	×	×

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
	A, ES:[HL+B]	3	2	5	A, CY ← A - ((ES:HL) + B) - CY	×	×	×
	A, ES:[HL+C]	3	2	5	A, CY ← A - ((ES:HL) + C) - CY	×	×	×
AND	A, #byte	2	1	—	A ← A ∧ byte	×		
	saddr, #byte	3	2	—	(saddr) ← (saddr) ∧ byte	×		
	A, r 注3	2	1	—	A ← A ∧ r	×		
	r, A	2	1	—	r ← r ∧ A	×		
	A, saddr	2	1	—	A ← A ∧ (saddr)	×		
	A, !addr16	3	1	4	A ← A ∧ (addr16)	×		
	A, [HL]	1	1	4	A ← A ∧ (HL)	×		
	A, [HL+byte]	2	1	4	A ← A ∧ (HL + byte)	×		
	A, [HL+B]	2	1	4	A ← A ∧ (HL + B)	×		
	A, [HL+C]	2	1	4	A ← A ∧ (HL + C)	×		
	A, ES:!addr16	4	2	5	A ← A ∧ (ES:addr16)	×		
	A, ES:[HL]	2	2	5	A ← A ∧ (ES:HL)	×		
	A, ES:[HL+byte]	3	2	5	A ← A ∧ ((ES:HL) + byte)	×		
	A, ES:[HL+B]	3	2	5	A ← A ∧ ((ES:HL) + B)	×		
	A, ES:[HL+C]	3	2	5	A ← A ∧ ((ES:HL) + C)	×		
OR	A, #byte	2	1	—	A ← A ∨ byte	×		
	saddr, #byte	3	2	—	(saddr) ← (saddr) ∨ byte	×		
	A, r 注3	2	1	—	A ← A ∨ r	×		
	r, A	2	1	—	r ← r ∨ A	×		
	A, saddr	2	1	—	A ← A ∨ (saddr)	×		
	A, !addr16	3	1	4	A ← A ∨ (addr16)	×		
	A, [HL]	1	1	4	A ← A ∨ (HL)	×		
	A, [HL+byte]	2	1	4	A ← A ∨ (HL + byte)	×		
	A, [HL+B]	2	1	4	A ← A ∨ (HL + B)	×		
	A, [HL+C]	2	1	4	A ← A ∨ (HL + C)	×		
	A, ES:!addr16	4	2	5	A ← A ∨ (ES:addr16)	×		
	A, ES:[HL]	2	2	5	A ← A ∨ (ES:HL)	×		
	A, ES:[HL+byte]	3	2	5	A ← A ∨ ((ES:HL) + byte)	×		
	A, ES:[HL+B]	3	2	5	A ← A ∨ ((ES:HL) + B)	×		
	A, ES:[HL+C]	3	2	5	A ← A ∨ ((ES:HL) + C)	×		

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
XOR	A, #byte	2	1	—	A ← A ∀ byte	×		
	saddr, #byte	3	2	—	(saddr) ← (saddr) ∀ byte	×		
	A, r 注3	2	1	—	A ← A ∀ r	×		
	r, A	2	1	—	r ← r ∀ A	×		
	A, saddr	2	1	—	A ← A ∀ (saddr)	×		
	A, !addr16	3	1	4	A ← A ∀ (addr16)	×		
	A, [HL]	1	1	4	A ← A ∀ (HL)	×		
	A, [HL+byte]	2	1	4	A ← A ∀ (HL + byte)	×		
	A, [HL+B]	2	1	4	A ← A ∀ (HL + B)	×		
	A, [HL+C]	2	1	4	A ← A ∀ (HL + C)	×		
	A, ES:!addr16	4	2	5	A ← A ∀ (ES:addr16)	×		
	A, ES:[HL]	2	2	5	A ← A ∀ (ES:HL)	×		
	A, ES:[HL+byte]	3	2	5	A ← A ∀ ((ES:HL) + byte)	×		
	A, ES:[HL+B]	3	2	5	A ← A ∀ ((ES:HL) + B)	×		
	A, ES:[HL+C]	3	2	5	A ← A ∀ ((ES:HL) + C)	×		
CMP	A, #byte	2	1	—	A - byte	×	×	×
	saddr, #byte	3	1	—	(saddr) - byte	×	×	×
	A, r 注3	2	1	—	A - r	×	×	×
	r, A	2	1	—	r - A	×	×	×
	A, saddr	2	1	—	A - (saddr)	×	×	×
	A, !addr16	3	1	4	A - (addr16)	×	×	×
	A, [HL]	1	1	4	A - (HL)	×	×	×
	A, [HL+byte]	2	1	4	A - (HL + byte)	×	×	×
	A, [HL+B]	2	1	4	A - (HL + B)	×	×	×
	A, [HL+C]	2	1	4	A - (HL + C)	×	×	×
	!addr16, #byte	4	1	4	(addr16) - byte	×	×	×
	A, ES:!addr16	4	2	5	A - (ES:addr16)	×	×	×
	A, ES:[HL]	2	2	5	A - (ES:HL)	×	×	×
	A, ES:[HL+byte]	3	2	5	A - ((ES:HL) + byte)	×	×	×
	A, ES:[HL+B]	3	2	5	A - ((ES:HL) + B)	×	×	×
	A, ES:[HL+C]	3	2	5	A - ((ES:HL) + C)	×	×	×
	ES:!addr16, #byte	5	2	5	(ES:addr16) - byte	×	×	×

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
CMP0	A	1	1	—	A - 00H	×	×	×
	X	1	1	—	X - 00H	×	×	×
	B	1	1	—	B - 00H	×	×	×
	C	1	1	—	C - 00H	×	×	×
	saddr	2	1	—	(saddr) - 00H	×	×	×
	!addr16	3	1	4	(addr16) - 00H	×	×	×
	ES:!addr16	4	2	5	(ES:addr16) - 00H	×	×	×
CMPS	X, [HL+byte]	3	1	4	X - (HL + byte)	×	×	×
	X, ES:[HL+byte]	4	2	5	X - ((ES:HL) + byte)	×	×	×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. r = A を除く。

4. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（d）16 ビット演算命令

表 4—34 オペレーション一覧（16 ビット演算命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注 3		
			注 1	注 2		Z	AC	CY
ADDW	AX, #word	3	1	—	AX, CY ← AX + word	×	×	×
	AX, AX	1	1	—	AX, CY ← AX + AX	×	×	×
	AX, BC	1	1	—	AX, CY ← AX + BC	×	×	×
	AX, DE	1	1	—	AX, CY ← AX + DE	×	×	×
	AX, HL	1	1	—	AX, CY ← AX + HL	×	×	×
	AX, saddrp	2	1	—	AX, CY ← AX + (saddrp)	×	×	×
	AX, !addr16	3	1	4	AX, CY ← AX + (addr16)	×	×	×
	AX, [HL+byte]	3	1	4	AX, CY ← AX + (HL + byte)	×	×	×
	AX, ES:!addr16	4	2	5	AX, CY ← AX + (ES:addr16)	×	×	×
	AX, ES:[HL+byte]	4	2	5	AX, CY ← AX + ((ES:HL) + byte)	×	×	×
SUBW	AX, #word	3	1	—	AX, CY ← AX - word	×	×	×
	AX, BC	1	1	—	AX, CY ← AX - BC	×	×	×
	AX, DE	1	1	—	AX, CY ← AX - DE	×	×	×
	AX, HL	1	1	—	AX, CY ← AX - HL	×	×	×
	AX, saddrp	2	1	—	AX, CY ← AX - (saddrp)	×	×	×
	AX, !addr16	3	1	4	AX, CY ← AX - (addr16)	×	×	×
	AX, [HL+byte]	3	1	4	AX, CY ← AX - (HL - byte)	×	×	×
	AX, ES:!addr16	4	2	5	AX, CY ← AX - (ES:addr16)	×	×	×
	AX, ES:[HL+byte]	4	2	5	AX, CY ← AX - ((ES:HL) + byte)	×	×	×
CMPW	AX, #word	3	1	—	AX - word	×	×	×
	AX, BC	1	1	—	AX - BC	×	×	×
	AX, DE	1	1	—	AX - DE	×	×	×
	AX, HL	1	1	—	AX - HL	×	×	×
	AX, saddrp	2	1	—	AX - (saddrp)	×	×	×
	AX, !addr16	3	1	4	AX - (addr16)	×	×	×
	AX, [HL+byte]	3	1	4	AX - (HL + byte)	×	×	×
	AX, ES:!addr16	4	2	5	AX - (ES:addr16)	×	×	×
	AX, ES:[HL+byte]	4	2	5	AX - ((ES:HL) + byte)	×	×	×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄　：変化なし
0　　：0にクリアされる
1　　：1にセットされる
×　　：結果にしたがってセット／リセットされる
R　　：以前に退避した値がリストアされる

備考 1. 命令の1クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択したCPUクロック（fCLK）の1クロック分です。

2. クロック数は内部ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大16バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(e) 乗除積和算命令

表 4—35 オペレーション一覧（乗除積和算命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
MULU	X	1	1	—	AX ← A × X			
MULHU 注4	—	3	2	—	BCAX ← AX × BC			
MULH 注4	—	3	2	—	BCAX ← AX × BC			
DIVHU 注4	—	3	9	—	AX（商），DE（余り）← AX ÷ DE			
DIVWU 注4	—	3	17	—	BCAX（商），HLDE（余り）← BCAX ÷ HLDE			
MACHU 注4	—	3	3	—	MACR ← MACR + AX × BC		×	×
MACH 注4	—	3	3	—	MACR ← MACR + AX × BC		×	×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

4. 拡張命令のため，製品ごとに搭載／非搭載が異なります。各製品のユーザーズ・マニュアルを参照してください。

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(f) 増減命令

表 4—36 オペレーション一覧（増減命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
INC	r	1	1	—	r ← r + 1	×	×	
	saddr	2	2	—	(saddr) ← (saddr) + 1	×	×	
	!addr16	3	2	—	(addr16) ← (addr16) + 1	×	×	
	[HL+byte]	3	2	—	(HL + byte) ← (HL + byte) + 1	×	×	
	ES:!addr16	4	3	—	(ES, addr16) ← (ES, addr16) + 1	×	×	
	ES: [HL+byte]	4	3	—	((ES:HL) + byte) ← ((ES:HL) + byte) + 1	×	×	
DEC	r	1	1	—	r ← r - 1	×	×	
	saddr	2	2	—	(saddr) ← (saddr) - 1	×	×	
	!addr16	3	2	—	(addr16) ← (addr16) - 1	×	×	
	[HL+byte]	3	2	—	(HL + byte) ← (HL + byte) - 1	×	×	
	ES:!addr16	4	3	—	(ES, addr16) ← (ES, addr16) - 1	×	×	
	ES: [HL+byte]	4	3	—	((ES:HL) + byte) ← ((ES:HL) + byte) - 1	×	×	
INCW	rp	1	1	—	rp ← rp + 1			
	saddrp	2	2	—	(saddrp) ← (saddrp) + 1			
	!addr16	3	2	—	(addr16) ← (addr16) + 1			
	[HL+byte]	3	2	—	(HL + byte) ← (HL + byte) + 1			
	ES:!addr16	4	3	—	(ES, addr16) ← (ES, addr16) + 1			
	ES: [HL+byte]	4	3	—	((ES:HL) + byte) ← ((ES:HL) + byte) + 1			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
DECW	rp	1	1	—	rp ← rp - 1			
	saddrp	2	2	—	(saddrp) ← (saddrp) - 1			
	!addr16	3	2	—	(addr16) ← (addr16) - 1			
	[HL+byte]	3	2	—	(HL + byte) ← (HL + byte) - 1			
	ES:!addr16	4	3	—	(ES, addr16) ← (ES, addr16) - 1			
	ES: [HL+byte]	4	3	—	((ES:HL) + byte) ← ((ES:HL) + byte) - 1			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（g）シフト命令

表 4—37 オペレーション一覧（シフト命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
SHR	A, cnt	2	1	—	(CY ← A_0, A_{m-1} ← A_m, A_7 ← 0) × cnt			×
SHRW	AX, cnt	2	1	—	(CY ← AX_0, AX_{m-1} ← AX_m, AX_{15} ← 0) × cnt			×
SHL	A, cnt	2	1	—	(CY ← A_7, A_m ← A_{m-1}, A_0 ← 0) × cnt			×
	B, cnt	2	1	—	(CY ← B_7, B_m ← B_{m-1}, B_0 ← 0) × cnt			×
	C, cnt	2	1	—	(CY ← C_7, C_m ← C_{m-1}, C_0 ← 0) × cnt			×
SHLW	AX, cnt	2	1	—	(CY ← AX_{15}, AX_m ← AX_{m-1}, AX_0 ← 0) × cnt			×
	BC, cnt	2	1	—	(CY ← BC_{15}, BC_m ← BC_{m-1}, BC_0 ← 0) × cnt			×
SAR	A, cnt	2	1	—	(CY ← A_0, A_{m-1} ← A_m, A_7 ← A_7) × cnt			×
SARW	AX, cnt	2	1	—	(CY ← AX_0, AX_{m-1} ← AX_m, AX_{15} ← AX_{15}) × cnt			×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. cnt はビット・シフト数です。

4. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまう

ため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（h）ローテート命令

表 4—38 オペレーション一覧（ローテート命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
ROR	A, 1	2	1	—	(CY, A_7 ← A_0, A_{m-1} ← A_m) × 1			×
ROL	A, 1	2	1	—	(CY, A_0 ← A_7, A_{m+1} ← A_m) × 1			×
RORC	A, 1	2	1	—	(CY ← A_0, A_7 ← CY, A_{m-1} ← A_m) × 1			×
ROLC	A, 1	2	1	—	(CY ← A_7, A_0 ← CY, A_{m+1} ← A_m) × 1			×
ROLWC	AX, 1	2	1	—	(CY ← AX_{15}, AX_0 ← CY, AX_{m+1} ← AX_m) × 1			×
	BC, 1	2	1	—	(CY ← BC_{15}, BC_0 ← CY, BC_{m+1} ← BC_m) × 1			×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「（b）外部メモリの内容をデータ・アクセス」を参照してください。

(i) ビット操作命令

表 4—39 オペレーション一覧（ビット操作命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
MOV1	CY, saddr.bit	3	1	—	CY ← (saddr).bit			×
	CY, sfr.bit	3	1	—	CY ← sfr.bit			×
	CY, A.bit	2	1	—	CY ← A.bit			×
	CY, PSW.bit	3	1	—	CY ← PSW.bit			×
	CY, [HL].bit	2	1	4	CY ← (HL).bit			×
	saddr.bit, CY	3	2	—	(saddr).bit ← CY			
	sfr.bit, CY	3	2	—	sfr.bit ← CY			
	A.bit, CY	2	1	—	A.bit ← CY			
	PSW.bit, CY	3	4	—	PSW.bit ← CY	×	×	
	[HL].bit, CY	2	2	—	(HL).bit ← CY			
	CY, ES:[HL].bit	3	2	5	CY ← (ES, HL).bit			×
	ES:[HL].bit, CY	3	3	—	(ES, HL).bit ← CY			
AND1	CY, saddr.bit	3	1	—	CY ← CY ∧ (saddr).bit			×
	CY, sfr.bit	3	1	—	CY ← CY ∧ sfr.bit			×
	CY, A.bit	2	1	—	CY ← CY ∧ A.bit			×
	CY, PSW.bit	3	1	—	CY ← CY ∧ PSW.bit			×
	CY, [HL].bit	2	1	4	CY ← CY ∧ (HL).bit			×
	CY, ES:[HL].bit	3	2	5	CY ← CY ∧ (ES, HL).bit			×
OR1	CY, saddr.bit	3	1	—	CY ← CY ∨ (saddr).bit			×
	CY, sfr.bit	3	1	—	CY ← CY ∨ sfr.bit			×
	CY, A.bit	2	1	—	CY ← CY ∨ A.bit			×
	CY, PSW.bit	3	1	—	CY ← CY ∨ PSW.bit			×
	CY, [HL].bit	2	1	4	CY ← CY ∨ (HL).bit			×
	CY, ES:[HL].bit	3	2	5	CY ← CY ∨ (ES, HL).bit			×
XOR1	CY, saddr.bit	3	1	—	CY ← CY ∀ (saddr).bit			×
	CY, sfr.bit	3	1	—	CY ← CY ∀ sfr.bit			×
	CY, A.bit	2	1	—	CY ← CY ∀ A.bit			×
	CY, PSW.bit	3	1	—	CY ← CY ∀ PSW.bit			×
	CY, [HL].bit	2	1	4	CY ← CY ∀ (HL).bit			×
	CY, ES:[HL].bit	3	2	5	CY ← CY ∀ (ES, HL).bit			×

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
SET1	saddr.bit	3	2	—	(saddr).bit ← 1			
	sfr.bit	3	2	—	sfr.bit ← 1			
	A.bit	2	1	—	A.bit ← 1			
	!addr16.bit	4	2	—	(addr16).bit ← 1			
	PSW.bit	3	4	—	PSW.bit ← 1	×	×	×
	[HL].bit	2	2	—	(HL).bit ← 1			
	ES:!addr16.bit	5	3	—	(ES, addr16).bit ← 1			
	ES:[HL].bit	3	3	—	(ES, HL).bit ← 1			
	CY	2	1	—	CY ← 1			1
CLR1	saddr.bit	3	2	—	(saddr).bit ← 0			
	sfr.bit	3	2	—	sfr.bit ← 0			
	A.bit	2	1	—	A.bit ← 0			
	!addr16.bit	4	2	—	(addr16).bit ← 0			
	PSW.bit	3	4	—	PSW.bit ← 0	×	×	×
	[HL].bit	2	2	—	(HL).bit ← 0			
	ES:!addr16.bit	5	3	—	(ES, addr16).bit ← 0			
	ES:[HL].bit	3	3	—	(ES, HL).bit ← 0			
	CY	2	1	—	CY ← 0			0
NOT1	CY	2	1	—	CY ← $\overline{CY}$			×

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(j) コール・リターン命令

表 4—40 オペレーション一覧（コール・リターン命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
CALL	rp	2	3	—	(SP - 2) ← (PC + 2)S, (SP - 3) ← (PC + 2)H, (SP - 4) ← (PC + 2)L, PC ← CS, rp, SP ← SP - 4			
	$!addr20	3	3	—	(SP - 2) ← (PC + 3)S, (SP - 3) ← (PC + 3)H, (SP - 4) ← (PC + 3)L, PC ← PC + 3 + jdisp16, SP ← SP - 4			
	!addr16	3	3	—	(SP - 2) ← (PC + 3)S, (SP - 3) ← (PC + 3)H, (SP - 4) ← (PC + 3)L, PC ← 0000, addr16, SP ← SP - 4			
	!!addr20	4	3	—	(SP - 2) ← (PC + 4)S, (SP - 3) ← (PC + 4)H, (SP - 4) ← (PC + 4)L, PC ← addr20, SP ← SP - 4			
CALLT	[addr5]	2	5	—	(SP - 2) ← (PC + 2)S, (SP - 3) ← (PC + 2)H, (SP - 4) ← (PC + 2)L, PCS ← 0000, PCH ← (000000000000, addr5 + 1), PCL ← (000000000000, addr5), SP ← SP - 4			
BRK	—	2	5	—	(SP - 1) ← PSW, (SP - 2) ← (PC + 2)S, (SP - 3) ← (PC + 2)H, (SP - 4) ← (PC + 2)L, PCS ← 0000, PCH ← (0007FH), PCL ← (0007EH), SP ← SP - 4, IE ← 0			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
RET	−	1	6	−	$PC_L \leftarrow (SP)$, $PC_H \leftarrow (SP + 1)$, $PC_S \leftarrow (SP + 2)$, $SP \leftarrow SP + 4$			
RETI	−	2	6	−	$PC_L \leftarrow (SP)$, $PC_H \leftarrow (SP + 1)$, $PC_S \leftarrow (SP + 2)$, $PSW \leftarrow (SP + 3)$, $SP \leftarrow SP + 4$	R	R	R
RETB	−	2	6	−	$PC_L \leftarrow (SP)$, $PC_H \leftarrow (SP + 1)$, $PC_S \leftarrow (SP + 2)$, $PSW \leftarrow (SP + 3)$, $SP \leftarrow SP + 4$	R	R	R

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(k) スタック操作命令

表 4—41 オペレーション一覧（スタック操作命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
PUSH	PSW	2	1	—	(SP - 1) ← PSW, (SP - 2) ← 00H, SP ← SP - 2			
	rp	1	1	—	(SP - 1) ← rp_H, (SP - 2) ← rp_L, SP ← SP - 2			
POP	PSW	2	3	—	PSW ← (SP + 1), SP ← SP + 2	R	R	R
	rp	1	1	—	rp_L ← (SP), rp_H ← (SP + 1), SP ← SP + 2			
MOVW	SP, #word	4	1	—	SP ← word			
	SP, AX	2	1	—	SP ← AX			
	AX, SP	2	1	—	AX ← SP			
	HL, SP	3	1	—	HL ← SP			
	BC, SP	3	1	—	BC ← SP			
	DE, SP	3	1	—	DE ← SP			
ADDW	SP, #byte	2	1	—	SP ← SP + byte			
SUBW	SP, #byte	2	1	—	SP ← SP - byte			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまう

ため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

(I) 無条件分岐命令

表 4—42 オペレーション一覧（無条件分岐命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注3		
			注1	注2		Z	AC	CY
BR	AX	2	3	—	PC ← CS, AX			
	$addr20	2	3	—	PC ← PC + 2 + jdisp8			
	$!addr20	3	3	—	PC ← PC + 3 + jdisp16			
	!addr16	3	3	—	PC ← 0000, addr16			
	!!addr20	4	3	—	PC ← addr20			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（m）条件付き分岐命令

表 4—43 オペレーション一覧（条件付き分岐命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
BC	$addr20	2	2/4 注3	—	PC ← PC + 2 + jdisp8 if CY = 1			
BNC	$addr20	2	2/4 注3	—	PC ← PC + 2 + jdisp8 if CY = 0			
BZ	$addr20	2	2/4 注3	—	PC ← PC + 2 + jdisp8 if Z = 1			
BNZ	$addr20	2	2/4 注3	—	PC ← PC + 2 + jdisp8 if Z = 0			
BH	$addr20	3	2/4 注3	—	PC ← PC + 3 + jdisp8 if (Z ∨ CY) = 0			
BNH	$addr20	3	2/4 注3	—	PC ← PC + 3 + jdisp8 if (Z ∨ CY) = 1			
BT	saddr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if (saddr).bit = 1			
	sfr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if sfr.bit = 1			
	A.bit, $addr20	3	3/5 注3	—	PC ← PC + 3 + jdisp8 if A.bit = 1			
	PSW.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if PSW.bit = 1			
	[HL].bit, $addr20	3	3/5 注3	6/8	PC ← PC + 3 + jdisp8 if (HL).bit = 1			
	ES:[HL].bit, $addr20	4	4/6 注3	7/9	PC ← PC + 4 + jdisp8 if (ES, HL).bit = 1			
BF	saddr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if (saddr).bit = 0			
	sfr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if sfr.bit = 0			
	A.bit, $addr20	3	3/5 注3	—	PC ← PC + 3 + jdisp8 if A.bit = 0			
	PSW.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if PSW.bit = 0			
	[HL].bit, $addr20	3	3/5 注3	6/8	PC ← PC + 3 + jdisp8 if (HL).bit = 0			
	ES:[HL].bit, $addr20	4	4/6 注3	7/9	PC ← PC + 4 + jdisp8 if (ES, HL).bit = 0			

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注4		
			注1	注2		Z	AC	CY
BTCLR	saddr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if (saddr).bit = 1 then reset (saddr).bit			
	sfr.bit, $addr20	4	3/5 注3	—	PC ← PC + 4 + jdisp8 if sfr.bit = 1 then reset sfr.bit			
	A.bit, $addr20	3	3/5 注3	—	PC ← PC + 3 + jdisp8 if A.bit = 1 then reset A.bit			
	PSW.bit, $addr20	4	5/7 注3	—	PC ← PC + 4 + jdisp8 if PSW.bit = 1 then reset PSW.bit	×	×	×
	[HL].bit, $addr20	3	3/5 注3	—	PC ← PC + 3 + jdisp8 if (HL).bit = 1 then reset (HL).bit			
	ES:[HL].bit, $addr20	4	4/6 注3	—	PC ← PC + 4 + jdisp8 if (ES, HL).bit = 1 then reset (ES, HL).bit			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. クロック数は“条件不成立時／条件成立時”を表しています。

4. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（n）条件付きステップ命令

表 4—44 オペレーション一覧（条件付きステップ命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注 3		
			注 1	注 2		Z	AC	CY
SKC	—	2	1	—	Next instruction skip if CY = 1			
SKNC	—	2	1	—	Next instruction skip if CY = 0			
SKZ	—	2	1	—	Next instruction skip if Z = 1			
SKNZ	—	2	1	—	Next instruction skip if Z = 0			
SKH	—	2	1	—	Next instruction skip if (Z ∨ CY) = 0			
SKNH	—	2	1	—	Next instruction skip if (Z ∨ CY) = 1			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ： 変化なし
0 ： 0 にクリアされる
1 ： 1 にセットされる
× ： 結果にしたがってセット／リセットされる
R ： 以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

（o）CPU 制御命令

表 4—45 オペレーション一覧（CPU 制御命令）

ニモニック	オペランド	バイト	クロック		オペレーション	フラグ注 3		
			注 1	注 2		Z	AC	CY
SEL	RBn	2	1	—	RBS[1:0] ← n			
NOP	—	1	1	—	No Operation			
EI	—	3	4	—	IE ← 1 (Enable Interrupt)			
DI	—	3	4	—	IE ← 0 (Disable Interrupt)			
HALT	—	2	3	—	Set HALT Mode			
STOP	—	2	3	—	Set STOP Mode			

注 1. 内部 RAM 領域，SFR 領域をアクセスしたとき，またはデータ・アクセスをしない命令のとき。

2. プログラム・メモリ領域をアクセスしたとき。

3. フラグ欄の記号は，その命令実行時のフラグの変化を表しています。

空欄 ：変化なし
0 ：0 にクリアされる
1 ：1 にセットされる
× ：結果にしたがってセット／リセットされる
R ：以前に退避した値がリストアされる

備考 1. 命令の 1 クロックはプロセッサ・クロック・コントロール・レジスタ（PCC）で選択した CPU クロック（fCLK）の 1 クロック分です。

2. クロック数は内部 ROM（フラッシュ・メモリ）領域にプログラムがある場合です。

3. n はレジスタ・バンク数です（n = 0-3）。

4. 外部メモリ領域が内蔵フラッシュ領域と接している製品で，外部バス・インタフェース機能使用時には，フラッシュの最終アドレス（最大 16 バイト）に配置された命令の実行クロック数に，ウエイト数が加算されます。これは，命令コードの先読みを行う際にフラッシュ空間を越え外部メモリ空間をアクセスしてしまうため，外部メモリのウエイトが入ってしまうためです。ウエイト数は「(b) 外部メモリの内容をデータ・アクセス」を参照してください。

4. 6. 6 命令の説明

ここでは，RL78 ファミリ，78K0R マイクロコントローラ製品の命令を説明します。

表 4—46 アセンブリ言語命令一覧

機能	命令
8 ビット・データ転送命令	MOV，XCH，ONEB，CLRB，MOVS
16 ビット・データ転送命令	MOVW，XCHW，ONEW，CLRW
8 ビット演算命令	ADD，ADDC，SUB，SUBC，AND，OR，XOR，CMP，CMP0，CMPS
16 ビット演算命令	ADDW，SUBW，CMPW
乗除積和算命令	MULU，MULHU，MULH，DIVHU，DIVWU，MACHU，MACH
増減命令	INC，DEC，INCW，DECW
シフト命令	SHR，SHRW，SHL，SHLW，SAR，SARW
ローテート命令	ROR，ROL，RORC，ROLC，ROLWC
ビット操作命令	MOV1，AND1，OR1，XOR1，SET1，CLR1，NOT1
コール・リターン命令	CALL，CALLT，BRK，RET，RETI，RETB
スタック操作命令	PUSH，POP，MOVW，ADDW，SUBW
無条件分岐命令	BR
条件付き分岐命令	BC，BNC，BZ，BNZ，BH，BNH，BT，BF，BTCLR
条件付きスキップ命令	SKC，SKNC，SKZ，SKNZ，SKH，SKNH
CPU 制御命令	SEL，NOP，EI，DI，HALT，STOP

個々の命令について，次の内容を説明します。

[命令形式]

命令の基本記述形式を示します。

[オペレーション]

命令のオペレーションを略号を用いて示します。

[オペランド]

この命令で指定できるオペランドを示します。各オペランドの略号の説明は，「(2) オペレーション欄の記号」を参照してください。

[フラグ]

命令実行により変化するフラグの動作を示します。

各フラグの動作記号を凡例に示します。

記号	解説
ブランク	変化なし
0	0 にクリアされる
1	1 にセットされる
×	結果に従ってセットまたはクリアされる
R	以前に退避した値がリストアされる

[説明]

命令のオペレーションの詳細を解説します。

[記述例]

命令の記述例を示します。

(1) 8 ビット・データ転送命令

8 ビット・データ転送命令には，次の命令があります。

命令	概要
MOV	バイト・データの転送
XCH	バイト・データの交換
ONEB	バイト・データの 01H セット
CLRB	バイト・データのクリア
MOVS	バイト・データの転送と PSW 変化

MOV

バイト・データの転送を行います。

[命令形式]

MOV dst, src

[オペレーション]

dst ← src

[オペランド]

オペランド（dst, src）
r, #byte
saddr, #byte
sfr, #byte
!addr16, #byte
A, r 注
r, A 注
A, saddr
saddr, A
A, sfr
sfr, A
A, !addr16
!addr16, A
PSW, #byte
A, PSW
PSW, A
ES, #byte
ES, saddr
A, ES
ES, A
CS, #byte
A, CS
CS, A
A, [DE]
[DE], A

オペランド（dst, src）
[DE+byte], #byte
A, [DE+byte]
[DE+byte], A
A, [HL]
[HL], A
[HL+byte], #byte
A, [HL+byte]
[HL+byte], A
A, [HL+B]
[HL+B], A
A, [HL+C]
[HL+C], A
word[B], #byte
A, word[B]
word[B], A
word[C], #byte
A, word[C]
word[C], A
word[BC], #byte
A, word[BC]
word[BC], A
[SP+byte], #byte
A, [SP+byte]
[SP+byte], A
B, saddr
B, !addr16
C, saddr
C, !addr16
X, saddr
X, !addr16
ES:!addr16, #byte
A, ES:!addr16
ES:!addr16, A
A, ES:[DE]
ES:[DE], A
ES:[DE+byte], #byte
A, ES:[DE+byte]

オペランド（dst, src）
ES:[DE+byte], A
A, ES:[HL]
ES:[HL], A
ES:[HL+byte], #byte
A, ES:[HL+byte]
ES:[HL+byte], A
A, ES:[HL+B]
ES:[HL+B], A
A, ES:[HL+C]
ES:[HL+C], A
ES:word[B], #byte
A, ES:word[B]
ES:word[B], A
ES:word[C], #byte
A, ES:word[C]
ES:word[C], A
ES:word[BC], #byte
A, ES:word[BC]
ES:word[BC], A
B, ES:!addr16
C, ES:!addr16
X, ES:!addr16

注　r = A を除く。

[フラグ]

（1）PSW, #byte と PSW, A のオペランドの場合

Z	AC	CY
×	×	×

×　　：結果にしたがってセット／リセットされる

（2）上記以外

Z	AC	CY

空欄　：変化なし

［説明］

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に，第 2 オペランドで指定されるソース・オペランド（src）の内容を転送します。
- MOV PSW, #byte 命令，MOV PSW, A 命令と次に続く命令の間では，すべての割り込みを受け付けません。

［記述例］

```
MOV     A, #4DH         ; （1）
```

（1）A レジスタに 4DH を転送します。

XCH

バイト・データの交換を行います。

[命令形式]

XCH dst, src

[オペレーション]

dst ←→ src

[オペランド]

オペランド（dst, src）
A, r 注
A, saddr
A, sfr
A, !addr16
A, [DE]
A, [DE+byte]
A, [HL]
A, [HL + byte]
A, [HL + B]
A, [HL + C]
A, ES:!addr16
A, ES:[DE]
A, ES:[DE+byte]
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注　r = A を除く。

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- 第 1 オペランドと第 2 オペランドの内容を交換します。

[記述例]

```
XCH     A, FFEBCH       ; (1)
```

(1) A レジスタの内容と FFEBCH 番地の内容を交換します。

ONEB

バイト・データの 01H セットを行います。

[命令形式]

ONEB dst

[オペレーション]

dst ← 01H

[オペランド]

オペランド（dst）
A
X
B
C
saddr
!addr16
ES:!addr16

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に 01H を転送します。

[記述例]

```
ONEB    A       ; (1)
```

(1) A レジスタに 01H を転送します。

CLRB

バイト・データのクリアを行います。

[命令形式]

CLRB dst

[オペレーション]

dst ← 00H

[オペランド]

オペランド（dst）
A
X
B
C
saddr
!addr16
ES:!addr16

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に 00H を転送します。

[記述例]

```
CLRB    A       ; (1)
```

(1) A レジスタに 00H を転送します。

MOVS

バイト・データの転送と PSW 変化を行います。

[命令形式]

MOVS dst, src

[オペレーション]

dst ← src

[オペランド]

オペランド（dst, src）
[HL+byte], X
ES:[HL+byte], X

[フラグ]

Z	AC	CY
×		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に，第 2 オペランドで指定されるソース・オペランドの内容を転送します。
- src の値が 0 の場合，Z フラグがセット（1），その他の場合は Z フラグはクリア（0）されます。
- A レジスタの値が 0，または src の値が 0 であった場合，CY フラグがセット（1），その他の場合は CY フラグはクリア（0）されます。

[記述例]

```
MOVS    [HL+2H], X      ; (1)
```

（1）HL = FE00H, X = 55H, A = 0H の場合，X の 55H を FE02H 番地に格納します。
Z フラグ = 0，CY フラグ = 1（A レジスタ 0 のため）

（2）16 ビット・データ転送命令

16 ビット・データ転送命令には，次の命令があります。

命令	概要
MOVW	ワード・データの転送
XCHW	ワード・データの交換
ONEW	ワード・データの 0001H セット
CLRW	ワード・データのクリア

MOVW

ワード・データの転送を行います。

[命令形式]

MOVW dst, src

[オペレーション]

dst ← src

[オペランド]

オペランド（dst, src）
rp, #word
saddrp, #word
sfrp, #word
AX, saddrp
saddrp, AX
AX, sfrp
sfrp, AX
AX, rp 注
rp, AX 注
AX, !addr16
!addr16, AX
AX, [DE]
[DE], AX
AX, [DE+byte]
[DE+byte], AX
AX, [HL]
[HL], AX
AX, [HL+byte]
[HL+byte], AX
AX, word[B]
word[B], AX
AX, word[C]
word[C], AX
AX, word[BC]

オペランド（dst, src）
word[BC], AX
AX, [SP+byte]
[SP+byte], AX
BC, saddrp
BC, !addr16
DE, saddrp
DE, !addr16
HL, saddrp
HL, !addr16
AX, ES:!addr16
ES:!addr16, AX
AX, ES:[DE]
ES:[DE], AX
AX, ES:[DE+byte]
ES:[DE+byte], AX
AX, ES:[HL]
ES:[HL], AX
AX, ES:[HL+byte]
ES:[HL+byte], AX
AX, ES:word[B]
ES:word[B], AX
AX, ES:word[C]
ES:word[C], AX
AX, ES:word[BC]
ES:word[BC], AX
BC, ES:!addr16
DE, ES:!addr16
HL, ES:!addr16

注 rp = BC，DE，HL のときのみ。

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に，第 2 オペランドで指定されるソース・オペランド（src）の内容を転送します。

[記述例]

```
MOVW    AX, HL          ; (1)
```

(1) HL レジスタの内容を AX レジスタに転送します。

[注意]

- 偶数アドレスのみ指定できます。奇数アドレスは指定できません。

XCHW

ワード・データの交換を行います。

[命令形式]

XCHW dst, src

[オペレーション]

dst ←→ src

[オペランド]

オペランド（dst, src）
AX, rp 注

注 rp = BC，DE，HL のときのみ。

[フラグ]

Z	AC	CY

空欄 ：変化なし

[説明]

- 第 1 オペランドと第 2 オペランドの内容を交換します。

[記述例]

```
XCHW    AX, BC          ; (1)
```

(1) AX レジスタと BC レジスタの内容を交換します。

ONEW

ワード・データの 0001H セットを行います。

[命令形式]

ONEW dst

[オペレーション]

dst ← 0001H

[オペランド]

オペランド（dst）
AX
BC

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）にに 0001H を転送します。

[記述例]

```
ONEW    AX      ; (1)
```

(1) A レジスタに 0001H を転送します。

CLRW

ワード・データのクリアを行います。

[命令形式]

CLRW dst

[オペレーション]

dst ← 0000H

[オペランド]

オペランド（dst）
AX
BC

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に 0000H を転送します。

[記述例]

```
CLRW    AX      ; (1)
```

(1) A レジスタに 0000H を転送します。

(3) 8 ビット演算命令

8 ビット演算命令には，次の命令があります。

命令	概要
ADD	バイト・データの加算
ADDC	キャリーを含むバイト・データの加算
SUB	バイト・データの減算
SUBC	キャリーを含むバイト・データの減算
AND	バイト・データの論理積
OR	バイト・データの論理和
XOR	バイト・データの排他的論理和
CMP	バイト・データの比較
CMP0	バイト・データの 0 比較
CMPS	バイト・データの比較

ADD

バイト・データの加算を行います。

[命令形式]

ADD dst, src

[オペレーション]

dst, CY ← dst + src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注　r = A を除く。

[フラグ]

Z	AC	CY
×	×	×

×　：結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）を加算し，その結果を CY フラグとデスティネーション・オペランド（dst）に格納します。
- 加算の結果，dst が 0 になった場合，Z フラグがセット（1），その他の場合は Z フラグはクリア（0）されます。
- 加算の結果，ビット 7 からのキャリーが発生した場合は，CY フラグはセット（1），その他の場合は CY フラグはクリア（0）されます。
- 加算の結果，ビット 3 からビット 4 へのキャリーが発生した場合は，AC フラグはセット（1），その他の場合は AC フラグはクリア（0）されます。

[記述例]

```
ADD     CR10, #56H      ; (1)
```

(1) CR10 レジスタに 56H を加算し，結果を CR10 レジスタに格納します。

ADDC

キャリーを含むバイト・データの加算を行います。

[命令形式]

ADDC dst, src

[オペレーション]

dst, CY ← dst + src + CY

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 r = A を除く。

[フラグ]

Z	AC	CY
×	×	×

× : 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）と CY フラグを加算して，結果をデスティネーション・オペランド（dst）と CY フラグに格納します。
  CY フラグは最下位ビットへ加算されます。
  この命令は，おもに複数バイトの加算を行うときに使用します。
- 加算の結果，dst が 0 になった場合，Z フラグがセット（1），その他の場合は Z フラグはクリア（0）されます。
- 加算の結果，ビット 7 からのキャリーが発生した場合は，CY フラグはセット（1），その他の場合は CY フラグはクリア（0）されます。
- 加算の結果，ビット 3 からビット 4 へのキャリーが発生した場合は，AC フラグはセット（1），その他の場合は AC フラグはクリア（0）されます。

[記述例]

```
ADDC    A, [HL + B]     ; (1)
```

(1) A レジスタと（HL レジスタ +（B レジスタ））番地の内容と CY フラグを加算し，結果を A レジスタに格納します。

SUB

バイト・データの減算を行います。

[命令形式]

SUB dst, src

[オペレーション]

dst, CY ← dst - src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注　r = A を除く。

[フラグ]

Z	AC	CY
×	×	×

×　：結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から第 2 オペランドで指定されるソース・オペランド（src）を減算し，結果をデスティネーション・オペランド（dst）と CY フラグに格納します。
  ソース・オペランド（src）とデスティネーション・オペランド（dst）を同一のものとすることにより，デスティネーション・オペランドの 0 クリアが可能です。
- 減算の結果，dst が 0 なら，Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 減算の結果，ビット 7 でボローが発生した場合，CY フラグはセット（1），その他の場合はクリア（0）されます。
- 減算の結果，ビット 4 からビット 3 へのボローが発生した場合，AC フラグはセット（1），その他の場合はクリア（0）されます。

[記述例]

```
SUB     D, A            ; (1)
```

(1) D レジスタから A レジスタを減算し，結果を D レジスタに格納します。

SUBC

キャリーを含むバイト・データの減算を行います。

[命令形式]

SUBC dst, src

[オペレーション]

dst, CY ← dst - src - CY

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 r = A を除く。

[フラグ]

Z	AC	CY
×	×	×

× : 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から第 2 オペランドで指定されるソース・オペランド（src）と CY フラグを減算し，結果をデスティネーション・オペランド（dst）に格納します。
  CY フラグは最下位ビットから減算します。
  この命令は，主として複数バイトの減算を行うときに使用します。
- 減算の結果，dst が 0 なら，Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 減算の結果，ビット 7 でボローが発生した場合，CY フラグはセット（1），その他の場合はクリア（0）されます。
- 減算の結果，ビット 4 からビット 3 へのボローが発生した場合，AC フラグはセット（1），その他の場合はクリア（0）されます。

[記述例]

```
SUBC    A, [HL]          ; (1)
```

(1) A レジスタから（HL レジスタ）番地の内容と CY フラグを減算し，結果を A レジスタに格納します。

AND

バイト・データの論理積を行います。

[命令形式]

AND dst, src

[オペレーション]

dst ← dst ∧ src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 r = A を除く。

[フラグ]

Z	AC	CY
×		

空欄 ： 変化なし

× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）のビットごとの論理積をとり，結果をデスティネーション・オペランド（dst）に格納します。
- 論理積をとった結果，全ビットが 0 であれば Z フラグはセット（1），その他の場合は，Z フラグはクリア（0）されます。

[記述例]

```
AND     FFEBAH, #11011100B      ; (1)
```

（1）FFEBAH の内容と 11011100B のビットごとの論理積をとり，結果を FFEBAH に格納します。

OR

バイト・データの論理和を行います。

[命令形式]

OR dst, src

[オペレーション]

dst ← dst ∨ src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 r = A を除く。

[フラグ]

Z	AC	CY
×		

空欄 ： 変化なし

× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）のビットごとの論理和をとり，結果をデスティネーション・オペランド（dst）に格納します。
- 論理和をとった結果，全ビットが 0 であれば Z フラグはセット（1），その他の場合は，Z フラグはクリア（0）されます。

[記述例]

```
OR      A, FFE98H       ; (1)
```

(1) A レジスタと FFE98H のビットごとの論理和をとり，結果を A レジスタに格納します。

XOR

バイト・データの排他的論理和を行います。

[命令形式]

XOR dst, src

[オペレーション]

dst ← dst ∀ src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

注 r = A を除く。

[フラグ]

Z	AC	CY
×		

空欄 ： 変化なし

× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）のビットごとの排他的論理和をとり，結果をデスティネーション・オペランド（dst）に格納します。
  この命令でソース・オペランド（src）に #0FFH を選択することにより，デスティネーション・オペランド（dst）の全ビットの論理否定がとれます。
- 排他的論理和の結果，全ビットが 0 であれば Z フラグはセット（1），その他の場合はクリア（0）されます。

[記述例]

```
XOR     A, L            ; (1)
```

(1) A レジスタと L レジスタのビットごとの排他的論理和をとり，結果を A レジスタに格納します。

CMP

バイト・データの比較を行います。

[命令形式]

CMP dst, src

[オペレーション]

dst - src

[オペランド]

オペランド（dst, src）
A, #byte
saddr, #byte
A, r 注
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
!addr16, #byte
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]
ES:!addr16, #byte

注　r = A を除く。

[フラグ]

Z	AC	CY
×	×	×

× : 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から第 2 オペランドで指定されるソース・オペランド（src）を減算します。
  減算の結果はどこへも格納せずに Z，AC，CY の各フラグだけを変化させます。
- 減算の結果，0 なら Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 減算の結果，ビット 7 でボローが発生した場合，CY フラグはセット（1），その他の場合はクリア（0）されます。
- 減算の結果，ビット 4 からビット 3 へのボローが発生した場合，AC フラグはセット（1），その他の場合はクリア（0）されます。

[記述例]

```
CMP     FFE38H, #38H            ; (1)
```

(1) FFE38H 番地の内容から 38H を減算し，フラグだけを変化させます。
(FFE38H 番地の内容とイミーディエト・データの比較)

CMP0

バイト・データの 0 比較を行います。

[命令形式]

CMP0 dst

[オペレーション]

dst - 00H

[オペランド]

オペランド（dst）
A
X
B
C
saddr
!addr16
ES:!addr16

[フラグ]

Z	AC	CY
×	×	×

× ：結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から 00H を減算します。
- 減算結果はどこへも格納せずに Z, AC, CY の各フラグだけ変化させます。
- dst の値が 00H であった場合，Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- AC, CY フラグは常にクリア（0）されます。

［記述例］

```
CMP0    A               ; (1)
```

(1) A レジスタの内容が 0 であった場合は Z フラグがセットされます。

CMPS

バイト・データの比較を行います。

[命令形式]

CMPS dst, src

[オペレーション]

dst - src

[オペランド]

オペランド（dst, src）
X, [HL+byte]
X, ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	×

× : 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（[dst]）から，第 2 オペランドで指定されるソース・オペランド（src）を減算します。
  減算の結果はどこへも格納せず Z, AC, CY の各フラグだけを変化させます。
- 減算の結果，0 なら Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 演算の結果が 0 以外，または A レジスタの値が 0，または dst の値が 0 であった場合，CY フラグがセット（1），その他の場合は CY フラグはクリア（0）されます。
- 減算の結果，ビット 4 からビット 3 へのボローが発生した場合，AC フラグはセット（1），その他の場合はクリア（0）されます。

［記述例］

```
CMPS    X, [HL+F0H]     ; (1)
```

(1) HL = FD12H の場合，X の値と FFE02H 番地の内容を比較し，同じ値であった場合は Z フラグをセットし，X の値と FFE02H 番地の内容を比較し，違う値であった場合は CY フラグをセットします。
A レジスタの値が 0 だった場合は CY フラグをセット，X レジスタの値が 0 だった場合は CY フラグをセット，AC フラグは CMP 命令と同様にビット 4 からビット 3 へのボローによってセットされます。

（4）16 ビット演算命令

16 ビット演算命令には，次の命令があります。

命令	概要
ADDW	ワード・データの加算
SUBW	ワード・データの減算
CMPW	ワード・データの比較

ADDW

ワード・データの加算を行います。

[命令形式]

ADDW dst, src

[オペレーション]

dst, CY ← dst + src

[オペランド]

オペランド（dst, src）
AX, #word
AX, AX
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	×

×　　：結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）と第 2 オペランドで指定されるソース・オペランド（src）の加算を行い，結果をデスティネーション・オペランド（dst）に格納します。
- 加算の結果，dst が 0 になった場合，Z フラグがセット（1），その他の場合は Z フラグはクリア（0）されます。
- 加算の結果，ビット 15 からのキャリーが発生した場合は，CY フラグはセット（1），その他の場合は CY フラグはクリア（0）されます。
- 加算の結果，AC フラグは不定となります。

[記述例]

```
ADDW    AX, #ABCDH      ; (1)
```

(1) AX レジスタと ABCDH を加算し，結果を AX レジスタに格納します。

SUBW

ワード・データの減算を行います。

[命令形式]

SUBW dst, src

[オペレーション]

dst, CY ← dst - src

[オペランド]

オペランド（dst, src）
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	×

× : 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から第 2 オペランドで指定されるソース・オペランド（src）を減算し，結果をデスティネーション・オペランド（dst）と CY フラグに格納します。
- 減算の結果，dst が 0 なら Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 減算の結果，ビット 15 でボローが発生した場合，CY フラグはセット（1），その他の場合はクリア（0）されます。
- 減算の結果，AC フラグは不定となります。

［記述例］

```
SUBW    AX, #ABCDH      ; (1)
```

(1) AX レジスタの内容から ABCDH を減算し，結果を AX レジスタに格納します。

CMPW

ワード・データの比較を行います。

[命令形式]

CMPW dst, src

[オペレーション]

dst - src

[オペランド]

オペランド（dst, src）
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	×

×　：結果にしたがってセット／リセットされる

［説明］

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）から第 2 オペランドで指定されるソース・オペランド（src）を減算します。
  減算の結果はどこへも格納せずに Z，AC，CY の各フラグだけを変化させます。
- 減算の結果，0 なら Z フラグはセット（1），その他の場合は Z フラグはクリア（0）されます。
- 減算の結果，ビット 15 でボローが発生した場合，CY フラグはセット（1），その他の場合はクリア（0）されます。
- 減算の結果，AC フラグは不定となります。

［記述例］

```
CMPW    AX, #ABCDH              ; (1)
```

（1）AX レジスタから ABCDH を減算し，フラグだけを変化させます。
（AX レジスタとイミーディエト・データとの比較）

（5）乗除積和算命令

乗除積和算命令には，次の命令があります。

命令	概要
MULU	データの符号なし乗算
MULHU 注	データの符号なし 16 ビット乗算
MULH 注	データの符号付き 16 ビット乗算
DIVHU 注	データの符号なし 16 ビット除算
DIVWU 注	データの符号なし 32 ビット除算
MACHU 注	データの符号なし積和算
MACH 注	データの符号付き積和算

注　拡張命令のため，製品ごとに搭載／非搭載が異なります。各製品のユーザーズ・マニュアルを参照してください。

MULU

データの符号なし乗算を行います。

[命令形式]

MULU src

[オペレーション]

AX ← A × src

[オペランド]

オペランド（src）
X

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- A レジスタの内容とソース・オペランド（src）のデータを符号なしのデータとして乗算し，結果を AX レジスタに格納します。

[記述例]

```
MULU    X       ; (1)
```

(1) A レジスタの内容と X レジスタの内容を乗算し，結果を AX レジスタに格納します。

MULHU

データの符号なし 16 ビット乗算を行います。

[命令形式]

MULHU

[オペレーション]

BCAX ← AX × BC

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- AX レジスタの内容と BC レジスタの内容を符号なしのデータとして乗算し，結果の上位 16 ビットを BC レジスタ，結果の下位 16 ビットを AX レジスタに格納します。

[記述例]

```
MOVW    AX, #0C000H
MOVW    BC, #1000H
MULHU
MOVW    !addr16, AX
MOVW    AX, BC
MOVW    !addr16, AX     ;  (1)
```

(1) C000H と 1000H を乗算し，結果の C000000H を !addr16 で示すメモリに格納します。

MULH

データの符号付き 16 ビット乗算を行います。

[命令形式]

MULH

[オペレーション]

BCAX ← AX × BC

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- AX レジスタの内容と BC レジスタの内容を符号付きのデータとして乗算し，結果の上位 16 ビットを BC レジスタ，結果の下位 16 ビットを AX レジスタに格納します。

[記述例]

```
MOVW    AX, #0C000H
MOVW    BC, #1000H
MULH
MOVW    !addr16, AX
MOVW    AX, BC
MOVW    !addr16, AX     ;  (1)
```

(1) C000H と 1000H を乗算し，結果の FC000000H を !addr16 で示すメモリに格納します。

DIVHU

データの符号なし 16 ビット除算を行います。

[命令形式]

DIVHU

[オペレーション]

AX（商），DE（余り）← AX ÷ DE

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- AX レジスタの内容を DE レジスタの内容で除算し，商を AX レジスタに，余りを DE レジスタに格納します。
  除算は AX レジスタおよび DE レジスタの内容を符号なしのデータとして行います。
  ただし，DE レジスタの内容が 0 のときは，DE レジスタには AX レジスタの内容が格納され，AX レジスタ = 0FFFFH となります。

［記述例］

```
MOVW    AX, #8081H
MOVW    DE, #0002H
DIVHU
MOVW    !addr16, AX
MOVW    AX, DE
MOVW    !addr16, AX     ;  (1)
```

（1）8081H を 0002H で除算し，AX レジスタの商（4040H）と DE レジスタの余り（0001H）を !addr16 で示すメモリに格納します。

DIVWU

データの符号なし 32 ビット除算を行います。

[命令形式]

DIVWU

[オペレーション]

BCAX（商），HLDE（余り）← BCAX ÷ HLDE

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- BCAX レジスタの内容を HLDE レジスタの内容で除算し，商を BCAX レジスタに，余りを HLDE レジスタに格納します。
  除算は BCAX レジスタおよび HLDE レジスタの内容を符号なしのデータとして行います。
  ただし，HLDE レジスタの内容が 0 のときは，HLDE レジスタには BCAX レジスタの内容が格納され，BCAX レジスタ = 0FFFFFFFFH となります。

［記述例］

```
MOVW    AX, #8081H
MOVW    BC, #8080H
MOVW    DE, #0002H
MOVW    HL, #0000H
DIVWU
MOVW    !addr16, AX
MOVW    AX, BC
MOVW    !addr16, AX
MOVW    AX, DE
MOVW    !addr16, AX
MOVW    AX, HL
MOVW    !addr16, AX     ;  (1)
```

（1）80808081H を 00000002H で除算し，BCAX レジスタの商（40404040H）と HLDE レジスタの余り（00000001H）を !addr16 で示すメモリに格納します。

MACHU

データの符号なし積和演算を行います。

[命令形式]

MACHU

[オペレーション]

MACR ← MACR ＋ AX × BC

[オペランド]

なし

[フラグ]

Z	AC	CY
	×	×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- AX レジスタの内容と BC レジスタの内容を符号なしのデータとして乗算した結果を MACR レジスタと加算を行い，MACR レジスタに格納します。
- 加算の結果，オーバフローが発生した場合は CY フラグがセット（1）され，その他の場合は CY フラグがクリア（0）されます。
- AC フラグは 0 となります。
- MACR レジスタは，積和演算前に初期値を設定してください。また，MACR レジスタは固定であるため，積和演算結果を複数必要とする場合は MACR レジスタを退避して使用してください。

［記述例］

```
MOVW    AX, #00000H
MOVW    !0FFF2H, AX
MOVW    !0FFF0H, AX
MOVW    AX, #0C000H
MOVW    BC, #01000H
MACHU
MOVW    AX, !0FFF2H
MOVW    !addr16, AX
MOVW    AX, !0FFF0H
MOVW    !addr16, AX     ;  (1)
```

（1）AX レジスタの内容と BC レジスタの内容を乗算し，MACR レジスタの内容と加算した結果を MACR レジスタに格納します。

MACH

データの符号付き積和演算を行います。

[命令形式]

MACH

[オペレーション]

MACR ← MACR ＋ AX × BC

[オペランド]

なし

[フラグ]

Z	AC	CY
	×	×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- AX レジスタの内容と BC レジスタの内容を符号付きのデータとして乗算した結果を MACR レジスタと加算を行い，MACR レジスタに格納します。
- 加算の結果，オーバフローが発生した場合は CY フラグがセット（1）され，その他の場合は CY フラグがクリア（0）されます。オーバフローとは，正の累計値に正の積を加算した結果が 7FFFFFFFH を越えた 場合と，負の累計値に負の積を加算した結果が 80000000H を越えた 場合となります。
- 演算の結果，MACR レジスタの値が正の場合は AC フラグがクリア（0），負の場合は AC フラグがセット（1）されます。
- MACR レジスタは，積和演算前に初期値を設定してください。また，MACR レジスタは固定であるため，積和演算結果を複数必要とする場合は MACR レジスタを退避して使用してください。

［記述例］

```
MOVW    AX, #00000H
MOVW    !0FFF0H, AX
MOVW    AX, #08000H
MOVW    !0FFF2H, AX
MOVW    AX, #00001H
MOVW    !0FFF0H, AX
MOVW    AX, #07FFFH
MOVW    BC, #0FFFFH
MACH
MOVW    AX, !0FFF2H
MOVW    !addr16, AX
MOVW    AX, !0FFF0H
MOVW    !addr16, AX     ;  (1)
```

（1）AX レジスタの内容と BC レジスタの内容を乗算し，MACR レジスタの内容と加算した結果を MACR レジスタに格納します。

（6）増減命令

増減命令には，次の命令があります。

命令	概要
INC	バイト・データのインクリメント
DEC	バイト・データのデクリメント
INCW	ワード・データのインクリメント
DECW	ワード・データのデクリメント

INC

バイト・データのインクリメントを行います。

[命令形式]

INC dst

[オペレーション]

dst ← dst + 1

[オペランド]

オペランド（src）
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	

空欄　：変化なし
×　　：結果にしたがってセット／リセットされる

[説明]

- デスティネーション・オペランド（dst）の内容を 1 だけインクリメントします。
- インクリメントした結果が 0 になれば Z フラグはセット（1），その他の場合はクリア（0）されます。
- インクリメントした結果，ビット 3 からビット 4 へのキャリーがあれば，AC フラグはセット（1），その他の場合はクリア（0）されます。
- 繰り返し処理のカウンタやインデクスト・アドレッシングのオフセット・レジスタのインクリメントに使用することが多いため，CY フラグの内容は変化させません（複数バイトの演算時に，CY フラグの内容を保持させるため）。

［記述例］

```
INC     B               ; (1)
```

(1) B レジスタをインクリメントします。

DEC

バイト・データのデクリメントを行います。

[命令形式]

DEC dst

[オペレーション]

dst ← dst - 1

[オペランド]

オペランド（src）
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[フラグ]

Z	AC	CY
×	×	

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- デスティネーション・オペランド（dst）の内容を 1 だけデクリメントします。
- デクリメントした結果が 0 であれば，Z フラグはセット（1），その他の場合はクリア（0）されます。
- デクリメントした結果がビット 4 からビット 3 へのキャリーがあれば，AC フラグはセット（1），その他の場合はクリア（0）されます。
- 繰り返し処理のカウンタに使用することが多いため，CY フラグの内容は変化させません（複数バイトの演算時に CY フラグを保持させるため）。
- dst が B レジスタ，C レジスタ，または saddr の場合で AC，CY の各フラグを変化させたくない場合，DBNZ 命令を使用できます。

［記述例］

```
DEC     FFE92H          ; (1)
```

(1) FFE92H 番地の内容をデクリメントします。

INCW

ワード・データのインクリメントを行います。

[命令形式]

INCW dst

[オペレーション]

dst ← dst + 1

[オペランド]

オペランド（src）
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- デスティネーション・オペランド（dst）の内容を 1 だけインクリメントします。
- レジスタを使用するアドレッシングで，使用するレジスタ（ポインタ）のインクリメントに使用することが多いため，Z，AC，CY の各フラグを変化させません。

[記述例]

```
INCW    HL              ; (1)
```

(1) HL レジスタをインクリメントします。

DECW

ワード・データのデクリメントを行います。

[命令形式]

DECW dst

[オペレーション]

dst ← dst - 1

[オペランド]

オペランド（src）
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- デスティネーション・オペランド（dst）の内容を 1 だけデクリメントします。
- レジスタを使用するアドレッシングで，使用するレジスタ（ポインタ）のデクリメントに使用することが多いため，Z，AC，CY の各フラグを変化させません。

[記述例]

```
DECW    DE              ; (1)
```

(1) DE レジスタをデクリメントします。

（7）シフト命令

シフト命令には，次の命令があります。

命令	概要
SHR	右方向の論理シフト
SHRW	右方向の論理シフト
SHL	左方向の論理シフト
SHLW	左方向の論理シフト
SAR	右方向の算術シフト
SARW	右方向の算術シフト

SHR

右方向の論理シフトを行います。

[命令形式]

SHR dst, cnt

[オペレーション]

(CY ← dst_0, dst_{m-1} ← dst_m, dst_7 ← 0) × cnt

[オペランド]

オペランド（dst, cnt）
A, cnt

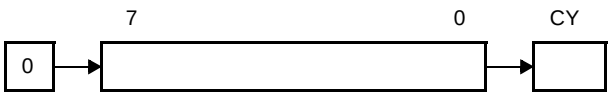
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回右にシフトします。
- 上位ビットには 0 が埋め込まれ，CY へは最後にビット 0 からシフトした値が入ります。
- cnt は 1-7 を指定できます。



［記述例］

```
SHR     A, 3    ; (1)
```

(1) A レジスタの内容が F5H であった場合の結果は，A = 1EH，CY = 1 となります。

```
A = F5H  CY = 0
A = 7AH  CY = 1  1 回
A = 3DH  CY = 0  2 回
A = 1EH  CY = 1  3 回
```

SHRW

右方向の論理シフトを行います。

[命令形式]

SHRW dst, cnt

[オペレーション]

(CY ← dst_0, dst_{m-1} ← dst_m, dst_{15} ← 0) × cnt

[オペランド]

オペランド（dst, cnt）
AX, cnt

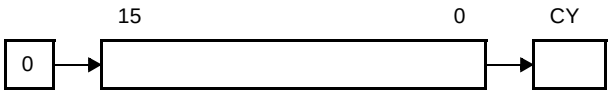
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回右にシフトします。
- 上位ビットには 0 が埋め込まれ，CY へは最後にビット 0 からシフトした値が入ります。
- cnt は 1-15 を指定できます。



［記述例］

```
SHRW    AX, 3   ; (1)
```

(1) AX レジスタの内容が AAF5H であった場合の結果は，AX = 155EH，CY = 1 となります。

```
AX = AAF5H  CY = 0
AX = 557AH  CY = 1  1回
AX = 2ABDH  CY = 0  2回
AX = 155EH  CY = 1  3回
```

SHL

左方向の論理シフトを行います。

[命令形式]

SHL dst, cnt

[オペレーション]

(CY ← dst_7, dst_m ← dst_{m-1}, dst_0 ← 0) × cnt

[オペランド]

オペランド（dst, cnt）
A, cnt
B, cnt
C, cnt

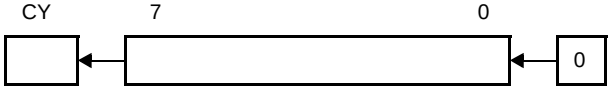
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回左にシフトします。
- 下位ビットには 0 が埋め込まれ，CY へは最後にビット 7 からシフトした値が入ります。
- cnt は 1-7 を指定できます。



［記述例］

```
SHL     A, 3    ; (1)
```

(1) A レジスタの内容が 5DH であった場合の結果は，A = E8H，CY = 0 となります。

```
A = 5DH  CY = 0
A = BAH  CY = 0  1 回
A = 74H  CY = 1  2 回
A = E8H  CY = 0  3 回
```

SHLW

左方向の論理シフトを行います。

[命令形式]

SHLW dst, cnt

[オペレーション]

(CY ← dst_{15}, dst_m ← dst_{m-1}, dst_0 ← 0) × cnt

[オペランド]

オペランド（dst, cnt）
AX, cnt
BC, cnt

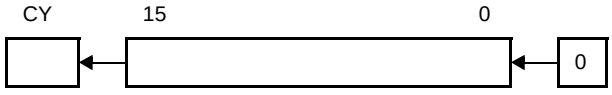
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回左にシフトします。
- 下位ビットには 0 が埋め込まれ，CY へは最後にビット 15 からシフトした値が入ります。
- cnt は 1-15 を指定できます。



［記述例］

```
SHLW    BC, 3   ; (1)
```

(1) BC レジスタの内容が C35DH であった場合の結果は，BC = 1AE8H，CY = 0 となります。

```
BC = C35DH  CY = 0
BC = 86BAH  CY = 1  1回
BC = 0D74H  CY = 1  2回
BC = 1AE8H  CY = 0  3回
```

SAR

右方向の算術シフトを行います。

[命令形式]

SAR dst, cnt

[オペレーション]

(CY ← dst_0, dst_{m-1} ← dst_m, dst_7 ← dst_7) × cnt

[オペランド]

オペランド（dst, cnt）
A, cnt

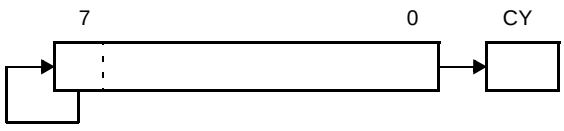
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回右にシフトします。
- 上位ビットには同じ値を保持し，CY へは最後にビット 0 からシフトした値が入ります。
- cnt は 1-7 を指定できます。



［記述例］

```
SAR     A, 4    ; (1)
```

(1) A レジスタの内容が 8CH であった場合の結果は，A = F8H，CY = 1 となります。

```
A = 8CH  CY = 0
A = C6H  CY = 0  1 回
A = E3H  CY = 0  2 回
A = F1H  CY = 1  3 回
A = F8H  CY = 1  4 回
```

SARW

右方向の算術シフトを行います。

[命令形式]

SARW dst, cnt

[オペレーション]

(CY ← dst_0, dst_{m-1} ← dst_m, dst_{15} ← dst_{15}) × cnt

[オペランド]

オペランド（dst, cnt）
AX, cnt

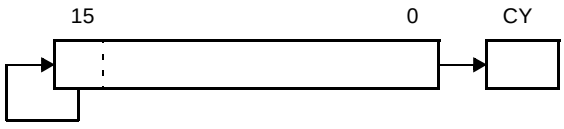
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を cnt 回右にシフトします。
- 上位ビットには同じ値を保持し，CY へは最後にビット 0 からシフトした値が入ります。
- cnt は 1-15 を指定できます。



［記述例］

```
SAR     AX, 4   ; (1)
```

(1) A レジスタの内容が A28CH であった場合の結果は，AX = FA28H，CY = 1 となります。

```
AX = A28CH  CY = 0
AX = D146H  CY = 0  1回
AX = E8A3H  CY = 0  2回
AX = F451H  CY = 1  3回
AX = FA28H  CY = 1  4回
```

（8）ローテート命令

ローテート命令には，次の命令があります。

命令	概要
ROR	バイト・データの右方向のローテート
ROL	バイト・データの左方向のローテート
RORC	キャリーを含むバイト・データの右方向のローテート
ROLC	キャリーを含むバイト・データの左方向のローテート
ROLWC	キャリーを含むワード・データの左方向ローテート

ROR

バイト・データの右方向のローテートを行います。

[命令形式]

ROR dst, cnt

[オペレーション]

(CY, dst_7 ← dst_0, dst_{m-1} ← dst_m) × 1 回

[オペランド]

オペランド（dst, cnt）
A, 1

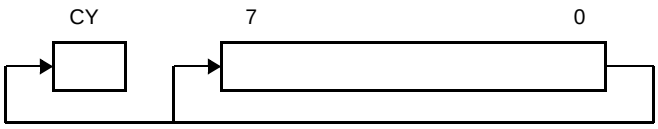
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）の内容を 1 回だけ右方向へ回転させます。
- LSB（ビット 0）の内容は，MSB（ビット 7）へ回転されると同時に CY フラグへも転送されます。



[記述例]

```
ROR     A, 1            ; (1)
```

(1) A レジスタの内容を右へ 1 ビット回転します。

ROL

バイト・データの左方向のローテートを行います。

[命令形式]

ROL dst, cnt

[オペレーション]

(CY, dst_0 ← dst_7, dst_{m+1} ← dst_m) × 1 回

[オペランド]

オペランド（dst, cnt）
A, 1

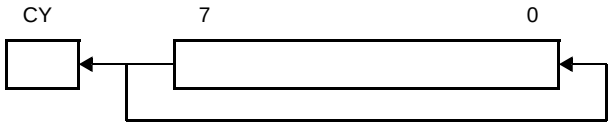
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）の内容を 1 回だけ左方向へ回転させます。
- MSB（ビット 7）の内容は，LSB（ビット 0）へ回転されると同時に CY フラグへも転送されます。



[記述例]

```
ROL     A, 1            ; (1)
```

(1) A レジスタの内容を左へ 1 ビット回転します。

RORC

キャリーを含むバイト・データの右方向のローテートを行います。

[命令形式]

RORC dst, cnt

[オペレーション]

(CY ← dst_0, dst_7 ← CY, dst_{m-1} ← dst_m) × 1 回

[オペランド]

オペランド（dst, cnt）
A, 1

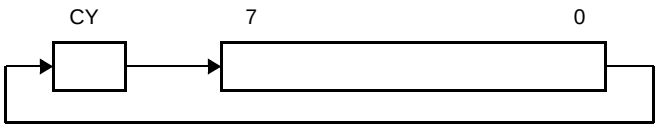
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）の内容を CY フラグを含め，1 回だけ右へ回転させます。



[記述例]

```
RORC    A, 1            ; (1)
```

(1) A レジスタの内容を CY フラグを含めて 1 ビット右方向へ回転します。

ROLC

キャリーを含むバイト・データの左方向のローテートを行います。

[命令形式]

ROLC dst, cnt

[オペレーション]

(CY ← dst_7, dst_0 ← CY, dst_{m+1} ← dst_m) × 1 回

[オペランド]

オペランド（dst, cnt）
A, 1

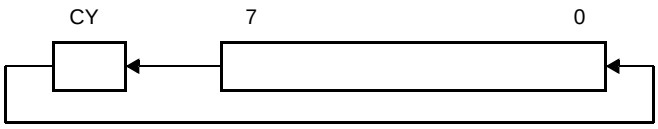
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）の内容を CY フラグを含め，1 回だけ左へ回転させます。



[記述例]

```
ROLC    A, 1            ; (1)
```

(1) A レジスタの内容を CY フラグを含めて 1 ビット左方向へ回転します。

ROLWC

キャリーを含むワード・データの左方向ローテートを行います。

[命令形式]

ROLWC dst, cnt

[オペレーション]

(CY ← dst_{15}, dst_0 ← CY, dst_{m+1} ← dst_m) × 1 回

[オペランド]

オペランド（dst, cnt）
AX, 1
BC, 1

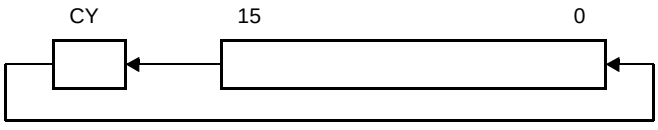
[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）を CY フラグを含め，1 回だけ左方向へ回転させます。



[記述例]

```
ROLWC   BC, 1           ; (1)
```

(1) BC レジスタの内容を CY フラグを含めて 1 ビット左へ回転します。

(9) ビット操作命令

ビット操作命令には，次の命令があります。

命令	概要
MOV1	1 ビット・データの転送
AND1	1 ビット・データの論理積
OR1	1 ビット・データの論理和
XOR1	1 ビット・データの排他的論理和
SET1	1 ビット・データのセット
CLR1	1 ビット・データのクリア
NOT1	1 ビット・データの論理否定

MOV1

1ビット・データの転送を行います。

[命令形式]

MOV1 dst, src

[オペレーション]

dst ← src

[オペランド]

オペランド（dst, src）
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
saddr.bit, CY
sfr.bit, CY
A.bit, CY
PSW.bit, CY
[HL].bit, CY
CY, ES:[HL].bit
ES:[HL].bit, CY

[フラグ]

（1）dst が CY の場合

Z	AC	CY
		×

空欄 ： 変化なし

× ： 結果にしたがってセット／リセットされる

(2) dst が PSW.bit の場合

Z	AC	CY
×	×	

空欄 : 変化なし
× : 結果にしたがってセット／リセットされる

(3) 上記以外

Z	AC	CY

空欄 : 変化なし

[説明]

- 第 1 オペランドで指定されたデスティネーション・オペランド（dst）に，第 2 オペランドで指定されたソース・オペランド（src）のビット・データを転送します。
- デスティネーション・オペランド（dst）が CY，または PSW.bit の場合，該当するフラグのみが変化します。

[記述例]

```
MOV1    P3.4, CY    ; (1)
```

(1) CY フラグの内容をポート 3 のビット 4 に転送します。

AND1

1ビット・データの論理積を行います。

[命令形式]

AND1 dst, src

[オペレーション]

dst ← dst ∧ src

[オペランド]

オペランド（dst, src）
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第1オペランドで指定されるデスティネーション・オペランド（dst）と第2オペランドで指定されるソース・オペランド（src）のビット・データとの論理積をとり，結果をデスティネーション・オペランド（dst）に格納します。
- CYフラグは，演算結果が格納されます（デスティネーション・オペランド（dst）であるため）。

［記述例］

```
AND1   CY, FFE7FH.3   ;  (1)
```

（1）FFE7FH のビット 3 と CY フラグの論理積をとり，結果を CY フラグに格納します。

OR1

1ビット・データの論理和を行います。

[命令形式]

OR1 dst, src

[オペレーション]

dst ← dst ∨ src

[オペランド]

オペランド（dst, src）
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第1オペランドで指定されるデスティネーション・オペランド（dst）と第2オペランドで指定されるソース・オペランド（src）のビット・データとの論理和をとり，結果をデスティネーション・オペランド（dst）に格納します。
- CYフラグは，演算結果が格納されます（デスティネーション・オペランド（dst）であるため）。

［記述例］

```
OR1     CY, P2.5    ; (1)
```

(1) ポート 2 のビット 5 と CY フラグの論理和をとり，結果を CY フラグに格納します。

XOR1

1ビット・データの排他的論理和を行います。

[命令形式]

XOR1 dst, src

[オペレーション]

dst ← dst ∀ src

[オペランド]

オペランド（dst, src）
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- 第1オペランドで指定されるデスティネーション・オペランド（dst）と第2オペランドで指定されるソース・オペランド（src）のビット・データとの排他的論理和をとり，結果をデスティネーション・オペランド（dst）に格納します。
- CYフラグは，演算結果が格納されます（デスティネーション・オペランド（dst）であるため）。

［記述例］

```
XOR1  CY, A.7 ;  (1)
```

（1）A レジスタのビット 7 と CY フラグの排他的論理和をとり，結果を CY フラグに格納します。

SET1

1ビット・データのセットを行います。

[命令形式]

SET1 dst

[オペレーション]

dst ← 1

[オペランド]

オペランド（dst）
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[フラグ]

（1）dst が PSW.bit の場合

Z	AC	CY
×	×	×

× ： 結果にしたがってセット／リセットされる

（2）dst が CY の場合

Z	AC	CY
		1

空欄 ： 変化なし
1 ： 1 にセットされる

（3）上記以外

Z	AC	CY

空欄 ： 変化なし

［説明］

- デスティネーション・オペランド（dst）をセット（1）します。
- デスティネーション・オペランド（dst）が CY，または PSW.bit の場合，該当するフラグのみがセット（1）されます。

［記述例］

```
SET1    FFE55H.1        ;  (1)
```

（1）FFE55H のビット 1 をセット（1）します。

CLR1

1ビット・データのクリアを行います。

[命令形式]

CLR1 dst

[オペレーション]

dst ← 0

[オペランド]

オペランド（dst）
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[フラグ]

（1）dst が PSW.bit の場合

Z	AC	CY
×	×	×

×　：結果にしたがってセット／リセットされる

(2) dst が CY の場合

Z	AC	CY
		0

空欄 : 変化なし
0 : 0 にクリアされる

(3) 上記以外

Z	AC	CY

空欄 : 変化なし

[説明]

- デスティネーション・オペランド（dst）をクリア（0）します。
- デスティネーション・オペランド（dst）が CY，または PSW.bit の場合，該当するフラグのみがクリア（0）されます。

[記述例]

```
CLR1    P3.7            ; (1)
```

(1) ポート 3 のビット 7 をクリア（0）します。

NOT1

1ビット・データの論理否定を行います。

[命令形式]

NOT1 dst

[オペレーション]

dst ← $\overline{dst}$

[オペランド]

オペランド（dst）
CY

[フラグ]

Z	AC	CY
		×

空欄 ： 変化なし
× ： 結果にしたがってセット／リセットされる

[説明]

- CY フラグを反転します。

[記述例]

```
NOT1    CY      ; (1)
```

(1) CY フラグを反転します。

（10）コール・リターン命令

コール・リターン命令には，次の命令があります。

命令	概要
CALL	サブルーチン・コール
CALLT	サブルーチン・コール（コール・テーブル参照）
BRK	ソフトウエア・ベクタ割り込み
RET	サブルーチンからの復帰
RETI	ハードウエア・ベクタ割り込みからの復帰
RETB	ソフトウエア割り込みからの復帰

CALL

サブルーチン・コールを行います。

[命令形式]

CALL target

[オペレーション]

(SP - 2) ← (PC + n)$_S$,

(SP - 3) ← (PC + n)$_H$,

(SP - 4) ← (PC + n)$_L$,

SP ← SP - 4,

PC ← target

備考　n は !!addr20 のときは 4，!addr16/$!addr20 のときは 3，AX/BC/DE/HL のときは 2 となります。

[オペランド]

オペランド（target）
AX
BC
DE
HL
$!addr20
!addr16
!!addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- 20/16 ビットの絶対アドレスまたはレジスタ間接アドレスによるサブルーチン・コールです。
- 次の命令の先頭アドレス（PC+n）をスタックに退避し，ターゲット・オペランド（target）で指定されるアドレスに分岐します。

［記述例］

```
CALL    !!3E000H        ;  (1)
```

（1）3E000H 番地にサブルーチン・コールします。

CALLT

サブルーチン・コール（コール・テーブル参照）を行います。

[命令形式]

CALLT [addr5]

[オペレーション]

(SP - 2) ← (PC + 2)$_S$,

(SP - 3) ← (PC + 2)$_H$,

(SP - 4) ← (PC + 2)$_L$,

PC$_S$ ← 0000,

PC$_H$ ← (000000000000, addr5 + 1),

PC$_L$ ← (000000000000, addr5),

SP ← SP - 4

[オペランド]

オペランド（[addr5]）
[addr5]

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- コール・テーブル参照のサブルーチン・コールです。
- 次の命令の先頭アドレス（PC + 2）をスタックに退避し，コール・テーブル（アドレスの上位 12 ビットは 000000000000B に固定で，次の 8 ビット中の 5 ビットを addr5 で指定します）のワード・データで示されるアドレスに分岐します。

［記述例］

```
CALLT   [80H]           ;  (1)
```

（1）00080H, 00081H 番地にあるワード・データをアドレスとして，そのアドレスにサブルーチン・コールします。

［備考］

- アドレスの指定は偶数アドレスのみです。奇数アドレスは指定できません。

addr5：00080H-000BEH のイミーディエト・データまたはラベル（偶数アドレスのみ）

BRK

ソフトウエア・ベクタ割り込みを行います。

[命令形式]

BRK

[オペレーション]

(SP - 1) ← PSW,

(SP - 2) ← (PC + 2)$_S$,

(SP - 3) ← (PC + 2)$_H$,

(SP - 4) ← (PC + 2)$_L$,

PC$_S$ ← 0000,

PC$_H$ ← (0007FH),

PC$_L$ ← (0007EH),

SP ← SP - 4,

IE ← 0

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- ソフトウエア割り込み命令です。
- PSW と次の命令のアドレス（PC + 2）をスタックに退避し，次に IE フラグをクリア（0）して，ベクタ・アドレス（0007EH, 0007FH）のワード・データで指示されるアドレスに分岐します。
  IE フラグがクリア（0）されるため，以後のマスカブル・ベクタ割り込みは禁止されます。
- この命令で発生したソフトウエア・ベクタ割り込みからの復帰には，RETB 命令を使用します。

RET

サブルーチンからの復帰を行います。

[命令形式]

RET

[オペレーション]

PC_L ← (SP),

PC_H ← (SP + 1),

PC_S ← (SP + 2),

SP ← SP + 4,

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- CALL, , CALLT 命令でコールされたサブルーチン・コールからのリターン命令です。
- スタックに退避されているワード・データを PC に復帰し，サブルーチンからリターンします。

RETI

ハードウエア・ベクタ割り込みからの復帰を行います。

[命令形式]

RETI

[オペレーション]

PC_L ← (SP),

PC_H ← (SP + 1),

PC_S ← (SP + 2),

PSW ← (SP + 3),

SP ← SP + 4

[オペランド]

なし

[フラグ]

Z	AC	CY
R	R	R

R : 以前に退避した値がリストアされる

[説明]

- ベクタ割り込みからの復帰命令です。
- スタックに退避されているデータを PC と PSW に復帰し，割り込み処理ルーチンからリターンします。
- BRK 命令によるソフトウエア割り込みからの復帰には使用できません。
- この命令と次に実行する命令の間では，すべての割り込みを受け付けません。
- NMIS フラグはノンマスカブル割り込み受け付けにより 1 にセットされ，RETI 命令により 0 にクリアされます。

[注意]

- ノンマスカブル割り込み処理からの復帰を RETI 命令以外の命令で行うと，NMIS フラグが 0 にクリアされないため，すべての割り込み（ノンマスカブル割り込みを含む）を受け付けなくなります。

RETB

ソフトウエア割り込みからの復帰を行います。

[命令形式]

RETB

[オペレーション]

PC_L ← (SP),

PC_H ← (SP + 1),

PC_S ← (SP + 2),

PSW ← (SP + 3),

SP ← SP + 4

[オペランド]

なし

[フラグ]

Z	AC	CY
R	R	R

R : 以前に退避した値がリストアされる

[説明]

- BRK 命令で発生したソフトウエア割り込みからの復帰命令です。
- スタックに退避されている PC と PSW を復帰し，割り込み処理ルーチンからリターンします。
- この命令と次に実行する命令の間では，すべての割り込みを受け付けません。

（11）スタック操作命令

スタック操作命令には，次の命令があります。

命令	概要
PUSH	プッシュ
POP	ポップ
MOVW	スタック・ポインタとのワード・データの転送
ADDW	スタック・ポインタの加算
SUBW	スタック・ポインタの減算

PUSH

プッシュを行います。

[命令形式]

PUSH src

[オペレーション]

（1）src が rp の場合

(SP - 1) ← rp_H,

(SP - 2) ← rp_L,

SP ← SP - 2

（2）src が PSW の場合

(SP - 1) ← PSW,

(SP - 2) ← 00H,

SP ← SP - 2

[オペランド]

オペランド（src）
PSW
rp

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- ソース・オペランド（src）で指定されたレジスタのデータをスタックに退避します。

［記述例］

```
PUSH    AX              ; (1)
```

(1) AX レジスタの内容をスタックに退避します。

POP

ポップを行います。

[命令形式]

POP dst

[オペレーション]

(1) dst が rp の場合

rp_L ← (SP),

rp_H ← (SP + 1),

SP ← SP + 2

(2) dst が PSW の場合

PSW ← (SP + 1),

SP ← SP + 2

[オペランド]

オペランド（dst）
PSW
rp

[フラグ]

(1) dst が rp の場合

Z	AC	CY

空欄 ： 変化なし

(2) dst が PSW の場合

Z	AC	CY
R	R	R

R ： 以前に退避した値がリストアされる

[説明]

- デスティネーション・オペランド（dst）で指定されたレジスタに，データをスタックから復帰します。
- オペランドが PSW の場合，各フラグはスタックのデータで置き換わります。
- POP PSW 命令と次に続く命令の間では，すべての割り込みを受け付けません。

[記述例]

```
POP     AX              ; (1)
```

(1) AX レジスタにスタックのデータを復帰します。

MOVW

スタック・ポインタとのワード・データの転送を行います。

[命令形式]

MOVW dst, src

[オペレーション]

dst ← src

[オペランド]

オペランド（dst, src）
SP, #word
SP, AX
AX, SP
HL, SP
BC, SP
DE, SP

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- スタック・ポインタの内容を操作するための命令です。
- 第 1 オペランドで指定されるデスティネーション・オペランド（dst）に第 2 オペランドで指定されるソース・オペランド（src）を格納します。

[記述例]

```
MOVW    SP, #FE1FH              ; (1)
```

(1) スタック・ポインタに FE1FH を格納します。

ADDW

スタック・ポインタの加算を行います。

[命令形式]

ADDW SP, src

[オペレーション]

SP ← SP + src

[オペランド]

オペランド（SP, src）
SP, #byte

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるスタック・ポインタと第 2 オペランドで指定されるソース・オペランド（src）の加算を行い，結果をスタック・ポインタに格納します。

[記述例]

```
ADDW    SP, #12H    ; (1)
```

(1) スタック・ポインタと 12H を加算し，結果をスタック・ポインタに格納します。

SUBW

スタック・ポインタの減算を行います。

[命令形式]

SUBW SP, src

[オペレーション]

SP ← SP - src

[オペランド]

オペランド（SP, src）
SP, #byte

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランドで指定されるスタック・ポインタから第 2 オペランドで指定されるソース・オペランド（src）を減算し，結果をスタック・ポインタに格納します。

[記述例]

```
SUBW    SP, #12H    ; (1)
```

(1) スタック・ポインタから 12H を減算し，結果をスタック・ポインタに格納します。

（12）無条件分岐命令

無条件分岐命令には，次の命令があります。

命令	概要
BR	無条件分岐

BR

無条件分岐を行います。

[命令形式]

BR target

[オペレーション]

PC ← target

[オペランド]

オペランド（target）
AX
$addr20
$!addr20
!addr16
!!addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- 無条件に分岐を行う命令です。
- ターゲット・アドレス・オペランド（target）のワード・データを PC に転送し，分岐します。

[記述例]

```
BR      !!12345H        ; (1)
```

(1) 12345H 番地に分岐します。

（13）条件付き分岐命令

条件付き分岐命令には，次の命令があります。

命令	概要
BC	キャリー・フラグによる条件分岐（CY = 1）
BNC	キャリー・フラグによる条件分岐（CY = 0）
BZ	ゼロ・フラグによる条件分岐（Z = 1）
BNZ	ゼロ・フラグによる条件分岐（Z = 0）
BH	数値の大小による条件分岐（(Z ∨ CY) = 0）
BNH	数値の大小による条件分岐（(Z ∨ CY) = 1）
BT	ビット・テストによる条件分岐（バイト・データのビット = 1）
BF	ビット・テストによる条件分岐（バイト・データのビット = 0）
BTCLR	ビット・テストによる条件分岐とクリア（バイト・データのビット = 1）

BC

キャリー・フラグによる条件分岐（CY = 1）を行います。

[命令形式]

BC $addr20

[オペレーション]

PC ← PC + 2 + jdisp8 if CY = 1

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- CY = 1 の場合に，オペランドで指定されたアドレスに分岐します。
  CY = 0 の場合は，何も処理を行わず，次に続く命令を実行します。

[記述例]

```
BC      $00300H         ; (1)
```

（1）CY = 1 なら 00300H 番地に分岐します（ただし，この命令の先頭は 0027FH-0037EH 番地内にあります）。

BNC

キャリー・フラグによる条件分岐（CY = 0）を行います。

[命令形式]

BNC $addr20

[オペレーション]

PC ← PC + 2 + jdisp8 if CY = 0

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- CY = 0 の場合に，オペランドで指定されたアドレスに分岐します。
  CY = 1 の場合は，何も処理を行わず，次に続く命令を実行します。

[記述例]

```
BNC     $00300H         ; (1)
```

(1) CY = 0 なら 00300H 番地に分岐します（ただし，この命令の先頭は 0027FH-0037EH 番地内にあります）。

BZ

ゼロ・フラグによる条件分岐（Z = 1）を行います。

[命令形式]

BZ $addr20

[オペレーション]

PC ← PC + 2 + jdisp8 if Z = 1

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- Z = 1 の場合に，オペランドで指定されたアドレスに分岐します。
  Z = 0 の場合は，何も処理を行わず，次に続く命令を実行します。

[記述例]

```
DEC B
BZ      $003C5H         ; (1)
```

(1) B レジスタが 0 なら 003C5H 番地に分岐します（ただし，この命令の先頭は，00344H-00443H 番地内にあります）。

BNZ

ゼロ・フラグによる条件分岐（Z = 0）を行います。

[命令形式]

BNZ $addr20

[オペレーション]

PC ← PC + 2 + jdisp8 if Z = 0

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- Z = 0 の場合に，オペランドで指定されたアドレスに分岐します。
  Z = 1 の場合は，何も処理を行わず，次に続く命令を実行します。

[記述例]

```
CMP     A, #55H
BNZ     $00A39H         ; (1)
```

(1) A レジスタが 55H でないとき，00A39H 番地に分岐します（ただし，この命令の先頭は 009B8H-00AB7H 番地内にあります）。

BH

数値の大小による条件分岐（(Z ∨ CY) = 0）を行います。

[命令形式]

BH $addr20

[オペレーション]

PC ← PC + 3 + jdisp8 if (Z ∨ CY) = 0

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- (Z ∨ CY) = 0 の場合に，オペランドで指定されたアドレスに分岐します。
  (Z ∨ CY) = 1 の場合は，何も処理を行わず，次に続く命令を実行します。
- この命令は，符号なしの大小を判定するのに使用します。直前の CMP 命令の第 1 オペランドが第 2 オペランドの値より大きいことを調べます。

[記述例]

```
CMP     A, C
BH      $00356H     ; (1)
```

(1) A レジスタの内容が C レジスタの内容より大きい場合，00356H 番地に分岐します（ただし，BH 命令の先頭は 002D4H-003D3H 番地内にあります）。

BNH

数値の大小による条件分岐（(Z ∨ CY) = 1）を行います。

[命令形式]

BNH $addr20

[オペレーション]

PC ← PC + 3 + jdisp8 if (Z ∨ CY) = 1

[オペランド]

オペランド（$addr20）
$addr20

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- (Z ∨ CY) = 1 の場合に，オペランドで指定されたアドレスに分岐します。
  (Z ∨ CY) = 0 の場合は，何も処理を行わず，次に続く命令を実行します。
- この命令は，符号なしの大小を判定するのに使用します。直前の CMP 命令の第 1 オペランドが第 2 オペランドの値より大きくない（第 1 オペランドが第 2 オペランド以下）ことを調べます。

[記述例]

```
CMP     A, C
BNH     $00356H     ; (1)
```

(1) A レジスタの内容が C レジスタの内容より小さいか等しい場合，00356H 番地に分岐します（ただし，BNH 命令の先頭は 002D4H-003D3H 番地内にあります）。

BT

ビット・テストによる条件分岐（バイト・データのビット = 1）を行います。

[命令形式]

BT bit, $addr20

[オペレーション]

PC ← PC + b + jdisp8 if bit = 1

[オペランド]

オペランド（bit, $addr20）	b（バイト数）
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランド（bit）の内容がセット（1）されているとき，第 2 オペランド（$addr20）で指定されるアドレスに分岐します。
  第 1 オペランド（bit）の内容がセット（1）されていないときは，何も処理を行わず，次に続く命令を実行します。

［記述例］

```
BT      FFE47H.3, $0055CH    ; (1)
```

（1）FFE47H 番地のビット 3 が 1 のとき，0055CH 番地に分岐します（ただし，この命令の先頭は，004DAH-005D9H 番地内にあります）。

BF

ビット・テストによる条件分岐（バイト・データのビット = 0）を行います。

[命令形式]

BF bit, $addr20

[オペレーション]

PC ← PC + b + jdisp8 if bit = 0

[オペランド]

オペランド（bit, $addr20）	b（バイト数）
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランド（bit）の内容がクリア（0）されているとき，第 2 オペランド（$addr20）で指定されるアドレスに分岐します。
  第 1 オペランド（bit）の内容がクリア（0）されていないときは，何も処理を行わず，次に続く命令を実行します。

［記述例］

```
BF      P2.2, $01549H       ; (1)
```

(1) ポート 2 のビット 2 が 0 のとき，01549H 番地に分岐します（ただし，この命令の先頭は，014C6H-015C5H 番地内にあります）。

BTCLR

ビット・テストによる条件分岐とクリア（バイト・データのビット = 1）を行います。

[命令形式]

BTCLR bit, $addr20

[オペレーション]

PC ← PC + b + jdisp8 if bit = 1, then bit ← 0

[オペランド]

オペランド（bit, $addr20）	b（バイト数）
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[フラグ]

（1）bit が PSW.bit の場合

Z	AC	CY
×	×	×

× ： 結果にしたがってセット／リセットされる

（2）上記以外

Z	AC	CY

空欄 ： 変化なし

[説明]

- 第 1 オペランド（bit）の内容がセット（1）されているとき，第 1 オペランド（bit）の内容をクリア（0）し，第 2 オペランドで指定されたアドレスに分岐します。
  第 1 オペランド（bit）の内容がセット（1）されていないときは，何も処理を行わず，次に続く命令を実行します。
- 第 1 オペランド（bit）が PSW.bit の場合，該当するフラグの内容がクリア（0）されます。

[記述例]

```
BTCLR PSW.0, $00356H    ; (1)
```

(1) PSW のビット 0（CY フラグ）が 1 の場合，CY フラグをクリアして，00356H 番地に分岐します（ただし，この命令の先頭は，002D4H-003D3H 番地内にあります）。

（14）条件付きスキップ命令

条件付きスキップ命令には，次の命令があります。

命令	概要
SKC	キャリー・フラグによるスキップ（CY = 1）
SKNC	キャリー・フラグによるスキップ（CY = 0）
SKZ	ゼロ・フラグによるスキップ（Z = 1）
SKNZ	ゼロ・フラグによるスキップ（Z = 0）
SKH	数値の大小によるスキップ（(Z ∨ CY) = 0）
SKNH	数値の大小によるスキップ（(Z ∨ CY) =1）

SKC

キャリー・フラグによるスキップ（CY = 1）を行います。

[命令形式]

SKC

[オペレーション]

Next instruction skip if CY = 1

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- CY = 1 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- CY = 0 のときには次の命令を実行します。

[記述例]

```
MOV     A, #55H
SKC
ADD     A, #55H     ; (1)
```

(1) CY = 0 のときは A レジスタ = AAH，CY = 1 のときは A レジスタ = 55H となります。

SKNC

キャリー・フラグによるスキップ（CY = 0）を行います。

[命令形式]

SKNC

[オペレーション]

Next instruction skip if CY = 0

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- CY = 0 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- CY = 1 のときには次の命令を実行します。

[記述例]

```
MOV     A, #55H
SKNC
ADD     A, #55H     ; (1)
```

（1）CY = 1 のときは A レジスタ = AAH，CY = 0 のときは A レジスタ = 55H となります。

SKZ

ゼロ・フラグによるスキップ（Z = 1）を行います。

[命令形式]

SKZ

[オペレーション]

Next instruction skip if Z = 1

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- Z = 1 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。
  ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- Z = 0 のときには次の命令を実行します。

[記述例]

```
MOV     A, #55H
SKZ
ADD     A, #55H     ; (1)
```

(1) Z = 0 のときは A レジスタ = AAH，Z = 1 のときは A レジスタ = 55H となります。

SKNZ

ゼロ・フラグによるスキップ（Z = 0）を行います。

[命令形式]

SKNZ

[オペレーション]

Next instruction skip if Z = 0

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- Z = 0 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。
  ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- Z = 1 のときには次の命令を実行します。

[記述例]

```
MOV     A, #55H
SKNZ
ADD     A, #55H     ; (1)
```

(1) Z = 1 のときは A レジスタ = AAH，Z = 0 のときは A レジスタ = 55H となります。

SKH

数値の大小によるスキップ（(Z ∨ CY) = 0）を行います。

[命令形式]

SKH

[オペレーション]

Next instruction skip if (Z ∨ CY) = 0

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- (Z ∨ CY) = 0 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- (Z ∨ CY) = 1 のときには次の命令を実行します。

[記述例]

```
CMP     A, #80H
SKH
CALL    !!TARGET        ; (1)
```

(1) A レジスタの内容が 80H より大きい場合は CALL 命令をスキップし，その次の命令を実行，A レジスタの内容が 80H 以下の場合は次の CALL 命令を実行し，TARGET 番地へ分岐します。

SKNH

数値の大小によるスキップ（(Z ∨ CY) =1）を行います。

[命令形式]

SKNH

[オペレーション]

Next instruction skip if (Z ∨ CY) = 1

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- (Z ∨ CY) = 1 のときには次の命令をスキップします。次に続く命令は NOP になり，1 クロックの実行時間は消費します。ただし，ES: で示される PREFIX 命令が次の命令であった場合は，2 クロックの実行時間を消費します。
- (Z ∨ CY) = 0 のときには次の命令を実行します。

[記述例]

```
CMP     A, #80H
SKNH
CALL    !!TARGET        ; (1)
```

(1) A レジスタの内容が 80H 以下の場合は CALL 命令をスキップし，その次の命令を実行，A レジスタの内容が 80H より大きい場合は次の CALL 命令を実行し，TARGET 番地へ分岐します。

（15）CPU 制御命令

CPU 制御命令には，次の命令があります。

命令	概要
SEL	レジスタ・バンクの選択
NOP	ノー・オペレーション
EI	割り込みの許可
DI	割り込みの禁止
HALT	ホルト・モードの設定
STOP	ストップ・モードの設定

SEL

レジスタ・バンクの選択を行います。

[命令形式]

SEL RBn

[オペレーション]

RBS0, RBS1 ← n ;（n = 0 ～ 3）

[オペランド]

オペランド（RBn）
RBn

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- オペランド（RBn）で指定されたレジスタ・バンクを次命令以降で使用するレジスタ・バンクとします。
- RBn には，RB0-RB3 まであります。

[記述例]

```
SEL     RB2     ; (1)
```

(1) 次命令以降で使用するレジスタ・バンクとして，レジスタ・バンク 2 を選択します。

NOP

ノー・オペレーションです。

[命令形式]

NOP

[オペレーション]

no operation

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

- 何も処理をせずに時間だけを消費します。

EI

割り込みの許可を行います。

[命令形式]

EI

[オペレーション]

IE ← 1

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- マスカブル割り込みの受け付け可能な状態にします（割り込み許可フラグ（IE）をセット（1）します）。
- この命令と次に続く 1 命令の間では，すべての割り込みを受け付けません。
- この命令を実行しても，他の要因によりベクタ割り込みの受け付けを行わないようにすることができます。詳細については，各製品のユーザーズ・マニュアルの割り込み機能を参照してください。

DI

割り込みの禁止を行います。

[命令形式]

DI

[オペレーション]

IE ← 0

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- マスカブル割り込みのベクタ割り込みによる受け付けを禁止します（割り込み許可フラグ（IE）をクリア（0）します）。
- この命令と次に続く 1 命令の間では，すべての割り込みを受け付けません。
- 割り込み処理の詳細については，各製品のユーザーズ・マニュアルの割り込み機能を参照してください。

HALT

ホルト・モードの設定を行います。

[命令形式]

HALT

[オペレーション]

Set HALT Mode

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄 ： 変化なし

[説明]

- HALT モードになります。CPU の動作クロックを停止させるモードです。通常動作モードとの組み合わせによる間欠動作により，システムのトータル消費電力を低下させることができます。

STOP

ストップ・モードの設定を行います。

[命令形式]

STOP

[オペレーション]

Set STOP Mode

[オペランド]

なし

[フラグ]

Z	AC	CY

空欄　：変化なし

[説明]

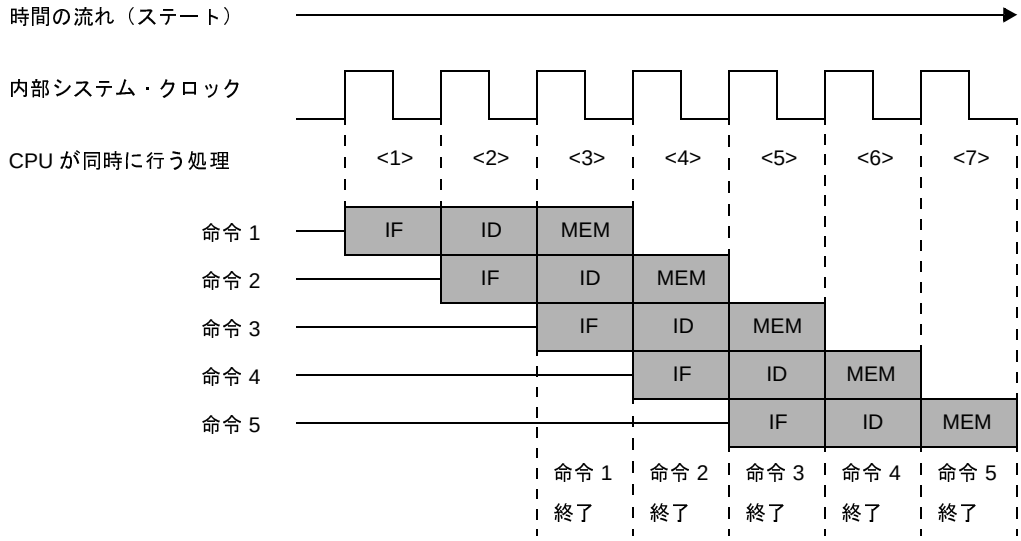
- STOP モードになります。メイン・システム・クロック発振回路を停止させ，システム全体が停止するモードです。リーク電流だけの超低消費電力にすることができます。

4. 6. 7 パイプライン

(1) 特長

RL78 ファミリ，78K0R マイクロコントローラは，3 段パイプラインの制御により，ほとんどの命令を 1 クロックで実行します。命令実行手順は，インストラクション・フェッチ（IF），インストラクション・デコード（ID），メモリ・アクセス（MEM）の 3 つのステージで構成されています。

図 4―38 標準的な命令を 5 つ続けて実行する例



IF（インストラクション・フェッチ）	命令のフェッチを行い，フェッチ・ポインタをインクリメントします。
ID（インストラクション・デコード）	命令をデコードし，アドレス計算を行います。
MEM（メモリ・アクセス）	デコードした命令を実行し，対象となるアドレスのメモリにアクセスします。

(2) 動作クロック数

RL78 ファミリ，78K0R マイクロコントローラでは，他のパイプライン・マイコンに見られるクロック数が数えられないという問題が解決され，常に同じクロック数で動作することにより，安定したプログラムを提供できます。

次の場合を除き，「(5) オペレーション一覧」に記載してある動作クロック数となります。

(a) フラッシュ・メモリの内容をデータ・アクセス

フラッシュ・メモリの内容をデータとしてアクセスした場合は，MEM ステージでパイプラインが停止しますので，一覧にある動作クロック数より増加します。詳細は「(5) オペレーション一覧」を参照してください。

(b) 外部メモリの内容をデータ・アクセス

外部メモリの内容をデータとしてアクセスした場合は，CPU はウエイトが発生します。したがって，一覧にある動作クロック数より増加します。

増加するクロック数については，以下を参照してください。

外部拡張クロック出力（CLKOUT）選択クロック	ウエイト数
fCLK	3 クロック
fCLK/2	5 ～ 6 クロック
fCLK/3	7 ～ 9 クロック
fCLK/4	9 ～ 12 クロック

(c) RAM からの命令フェッチ

RAM の内容をフェッチ・データとした場合は，RAM の読み出しが間に合わず，命令キューが空になりますので，命令キューにデータが揃うまで CPU はウエイトします。また，RAM からフェッチ中に RAM へのアクセスが発生した場合も CPU のフェッチ動作はウエイトします。

(d) 外部メモリからの命令フェッチ

外部メモリの内容をフェッチ・データとした場合は，外部メモリの読み出しが間に合わず，命令キューが空になりますので，命令キューにデータが揃うまで CPU はウエイトします。また，外部メモリからフェッチ中に外部メモリへのアクセスが発生した場合も CPU のフェッチ動作はウエイトします。

増加するクロック数については，以下を参照してください。

外部拡張クロック出力（CLKOUT）選択クロック	ウエイト数
fCLK	3 クロック
fCLK/2	5 ～ 6 クロック
fCLK/3	7 ～ 9 クロック
fCLK/4	9 ～ 12 クロック

(e) 命令の組み合わせによるハザード

間接アクセスに使用するレジスタへの書き込み直後に，そのレジスタの内容のデータを間接アクセスする場合は，1 クロックのウエイトが入ります。

レジスタ名	前命令	次の命令のオペランド，または命令
DE	D レジスタへのライト命令[注] E レジスタへのライト命令[注] DE レジスタへのライト命令[注] SEL RBn	[DE]，[DE+byte]
HL	H レジスタへのライト命令[注] L レジスタへのライト命令[注] HL レジスタへのライト命令[注] SEL RBn	[HL]，[HL+byte]，[HL+B]，[HL+C]，[HL].bit

レジスタ名	前命令	次の命令のオペランド，または命令
B	B レジスタへのライト命令[注] SEL RBn	word[B]，[HL+B]
C	C レジスタへのライト命令[注] SEL RBn	word[C]，[HL+C]
BC	B レジスタへのライト命令[注] C レジスタへのライト命令[注] BC レジスタへのライト命令[注] SEL RBn	word[BC]，[HL+B]，[HL+C]

SP	MOVW SP, #word
	MOVW SP, AX
	ADDW SP, #byte
	SUBW SP, #byte
		[SP+byte]
		CALL 命令，CALLT 命令，BRK 命令，SOFT 命令，RET 命令，RETI 命令，RETB 命令，割り込み，PUSH 命令，POP 命令

CS	MOV CS, #byte MOV CS, A	CALL rp BR AX
AX	A レジスタへのライト命令[注] X レジスタへのライト命令[注] AX レジスタへのライト命令[注] SEL RBn	BR AX
AX BC DE HL	A レジスタへのライト命令[注] X レジスタへのライト命令[注] B レジスタへのライト命令[注] C レジスタへのライト命令[注] D レジスタへのライト命令[注] E レジスタへのライト命令[注] H レジスタへのライト命令[注] L レジスタへのライト命令[注] AX レジスタへのライト命令[注] BC レジスタへのライト命令[注] DE レジスタへのライト命令[注] HL レジスタへのライト命令[注] SEL RBn	CALL rp

注 レジスタへのライト命令はダイレクト・アドレッシング，ショート・ダイレクト・アドレッシング，レジスタ・インダイレクト・アドレッシング，ベースト・アドレッシング，ベースト・インデクスト・アドレッシングにて，対象となるレジスタの値を書き換えたときにもウエイトが入ります。

第 5 章　リンク・ディレクティブ仕様

この章では，リンク・ディレクティブに必要となる項目や，ディレクティブ・ファイルの記述方法について説明します。

5. 1　コーディング方法

ここでは，リンク・ディレクティブのコーディング方法について説明します。

5. 1. 1　リンク・ディレクティブ

リンク・ディレクティブ（以降ディレクティブと略します）とは，リンカに対して入力ファイルや使用可能なメモリ領域，セグメントの配置など，リンク時の各種指示を行うための命令群です。

ディレクティブには，次の 2 種類があります。

ディレクティブの種類	役割
メモリ・ディレクティブ	- 実装メモリのアドレスを宣言します。 - メモリをいくつかの領域に分割して，メモリ領域を指定します。 例を示します。 CALLT 領域 内蔵 ROM 外付け ROM SADDR SADDR 以外の内蔵 RAM
セグメント配置ディレクティブ	- セグメントの配置を指定します。 各セグメントに対し，次の内容を指定します。 アブソリュート・アドレス メモリ領域のみ指定

エディタなどを使用してディレクティブを記述したファイル（ディレクティブ・ファイル）を作成し，リンカの起動時に，-d オプションを指定します。

これにより，リンカはディレクティブ・ファイルを読み込み，解釈しながらリンク処理を行います。

(1) ディレクティブ・ファイル

ディレクティブ・ファイル中に記述するディレクティブの記述フォーマットを次に示します。

- メモリ・ディレクティブ

```
MEMORY　メモリ領域名：（ スタート・アドレス値，サイズ ）［ / メモリ空間名 ］
```

- セグメント配置ディレクティブ

```
MERGE セグメント名 : [ AT ( スタート・アドレス ) ] [ = メモリ領域名指定 ] [ / メモリ空間名 ]
MERGE セグメント名 : [ 結合属性 ] [ = メモリ領域名指定 ] [ / メモリ空間名 ]
```

なお，ディレクティブは，1 つのディレクティブ・ファイル中に複数記述することができます。

各ディレクティブの詳細については，「(2) メモリ・ディレクティブ」，「(3) セグメント配置ディレクティブ」を参照してください。

(a) シンボル

セグメント名，メモリ領域名，メモリ空間名の記述では，大文字と小文字は区別されます。

(b) 数値

各ディレクティブの項目のうち，数値定数を記述する場合は，10 進数，または 16 進数を記述することができます。

記述方法はソースと同じで，16 進数の場合は最後に“H”を付けます。また，先頭が A ～ F の場合は前に“0”を付けます。

例を以下に示します。

```
23H, 0FC80H
```

(c) コメント文

ディレクティブ・ファイル中に，“;”，または“#”を記述した場合，そこから改行文字（LF）まではコメントとして扱われます。なお，改行文字が現れる前にディレクティブ・ファイルが終了した場合は，終了までをコメントとして扱います。

例を以下に示します。

下線部がコメントとなります。

```
; DIRECTIVE FILE FOR 78F1166_A0
MEMORY MEM1 : ( 40000H, 10000H ) #SECOND MEMORY AREA
```

(2) メモリ・ディレクティブ

メモリ・ディレクティブは，メモリ領域（実装するメモリのアドレスと名前）を定義するディレクティブです。

定義したメモリ領域は，その名前（メモリ領域名）によってセグメント配置ディレクティブで参照することができます。

メモリ領域は，デフォルトで定義されているメモリ領域を含め，100 個まで定義することができます。

構文を以下に示します。

```
MEMORY メモリ領域名 : ( スタート・アドレス , サイズ ) [/ メモリ空間名 ]
```

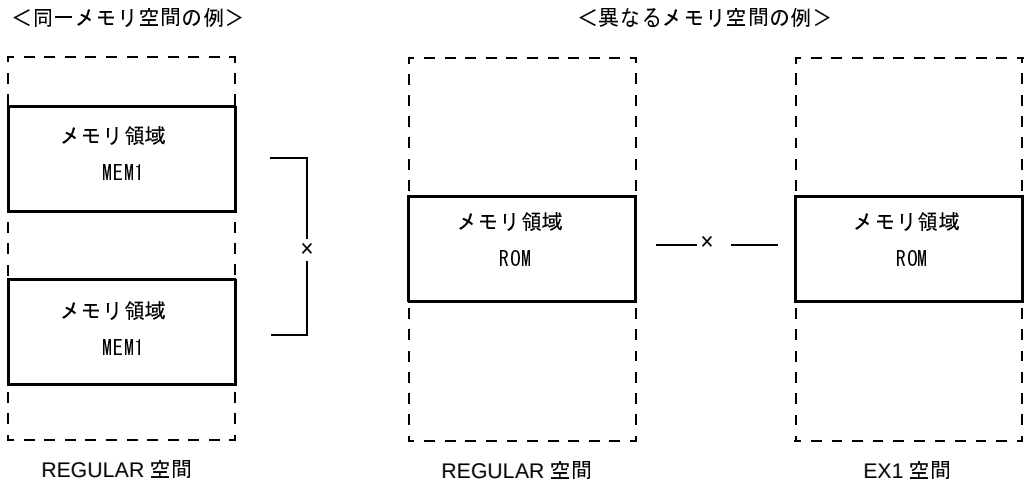
(a) メモリ領域名

定義するメモリ領域の名前を指定します。

指定時の条件は，次のとおりです。

- メモリ領域名に使用できる文字は，A ～ Z，a ～ z，0 ～ 9，_，?，@ です。
  ただし，0 ～ 9 はメモリ領域名の先頭には使用できません。
- 大文字と小文字は別の文字として区別します。
- 大文字と小文字は混在できます。
- メモリ領域名の長さは，最大 256 文字です。
  257 文字以上記述すると，エラーとなります。
- 各メモリ領域名は，全メモリ空間を通じて 1 つでなくてはなりません。
  異なるメモリ領域に同じメモリ領域名を付けることは，メモリ空間が同一である場合でも，異なる場合でも許されません。

図 5―1　メモリ領域名に指定できない例



(b) スタート・アドレス

定義するメモリ領域の先頭アドレスを指定します。

0H ～ 0FFFFFH までの数値定数を記述します。

(c) サイズ

定義するメモリ領域のサイズを指定します。

指定時の条件は，次のとおりです。

- 1 以上の数値定数を記述します。
- リンカがデフォルトで定義しているメモリ領域のサイズを指定し直す場合には，定義可能な範囲の制約があります。
  各デバイスのデフォルトで定義されているメモリ領域のサイズと再定義可能な範囲は，「CubeSuite+ 対応機能一覧」を参照してください。

(d) メモリ空間名

メモリ空間名は，次の 16 個の名前で表されます。

REGULAR，EX1，EX2，EX3，EX4，EX5，EX6，EX7，EX8，EX9，EX10，EX11，EX12，EX13，EX14，EX15

メモリ空間名は，メモリ領域をどのメモリ空間に割り付けるかを指定するときに使います。

次に指定時の条件を示します。

- メモリ空間名は，すべて大文字で記述します。
- メモリ空間名を省略した場合，REGULAR を指定したものとみなされます。
- “ / ”を記述したあとにメモリ空間名を省略した場合は，エラーとなります。

機能を以下に示します。

- メモリ領域名で指定した名前を持つメモリ領域を指定したメモリ空間に定義します。
- 1 つのメモリ・ディレクティブで，1 つのメモリ領域を定義することができます。
- メモリ・ディレクティブ自体は，複数の記述が可能です。このとき，指定した順番に複数回定義された場合は，エラーとなります。
- デフォルトのメモリ領域は，メモリ・ディレクティブで同一のメモリ領域を再定義しないかぎり有効です。メモリ・ディレクティブの記述を省略した場合，リンカが持つ各デバイスごとのデフォルトのメモリ領域のみを指定したものとします。
- デフォルトのメモリ空間を使用せずに，別の領域名で使用するときは，デフォルトの領域名のサイズを“0”に設定してください。

使用例を以下に示します。

- メモリ空間のアドレス 0H から 1FFH までをメモリ領域 ROMA として定義します。

```
MEMORY ROMA : ( 0H, 200H )
```

(3) セグメント配置ディレクティブ

セグメント配置ディレクティブは，指定したセグメントを指定したメモリ領域上か，特定番地に配置するディレクティブです。

構文を以下に示します。

```
MERGE セグメント名 : [AT ( スタート・アドレス )][ = メモリ領域名 ][ / メモリ空間名 ]
MERGE セグメント名 : [ 結合属性 ][ = メモリ領域名 ][ / メモリ空間名 ]
```

(a) セグメント名

リンカに入力するオブジェクト・モジュール・ファイル中に含まれるセグメント名です。

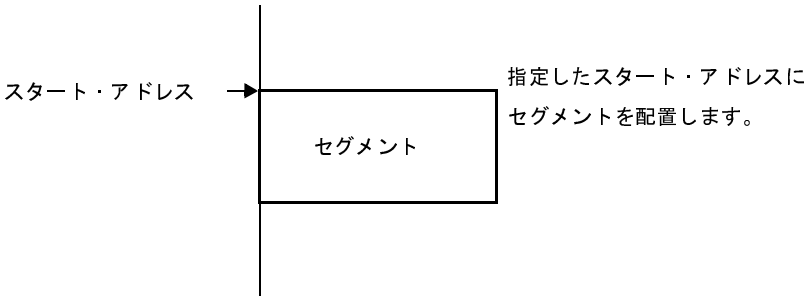
- セグメント名として，入力セグメント以外は指定できません。
- セグメント名は，アセンブル・ソース上に記述したとおりに指定しなければなりません。

（b）スタート・アドレス

セグメントを“スタート・アドレス”で指定した領域に配置します。

- 予約語 AT は，大文字，または小文字のいずれか一方で記述しなければなりません。大文字と小文字を混在することはできません。
- スタート・アドレスには，数値定数を記述します。

図 5—2 スタート・アドレス指定とセグメントの配置



注意 1. 指定したスタート・アドレスによって配置を行うと，セグメントが配置されるメモリ領域の範囲を越えてしまう場合は，エラーとなります。

2. セグメント疑似命令の AT 指定，または ORG 疑似命令によって配置アドレスを指定したセグメントに対して，リンク・ディレクティブでスタート・アドレスを指定することはできません。

（c）結合属性

ソース中に同名のセグメントが複数あった場合，結合させずにエラーとしたいときは“COMPLETE”，結合したいときは“SEQUENT（デフォルト）”をディレクティブ中に指定します。

SEQUENT	セグメントを出現順に，順次空きを作らないようにマージします。 BSEG はビット単位で出現順にマージします。
COMPLETE	同名のセグメントが複数存在する場合はエラーとします。

例を以下に示します。

```
MERGE DSEG1 : COMPLETE = RAM
```

（d）メモリ空間名

メモリ空間名は，セグメントを配置するメモリ空間を指定します。

- メモリ空間名として指定できるのは，次の 16 種類のうちのいずれかです。
  REGULAR，EX1，EX2，EX3，EX4，EX5，EX6，EX7，EX8，EX9，EX10，EX11，EX12，EX13，EX14，EX15
- メモリ空間名は，すべて大文字で記述します。
- メモリ空間名を省略した場合，REGULAR を指定したものとみなされます。

次にセグメントの配置先を示します。

メモリ領域	メモリ空間	セグメントの配置先
指定なし	指定なし	REGULAR 空間中のデフォルト状態のとき配置されるメモリ領域
指定なし	メモリ空間名	指定されたメモリ空間中の任意のメモリ領域
メモリ領域名	指定なし	REGULAR 空間の指定されたメモリ領域
メモリ領域名	メモリ空間名	指定されたメモリ空間の指定されたメモリ領域

この表では，セグメントの配置の対象となるメモリ領域を定義するということを中心として説明しています。なお，実際の配置アドレス決定時には，“AT（スタート・アドレス）”が指定されていれば，そのアドレスからセグメントを配置します。

たとえば，再配置属性が“CSEG FIXED”であるセグメントに，メモリ名“EX1”が指定された場合，セグメントが C0H ～ FFFFH の中に納まるように配置します。

注意を以下に示します。

- セグメント配置ディレクティブが指定しなかった入力セグメントは，アセンブル時にセグメント定義疑似命令で指定した再配置属性に従って配置アドレスが決定されます。
- セグメント名として指定したセグメントが存在しない場合は，エラーとなります。
- 同一のセグメントに対して，セグメント配置ディレクティブを複数回指定した場合は，エラーとなります。

5.2 予約語

ディレクティブ・ファイル中での予約語を次に示します。

予約語は，ディレクティブ・ファイル中で，ほかの意味（セグメント名やメモリ領域名など）に使用することはできません。

予約語	説明
MEMORY	メモリ・ディレクティブを指定
MERGE	セグメント配置ディレクティブを指定
AT	セグメント配置ディレクティブの配置属性（スタート・アドレス）を指定
SEQUENT	セグメント配置ディレクティブの結合属性（セグメントを結合する）を指定
COMPLETE	セグメント配置ディレクティブの結合属性（セグメントを結合しない）を指定
WALIGN	セグメント配置ディレクティブの結合属性（2 バイトアライン型結合）を指定
LALIGN	セグメント配置ディレクティブの結合属性（4 バイトアライン型結合）を指定

注意　予約語の記述は，大文字でも小文字でもかまいません。ただし，大文字と小文字を混在して記述することはできません。

例 MEMORY : 使用可
memory : 使用可
Memory : 使用不可

5. 3 コーディング例

リンク・ディレクティブのコーディング例を次に示します。

5. 3. 1 リンク・ディレクティブを指定する場合

- セグメント・タイプ，再配置属性が“CSEG UNIT”であるセグメント SEG1 に対して，アドレスを割り付けます。
領域は次のように宣言してあるものとします。

```
MEMORY ROM  : ( 0000H, 1000H )
MEMORY MEM1 : ( 1000H, 2000H )
```

- 入力セグメント SEG1 を ROM 領域中の 500H に割り付ける場合（次図（1）参照）

```
MERGE  SEG1 : AT ( 500H )
```

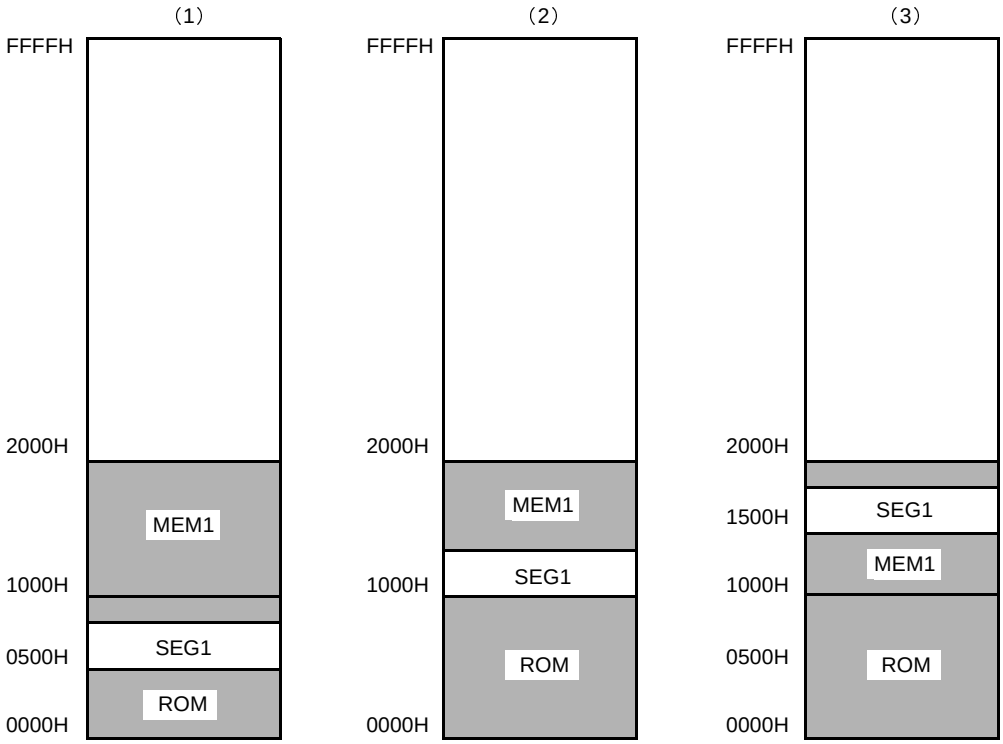
- 入力セグメント SEG1 をメモリ領域 MEM1 中に割り付ける場合（次図（2）参照）

```
MERGE  SEG1 : = MEM1
```

- 入力セグメント SEG1 をメモリ領域 MEM1 中の 1500H に割り付ける場合（次図 （3）参照）

```
MERGE SEG1 : AT ( 1500H ) = MEM178
```

図 5—3　入力セグメント SEG1 の割り付け例



5. 3. 2　コンパイラを使用する場合

コンパイラを使用する場合の，リンク・ディレクティブ・ファイルの作成方法を説明します。実際のターゲット・システムに合わせて作成し，リンク時に -d オプションで作成したファイルを指定してください。

なお，作成の際には，次のことに注意してください。

- RL78,78K0R C コンパイラは，ショート・ダイレクト・アドレス領域（saddr 領域）の一部を次のような RL78,78K0R C コンパイラ固有の目的で使用する場合があります。
  具体的には，FFEB4H ～ FFEDFH までの 44 バイトの領域です。

（a）-qr オプションを指定した場合の register 変数 [FFEB4H ～ FFEC3H]

（b）norec 関数の引数，自動変数 [FFEC4H ～ FFED3H]

（c）セグメント情報 [FFED4H ～ FFED7H]

（d）ランタイム・ライブラリの引数 [FFED8H ～ FFEDFH]

（e）標準ライブラリの作業用 （（a）と（b）の領域の一部）

注意　ユーザが標準ライブラリを使用しない場合，（e）の領域は使用されません。

次にリンク・ディレクティブ・ファイル（lk78k0r.dr）で RAM サイズを変更する例を示します。

メモリ・サイズの変更をする場合は，他の領域と重ならないように注意してください。変更の際には，使用するターゲット・デバイスのメモリ・マップを参照してください。

```
                        先頭アドレス  サイズ
memory RAM :     ( 0fcf00H, 002f20H )     → このサイズを大きくします。
memory SDR :     ( 0ffe20H, 000098H )       （必要に応じて，先頭アドレスも変更します。）
merge @@INIS : = SDR                      → セグメントの配置を指定しています。
merge @@DATS : = SDR                      → セグメントの配置を指定しています。
merge @@BITS : = SDR                      → セグメントの配置を指定しています。
```

セグメントの配置を変更したい場合は，merge 文を追加します。コンパイラ出力セクション名の変更機能を使用した場合，セグメントを独自に配置することができます（詳細については，「コンパイラ出力セクション名の変更（#pragma section）」を参照してください）。

セグメントの配置を変更した結果，配置するメモリが足りなくなった場合は，対応する memory 文を変更してください。

第 6 章　関数仕様

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，RL78,78K0R C コンパイラには，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。

この章では，RL78,78K0R C コンパイラが持つライブラリ関数，シミュレータで使用可能な関数について説明します。

6. 1　提供ライブラリ

RL78,78K0R C コンパイラで提供しているライブラリは，以下のとおりです。

標準ライブラリは，アプリケーション内で使用するときは，関連するヘッダ・ファイルをインクルードして，ライブラリ関数を使用します。

ランタイム・ライブラリは，標準ライブラリの一部ですが，RL78,78K0R C コンパイラが自動的に呼び出すルーチンで，C 言語ソースやアセンブリ言語ソースで記述する関数ではありません。

表 6―1　提供ライブラリ

ライブラリの種類	収録されている機能
標準ライブラリ	- 文字／文字列関数 - プログラム制御関数 - 特殊関数 - 入出力関数 - ユーティリティ関数 - 文字列／メモリ関数 - 数学関数 - 診断関数
ランタイム・ライブラリ	- インクリメント - デクリメント - 符号反転 - 1 の補数 - 論理否定 - 乗算 - 除算 - 剰余算 - 加算 - 減算 - 左シフト - 右シフト - 比較 - ビット AND - ビット OR

ライブラリの種類	収録されている機能
	- ビット XOR - 浮動小数点数からの変換 - 浮動小数点への変換 - bit からの変換 - スタートアップ・ルーチン - フラッシュ用スタートアップ・ルーチン - ブート用 main - フラッシュ用ベクタ・テーブル - 関数前後処理 - 初期化 - BCD 型変換 - 補助

6. 1. 1 標準ライブラリ

標準ライブラリに収録されている関数を示します。

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。

(1) 文字／文字列関数

関数名	機能	ヘッダ・ファイル	リエントラント性
isalpha	文字が英字（A ～ Z，a ～ z）であるかを判定する	ctype.h	○
isupper	文字が英大文字（A ～ Z）であるかを判定する	ctype.h	○
islower	文字が英小文字（a ～ z）であるかを判定する	ctype.h	○
isdigit	文字が数字（0 ～ 9）であるかを判定する	ctype.h	○
isalnum	文字が英数字（0 ～ 9，A ～ Z，a ～ z）であるかを判定する	ctype.h	○
isxdigit	文字が 16 進数字（0 ～ 9，A ～ F，a ～ f）であるかを判定する	ctype.h	○
isspace	文字が空白文字（空白，タブ，復帰，改行，垂直，タブ，改ページ）であるかを判定する	ctype.h	○
ispunct	文字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する	ctype.h	○
isprint	文字が表示可能文字であるかを判定する	ctype.h	○
isgraph	文字が空白以外の表示可能文字であるかを判定する	ctype.h	○
iscntrl	文字がコントロール文字であるかを判定する	ctype.h	○
isascii	文字が ASCII コードであるかを判定する	ctype.h	○
toupper	英小文字を英大文字に変換する	ctype.h	○
tolower	英大文字を英小文字に変換する	ctype.h	○
toascii	ASCII コードへ変換する	ctype.h	○
_toupper	入力文字から“a”を引き，“A”を加える	ctype.h	○
toup		ctype.h	○

関数名	機能	ヘッダ・ファイル	リエントラント性
_tolower	入力文字から“A”を引き，“a”を加える	ctype.h	○
tolow		ctype.h	○

○ ：リエントラント

(2) プログラム制御関数

関数名	機能	ヘッダ・ファイル	リエントラント性
setjmp	呼び出し時の環境をセーブする	setjmp.h	×
longjmp	setjmp でセーブされた環境を復帰する	setjmp.h	×

× ：リエントラントでない

(3) 特殊関数

関数名	機能	ヘッダ・ファイル	リエントラント性
va_start	可変個の引数の処理のための設定を行う	stdarg.h	○
va_starttop	可変個の引数の処理のための設定を行う	stdarg.h	○
va_arg	可変個の引数を処理する	stdarg.h	○
va_end	可変個の引数の処理の終了を知らせる	stdarg.h	○

○ ：リエントラント

(4) 入出力関数

関数名	機能	ヘッダ・ファイル	リエントラント性
sprintf	フォーマットに従ってデータを文字列に書き込む	stdio.h	△
sscanf	入力文字列からフォーマットに従ってデータを読み込む	stdio.h	△
printf	フォーマットに従ってデータを SFR に出力する	stdio.h	△
scanf	SFR からフォーマットに従ってデータを読み込む	stdio.h	△
vprintf	フォーマットに従ってデータを SFR に出力する	stdio.h	△
vsprintf	フォーマットに従ってデータを文字列に書き込む	stdio.h	△
getchar	SFR から 1 文字読み込む	stdio.h	○
gets	文字列の読み取る	stdio.h	○
putchar	SFR に 1 文字出力する	stdio.h	○
puts	文字列を出力する	stdio.h	○

関数名	機能	ヘッダ・ファイル	リエントラント性
__putc	opaque に 1 文字出力	stdio.h	○

○ ：リエントラント

△ ：浮動小数点未対応のものはリエントラント

(5) ユーティリティ関数

関数名	機能	ヘッダ・ファイル	リエントラント性
atoi	10 進整数文字列を int に変換する	stdlib.h	○
atol	10 進整数文字列を long に変換する	stdlib.h	○
strtol	文字列を long に変換する	stdlib.h	○
strtoul	文字列を unsigned long に変換する	stdlib.h	○
calloc	配列の領域を割り付けて 0 で初期化する	stdlib.h	○
free	割り付けられているブロックを解放する	stdlib.h	○
malloc	ブロックを割り付ける	stdlib.h	○
realloc	ブロックを再度割り付ける	stdlib.h	○
abort	プログラムを異常終了する	stdlib.h	○
atexit	正常終了時に呼び出される関数を登録する	stdlib.h	×
exit	プログラムを終了する	stdlib.h	×
abs	int 型の値の絶対値を求める	stdlib.h	○
labs	long 型の値の絶対値を求める	stdlib.h	○
div	int 型の除算を行い，商と剰余を求める	stdlib.h	×
ldiv	long 型の除算を行い，商と剰余を求める	stdlib.h	×
brk	ブレーク値をセットする	stdlib.h	×
sbrk	ブレーク値を増減する	stdlib.h	×
atof	10 進整数文字列を double に変換する	stdlib.h	×
strtod	文字列を double に変換する	stdlib.h	×
itoa	int を文字列に変換する	stdlib.h	○
ltoa	long を文字列に変換する	stdlib.h	○
ultoa	unsigned long を文字列に変換する	stdlib.h	○
rand	疑似乱数を発生する	stdlib.h	×
srand	疑似乱数の発生状態を初期化する	stdlib.h	×
bsearch	バイナリ・サーチを行う	stdlib.h	○
qsort	クイック・ソートを行う	stdlib.h	○
strbrk	ブレーク値をセットする	stdlib.h	○
strsbrk	ブレーク値を増減する	stdlib.h	○
stritoa	int を文字列に変換する	stdlib.h	○

関数名	機能	ヘッダ・ファイル	リエントラント性
strltoa	long を文字列に変換する	stdlib.h	○
strultoa	unsigned long を文字列に変換する	stdlib.h	○

○ ：リエントラント
× ：リエントラントでない

(6) 文字列／メモリ関数

関数名	機能	ヘッダ・ファイル	リエントラント性
memcpy	バッファを指定文字数分コピーする	string.h	○
memmove	バッファを指定文字数分コピーする	string.h	○
strcpy	文字列をコピーする	string.h	○
strncpy	文字列の先頭から指定の文字数分コピーする	string.h	○
strcat	文字列に文字列を追加する	string.h	○
strncat	文字列に指定文字数分の文字列を追加する	string.h	○
memcmp	2 つのバッファの指定文字数分を比較する	string.h	○
strcmp	2 つの文字列を比較する	string.h	○
strncmp	2 つの文字列の指定文字数分を比較する	string.h	○
memchr	指定文字数分のバッファから指定文字を探す	string.h	○
strchr	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す	string.h	○
strrchr	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す	string.h	○
strspn	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る	string.h	○
strcspn	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る	string.h	○
strpbrk	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る	string.h	○
strstr	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る	string.h	○
strtok	文字列を区切り文字以外からなる文字列に分解する	string.h	×
memset	バッファの指定文字数分を指定文字で初期化する	string.h	○
strerror	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す	string.h	○
strlen	文字列の長さを求める	string.h	○
strcoll	地域特有の情報に基づいて 2 つの文字列を比較する	string.h	○
strxfrm	地域特有の情報に基づいて文字列を変換する	string.h	○

○ ：リエントラント

× ：リエントラントでない

(7) 数学関数

関数名	機能	ヘッダ・ファイル	リエントラント性
acos	acos を求める	math.h	×
asin	asin を求める	math.h	×
atan	atan を求める	math.h	×
atan2	atan2 を求める	math.h	×
cos	cos を求める	math.h	×
sin	sin を求める	math.h	×
tan	tan を求める	math.h	×
cosh	cosh を求める	math.h	×
sinh	sinh を求める	math.h	×
tanh	tanh を求める	math.h	×
exp	指数関数を求める	math.h	×
frexp	仮数部と指数部を求める	math.h	×
ldexp	x * 2 ^ exp を求める	math.h	×
log	自然対数を求める	math.h	×
log10	10 を底とした対数を求める	math.h	×
modf	小数部と整数部を求める	math.h	×
pow	x の y 乗を求める	math.h	×
sqrt	平方根を求める	math.h	×
ceil	x より小さくない最小の整数を求める	math.h	×
fabs	浮動小数点数 x の絶対値を求める	math.h	×
floor	x より大きくない最大の整数を求める	math.h	×
fmod	x/y の余りを求める	math.h	×
matherr	浮動小数点数を扱うライブラリの例外処理を求める	math.h	×
acosf	acos を求める	math.h	×
asinf	asin を求める	math.h	×
atanf	atan を求める	math.h	×
atan2f	y/x の atan を求める	math.h	×
cosf	cos を求める	math.h	×
sinf	sin を求める	math.h	×
tanf	tan を求める	math.h	×
coshf	cosh を求める	math.h	×
sinhf	sinh を求める	math.h	×
tanhf	tanh を求める	math.h	×

関数名	機能	ヘッダ・ファイル	リエントラント性
expf	指数関数を求める	math.h	×
frexpf	仮数部と指数部を求める	math.h	×
ldexpf	x * 2 ^ exp を求める	math.h	×
logf	自然対数を求める	math.h	×
log10f	10 を底とした対数を求める	math.h	×
modff	小数部と整数部を求める	math.h	×
powf	x の y 乗を求める	math.h	×
sqrtf	平方根を求める	math.h	×
ceilf	x より小さくない最小の整数を求める	math.h	×
fabsf	浮動小数点数 x の絶対値を求める	math.h	×
floorf	x より大きくない最大の整数を求める	math.h	×
fmodf	x/y の余りを求める	math.h	×

×　：リエントラントでない

(8) 診断関数

関数名	機能	ヘッダ・ファイル	リエントラント性
__assertfail	assert マクロをサポートする	assert.h	○

○　：リエントラント

6. 1. 2　ランタイム・ライブラリ

ランタイム・ライブラリに収録されている関数を示します。

これらの演算の命令は，@@ などを関数名の頭に付けた形式で呼び出されます。ただし，cstart，cstarte，cprep，cdisp は，先頭に _@ を付加した形式で呼び出されます。

なお，以下の表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。

(1) インクリメント

関数名	機能
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
lsincr	signed long をインクリメントする（RAM 配置用）

関数名	機能
luincr	unsigned long をインクリメントする（RAM 配置用）
fincr	float をインクリメントする（RAM 配置用）

（2）デクリメント

関数名	機能
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
lsdecr	signed long をデクリメントする（RAM 配置用）
ludecr	unsigned long をデクリメントする（RAM 配置用）
fdecr	float をデクリメントする（RAM 配置用）

（3）符号反転

関数名	機能
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
lsrevr	signed long を符号反転する（RAM 配置用）
lurevr	unsigned long を符号反転する（RAM 配置用）
frevr	float を符号反転する（RAM 配置用）

（4）1 の補数

関数名	機能
lscom	signed long の 1 の補数を求める
lucom	unsigned long の 1 の補数を求める
lscomr	signed long の 1 の補数を求める（RAM 配置用）
lucomr	unsigned long の 1 の補数を求める（RAM 配置用）

（5）論理否定

関数名	機能
lsnot	signed long の否定を求める
lunot	unsigned long の否定を求める

（6）乗算

関数名	機能
csmul	signed char 同士の乗算を行う
cumul	unsigned char 同士の乗算を行う
ismul	signed int 同士の乗算を行う
iumul	unsigned int 同士の乗算を行う
lsmul	signed long 同士の乗算を行う（MDUC レジスタがゼロとして処理）
lumul	unsigned long 同士の乗算を行う（MDUC レジスタがゼロとして処理）
fmul	float 同士の乗算を行う
iumulr	unsigned int 同士の乗算を行う（RAM 配置用）
lsmulr	signed long 同士の乗算を行う（RAM 配置用）
lumulr	unsigned long 同士の乗算を行う（RAM 配置用）
fmulr	float 同士の乗算を行う（RAM 配置用）

（7）除算

関数名	機能
csdiv	signed char 同士の除算を行う
cudiv	unsigned char 同士の除算を行う
isdiv	signed int 同士の除算を行う
iudiv	unsigned int 同士の除算を行う
lsdiv	signed long 同士の除算を行う
ludiv	unsigned long 同士の除算を行う
fdiv	float 同士の除算を行う
csdivr	signed char 同士の除算を行う（RAM 配置用）
cudivr	unsigned char 同士の除算を行う（RAM 配置用）
isdivr	signed int 同士の除算を行う（RAM 配置用）
iudivr	unsigned int 同士の除算を行う（RAM 配置用）
lsdivr	signed long 同士の除算を行う（RAM 配置用）
ludivr	unsigned long 同士の除算を行う（RAM 配置用）
fdivr	float 同士の除算を行う（RAM 配置用）

（8）剰余算

関数名	機能
csrem	signed char 同士の剰余算を行う
curem	unsigned char 同士の剰余算を行う
isrem	signed int 同士の剰余算を行う
iurem	unsigned int 同士の剰余算を行う

関数名	機能
lsrem	signed long 同士の剰余算を行う
lurem	unigned long 同士の剰余算を行う
csremr	signed char 同士の剰余算を行う（RAM 配置用）
curemr	unsigned char 同士の剰余算を行う（RAM 配置用）
isremr	signed int 同士の剰余算を行う（RAM 配置用）
iuremr	unsigned int 同士の剰余算を行う（RAM 配置用）
lsremr	signed long 同士の剰余算を行う（RAM 配置用）
luremr	unigned long 同士の剰余算を行う（RAM 配置用）

（9）加算

関数名	機能
lsadd	signed long 同士の加算を行う
luadd	unsigned long 同士の加算を行う
fadd	float 同士の加算を行う
lsaddr	signed long 同士の加算を行う（RAM 配置用）
luaddr	unsigned long 同士の加算を行う（RAM 配置用）
faddr	float 同士の加算を行う（RAM 配置用）

（10）減算

関数名	機能
lssub	signed long 同士の減算を行う
lusub	unsigned long 同士の減算を行う
fsub	float 同士の減算を行う
lssubr	signed long 同士の減算を行う（RAM 配置用）
lusubr	unsigned long 同士の減算を行う（RAM 配置用）
fsubr	float 同士の減算を行う（RAM 配置用）

（11）左シフト

関数名	機能
lslsh	signed long の左シフトを行う
lulsh	unsigned long の左シフトを行う
lslshr	signed long の左シフトを行う（RAM 配置用）
lulshr	unsigned long の左シフトを行う（RAM 配置用）

（12）右シフト

関数名	機能
lsrsh	signed long の右シフトを行う
lursh	unsigned long の右シフトを行う
lsrshr	signed long の右シフトを行う（RAM 配置用）
lurshr	unsigned long の右シフトを行う（RAM 配置用）

（13）比較

関数名	機能
cscmp	signed char 同士の比較を行う
iscmp	signed int 同士の比較を行う
lscmp	signed long 同士の比較を行う
lucmp	unsigned long 同士の比較を行う
fcmp	float 同士の比較を行う
cscmpr	signed char 同士の比較を行う（RAM 配置用）
iscmpr	signed int 同士の比較を行う（RAM 配置用）
lscmpr	signed long 同士の比較を行う（RAM 配置用）
lucmpr	unsigned long 同士の比較を行う（RAM 配置用）
fcmpr	float 同士の比較を行う（RAM 配置用）

（14）ビット AND

関数名	機能
lsband	signed long 同士の AND をとる
luband	unsigned long 同士の AND をとる
lsbandr	signed long 同士の AND をとる（RAM 配置用）
lubandr	unsigned long 同士の AND をとる（RAM 配置用）

（15）ビット OR

関数名	機能
lsbor	signed long 同士の OR をとる
lubor	unsigned long 同士の OR をとる
lsborr	signed long 同士の OR をとる（RAM 配置用）
luborr	unsigned long 同士の OR をとる（RAM 配置用）

（16）ビット XOR

関数名	機能
lsbxor	signed long 同士の XOR をとる
lubxor	unsigned long 同士の XOR をとる
lsbxorr	signed long 同士の XOR をとる（RAM 配置用）
lubxorr	unsigned long 同士の XOR をとる（RAM 配置用）

（17）浮動小数点数からの変換

関数名	機能
ftols	float から signed long に変換する
ftolu	float から unsigned long に変換する
ftolsr	float から signed long に変換する（RAM 配置用）
ftolur	float から unsigned long に変換する（RAM 配置用）

（18）浮動小数点への変換

関数名	機能
lstof	signed long から float に変換する
lutof	unsigned long から float に変換する
lstofr	signed long から float に変換する（RAM 配置用）
lutofr	unsigned long から float に変換する（RAM 配置用）

（19）bit からの変換

関数名	機能
btol	bit を long に変換する
btolr	bit を long に変換する（RAM 配置用）

（20）スタートアップ・ルーチン

関数名	機能

cstart	スタートアップ・ルーチン
	- atexit 関数で関数を登録する領域（4 * 32 バイト）を確保し，先頭のラベル名を _@FNCTBL とする
	- ブレーク領域（32 バイト）を確保し，先頭のラベル名を _@MEMTOP とし，領域の次のアドレスのラベル名を _@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・ルーチンの先頭アドレスを指定する
	`@@VECT00 CSEG AT 0000H`
	`DW _@cstart`
	- レジスタ・バンクを RB0 に設定する
	- ミラー領域を設定する
	- SP レジスタにスタック領域の最終アドレス +1 をセットする
	- hdwinit 関数を呼び出す
	- stkinit 関数を呼び出す
	- saddr 領域（レジスタ・バンク領域は除く）を 0 クリアする
	- 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
	- 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
	- 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
	- main 関数（ユーザ・プログラム）を呼び出す（スタート・アップ・モジュールの場合）
	- exit 関数を引数 0 で呼び出す（スタート・アップ・モジュールの場合）
	- boot_main 関数を呼び出す（ブート用スタート・アップ・モジュールの場合）
	- フラッシュ領域分岐テーブルの先頭（ITBLTOP）に分岐し，フラッシュ用のスタート・アップ・モジュールに処理を移す（ブート用スタート・アップ・モジュールの場合）

（21）フラッシュ用スタートアップ・ルーチン

関数名	機能

cstarte	フラッシュ用スタートアップ・ルーチン
	- 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・ルーチンの先頭アドレスを指定する
	- フラッシュ領域分岐テーブルを次のように定義する（ITBLTOR は，フラッシュ領域分岐テーブルの先頭アドレス）
	`@EVECT00 CSEG AT ITBLTOP`
	`BR _@cstarte`
	- SP レジスタにスタック領域の最終アドレス +1 をセットする
	- stkinit 関数を呼び出す
	- 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 関数を引数 0 で呼び出す

（22）ブート用 main

関数名	機能
boot_main	ブート領域の main 関数処理を行う

（23）フラッシュ用ベクタ・テーブル

関数名	機能
vect00 ～ vect7e	-zf オプション指定時の割り込みベクタ・テーブルの設定を行う

（24）関数前後処理

関数名	機能
hdwinit	CPU リセット直後に周辺装置（sfr）の初期化処理を行う
cprep3	関数の前処理（レジスタ変数用 saddr 領域を含む）を行う
cdisp3	関数の後処理（レジスタ変数用 saddr 領域を含む）を行う
cpre3e	関数の前処理（レジスタ変数用 saddr 領域を含む）を行う
cdis3e	関数の後処理（レジスタ変数用 saddr 領域を含む）を行う

（25）初期化

関数名	機能
stkinit	スタック領域の初期化を行う

（26）BCD 型変換

関数名	機能
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 2 バイト bcd に変換する

（27）補助

関数名	機能
indao	定型命令パターン置換用
ifdao	定型命令パターン置換用
inado	定型命令パターン置換用
ifado	定型命令パターン置換用
lnd0	定型命令パターン置換用

関数名	機能
lfd0	定型命令パターン置換用
ln0d	定型命令パターン置換用
lf0d	定型命令パターン置換用
lnd0o	定型命令パターン置換用
lfd0o	定型命令パターン置換用
ln0do	定型命令パターン置換用
lf0do	定型命令パターン置換用
df1in	定型命令パターン置換用
df1de	定型命令パターン置換用
dn4in	定型命令パターン置換用
dn4ip	定型命令パターン置換用
df4in	定型命令パターン置換用
df4ip	定型命令パターン置換用
dn4ino	定型命令パターン置換用
dn4ipo	定型命令パターン置換用
df4ino	定型命令パターン置換用
df4ipo	定型命令パターン置換用
dn4de	定型命令パターン置換用
dn4dp	定型命令パターン置換用
df4de	定型命令パターン置換用
df4dp	定型命令パターン置換用
dn4deo	定型命令パターン置換用
dn4dpo	定型命令パターン置換用
df4deo	定型命令パターン置換用
df4dpo	定型命令パターン置換用
divuw	78K0 divuw 命令互換
mulsw	符号付き int 乗算
muluw	符号なし int 乗算
macsw	符号付き積和演算
macuw	符号なし積和演算
divuwr	78K0 divuw 命令互換（RAM 配置用）

6.2 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，戻り値はレジスタにより受け渡しが行われます。

ただし，可能であれば，第 1 引数もレジスタにより受け渡します。

6.2.1 引　　数

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

詳細は，「3.3.2 通常関数呼び出しインタフェース」を参照してください。

6.2.2 戻 り 値

戻り値は，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，DE に格納します。

詳細は，「3.3.1 戻 り 値」を参照してください。

6.2.3 個々のライブラリによる使用レジスタの保存

HL 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。

引数と戻り値の受け渡し手順の例（スモール・モデル，ミディアム・モデルの場合）を次に示します。

呼び出す関数を示します。

```
"long func ( int a, long b, char *c ) ;"
```

（1）引数をスタックに積む（関数呼び出し元）

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（関数呼び出し元

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（関数呼び出し先）

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（関数呼び出し先）

（5）関数 func の処理を行い，戻り値をレジスタに格納する（関数呼び出し先）

戻り値“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（関数呼び出し先）

（7）退避したレジスタを復帰する（関数呼び出し先）

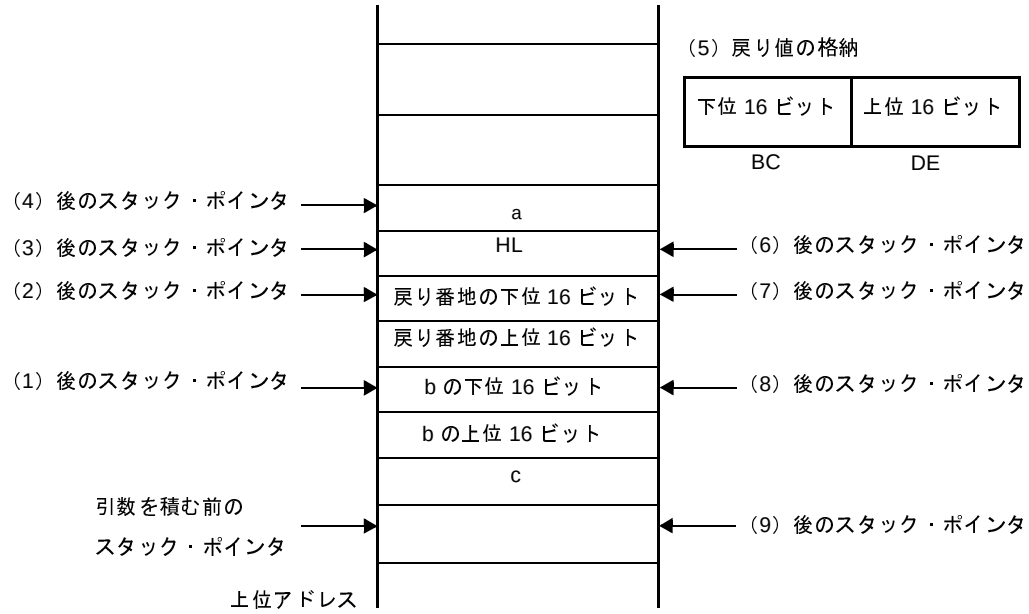
（8）ret 命令で呼び出した関数に制御を戻す（関数呼び出し先）

（9）引数をスタックから取り除く（関数呼び出し元）

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。

スタック・ポインタに 6 が加えられます。

図 6—1 関数呼び出し時のスタック領域



6. 3 ヘッダ・ファイル

RL78,78K0R 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。

RL78,78K0R C コンパイラのヘッダ・ファイルを次に示します。

6. 3. 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。

```
isalpha，isupper，islower，isdigit，isalnum，isxdigit，isspace，ispunct，isprint，isgraph，
iscntrl，isascii，toupper，tolower，toascii，_toupper/toup，_tolower/tolow
```

6. 3. 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。

```
setjmp，longjmp
```

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

```
typedef int     jmp_buf[12]
```

6. 3. 3 stdarg.h

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。

```
va_start，va_starttop，va_arg，va_end
```

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  ;
```

6. 3. 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。

```
sprintf，sscanf，printf，scanf，vprintf，vsprintf，getchar，gets，putchar，puts，__putc
```

次のマクロ名を宣言しています。

```
#define  EOF     ( -1 )
```

6. 3. 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。

```
atoi, atol, strtol, strtoul, calloc, free, malloc, realloc, abort, atexit, exit, abs, labs,
div, ldiv, brk, sbrk, atof, strtod, itoa, ltoa, ultoa, rand, srand, bsearch, qsort, strbrk,
strsbrk, stritoa, strltoa, strultoa
```

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

```
typedef struct {
        int     quot ;
        int     rem ;
} div_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. 3. 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。

```
memcpy, memmove, strcpy, strncpy, strcat, strncat, memcmp, strcmp, strncmp, memchr, strchr,
strrchr, strspn, strcspn, strpbrk, strstr, strtok, memset, strerror, strlen, strcoll,
strxfrm
```

6. 3. 7 error.h

error.h は，errno.h をインクルードしています。

6. 3. 8 errno.h

次のオブジェクトが宣言，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 ;
```

6. 3. 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 )
```

6. 3. 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 ) -> member ) )
```

備考 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。

6. 3. 11 math.h

math.h では，次の関数が定義されています。

```
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, logf, log10f, modff, powf, sqrtf, ceilf,
fabsf, floorf, fmodf
```

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
```

6. 3. 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 */
```

6. 3. 13 assert.h

assert.h では，次の関数が定義されています。

```
__assertfail
```

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 1 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，次のように宣言し，__assertfail の定義も行いません。

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

6.4 リエントラント性

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。

RL78,78K0R C 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，次の関数は，リエントラントでないので注意してください。

- リエントラント化できない関数

```
setjmp，longjmp，atexit，exit
```

- 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
```

- 浮動小数点を扱う関数

```
sprintf，sscanf，printf，scanf，vprintf，vsprintf 注
atof，strtod，すべての数学関数
```

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。

6.5 引数／戻り値に適した標準ライブラリの使用

標準ライブラリの引数／戻り値にポインタを指定するものは，メモリ・モデルに応じて適切なライブラリがリンクされます。

メモリ・モデルのデフォルトでないポインタを扱いたい場合，以下の標準関数名で呼び出すことで，そのポインタに適切なライブラリをリンクすることが可能です。

＜関数名＞_n ： ポインタを常に near として扱う

＜関数名＞_f ： ポインタを常に far として扱う

たとえば，スモール・モデル選択時に，strcmp 関数の引数として far ポインタを指定することができます。

例を以下に示します。

```
#include <string.h>

__far char *sf1 ;
__far char *sf2 ;

void main ( void ) {
          :
        r = strcmp_f ( sf1, sf2 ) ;
          :
}
```

注意事項を以下に示します。

- スモール・モデル，ミディアム・モデル指定時の可変個引数を取り扱う入出力関数 sprintf/printf/vprintf/vsprintf/sscanf/scanf のポインタ引数は，near ポインタとして扱います。関数ポインタは使用することができません。関数ポインタ使用時，あるいは far ポインタ使用時は，printf_f を使用し，可変個引数のポインタをすべて far ポインタにキャストする必要があります。
  ラージ・モデル指定時の可変個引数を取り扱う入出力関数 sprintf/printf/vprintf/vsprintf/sscanf/scanf のポインタ引数は，far ポインタとして扱います。
- スモール・モデル，ミディアム・モデル指定時の可変個引数を取り扱う特殊関数 va_start/va_starttop/va_arg/va_end のポインタ引数は，near ポインタとして扱います。関数ポインタは使用することができません。

6.6 文字／文字列関数

文字／文字列関数には，次のものがあります。

関数名	機能
isalpha	文字が英字（A ～ Z，a ～ z）であるかを判定
isupper	文字が英大文字（A ～ Z）であるかを判定
islower	文字が英小文字（a ～ z）であるかを判定
isdigit	文字が数字（0 ～ 9）であるかを判定
isalnum	文字が英数字（0 ～ 9，A ～ Z，a ～ z）であるかを判定
isxdigit	文字が 16 進数字（0 ～ 9，A ～ F，a ～ f）であるかを判定
isspace	文字が空白文字（空白，タブ，復帰，改行，垂直，タブ，改ページ）であるかを判定
ispunct	文字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
isprint	文字が表示可能文字であるかを判定
isgraph	文字が空白以外の表示可能文字であるかを判定
iscntrl	文字がコントロール文字であるかを判定
isascii	文字が ASCII コードであるかを判定
toupper	英小文字を英大文字に変換
tolower	英大文字を英小文字に変換
toascii	ASCII コードへ変換
_toupper	入力文字から“a”を引き，“A”を加える
toup	
_tolower	入力文字から“A”を引き，“a”を加える
tolow	

isalpha

c が英字（A ～ Z，a ～ z）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isalpha ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が英字である場合 ： 1 文字 *c* が英字でない場合 ： 0

[詳細説明]

- *c* が英字（A ～ Z，a ～ z）の場合は，1 を返します。
  それ以外の場合は，0 を返します。

isupper

c が英大文字（A ～ Z）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isupper ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が英大文字である場合 ： 1 文字 *c* が英大文字でない場合 ： 0

[詳細説明]

- *c* が英大文字（A ～ Z）の場合，1 を返します。
  それ以外の場合は，0 を返します。

islower

c が英小文字（a ～ z）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int islower ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が英小文字である場合 ： 1 文字 *c* が英小文字でない場合 ： 0

[詳細説明]

- *c* が英小文字（a ～ z）の場合，1 を返します。
  それ以外の場合は，0 を返します。

isdigit

c が数字（0 ～ 9）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isdigit ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が 10 進数である場合 ： 1 文字 *c* が 10 進数でない場合 ： 0

[詳細説明]

- *c* が数字（0 ～ 9）の場合，1 を返します。
  それ以外の場合は，0 を返します。

isalnum

c が英数字（0 ～ 9，A ～ Z，a ～ z）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isalnum ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が英数字である場合 ： 1 文字 *c* が英数字でない場合 ： 0

[詳細説明]

- *c* が英数字（0 ～ 9，A ～ Z，a ～ z）の場合，1 を返します。
  それ以外の場合は，0 を返します。

isxdigit

c が 16 進数字（0 ～ 9，A ～ F，a ～ f）であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isxdigit ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が 16 進数字である場合 ： 1 文字 *c* が 16 進数字でない場合 ： 0

[詳細説明]

- *c* が英数字 16 進数字（0 ～ 9，A ～ F，a ～ f）の場合，1 を返します。
  それ以外の場合は，0 を返します。

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。

[指定形式]

```
#include <ctype.h>
int isspace ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が空白文字である場合 ： 1 文字 *c* が空白文字でない場合 ： 0

[詳細説明]

- *c* が空白文字（空白，タブ，復帰，改行，垂直，タブ，改ページ）の場合，1 を返します。
  それ以外の場合は，0 を返します。

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。

[指定形式]

```
#include <ctype.h>
int ispunct ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が空白文字と英数字以外の表示可能文字である場合 ： 1 文字 *c* が空白文字と英数字以外の表示可能文字でない場合 ： 0

[詳細説明]

- *c* が空白文字と英数字以外の表示可能文字の場合，1 を返します。
  それ以外の場合は，0 を返します。

isprint

c が表示可能文字であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isprint ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c : 判定する文字	文字 *c* が表示可能な文字である場合 : 1 文字 *c* が表示可能な文字でない場合 : 0

[詳細説明]

- *c* が表示可能文字の場合，1 を返します。
  それ以外の場合は，0 を返します。

isgraph

c が空白以外の表示可能文字であるかを判定します。

[指定形式]

```
#include <ctype.h>
int isgraph ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が空白以外の表示可能文字である場合 ： 1 文字 *c* が空白であるか表示可能文字でない場合 ： 0

[詳細説明]

- *c* が空白以外の表示可能文字の場合，1 を返します。
  それ以外の場合は，0 を返します。

iscntrl

c がコントロール文字であるかを判定します。

[指定形式]

```
#include <ctype.h>
int iscntrl ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* がコントロール文字である場合 ： 1 文字 *c* がコントロール文字でない場合 ： 0

[詳細説明]

- *c* がコントロール文字の場合，1 を返します。
  それ以外の場合は，0 を返します。

isascii

c が ASCII コードであるかを判定します。

[指定形式]

```
#include <ctype.h>
int isascii( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 判定する文字	文字 *c* が ASCII コードである場合 ： 1 文字 *c* が ASCII コードでない場合 ： 0

[詳細説明]

- *c* が ASCII コードの場合，1 を返します。
  それ以外の場合は，0 を返します。

toupper

英小文字を英大文字に変換します。

[指定形式]

```
#include <ctype.h>
int toupper ( int c ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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

英大文字を英小文字に変換します。

[指定形式]

```
#include <ctype.h>
int tolower ( int c ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。

toascii

ASCII コードへの変換を行います。

[指定形式]

```
#include <ctype.h>
int toascii ( int c ) ;
```

[引数／戻り値]

引数	戻り値
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 *c*

[詳細説明]

- *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。

_toupper

_toupper は，*c* から“a”を引き，“A”を加えます。
（_toupper と toup はまったく同じです。）

備考 a：英小文字，A：英大文字

[指定形式]

```
#include <ctype.h>
int _toupper ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 変換される文字	*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

[詳細説明]

- _toupper は，toup 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。

toup

toup は，*c* から“a”を引き，“A”を加えます。
（_toupper と toup はまったく同じです。）

備考 a：英小文字，A：英大文字

[指定形式]

```
#include <ctype.h>
int toup ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 変換される文字	*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

[詳細説明]

- toup 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す。

_tolower

_tolower は，*c* から“A”を引き，“a”を加えます。
（_tolower と tolow はまったく同じです。）

備考 a：英小文字，A：英大文字

[指定形式]

```
#include <ctype.h>
int _tolower ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 変換される文字	*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

[詳細説明]

- _tolower は，tolow 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。

tolow

tolow は，*c* から“A”を引き，“a”を加えます。
（_tolower と tolow はまったく同じです。）

備考 a：英小文字，A：英大文字

[指定形式]

```
#include <ctype.h>
int tolow ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 変換される文字	*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

[詳細説明]

- tolow 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す。

6.7　プログラム制御関数

プログラム制御関数には，次のものがあります。

関数名	機能
setjmp	呼び出し時の環境をセーブ
longjmp	setjmp でセーブされた環境を復帰

setjmp

呼び出し時の環境をセーブします。

[指定形式]

```
#include <setjmp.h>
int setjmp ( jmp_buf env ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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

setjmp でセーブされた環境を復帰します。

[指定形式]

#include <setjmp.h>

void longjmp (jmp_buf *env*, int *val*) ;

[引数／戻り値]

引数	戻り値
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。

[詳細説明]

- 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。

6. 8　特殊関数

特殊関数には，次のものがあります。

関数名	機能
va_start	可変個の引数の処理のための設定を行う
va_starttop	可変個の引数の処理のための設定を行う
va_arg	可変個の引数を処理
va_end	可変個の引数の処理の終了を知らせる

va_start

可変個の引数の処理のための設定を行います（マクロ）。

[指定形式]

```
#include <stdarg.h>
void va_start ( va_list ap, parmN ) ;
```

備考 va_list は，stdarg.h で typedef 定義されています。

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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引数の名前です。
- *parmN* がレジスタ記憶クラスで宣言されている場合は，正常動作は保証されません。
- *parmN* が第一引数の場合は，正常動作は保証されません（代わりに va_starttop を使用してください）。

va_starttop

可変個の引数の処理のための設定を行います（マクロ）。

[指定形式]

#include <stdarg.h>

void va_starttop (va_list *ap*, *parmN*) ;

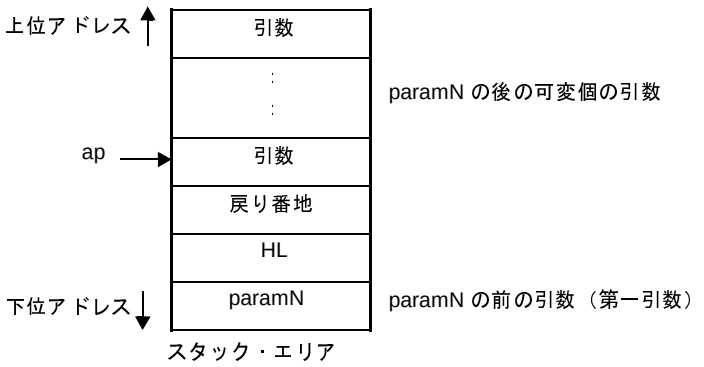
備考 va_list は，stdarg.h で typedef 定義されています。

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *ap* は，va_list 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ストアします。
- *parmN* は，関数定義上での右端，かつ 1 番目の引数の名前です。
- *parmN* がレジスタ記憶クラスで宣言されている場合は，正常動作は保証されません。
- *parmN* が第一引数以外の場合は，正常動作は保証されません。



va_arg

可変個の引数の処理を行います（マクロ）。

[指定形式]

#include <stdarg.h>

type va_arg (va_list *ap*, type) ;

備考 va_list は，stdarg.h で typedef 定義されています。

[引数／戻り値]

引数	戻り値
ap ： 引数リストの処理のための変数 type ： 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* がヌル・ポインタの場合 ： 0

[詳細説明]

- va_arg では，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されません）。
- 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* がヌル・ポインタの場合は，type 型の 0 を返します。
- RL78,78K0R C コンパイラでは，引数リストとしてポインタを指定する場合，ミディアム・モデルでは near データ・ポインタ（2 バイト長）をラージ・モデルでは far データ・ポインタ（4 バイト長）とする必要があります。また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数リストとして指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

va_end

可変個の引数の処理の終了を知らせます（マクロ）。

[指定形式]

```
#include <stdarg.h>
void va_end ( va_list ap ) ;
```

備考 va_list は，stdarg.h で typedef 定義されています。

[引数／戻り値]

引数	戻り値
ap ： 可変個の引数の処理のための変数	なし

[詳細説明]

- 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* にヌル・ポインタをセットします。

6.9 入出力関数

入出力関数には，次のものがあります。

関数名	機能
sprintf	フォーマットに従ってデータを文字列に書き込み
sscanf	入力文字列からフォーマットに従ってデータを読み込み
printf	フォーマットに従ってデータを SFR に出力
scanf	SFR からフォーマットに従ってデータを読み込み
vprintf	フォーマットに従ってデータを SFR に出力
vsprintf	フォーマットに従ってデータを文字列に書き込み
getchar	SFR から 1 文字読み込み
gets	文字列の読み取り
putchar	SFR に 1 文字出力
puts	文字列を出力
__putc	opaque に 1 文字出力

sprintf

フォーマットに従ってデータを文字列に書きます。

[指定形式]

```
#include <stdio.h>
int sprintf ( char *s, const char *format, … ) ;
```

[引数／戻り値]

引数	戻り値
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（終端のヌル文字は数えません）

[詳細説明]

- 書式に対して実引数が不足しているときの動作は保証されません。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）。

（1）0 個以上のフラグ（後述）は，変換仕様の意味を修飾します。

（2）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。

- 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 * を指定することができます。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。

フラグは次のとおりです。

フラグ	内容
-	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 - 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（# なしのデフォルトでは，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（# なしのデフォルトでは，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。
0	先頭フィールドを 0 で埋めます。 - フラグと同時に指定した場合は，0 フラグは無視されます。 d，i，o，u，x，X 変換で精度を指定している場合は，0 フラグは無視されます。

変換指定は次のとおりです。

変換指定	内容
d，i	int 引数を符号付き 10 進（d または i）表記に変換します。
o	int 引数を無符号 8 進（o）表記に変換します。
u	int 引数を無符号 10 進（u）表記に変換します。

変換指定	内容
x，X	int 引数を無符号 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。

精度指定	内容
f	double 引数を [-]*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 [-]*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 - 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，-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。
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ヌル文字（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，配列はヌ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 を付けます）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），動作は保証されません。

- 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)”

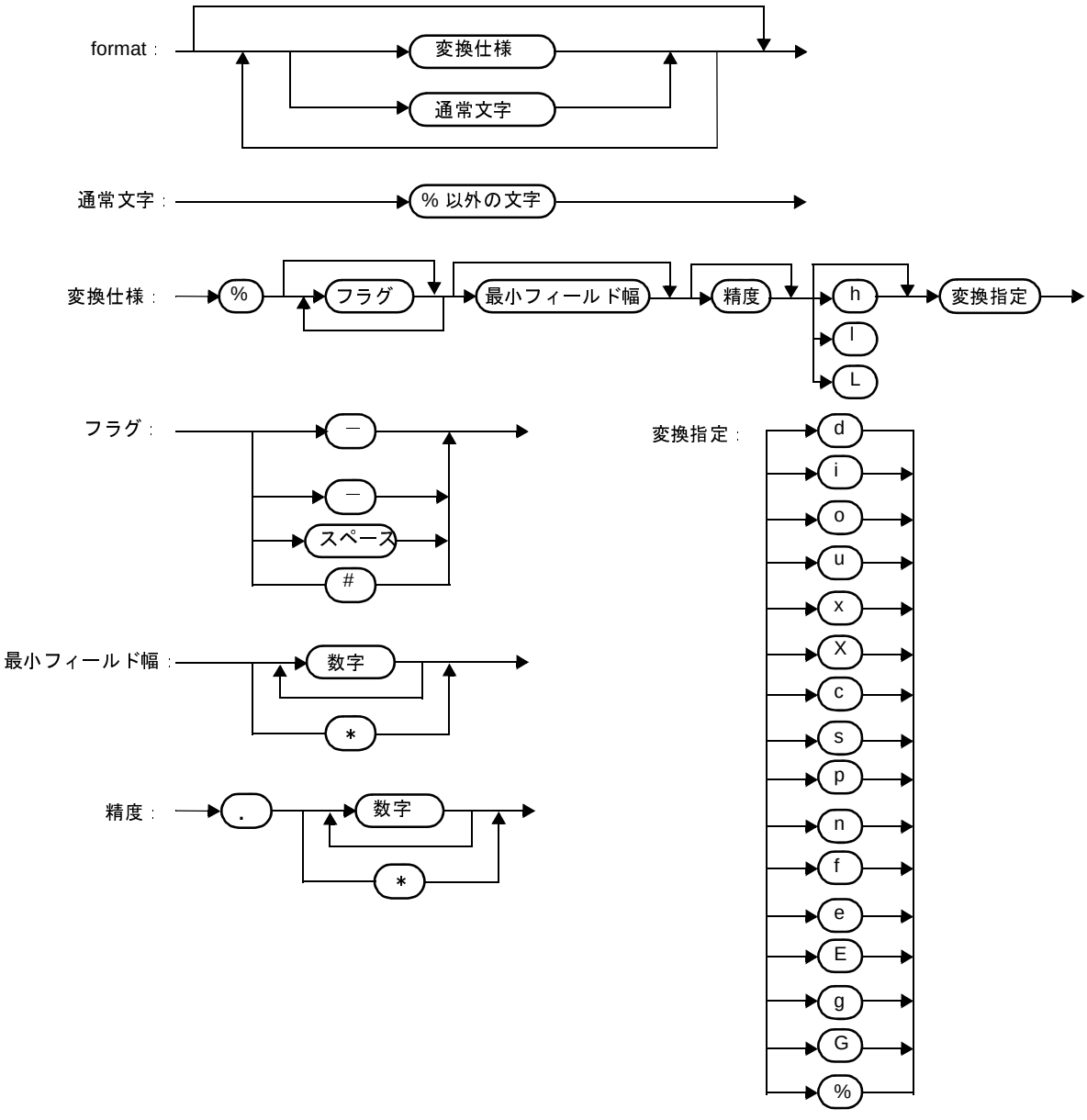
+ ∞ → “(+INF)”

- ∞ → “(-INF)”

文字列 *s* の末尾にヌル文字（戻り値のカウントには含まない）を書きます。

format の構文図を次に示します。

図 6—2 *format* の構文図



- RL78,78K0R C コンパイラでは，引数としてポインタを指定する変換指定 s/p/n について，ミディアム・モデルでは near データ・ポインタ（2 バイト長）をラージ・モデルでは far データ・ポインタ（4 バイト長）を指定する必要があります。
  また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数として指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

sscanf

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。

[指定形式]

```
#include <stdio.h>
int sscanf ( const char *s, const char *format, … ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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* 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されません。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
- 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。

（1）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）

（2）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）

（3）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 なしのデフォルトでは引数は 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。

（a）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。

（b）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。

（c）% 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格納されます。**

変換指示子は次のとおりです。

変換指示子	内容
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。

変換指示子	内容
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ヌ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。 終端のヌル文字は，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ヌ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。 終端のヌ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。 終端のヌル文字は，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，戻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。

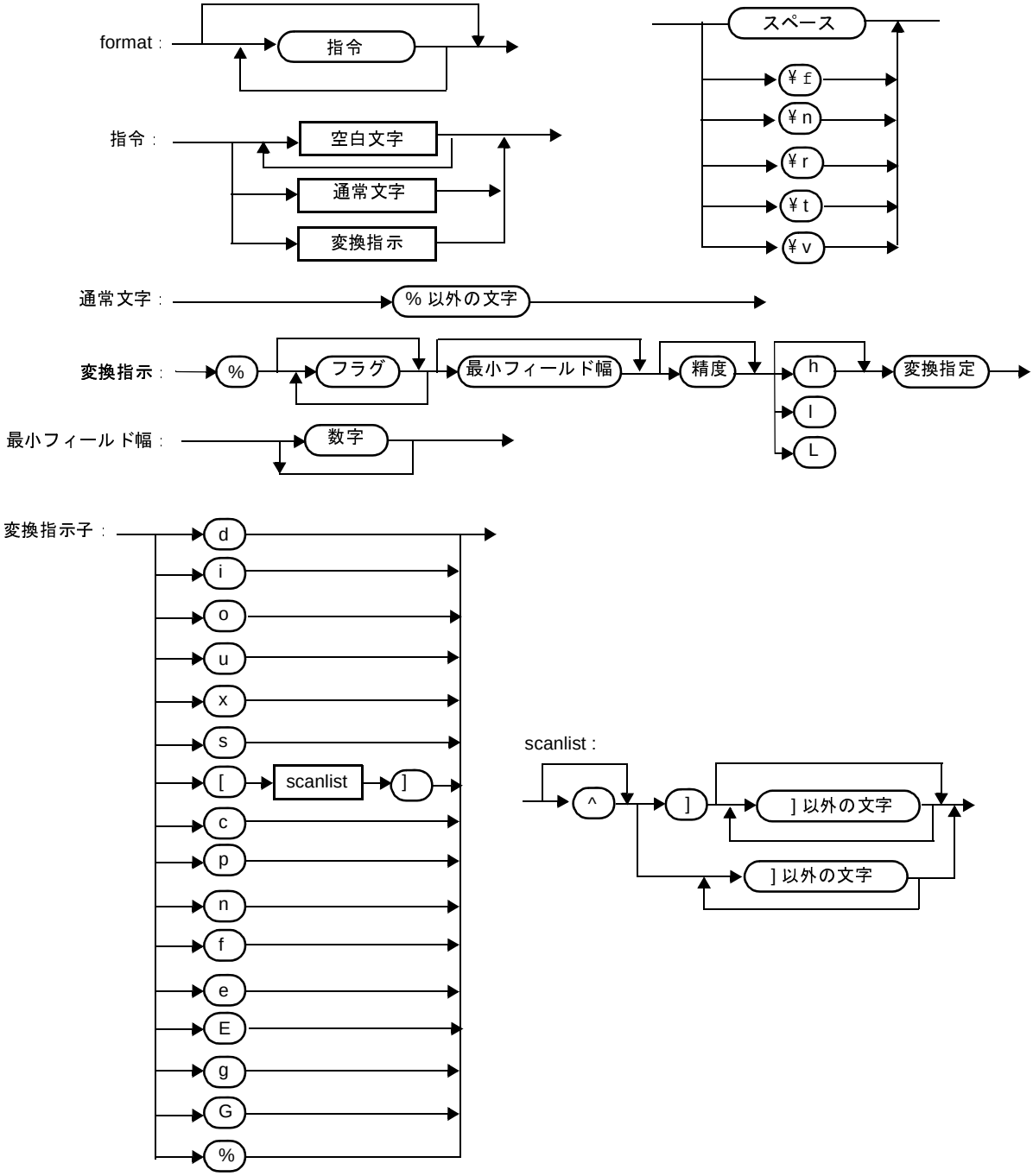
変換指示が不正な場合，指令は失敗します。

入力文字列に終端のヌ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 6—3 *format* の構文図



- RL78,78K0R C コンパイラでは，引数としてポインタを指定する変換指定 s/p/n について，ミディアム・モデルでは near データ・ポインタ（2 バイト長）をラージ・モデルでは far データ・ポインタ（4 バイト長）を指定する必要があります。
  また，関数ポインタは，どちらのモデルでも 4 バイト長固定ですが，引数として指定する場合には，それぞれ 2/4 バイト長で指定する必要があります。

printf

フォーマットに従ってデータを SFR に出力します。

[指定形式]

#include <stdio.h>

int printf (const char *format, …) ;

[引数／戻り値]

引数	戻り値
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（終端のヌル文字は数えません）

[詳細説明]

- *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。

scanf

SFR からフォーマットに従ってデータを読みます。

[指定形式]

```
#include <stdio.h>
int scanf ( const char *format, … ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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されません。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。

vprintf

フォーマットに従ってデータを SFR に出力します。

[指定形式]

```
#include <stdio.h>
int vprintf ( const char *format, va_list p ) ;
```

[引数／戻り値]

引数	戻り値
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（終端のヌル文字は数えません）

[詳細説明]

- *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。

vsprintf

フォーマットに従ってデータを文字列に書きます。

[指定形式]

```
#include <stdio.h>
int vsprintf ( char *s, const char *format, va_list p ) ;
```

[引数／戻り値]

引数	戻り値
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（終端のヌル文字は数えません）

[詳細説明]

- *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。

getchar

SFR から，1 文字読み込みます。

[指定形式]

```
#include <stdio.h>
int getchar ( void ) ;
```

[引数／戻り値]

引数	戻り値
なし	SFR から読み込んだ 1 文字

[詳細説明]

- 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。

gets

文字列を読み取ります。

[指定形式]

```
#include <stdio.h>
char *gets ( char *s ) ;
```

[引数／戻り値]

引数	戻り値
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 : ヌル・ポインタ

[詳細説明]

- 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ヌル文字を書きます。
- 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，ヌル・ポインタを返します。

putchar

SFR に 1 文字出力します。

[指定形式]

```
#include <stdio.h>
int putchar ( int c ) ;
```

[引数／戻り値]

引数	戻り値
c ： 出力する文字	出力した文字

[詳細説明]

- 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。

puts

文字列を出力します。

[指定形式]

```
#include <stdio.h>
int puts ( const char *s ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。
- 文字列の終端のヌ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。

__putc

opaque に 1 文字出力します。

[指定形式]

```
#include <stdio.h>
int  __putc ( int c, void *opaque ) ;
```

[引数／戻り値]

引数	戻り値
c ： 出力する文字 *opaque* ： 文字出力先へのポインタ	出力した文字

[詳細説明]

- *opaque* で示される文字出力先に，*c* で指定した文字を（unsigned char 型に変換して）書き込みます。また，*opaque* で示される文字出力先を 1 バイト進めます。
- *opaque* が 0 である場合，putchar 関数を呼び出し，putchar 関数の戻り値を返します。

6. 10 ユーティリティ関数

ユーティリティ関数には，次のものがあります。

関数名	機能
atoi	10 進整数文字列を int 変換
atol	10 進整数文字列を long に変換
strtol	文字列を long に変換
strtoul	文字列を unsigned long に変換
calloc	配列の領域を割り付けて 0 で初期化
free	割り付けられているブロックを解放
malloc	ブロックを割り付け
realloc	ブロックを再割り付け
abort	プログラムを異常終了
atexit	正常終了時に呼び出される関数を登録
exit	プログラムを終了
abs	int 型の値の絶対値を求める
labs	long 型の値の絶対値を求める
div	int 型の除算を行い，商と剰余を求める
ldiv	long 型の除算を行い，商と剰余を求める
brk	ブレーク値をセット
sbrk	ブレーク値を増減
atof	10 進整数文字列を double に変換
strtod	文字列を double に変換
itoa	int を文字列に変換
ltoa	long を文字列に変換
ultoa	unsigned long を文字列に変換
rand	疑似乱数を発生
srand	疑似乱数の発生状態を初期化
bsearch	バイナリ・サーチ
qsort	クイック・ソート
strbrk	ブレーク値をセット
strsbrk	ブレーク値を増減
stritoa	int を文字列に変換
strltoa	long を文字列に変換
strultoa	unsigned long を文字列に変換

atoi

10 進整数文字列を int に変換します。

[指定形式]

```
#include <stdlib.h>
int atoi ( const char *nptr ) ;
```

[引数／戻り値]

引数	戻り値
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 (32,767) 負のオーバフローの場合 ： INT_MIN (-32,768) 不正文字列の場合 ： 0

[詳細説明]

- *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ヌ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（32,767），負のときは INT_MIN（-32,768）を返します。

atol

10 進整数文字列を long に変換します。

[指定形式]

```
#include <stdlib.h>
long int atol ( const char *nptr ) ;
```

[引数／戻り値]

引数	戻り値
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 ： LONG_MAX (2,147,483,647) 負のオーバフローの場合 ： LONG_MIN (-2,147,483,648) 不正文字列の場合 ： 0

[詳細説明]

- *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ヌ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（2,147,483,647），負のときは LONG_MIN（-2,147,483,648）を返します。

strtol

文字列を long に変換します。

[指定形式]

#include <stdlib.h>

long int strtol (const char **nptr*, char ***endptr*, int *base*) ;

[引数／戻り値]

引数	戻り値
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 (2,147,483,647) 負のオーバフローの場合 ： LONG_MIN (-2,147,483,648) 変換が行われない場合 ： 0

[詳細説明]

- *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（終端のヌル文字を含む）の列

備考 （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* がヌル・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（2,147,483,647），負は LONG_MIN（-2,147,483,648）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* がヌル・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

文字列を unsigned long に変換します。

[指定形式]

#include <stdlib.h>

unsigned long int strtoul (const char *_nptr_, char **_endptr_, int _base_) ;

[引数／戻り値]

引数	戻り値
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 オーバフローの場合 ： ULONG_MAX (4,294,967,295U) 変換が行われない場合 ： 0

[詳細説明]

- *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（終端のヌル文字を含む）の列

備考 （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* がヌル・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（4,294,967,295U）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* がヌル・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。

calloc

配列の領域を割り付けて 0 で初期化します。

[指定形式]

```
#include <stdlib.h>
void *calloc ( size_t nmemb, size_t size ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- *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，ヌル・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。新たなブレーク値が奇数の場合には，偶数に補正します。ブレーク値は，brk で設定します。brk については「brk」を参照してください。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，calloc_n/calloc_f 関数は存在しません。

free

割り付けられているブロックを解放します。

[指定形式]

```
#include <stdlib.h>
void free ( void *ptr ) ;
```

[引数／戻り値]

引数	戻り値
ptr ： 解放するブロックの先頭へのポインタ	なし

[詳細説明]

- *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）。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，free_n/free_f 関数は存在しません。

malloc

ブロックを割り付けます。

[指定形式]

```
#include <stdlib.h>
void *malloc ( size_t size ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- *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，ヌル・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。新たなブレーク値が奇数の場合には，偶数に補正します。ブレーク値は，brk で設定します。brk については「brk」を参照してください。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，malloc_n/malloc_f 関数は存在しません。

realloc

ブロックの再割り付けを行います。

[指定形式]

```
#include <stdlib.h>
void *realloc ( void *ptr, size_t size ) ;
```

[引数／戻り値]

引数	戻り値
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* がヌル・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 ： ヌル・ポインタ

[詳細説明]

- *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* がヌル・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，何もせずにヌル・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。新たなブレーク値が奇数の場合には，偶数に補正します。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，realloc_n/realloc_f 関数は存在しません。

abort

プログラムを異常終了させます。

[指定形式]

```
#include <stdlib.h>
void abort ( void ) ;
```

[引数／戻り値]

引数	戻り値
なし	戻りません。

[詳細説明]

- ループして戻りません。
- ユーザは abort の処理を作成します。

atexit

正常終了時に呼び出される関数を登録します。

[指定形式]

```
#include <stdlib.h>
int atexit ( void ( *func ) ( void ) ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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 個まで登録することができます。登録することができた場合は，0 を返します。登録されている関数が 32 個あり，これ以上登録することができない場合は，登録せずに 1 を返します。

exit

プログラムを終了させます。

[指定形式]

```
#include <stdlib.h>
void exit ( int status ) ;
```

[引数／戻り値]

引数	戻り値
status ： 終了状態を示す値	戻りません。

[詳細説明]

- 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。

abs

int 型の値の絶対値を求めます。

[指定形式]

```
#include <stdlib.h>
int abs ( int j ) ;
```

[引数／戻り値]

引数	戻り値
j ： 絶対値を求める値	-32,767 ≦ *j* ≦ 32,767 の場合 ： *j* の絶対値 *j* が -32,768 の場合 ： -32,768（0x8000）

[詳細説明]

- abs は，*j* の値（int 型）の絶対値を求めます。
- *j* が -32768 の場合は，-32,768 を返します。

labs

long 型の値の絶対値を求めます。

[指定形式]

#include <stdlib.h>

long int labs (long int *j*) ;

[引数／戻り値]

引数	戻り値
j ： 絶対値を求める値	-2,147,483,647 ≦ *j* ≦ 2,147,483,647 の場合 ： *j* の絶対値 *j* が -2,147,483,648 の場合 ： -2147483,648（0x80000000）

[詳細説明]

- labs は，*j* の値（long 型）の絶対値を求めます。
- *j* が -2,147,483,648 の場合は，-2,147,483,648 を返します。

div

int 型の除算を行い，商と剰余を求めます。

[指定形式]

#include <stdlib.h>

div_t div (int *numer*, int *denom*) ;

[引数／戻り値]

引数	戻り値
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。

[詳細説明]

- 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。

  RL78 拡張命令搭載品の場合

  - *denom* が 0 の場合

    - *numer* ≧ 0 のとき，商は -1（FFFFH），

      numer ＜ 0 のとき，商は 1，剰余は *numer* です。
- *numer* が -32,768，*denom* が -1 の場合，商は -32,768，剰余は 0 です。

ldiv

long 型の除算を行い，商と剰余を求めます。

[指定形式]

#include <stdlib.h>

ldiv_t ldiv (long int *numer*, long int *denom*) ;

[引数／戻り値]

引数	戻り値
numer ： 被除数 *denom* ： 除数	ldiv_t 型のメンバ quot に商，rem に剰余を返します。

[詳細説明]

- 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。

  RL78 拡張命令搭載品の場合

  - *denom* が 0 の場合
    - *numer* ≧ 0 のとき，商は -1（FFFFFFFFH），

      numer ＜ 0 のとき，商は 1，剰余は *numer* です。
- *numer* が -2,147,483,648，*denom* が -1 の場合は，商は -2,147,483,648，剰余は 0 です。

brk

ブレーク値をセットします。

[指定形式]

#include <stdlib.h>

int brk (char *endds) ;

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 ptr は常に near ポインタとなります。したがって，brk_n/brk_f 関数は存在しません。

sbrk

ブレーク値を増減します。

[指定形式]

```
#include <stdlib.h>
char *sbrk ( int incr ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- sbrk は，ブレーク値を *incr* バイト増減（*incr* の符号による）します。
- *incr* に指定された増減量が奇数の場合には，偶数に補正します。
- 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。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，sbrk_n/sbrk_f 関数は存在しません。

atof

10 進整数文字列を double に変換します。

[指定形式]

```
#include <stdlib.h>
double atof ( const char *nptr ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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ヌ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

文字列を double に変換します。

[指定形式]

```
#include <stdlib.h>
double strtod ( const char *nptr, char **endptr ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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ヌル文字が現れるまで）を浮動小数点数に変換します。
  この書式を満たさない先頭文字が現れた場合には，スキャンを終了し，*endptr* が空ポインタでなければ，空白を含む書式先頭のポインタを *endptr* に格納します。
- 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。

itoa

int を文字列に変換します。

[指定形式]

```
#include <stdlib.h>
char *itoa ( int value, char *string, int radix ) ;
```

[引数／戻り値]

引数	戻り値
value ： 　変換する数値 *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 ： 　ヌル・ポインタ

[詳細説明]

- 指定した数値 *value* をヌ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，ヌル・ポインタを返します。

ltoa

long を文字列に変換します。

[指定形式]

```
#include <stdlib.h>
char *ltoa ( long value, char *string, int radix ) ;
```

[引数／戻り値]

引数	戻り値
value ： 変換する数値 *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 ： ヌル・ポインタ

[詳細説明]

- 指定した数値 *value* をヌ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，ヌル・ポインタを返します。

ultoa

unsigned long を文字列に変換します。

[指定形式]

#include <stdlib.h>

char *ultoa (unsigned long *value*, char **string*, int *radix*) ;

[引数／戻り値]

引数	戻り値
value ： 変換する数値 *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 ： ヌル・ポインタ

[詳細説明]

- 指定した数値 *value* をヌ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，ヌル・ポインタを返します。

rand

疑似乱数を発生させます。

[指定形式]

```
#include <stdlib.h>
int rand ( void ) ;
```

[引数／戻り値]

引数	戻り値
なし	0 から RAND_MAX の範囲の疑似乱数

[詳細説明]

- rand は，0 から RAND_MAX の範囲の疑似乱数を発生させます。

srand

疑似乱数の発生状態の初期化を行います。

[指定形式]

```
#include <stdlib.h>
void srand ( unsigned int seed ) ;
```

[引数／戻り値]

引数	戻り値
seed ： 疑似乱数の発生状態の初期値	なし

[詳細説明]

- 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戻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。

bsearch

バイナリ・サーチを行います。

[指定形式]

```
#include <stdlib.h>
void *bsearch ( const void *key, const void *base, size_t nmemb, size_t size,
                                                    int ( *compare ) ( const void *, const void * ) ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 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

qsort

クイック・ソートを行います。

[指定形式]

#include <stdlib.h>

void qsort (void *_base_, size_t _nmemb_, size_t _size_, int (*_compare_) (const void *, const void *)) ;

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。

strbrk

ブレーク値をセットします。

[指定形式]

```
#include <stdlib.h>
int strbrk ( char *endds ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，strbrk_n/strbrk_f 関数は存在しません。

strsbrk

ブレーク値を増減します。

[指定形式]

```
#include <stdlib.h>
char *strsbrk ( int incr ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。
- RL78,78K0R C コンパイラでは，割り付ける領域は内蔵 RAM に存在するため，引数 *ptr* は常に near ポインタとなります。したがって，strsbrk_n/strsbrk_f 関数は存在しません。

stritoa

int を文字列に変換します。

[指定形式]

```
#include <stdlib.h>
char *stritoa ( int value, char *string, int radix ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 指定した数値 value をヌ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，ヌル・ポインタを返します。

strltoa

long を文字列に変換します。

[指定形式]

#include <stdlib.h>

char *strltoa (long *value*, char **string*, int *radix*) ;

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 指定した数値 value をヌ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，ヌル・ポインタを返します。

strultoa

unsigned long を文字列に変換します。

[指定形式]

```
#include <stdlib.h>
char *strultoa ( unsigned long value, char *string, int radix ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 指定した数値 value をヌ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，ヌル・ポインタを返します。

6. 11 文字列／メモリ関数

文字列／メモリ関数には，次のものがあります。

関数名	機能
memcpy	バッファを指定文字数分コピー
memmove	バッファを指定文字数分コピー
strcpy	文字列をコピー
strncpy	文字列の先頭から指定の文字数分コピー
strcat	文字列に文字列を追加
strncat	文字列に指定文字数分の文字列を追加
memcmp	2 つのバッファの指定文字数分を比較
strcmp	2 つの文字列を比較
strncmp	2 つの文字列の指定文字数分を比較
memchr	指定文字数分のバッファから指定文字を探す
strchr	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す
strrchr	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す
strspn	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る
strcspn	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る
strpbrk	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る
strstr	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る
strtok	文字列を区切り文字以外からなる文字列に分解
memset	バッファの指定文字数分を指定文字で初期化
strerror	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す
strlen	文字列の長さを求める
strcoll	地域特有の情報に基づいて 2 つの文字列を比較
strxfrm	地域特有の情報に基づいて文字列を変換

memcpy

バッファを指定文字数分コピーします。

[指定形式]

```
#include <string.h>
void *memcpy ( void *s1, const void *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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，正常動作は保証されません（先頭から順にコピーするため）。

memmove

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）。

[指定形式]

```
#include <string.h>
void *memmove ( void *s1, const void *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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。

strcpy

文字列をコピーします。

[指定形式]

```
#include <string.h>
char *strcpy ( char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
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	*s1* の値

[詳細説明]

- strcpy は，*s2* が指す文字列（終端のヌル文字を含みます）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，正常動作は保証されません（先頭から順にコピーするため）。

strncpy

文字列の先頭から指定の文字数分コピーします。

[指定形式]

```
#include <string.h>
char *strncpy ( char *s1, const char *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- 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，正常動作は保証されません（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，終端のヌル文字までをコピーし，*n* 文字に達するまでヌル文字を付加します。

strcat

文字列に文字列を追加します。

[指定形式]

```
#include <string.h>
char *strcat ( char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
s1 ： 追加される文字列へのポインタ *s2* ： 追加する文字列へのポインタ	*s1* の値

[詳細説明]

- strcat は，*s1* が指す文字列の終わりに *s2* が指す文字列（終端のヌル文字を含みます）をコピーして追加します。
  s2 の始めの文字を *s1* の終端のヌ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，その動作は保証されません。

strncat

文字列に指定文字数分の文字列を追加します。

[指定形式]

```
#include <string.h>
char *strncat ( char *s1, const char *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
s1 ： 追加される文字列へのポインタ *s2* ： 追加する文字列へのポインタ *n* ： 追加する文字数	*s1* の値

[詳細説明]

- strncat は，*s1* が指す文字列の終わりに，*s2* が指す文字列（終端のヌル文字を含みません）のうち *n* 文字分を追加します。*s2* の始めの文字を *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，終端のヌ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。
- 終端のヌ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，その動作は保証されません。

memcmp

2 つのバッファの指定文字数分を比較します。

[指定形式]

```
#include <string.h>
int memcmp ( const void *s1, const void *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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）

[詳細説明]

- *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 - *s2* の文字）を返します。

strcmp

2 つの文字列を比較します。

[指定形式]

```
#include <string.h>
int strcmp ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
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）

[詳細説明]

- 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 - *s2* の文字）を返します。

strncmp

2 つの文字列の指定文字数分を比較します。

[指定形式]

```
#include <string.h>
int strncmp ( const char *s1, const char *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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）

[詳細説明]

- 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 - *s2* の文字）を返します。

memchr

指定文字数分のバッファから指定文字を探します。

[指定形式]

```
#include <string.h>
void *memchr ( const void *s, int c, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- *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，ヌル・ポインタを返します。

strchr

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。

[指定形式]

```
#include <string.h>
char *strchr ( const char *s, int c ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 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。
- 終端のヌル文字は，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，ヌル・ポインタを返します。

strrchr

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。

[指定形式]

```
#include <string.h>
char *strrchr ( const char *s, int c ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- 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。
- 終端のヌル文字は，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，ヌル・ポインタを返します。

strspn

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。

[指定形式]

```
#include <string.h>
size_t strspn ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
s1 ： 検索される文字列へのポインタ *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* で指定される文字で構成される部分の長さ

[詳細説明]

- 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* の終端のヌル文字は *s2* の一部とはみなしません。

strcspn

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。

[指定形式]

```
#include <string.h>
size_t strcspn ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
s1 ： 検索される文字列へのポインタ *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

[詳細説明]

- 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* の終端のヌル文字は *s2* の一部とはみなしません。

strpbrk

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。

[指定形式]

```
#include <string.h>
char *strpbrk ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- *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，ヌル・ポインタを返します。

strstr

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。

[指定形式]

```
#include <string.h>
char *strstr ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ *s2* が空文字列の場合 ： *s1* の値

[詳細説明]

- *s1* が指す文字列中で *s2* が指す文字列（終端のヌル文字を除く）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，ヌル・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。

strtok

文字列を区切り文字以外からなる文字列に分解します。

[指定形式]

```
#include <string.h>
char *strtok ( char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
s1 ： 分解される文字列へのポインタ，またはヌル・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	字句がある場合 ： 字句の第 1 文字へのポインタ 字句がない場合 ： ヌル・ポインタ

[詳細説明]

- 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* がヌル・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，保存ポインタがヌル・ポインタの場合は何もせずにヌル・ポインタを返します。
- *s1* がヌル・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ヌル・ポインタにして，ヌル・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，保存ポインタをヌル・ポインタにします。見つかれば，その文字の位置にヌル文字を上書きし，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。

memset

バッファの指定文字数分を指定文字で初期化します。

[指定形式]

```
#include <string.h>
void *memset ( void *s, int c, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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。

strerror

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。

[指定形式]

```
#include <string.h>
char *strerror ( int errnum ) ;
```

[引数／戻り値]

引数	戻り値
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 ： ヌル・ポインタ

[詳細説明]

- *errnum* の値に対応して，次の値を返します。

errnum の値	戻り値
0	文字列“Error 0”へのポインタ
1（EDOM）	文字列“Argument too large”へのポインタ
2（ERANGE）	文字列“Result too large”へのポインタ
3（ENOMEM）	文字列“Not enough memory”へのポインタ
その他	ヌル・ポインタ

- エラー・メッセージの文字列を far 領域に確保しているため，戻り値は常に far ポインタとなります。したがって，strerror_n/strerror_f 関数は存在しません。

strlen

文字列の長さを求めます。

[指定形式]

```
#include <string.h>
size_t strlen ( const char *s ) ;
```

[引数／戻り値]

引数	戻り値
s : 文字列へのポインタ	文字列 *s* の長さ

[詳細説明]

- *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ヌル文字の前までの文字数です。

strcoll

地域特有の情報に基づいて 2 つの文字列を比較します。

[指定形式]

```
#include <string.h>
int strcoll ( const char *s1, const char *s2 ) ;
```

[引数／戻り値]

引数	戻り値
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）

[詳細説明]

- RL78,78K0R C 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。

strxfrm

地域特有の情報に基づいて文字列を変換します。

[指定形式]

```
#include <string.h>
size_t strxfrm ( char *s1, const char *s2, size_t n ) ;
```

[引数／戻り値]

引数	戻り値
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。

[詳細説明]

- RL78,78K0R C 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 ) ) ;
```

6. 12 数学関数

数学関数には，次のものがあります。

関数名	機能
acos	acos を求める
asin	asin を求める
atan	atan を求める
atan2	atan2 を求める
cos	cos を求める
sin	sin を求める
tan	tan を求める
cosh	cosh を求める
sinh	sinh を求める
tanh	tanh を求める
exp	指数関数を求める
frexp	仮数部と指数部を求める
ldexp	x * 2 ^ exp を求める
log	自然対数を求める
log10	10 を底とした対数を求める
modf	小数部と整数部を求める
pow	x の y 乗を求める
sqrt	平方根を求める
ceil	x より小さくない最小の整数を求める
fabs	浮動小数点数 x の絶対値を求める
floor	x より大きくない最大の整数を求める
fmod	x/y の余りを求める
matherr	浮動小数点数を扱うライブラリの例外処理を求める
acosf	acos を求める
asinf	asin を求める
atanf	atan を求める
atan2f	y/x の atan を求める
cosf	cos を求める
sinf	sin を求める
tanf	tan を求める
coshf	cosh を求める
sinhf	sinh を求める
tanhf	tanh を求める
expf	指数関数を求める
frexpf	仮数部と指数部を求める

関数名	機能
ldexpf	x * 2 ^ exp を求める
logf	自然対数を求める
log10f	10 を底とした対数を求める
modff	小数部と整数部を求める
powf	x の y 乗を求める
sqrtf	平方根を求める
ceilf	x より小さくない最小の整数を求める
fabsf	浮動小数点数 x の絶対値を求める
floorf	x より大きくない最大の整数を求める
fmodf	x/y の余りを求める

acos

acos を求めます。

[指定形式]

#include <math.h>

double acos (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 ： *x* の acos *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 ： NaN

[詳細説明]

- *x* の acos（0 から π の範囲内）を計算します。
- *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。
- *x* が非数の場合は，NaN を返します。

asin

asin を求めます。

[指定形式]

#include <math.h>

double asin (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 ： *x* の asin *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 ： NaN *x* = -0 の場合 ： -0 アンダフロー時 ： 非正規化数

[詳細説明]

- *x* の asin（$-\pi/2$ から $+\pi/2$ 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。

atan

atan を求めます。

[指定形式]

#include <math.h>

double atan (double *x*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値	正常時 : *x* の atan *x* が非数の場合 : NaN *x* = -0 の場合 : -0 アンダフロー時 : 非正規化数

[詳細説明]

- *x* の atan（$-\pi/2$ から $+\pi/2$ 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。

atan2

y/x の atan を求めます。

[指定形式]

#include <math.h>

double atan2 (double *y*, double *x*) ;

[引数／戻り値]

引数	戻り値
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，*x*，*y* がともに±∞の場合，*x*，*y* のどちらかが非数の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

cos

cos を求めます。

[指定形式]

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

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の cos *x* が非数，*x* が無限大の場合 ： NaN

[詳細説明]

- *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。

sin

sin を求めます。

[指定形式]

#include <math.h>

double sin (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の sin *x* が非数，*x* が無限大の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

tan

tan を求めます。

[指定形式]

#include <math.h>

double tan (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の tan *x* が非数，*x* = ±∞の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

cosh

cosh を求めます。

[指定形式]

#include <math.h>

double cosh (double *x*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値	正常時 : *x* の cosh *x* が非数の場合 : NaN *x* = ±∞の場合 : + ∞ オーバフロー時 : HUGE_VAL（正の符号を持ちます）

[詳細説明]

- *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，+ ∞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。

sinh

sinh を求めます。

[指定形式]

#include <math.h>

double sinh (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の sinh *x* が非数の場合 ： 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

[詳細説明]

- *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。

tanh

tanh を求めます。

[指定形式]

#include <math.h>

double tanh (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の tanh *x* が非数の場合 ： NaN *x* = ±∞の場合 ： ± 1 アンダフロー時 ： ± 0

[詳細説明]

- *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。

exp

指数関数を求めます。

[指定形式]

```
#include <math.h>
double exp ( double x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の指数関数 *x* が非数の場合 ： NaN *x* = + ∞の場合 ： + ∞ *x* = - ∞の場合 ： +0 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 オーバフロー時 ： HUGE_VAL（正の符号を持ちます）

[詳細説明]

- *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。
- *x* が - ∞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。
- 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。

frexp

仮数部と指数部を求めます。

[指定形式]

```
#include <math.h>
double frexp ( double x, int *exp ) ;
```

[引数／戻り値]

引数	戻り値
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* が非数，*x* = ±∞の場合 ： NaN *x* = ± 0 の場合 ： ± 0

[詳細説明]

- 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。

ldexp

x * 2 ^ exp を求めます。

[指定形式]

#include <math.h>

double ldexp (double *x*, int *exp*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値 *exp* ： べき乗数	正常時 ： *x* * 2 ^ *exp* *x* が非数の場合 ： 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

[詳細説明]

- *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。

log

自然対数を求めます。

[指定形式]

#include <math.h>

double log (double *x*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値	正常時 : *x* の自然対数 *x* < 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* が無限大の場合 : + ∞

[詳細説明]

- *x* の自然対数を求めます。
- *x* < 0 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* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *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。

log10

10 を底とした対数を求めます。

[指定形式]

#include <math.h>

double log10 (double *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の 10 を底とした対数 *x* ＜ 0 の場合 ： NaN *x* = 0 の場合 ： - ∞ *x* が非数の場合 ： NaN *x* が無限大の場合 ： + ∞

[詳細説明]

- *x* の 10 を底とした対数を求めます。
- *x* ＜ 0 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* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *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。

modf

小数部と整数部を求めます。

[指定形式]

```
#include <math.h>
double modf ( double x, double *iptr ) ;
```

[引数／戻り値]

引数	戻り値
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 の場合 ： ± 0

[詳細説明]

- 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。

pow

x の y 乗を求めます。

[指定形式]

```
#include <math.h>
double pow ( double x, double y ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値 *y* ： 乗数	正常時 ： *x* ^ *y* *x* が非数，または *y* が非数の場合， *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

[詳細説明]

- *x* ^ *y* を計算します。
- *x* が非数，または *y* が非数の場合は，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。
- 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。
- 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。

sqrt

平方根を求めます。

[指定形式]

#include <math.h>

double sqrt (double *x*) ;

[引数／戻り値]

引数	戻り値
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* が非数の場合 : NaN *x* = ± 0 の場合 : ± 0

[詳細説明]

- *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。

ceil

x より小さくない最小の整数を求めます。

[指定形式]

```
#include <math.h>
double ceil ( double x ) ;
```

[引数／戻り値]

引数	戻り値
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，*x* が無限大の場合 ： 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

[詳細説明]

- *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* が無限大の場合は，NaN を返し errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *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。

fabs

浮動小数点数 x の絶対値を返します。

[指定形式]

```
#include <math.h>
double fabs ( double x ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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。

floor

x より大きくない最大の整数を求めます。

[指定形式]

```
#include <math.h>
double floor ( double x ) ;
```

[引数／戻り値]

引数	戻り値
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，*x* が無限大の場合 ： 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

[詳細説明]

- *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* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *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。

fmod

x/y の余りを求めます。

[指定形式]

#include <math.h>

double fmod (double *x*, double *y*)

[引数／戻り値]

引数	戻り値
x : 演算を行う数値 *y* : 演算を行う数値	正常時 : *x*/*y* の余り *x* が非数，または *y* が非数の場合， *y* が± 0，または *x* が±∞の場合 : NaN *x* ≠∞，かつ *y* = ±∞の場合 : x

[詳細説明]

- *x* - 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，戻り値は *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。
- *x* が非数，または *y* が非数の場合は，NaN を返します。
- *y* が± 0，または *x* = ±∞の場合は，NaN を返し，errno に EDOM をセットします。
- *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。

matherr

浮動小数点数を扱うライブラリの例外処理を行います。

[指定形式]

```
#include <math.h>
void matherr ( struct exception *x ) ;
```

[引数／戻り値]

引数	戻り値

struct exception {
int type ;
char *name ;
}
 	
type :
演算例外を示す値
name :
関数名
	なし

[詳細説明]

- 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，独自のエラー処理を行うことができます。

- 渡される構造体は内蔵 RAM に存在するため，引数は常に near ポインタとなります。したがって，matherr_n/matherr_f 関数は存在しません。

acosf

acos を求めます。

[指定形式]

```
#include <math.h>
float acosf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 ： *x* の acos *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 ： NaN

[詳細説明]

- *x* の acos（0 から π の範囲内）を計算します。
- *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。
- *x* が非数の場合は，NaN を返します。

asinf

asin を求めます。

[指定形式]

#include <math.h>

float asinf (float *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	-1 ≦ *x* ≦ 1 の場合 ： *x* の asin *x* ＜ -1，1 ＜ *x*，*x* が非数の場合 ： NaN *x* = -0 の場合 ： -0 アンダフロー時 ： 非正規化数

[詳細説明]

- *x* の asin（$-\pi/2$ から $+\pi/2$ の範囲内）を計算します。
- *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。
- *x* が非数の場合は，NaN を返します。
- *x* = -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。

atanf

atan を求めます。

[指定形式]

#include <math.h>

float atanf (float *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の atan *x* が非数の場合 ： NaN *x* = -0 の場合 ： -0 アンダフロー時 ： 非正規化数

[詳細説明]

- *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* = -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。

atan2f

y/x の atan を求めます。

[指定形式]

#include <math.h>

float atan2f (float *y*, float *x*) ;

[引数／戻り値]

引数	戻り値
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，*x*，*y* がともに無限大の場合，*x*，*y* のどちらかが非数の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

cosf

cos を求めます。

[指定形式]

#include <math.h>

float cosf (float *x*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値	正常時 : *x* の cos *x* が非数，*x* が無限大の場合 : NaN

[詳細説明]

- *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。

sinf

sin を求めます。

[指定形式]

#include <math.h>

float sinf (float *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の sin *x* が非数，*x* が無限大の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

tanf

tan を求めます。

[指定形式]

#include <math.h>

float tanf (float *x*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の tan *x* が非数，*x* が無限大の場合 ： NaN アンダフロー時 ： 非正規化数

[詳細説明]

- *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。

coshf

cosh を求めます。

[指定形式]

```
#include <math.h>
float coshf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の cosh *x* が非数の場合 ： NaN *x* が無限大の場合 ： + ∞ オーバフロー時： HUGE_VAL（正の符号を持ちます）

[詳細説明]

- *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。

sinhf

sinh を求めます。

[指定形式]

```
#include <math.h>
float sinhf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の sinh *x* が非数の場合 ： 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

[詳細説明]

- *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。

tanhf

tanh を求めます。

[指定形式]

```
#include <math.h>
float tanhf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の tanh *x* が非数の場合 ： NaN *x* = ±∞の場合 ： ± 1 アンダフロー時 ： ± 0

[詳細説明]

- *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。

expf

指数関数を求めます。

[指定形式]

```
#include <math.h>
float expf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の指数関数 *x* が非数の場合 ： NaN *x* = + ∞の場合 ： + ∞ *x* = - ∞の場合 ： +0 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

[詳細説明]

- *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。
- *x* が - ∞の場合は，+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。
- 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。

frexpf

仮数部と指数部を求めます。

[指定形式]

```
#include <math.h>
float frexpf ( float x, int *exp ) ;
```

[引数／戻り値]

引数	戻り値
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* が非数，*x* = ±∞の場合 ： NaN *x* = ± 0 の場合 ： ± 0

[詳細説明]

- 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。

ldexpf

x * 2 ^ exp を求めます。

[指定形式]

#include <math.h>

float ldexpf (float *x*, int *exp*) ;

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値 *exp* ： べき乗数	正常時 ： *x* * 2 ^ *exp* *x* が非数の場合 ： 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

[詳細説明]

- *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。

logf

自然対数を求めます。

[指定形式]

#include <math.h>

float logf (float *x*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値	正常時 : *x* の自然対数 *x* < 0 の場合 : NaN *x* = 0 の場合 : - ∞ *x* が非数の場合 : NaN *x* が無限大の場合 : + ∞

[詳細説明]

- *x* の自然対数を求めます。
- *x* < 0 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* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *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。

log10f

10 を底とした対数を求めます。

[指定形式]

```
#include <math.h>
float log10f ( float x ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値	正常時 ： *x* の 10 を底とした対数 *x* ＜ 0 の場合 ： NaN *x* = 0 の場合 ： - ∞ *x* が非数の場合 ： NaN *x* = + ∞の場合 ： + ∞

[詳細説明]

- *x* の 10 を底とした対数を求めます。
- *x* ＜ 0 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* = 0 の場合は，- ∞を返し，errno に ERANGE をセットします。
- *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。

modff

小数部と整数部を求めます。

[指定形式]

```
#include <math.h>
float modff ( float x, float *iptr ) ;
```

[引数／戻り値]

引数	戻り値
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，*x* が無限大の場合 ： NaN *x* = ± 0 の場合 ： ± 0

[詳細説明]

- 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。

powf

x の y 乗を求めます。

[指定形式]

```
#include <math.h>
float powf ( float x, float y ) ;
```

[引数／戻り値]

引数	戻り値
x ： 演算を行う数値 *y* ： 乗数	正常時 ： *x* ^ *y* *x* が非数，または *y* が非数の場合， *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

[詳細説明]

- *x* ^ *y* を計算します。
- *x* が非数，または *y* が非数の場合は，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。
- 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。
- 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。

sqrtf

平方根を求めます。

[指定形式]

```
#include <math.h>
float sqrtf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
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* が非数の場合 ： NaN *x* = ± 0 の場合 ： ± 0

[詳細説明]

- *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。

ceilf

x より小さくない最小の整数を求めます。

[指定形式]

```
#include <math.h>
float ceilf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
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，*x* が無限大の場合 ： 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*

[詳細説明]

- *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* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *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。

fabsf

浮動小数点数 x の絶対値を返します。

[指定形式]

```
#include <math.h>
float fabsf ( float x ) ;
```

[引数／戻り値]

引数	戻り値
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

[詳細説明]

- *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。

floorf

x より大きくない最大の整数を求めます。

[指定形式]

#include <math.h>

float floorf (float *x*) ;

[引数／戻り値]

引数	戻り値
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，*x* が無限大の場合 ： 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

[詳細説明]

- *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* が無限大の場合は，NaN を返し，errno に EDOM をセットします。
- *x* が -0 の場合は，+0 を返します。
- *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。

fmodf

x/y の余りを求めます。

[指定形式]

#include <math.h>

float fmodf (float *x*, float *y*) ;

[引数／戻り値]

引数	戻り値
x : 演算を行う数値 *y* : 演算を行う数値	正常時 : *x*/*y* の余り *y* が± 0，または *x* が±∞の場合， *x* が非数，または *y* が非数の場合 : NaN *x* ≠∞，かつ *y* = ±∞の場合 : x

[詳細説明]

- *x* - 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，戻り値は *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，または 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。

6. 13　診断関数

診断関数には，次のものがあります。

関数名	機能
__assertfail	assert マクロをサポート

__assertfail

assert マクロのサポートをします。

[指定形式]

#include <assert.h>

int __assertfail (char *__*msg*, char *__*cond*, char *__*file*, int __*line*) ;

[引数／戻り値]

引数	戻り値
__*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

[詳細説明]

- __assertfail 関数は，assert マクロ（「6. 3. 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。

6. 14 ライブラリ消費スタック一覧

この節では，ライブラリに含まれている各種関数のスタック消費量について説明します。

6. 14. 1 標準ライブラリ

標準ライブラリのスタック消費量を示します。

（1）ctype.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
isalpha	0	0
isupper	0	0
islower	0	0
isdigit	0	0
isalnum	0	0
isxdigit	0	0
isspace	0	0
ispunct	0	0
isprint	0	0
isgraph	0	0
iscntrl	0	0
isascii	0	0
toupper	0	0
tolower	0	0
toascii	0	0
_toupper	0	0
toup	0	0
_tolower	0	0
tolow	0	0

（2）setjmp.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
setjmp	4	4
longjmp	2	2

（3）stdarg.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
va_arg	0	0
va_start	0	0
va_starttop	0	0
va_end	0	0

（4）stdio.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
sprintf	58（130）注1	58（140）注1
sscanf	294（332）注1（350）注2	294（340）注1（358）注2
printf	70（128）注1	74（138）注1
scanf	308（330）注1（348）注2	312（338）注1（356）注2
vprintf	70（128）注1	76（140）注1
vsprintf	58（130）注1	58（140）注1
getchar	0	0
gets	8	14
putchar	0	0
puts	6	10
__putc	4	4

注 1. （ ）内は浮動小数点対応版使用時の値

2. （ ）内は浮動小数点対応版における，演算例外発生時の値

（5）stdlib.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
atoi	4	4
atol	10	10
strtol	18	18
strtoul	18	18
calloc	12	12
free	8	8
malloc	6	6
realloc	12	12
abort	0	0

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
atexit	0	0
exit	6 + n 注1	6 + n 注1
abs	0	0
labs	0	0
div	6（2）注2 （0）注3	6（2）注2 （0）注3
ldiv	18（8）注2 （4）注3	18（8）注2 （4）注3
brk	0	0
sbrk	2	2
atof	46（64）注4	46（64）注4
strtod	46（64）注4	48（66）注4
itoa	10	10
ltoa	16	16
ultoa	16	16
rand	18（14）注5 （14）注3	18（14）注5 （14）注3
srand	0	0
bsearch	36 + n 注6	40 + n 注6
qsort	16 + n 注7	18 + n 注7
strbrk	0	0
strsbrk	2	2
stritoa	10	10
strltoa	16	16
strultoa	16	16

注 1. 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

2. （ ）内は乗除算器を使用した場合

3. （ ）内は RL78 拡張命令搭載品の場合

4. （ ）内は浮動小数点対応版における，演算例外発生時の値

5. （ ）内は乗算器／乗除算器を使用した場合

6. n は bsearch から呼び出される外部関数のスタック消費量

7. n は（X + 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）

スモール・モデル，ミディアム・モデル兼用ライブラリ使用時 ： X = 38

ラージ・モデル兼用ライブラリ使用時 ： X = 40

（6）string.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
memcpy	4	8

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
memmove	4	6
strcpy	2	6
strncpy	4	10
strcat	2	6
strncat	4	8
memcmp	2	4
strcmp	2	2
strncmp	2	2
memchr	2	4
strchr	4	2
strrchr	4	6
strspn	4	6
strcspn	4	4
strpbrk	4	6
strstr	4	8
strtok	4	4
memset	4	6
strerror	0	0
strlen	0	0
strcoll	2	2
strxfrm	4	4

（7）math.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
acos	30（48）注	30（48）注
asin	30（48）注	30（48）注
atan	30（48）注	30（48）注
atan2	30（48）注	30（48）注
cos	28（46）注	28（46）注
sin	28（46）注	28（46）注
tan	34（52）注	34（52）注
cosh	34（52）注	34（52）注
sinh	34（52）注	34（52）注
tanh	40（58）注	40（58）注
exp	30（48）注	30（48）注

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
frexp	2（16）注	4（16）注
ldexp	0（16）注	0（16）注
log	30（48）注	30（48）注
log10	30（48）注	30（48）注
modf	2（16）注	4（16）注
pow	30（48）注	30（48）注
sqrt	22（40）注	22（40）注
ceil	2（16）注	2（16）注
fabs	0	0
floor	2（16）注	2（16）注
fmod	2（16）注	2（16）注
matherr	0	0
acosf	30（48）注	30（48）注
asinf	30（48）注	30（48）注
atanf	30（48）注	30（48）注
atan2f	30（48）注	30（48）注
cosf	28（46）注	28（46）注
sinf	28（46）注	28（46）注
tanf	34（52）注	34（52）注
coshf	34（52）注	34（52）注
sinhf	34（52）注	34（52）注
tanhf	40（58）注	40（58）注
expf	30（48）注	30（48）注
frexpf	2（16）注	4（16）注
ldexpf	0（16）注	0（16）注
logf	30（48）注	30（48）注
log10f	30（48）注	30（48）注
modff	2（16）注	4（16）注
powf	30（48）注	30（48）注
sqrtf	22（40）注	22（40）注
ceilf	2（16）注	2（16）注
fabsf	0	0
floorf	2（16）注	2（16）注
fmodf	2（16）注	2（16）注

注 （ ）内は演算例外発生時

（8）assert.h

関数名	スモール・モデル， ミディアムモデル兼用	ラージ・モデル
__assertfail	82（140）注	92（156）注

注 （ ）内は浮動小数点数対応版 printf 使用時

6. 14. 2 ランタイム・ライブラリ

ランタイム・ライブラリのスタック消費量を示します。

（1）インクリメント

関数名	スタック消費量
lsinc	0
luinc	0
finc	16（34）注
lsincr	0
luincr	0
fincr	16（34）注

注 （ ）内は演算例外発生時

（2）デクリメント

関数名	スタック消費量
lsdec	0
ludec	0
fdec	16（34）注
lsdecr	0
ludecr	0
fdecr	16（34）注

注 （ ）内は演算例外発生時

（3）符号反転

関数名	スタック消費量
lsrev	2
lurev	2
frev	0

関数名	スタック消費量
lsrevr	2
lurevr	2
frevr	0

（4）1の補数

関数名	スタック消費量
lscom	0
lucom	0
lscomr	0
lucomr	0

（5）論理否定

関数名	スタック消費量
lsnot	0
lunot	0

（6）乗算

関数名	スタック消費量
csmul	0
cumul	0
ismul	4（2）**注1** （2）**注2**
iumul	4（2）**注1** （2）**注2**
lsmul	8（4）**注1** （4）**注2**
lumul	8（4）**注1** （4）**注2**
fmul	8 （10）**注1** （10）**注2** （26）**注3**
iumulr	4（2）**注1** （2）**注2**
lsmulr	8（4）**注1** （4）**注2**
lumulr	8（4）**注1** （4）**注2**
fmulr	8 （10）**注1** （10）**注2** （26）**注3**

注1. （ ）内は乗算器／乗除算器を使用した場合

2. （ ）内は RL78 拡張命令搭載品の場合

3. （ ）内は演算例外発生時

（7）除算

関数名	スタック消費量
csdiv	8（10）注1 （10）注2
cudiv	2（4）注1 （4）注2
isdiv	12（8）注1 （8）注2
iudiv	6（2）注1 （2）注2
lsdiv	12（8）注1 （12）注2
ludiv	6（2）注1 （6）注2
fdiv	8（10）注1 （14）注2 （26）注3
csdivr	8（10）注1 （10）注2
cudivr	2（4）注1 （4）注2
isdivr	12（8）注1 （8）注2
iudivr	6（2）注1 （2）注2
lsdivr	12（8）注1 （12）注2
ludivr	6（2）注1 （6）注2
fdivr	8（10）注1 （14）注2 （26）注3

注 1. （ ）内は乗除算器を使用した場合

2. （ ）内は RL78 拡張命令搭載品の場合

3. （ ）内は演算例外発生時

（8）剰余算

関数名	スタック消費量
csrem	8（10）注1 （10）注2
curem	2（4）注1 （4）注2
isrem	12（8）注1 （8）注2
iurem	6（2）注1 （2）注2
lsrem	12（8）注1 （12）注2
lurem	6（2）注1 （6）注2
csremr	8（10）注1 （10）注2
curemr	2（4）注1 （4）注2
isremr	12（8）注1 （8）注2
iuremr	6（2）注1 （2）注2
lsremr	12（8）注1 （12）注2
luremr	6（2）注1 （6）注2

注 1. （ ）内は乗除算器を使用した場合

2. （ ）内は RL78 拡張命令搭載品の場合

（9）加算

関数名	スタック消費量
lsadd	0
luadd	0
fadd	8（26）注
lsaddr	0
luaddr	0
faddr	8（26）注

注 （ ）内は演算例外発生時

（10）減算

関数名	スタック消費量
lssub	2
lusub	2
fsub	8（26）注
lssubr	2
lusubr	2
fsubr	8（26）注

注 （ ）内は演算例外発生時

（11）左シフト

関数名	スタック消費量
lslsh	4
lulsh	4
lslshr	4
lulshr	4

（12）右シフト

関数名	スタック消費量
lsrsh	4
lursh	4
lsrshr	4
lurshr	4

（13）比較

関数名	スタック消費量
cscmp	0
iscmp	0
lscmp	2
lucmp	2
fcmp	4（24）注
cscmpr	0
iscmpr	0
lscmpr	2
lucmpr	2
fcmpr	4（24）注

注 （ ）内は演算例外発生時

（14）ビット AND

関数名	スタック消費量
lsband	0
luband	0
lsbandr	0
lubandr	0

（15）ビット OR

関数名	スタック消費量
lsbor	0
lubor	0
lsborr	0
luborr	0

（16）ビット XOR

関数名	スタック消費量
lsbxor	0
lubxor	0
lsbxorr	0
lubxorr	0

（17）浮動小数点数からの変換

関数名	スタック消費量
ftols	6
ftolu	6
ftolsr	6
ftolur	6

（18）浮動小数点への変換

関数名	スタック消費量
lstof	6
lutof	6
lstofr	6
lutofr	6

（19）bit からの変換

関数名	スタック消費量
btol	0
btolr	0

（20）スタートアップ・ルーチン

関数名	スタック消費量
cstart	4

（21）フラッシュ用スタートアップ・ルーチン

関数名	スタック消費量
cstarte	4

（22）ブート用 main

関数名	スタック消費量
boot_main	0

（23）関数前後処理

関数名	スタック消費量
hdwinit	0
cprep3	ベース・ポインタ + 第 1 引数 + レジスタ変数 + 自動変数のサイズ
cdisp3	0
cpre3e	ベース・ポインタ + 第 1 引数 + レジスタ変数 + 自動変数のサイズ
cdis3e	0

（24）初期化

関数名	スタック消費量
stkinit	0

（25）BCD 型変換

関数名	スタック消費量
bcdtob	6
btobcd	6
bcdtow	6
wtobcd	8
bbcd	6

（26）補助

関数名	スタック消費量
indao	0
ifdao	0
inado	0
ifado	0
lnd0	2
lfd0	2
ln0d	0
lf0d	0
lnd0o	2
lfd0o	2
ln0do	0
lf0do	0
df1in	0

関数名	スタック消費量
df1de	0
dn4in	0
dn4ip	4
df4in	0
df4ip	4
dn4ino	0
dn4ipo	4
df4ino	0
df4ipo	4
dn4de	0
dn4dp	4
df4de	0
df4dp	4
dn4deo	0
dn4dpo	4
df4deo	0
df4dpo	4
divuw	6（2）**注1** （4）**注2**
divuwr	6（2）**注1** （4）**注2**
mulsw	14（10）**注3** （2）**注4** （2）**注2**
muluw	14（10）**注3** （2）**注4** （2）**注2**
macsw	22（18）**注3** （2）**注4** （6）**注2**
macuw	22（18）**注3** （2）**注4** （6）**注2**

注1. （ ）内は乗除算器を使用した場合

2. （ ）内は RL78 拡張命令搭載品の場合

3. （ ）内は乗算器／乗除算器を使用した場合

4. （ ）内は積和演算器を使用した場合

6. 15 ライブラリ最大割り込み禁止時間一覧

乗算器／乗除算器／積和演算器を使用したライブラリの中では，割り込み時に演算内容が途中で壊されないように，割り込み禁止になる時間があります。

乗算器／乗除算器／積和演算器を使用したライブラリの中での，ライブラリの最大割り込み禁止時間一覧を次に示します。

乗算器／乗除算器／積和演算器を使用しないライブラリでは，割り込み禁止になる区間はありません。

RL78 拡張命令搭載品の場合は，ライブラリ関数 @@macuw，@@macsw のみ，それぞれ 17 クロックの割り込み禁止が発生します。それ以外のライブラリ関数については，割り込み禁止になる区間はありません。

表 6―2 ライブラリの最大割り込み禁止時間（クロック数）

分類	関数名	最大割り込み禁止時間			備考
		乗算器使用時	乗除算器使用時	積和演算器使用時	
乗算	@@ismul	12	12	12	signed int 同士の乗算
	@@iumul	12	12	12	unsigned int 同士の乗算
	@@lsmul	24	24	24	signed long 同士の乗算
	@@lumul	24	24	24	unsigned long 同士の乗算
	@@iumulr	12	12	12	unsigned int 同士の乗算 （RAM 配置用）
	@@lumulr	24	24	24	unsigned long 同士の乗算 （RAM 配置用）
	@@fmul	23	23	23	float 同士の乗算
	@@lsmulr	24	24	24	signed long 同士の乗算 （RAM 配置用）
	@@muluw	24	24	14	unsigned int 同士の乗算 （結果は unsigned long）
	@@mulsw	24	24	16	signed int 同士の乗算 （結果は signed long）
	@@fmulr	23	23	23	float 同士の乗算 （RAM 配置用）

分類	関数名	最大割り込み禁止時間			備考
		乗算器使用時	乗除算器使用時	積和演算器使用時	
除算	@@csudiv	–	40	40	signed char 同士の除算
	@@cudiv	–	40	40	unsigned char 同士の除算
	@@isdiv	–	39	39	signed int 同士の除算
	@@iudiv	–	39	39	unsigned int 同士の除算
	@@ludiv	–	43	43	unsigned long 同士の除算
	@@lsdiv	–	43	43	signed long 同士の除算
	@@cudivr	–	40	40	unsigned char 同士の除算（RAM 配置用）
	@@fdiv	–	44	44	float 同士の除算
	@@csdivr	–	40	40	signed char 同士の除算（RAM 配置用）
	@@iudivr	–	39	39	unsigned int 同士の除算（RAM 配置用）
	@@isdivr	–	39	39	signed int 同士の除算（RAM 配置用）
	@@ludivr	–	43	43	unsigned long 同士の除算（RAM 配置用）
	@@lsdivr	–	43	43	signed long 同士の除算（RAM 配置用）
	@@fdivr	–	44	44	float 同士の除算（RAM 配置用）

分類	関数名	最大割り込み禁止時間			備考
		乗算器使用時	乗除算器使用時	積和演算器使用時	
剰余算	@@curem	–	40	40	unsigned char 同士の剰余算
	@@csrem	–	40	40	signed char 同士の剰余算
	@@iurem	–	39	39	unsigned int 同士の剰余算
	@@isrem	–	39	39	signed int 同士の剰余算
	@@lurem	–	43	43	unsigned long 同士の剰余算
	@@lsrem	–	43	43	signed long 同士の剰余算
	@@curemr	–	40	40	unsigned char 同士の剰余算（RAM 配置用）
	@@csremr	–	40	40	signed char 同士の剰余算（RAM 配置用）
	@@iuremr	–	39	39	unsigned int 同士の剰余算（RAM 配置用）
	@@isremr	–	39	39	signed int 同士の剰余算（RAM 配置用）
	@@luremr	–	43	43	unsigned long 同士の剰余算（RAM 配置用）
	@@lsremr	–	43	43	signed long 同士の剰余算（RAM 配置用）
積和演算	@@macuw **注 1**	24	24	21	unsigned int × unsigned int ＋ unsigned long
	@@macsw **注 1**	24	24	21	signed int × signed int ＋ signed long
補助	@@divuw	–	43	43	divuw 命令互換
	@@divuwr	–	43	43	divuw 命令互換（RAM 配置用）
stdio.h	printf	–	43 **注 2**	43 **注 2**	データを SFR に出力
	sprintf	–	43 **注 2**	43 **注 2**	データを文字列に書き込み
	vprintf	–	43 **注 2**	43 **注 2**	データを SFR に出力
	vsprintf	–	43 **注 2**	43 **注 2**	データを文字列に書き込み
stdlib.h	div	–	41	41	int 型の除算
	ldiv	–	46	46	long 型の除算
	rand	24	24	24	@@lumul を使用
	qsort	12	12	12	@@iumul を使用

注 1. RL78 拡張命令搭載品の場合の最大割り込み禁止時間は，17 クロック

2. （ ）内は浮動小数点対応版使用時の値

6. 16 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

RL78,78K0R C 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 フォルダ下にあるバッチ・ファイルについて，次に示します。

表 6—3 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*.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，このバッチ・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，このバッチ・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。 デフォルトでは，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，このバッチ・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。 デフォルトでは，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，このバッチ・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 SM+ for 78K0R 対応に更新します。 SM+ for 78K0R で putchar 関数の出力を確認したい場合は，このバッチ・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（デフォルトは退避／復帰を行いません）。 -qr オプションを指定する場合は，このバッチ・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（デフォルトは退避／復帰を行いません）。 -qr オプションを指定しない場合は，このバッチ・ファイルを使用してライブラリを更新してください。
repvect.bat	フラッシュ領域に配置する割り込みベクタ・テーブルへの分岐テーブルのアドレス値の設定処理（vect*.asm）を更新します。 デフォルトでは，フラッシュ領域分岐テーブルの先頭アドレスが 2000H に設定されていますが，フラッシュ領域分岐テーブルの先頭アドレスを変更したい場合は，vect.inc 中の ITBLTOP の EQU 定義値を変更し，このバッチ・ファイルを使用してライブラリを更新してください。
repmul.bat	乗算器ライブラリを更新します。
repmuldiv.bat	乗除算器ライブラリを更新します。
repmac.bat	積和演算器ライブラリを更新します。
repmac_rl78.bat	積和演算命令使用ライブラリを更新します（RL78 拡張命令搭載品）。

6. 16. 1 バッチ・ファイルの使用法

サブフォルダ 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，ライブリアンの起動を行うバッチ・ファイルとなっているため，CubeSuite+ に同梱されているアセンブラなどが必要です。バッチ・ファイルを使用する前に，RL78,78K0R アセンブラの実行形式ファイルがあるフォルダを環境変数 PATH で設定してください。

バッチ・ファイルは，bat と同レベルのサブフォルダ（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 と同レベルのサブフォルダ 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・ファイルでアセンブルしたファイルは Src￥cc78k0r￥lib へ出力するので，lib78k0r ディレクトリにコピーしてリンクしてください。

バッチ・ファイルの使用方法は，カレント・フォルダをサブフォルダ bat に移動し，各バッチ・ファイルを実行します。その際，次の引数が必要です。

品種 = chiptype（ターゲット・チップの種別）

f1166a0 … uPD78F1166_A0 など

なお，デバイス・ファイルのパスを変更する場合は，品種指定の後ろに -y オプションを指定してください。

```
バッチ・ファイル名 品種 -y デバイス・ファイルのパス
```

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

```
mkstup 品種
```

例を以下に示します。

```
mkstup f1166a0
```

（2）ROM 化ルーチン更新用

```
reprom 品種
```

例を以下に示します。

```
reprom f1166a0
```

（3）getchar 関数更新用

```
regetc 品種
```

例を以下に示します。

```
repgetc f1166a0
```

（4）putchar 関数更新用

```
repputc 品種
```

例を以下に示します。

```
repputc f1166a0
```

（5）putchar 関数（SM78K0R 対応）更新用

```
repputcs 品種
```

例を以下に示します。

```
repputcs f1166a0
```

（6）setjmp/longjmp 関数更新用（復帰／退避処理あり）

```
repselo 品種
```

例を以下に示します。

```
repselo f1166a0
```

（7）setjmp/longjmp 関数更新用（復帰／退避処理なし）

```
repselon 品種
```

例を以下に示します。

```
repselon f1166a0
```

（8）割り込みベクタ・テーブル更新用

```
repvect 品種
```

例を以下に示します。

```
repvect f1166a0
```

（9）乗算器使用ライブラリ更新用

```
repmul.bat 品種
```

UPD78F1235_64 用に更新する例を以下に示します。

```
repmul.bat f123564
```

以下が更新されます。

Src￥cc78k0r￥lib￥cl0rxm.lib
cl0rxme.lib
cl0rxl.lib
cl0rxle.lib

（10）乗除算器使用ライブラリ更新用

```
repmuldiv.bat 品種
```

UPD78F1235_64 用に更新する例を以下に示します。

```
repmuldiv.bat f123564
```

以下が更新されます。

Src￥cc78k0r￥lib￥cl0rdm.lib
cl0rdme.lib
cl0rdl.lib
cl0rdle.lib

（11）積和演算器使用ライブラリ更新用

```
repmac.bat 品種
```

UPD78F1070_64 用に更新する例を以下に示します。

```
repmac.bat f107064
```

以下が更新されます。

Src￥cc78k0r￥lib￥cl0ram.lib
cl0rame.lib
cl0ral.lib
cl0rale.lib

（12）積和演算命令使用ライブラリ更新用

```
repmac_rl78.bat 品種
```

R5F104LE 用に更新する例を以下に示します。

```
repmac_rl78.bat f104le
```

以下が更新されます。

Src￥cc78k0r￥lib￥cl78m.lib
cl78me.lib
cl78l.lib
cl78le.lib

第 7 章　スタートアップ

この章では，スタートアップ・ルーチンについて説明します。

7.1　機能概要

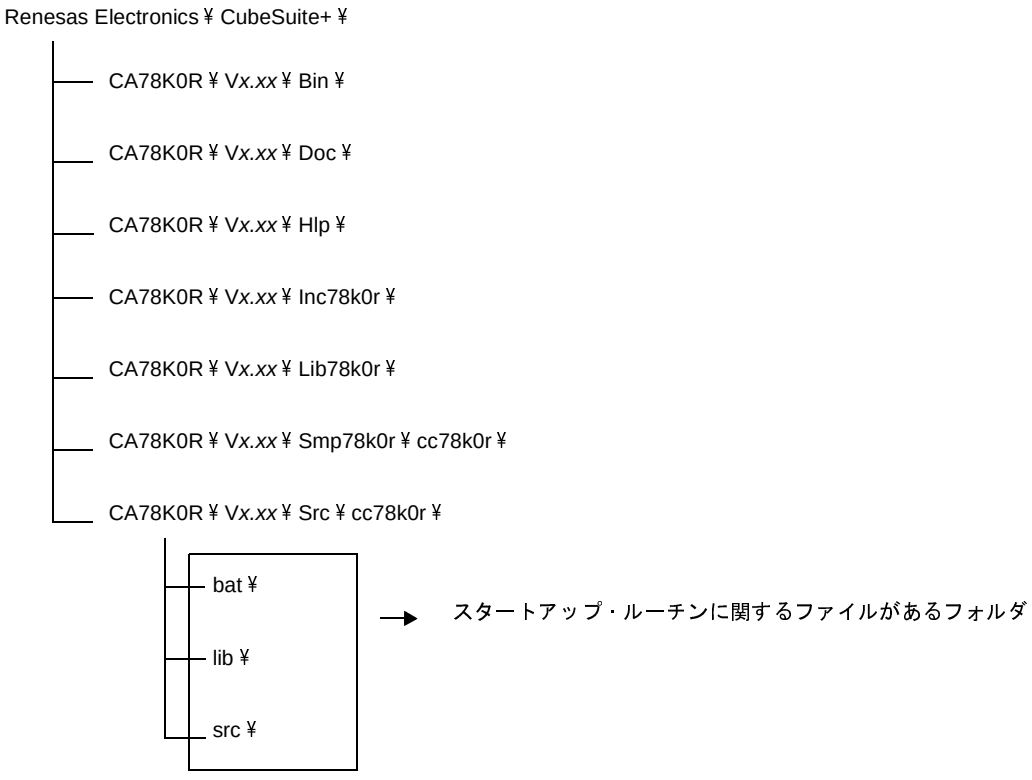
C 言語によるプログラムを実行させるには，システムへ組み込むための ROM 化処理，ユーザ・プログラム（main 関数）の起動などを行うプログラムが必要となります。このプログラムのことをスタートアップ・ルーチンと呼びます。

ユーザが作成したプログラムを実行させるためには，そのプログラムに応じたスタートアップ・ルーチンを作成しなければなりません。CA78K0R は，プログラム実行前に必要な処理を含むスタートアップ・ルーチンのオブジェクト・ファイルと，ユーザがシステムに合わせて変更できるようにスタートアップ・ルーチンのソース・ファイル（アセンブリ・ソース）を提供しています。スタートアップ・ルーチンのオブジェクト・ファイルをユーザ・プログラムとリンクすることにより，ユーザが実行前処理を記述しなくても実行可能なプログラムを作成することができます。

以降，スタートアップ・ルーチンの内容，使い方，改良のポイントなどについて説明します。

7.2　ファイルの構成

スタートアップ・ルーチンに関するファイルは，C コンパイラ・パッケージのフォルダ Src￥cc78k0r に格納されています。



次に，Src￥cc78k0r 以下にあるフォルダの内容について示します。

7. 2. 1　フォルダ bat の内容

このフォルダのバッチ・ファイルは，IDE 上では使用することができません。
これらのバッチ・ファイルは，スタートアップ・ルーチンなどのソース修正が必要な場合のみ使用してください。

表 7―1　フォルダ“bat”の内容

バッチ・ファイル名	説明
mkstup.bat	スタートアップ・ルーチンのアセンブル用バッチ・ファイル
reprom.bat	rom.asm 更新用バッチ・ファイル **注 1**
repgetc.bat	getchar.asm 更新用バッチ・ファイル
repputc.bat	putchar.asm 更新用バッチ・ファイル
repputcs.bat	_putchar.asm 更新用バッチ・ファイル
repselo.bat	setjmp.asm，longjmp.asm 更新用バッチ・ファイル （コンパイラ予約領域退避あり） **注 2**
repselon.bat	setjmp.asm，longjmp.asm 更新用バッチ・ファイル （コンパイラ予約領域退避なし） **注 2**
repvect.bat	vect*.asm 更新用バッチ・ファイル
repmul.bat	乗算器ライブラリ 更新用バッチ・ファイル
repmuldiv.bat	乗除算器ライブラリ 更新用バッチ・ファイル
repmac.bat	積和演算器ライブラリ 更新用バッチ・ファイル
repmac_rl78.bat	積和演算命令使用ライブラリ更新用バッチ・ファイル（RL78 拡張命令搭載品）

注 1. ROM 化ルーチンは，ライブラリに含まれているため，このバッチ・ファイルでライブラリも更新されます。

2. コンパイラ予約領域（KREGxx などのために確保される saddr 領域）の退避がある setjmp/longjmp と，退避がない（レジスタの退避のみ行う）setjmp/longjmp を作成します。

7. 2. 2　フォルダ lib の内容

フォルダ lib には，スタートアップ・ルーチン，ライブラリのソースをアセンブルしたオブジェクト・ファイルが入っています。このオブジェクト・ファイルは，RL78,78K0R であれば，どのターゲット・デバイス用のプログラムとでもリンクすることができます。特に修正が必要ない場合には，あらかじめ入っているオブジェクト・ファイルをそのままリンクしてください。CA78K0R が提供している mkstup.bat を実行すると，このオブジェクト・ファイルは上書きされます。

表 7—2　フォルダ“lib”の内容

ファイル名			説明
通常	ブート領域	フラッシュ領域	
cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0ram.lib cl0rxl.lib cl0rdl.lib cl0ral.lib cl78m.lib cl78l.lib cl78mf.lib cl78lf.lib	cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0ram.lib cl0rxl.lib cl0rdl.lib cl0ral.lib cl78m.lib cl78l.lib cl78mf.lib cl78lf.lib	cl0rme.lib cl0rle.lib cl0rmfe.lib cl0rlfe.lib cl0rxme.lib cl0rdme.lib cl0rame.lib cl0rxle.lib cl0rdle.lib cl0rale.lib cl78me.lib cl78le.lib cl78mfe.lib cl78lfe.lib	ライブラリ・ファイル（ランタイム・ライブラリ，標準ライブラリ）**注 1**
s0rm.rel s0rml.rel s0rl.rel s0rll.rel	s0rmb.rel s0rmlb.rel s0rlb.rel s0rllb.rel	s0rme.rel s0rmle.rel s0rle.rel s0rlle.rel	スタートアップ・ルーチンのオブジェクト・ファイル**注 2**

注 1.　ライブラリ・ファイルの命名規則は，次のようになっています。

```
lib78k0r￥cl<line><mul><model><float><flash>.lib
```

<line>

0r　：RL78 拡張命令非搭載品 /78K0R 用

78　：RL78 拡張命令搭載品用

<mul>

なし　：標準ライブラリ

x　：乗算器使用

d　：乗除算器使用

a　：積和演算器使用

<model>

m　：スモール・モデル，ミディアム・モデル

l　：ラージ・モデル

<float>

なし　：標準ライブラリ，ランタイム・ライブラリ（浮動小数点ライブラリ未使用）

f　：浮動小数点ライブラリ用

<flash>

なし　：通常／ブート領域用

e　　：フラッシュ領域用

2. スタートアップ・ルーチンの命名規則は，次のようになっています。

```
lib78k0r￥s0r<model><lib><flash>.rel
```

<model>

m　　：ミディアム・モデル（スモール・モデル兼用）

l　　：ラージ・モデル

<lib>

なし　：標準ライブラリ固定領域を使用しない場合

l　　：標準ライブラリ固定領域を使用する場合

<flash>

なし　：通常用

b　　：ブート領域用

e　　：フラッシュ領域用

RL78,78K0R C コンパイラのライブラリでは，次のようなデバイスの乗算器／乗除算器に対応しています。

RL78 拡張命令搭載品用ライブラリでは RL78 拡張命令に，RL78 拡張命令非搭載品 /78K0R 用ライブラリでは乗算器／乗除算器／積和演算器に対応しています。

ただし，演算途中に割り込みが入った場合に演算結果を壊さないように，割り込み禁止にしている部分があります。

対象となるライブラリ関数と，割り込み禁止時間については，「6. 15　ライブラリ最大割り込み禁止時間一覧」を参照してください。

乗算器／乗除算器／積和演算器や，RL78 拡張命令に関する搭載の有無については，各デバイスのユーザーズ・マニュアルを参照してください。

7. 2. 3　フォルダ src の内容

フォルダ src には，スタートアップ・ルーチン，ROM 化ルーチン，エラー処理ルーチン，標準ライブラリ関数（一部）のアセンブラ・ソースが入っています。システムに合わせて修正が必要な場合は，このアセンブラ・ソースを修正し，bat フォルダのバッチ・ファイルでアセンブルなどを行うことにより，リンクするオブジェクト・ファイルを作成することができます。

表 7―3　フォルダ“src”の内容

スタートアップ・ルーチン・ソース・ファイル名	説明
cstart.asm **注**	スタートアップ・ルーチンのソース・ファイル （標準ライブラリ使用時用）

スタートアップ・ルーチン・ソース・ファイル名	説明
cstartn.asm 注	スタートアップ・ルーチンのソース・ファイル （標準ライブラリ未使用時用）
rom.asm	ROM 化ルーチンのソース・ファイル
_putchar.asm	_putchar 関数
putchar.asm	putchar 関数
getchar.asm	getchar 関数
longjmp.asm	longjmp 関数
setjmp.asm	setjmp 関数
vectxx.asm	各割り込みベクタ・ソース（xx：ベクタ・アドレス）
def.inc	ライブラリ種別設定用
macro.inc	各種定型パターンについてのマクロ定義
vect.inc	フラッシュ領域分岐テーブルの先頭アドレス
library.inc	明示的にブート領域に配置するライブラリの選択
imul.asm, lmul.asm, mulsw.asm, muluw.asm	乗算器，乗除算器ライブラリ用関数
csdiv.asm, cudiv.asm, csrem.asm, curem.asm, isdiv.asm, iudiv.asm, isrem.asm, iurem.asm, lsdiv.asm, ludiv.asm, lsrem.asm, lurem.asm, divuw.asm, div.asm, ldiv.asm	乗除算器ライブラリ用関数
macsw.asm, macuw.asm	積和演算器，積和演算命令ライブラリ用関数

注　ファイル名に“n”が付加されたものは，標準ライブラリ処理がないスタートアップ・ルーチンです。標準ライブラリを使用しない場合のみに使用してください。また，cstartb*.asm はブート領域用スタートアップ・ルーチン，cstarte*.asm はフラッシュ領域用スタートアップ・ルーチンです。

7.3　バッチ・ファイル

この節では，バッチ・ファイルについて説明します。

7.3.1　スタートアップ・ルーチン作成用バッチ・ファイル

スタートアップ・ルーチンのオブジェクト・ファイルを作成するには，フォルダ bat にある mkstup.bat を使用します。

また，mkstup.bat では，CA78K0R の中のアセンブラが必要となります。したがって，PATH を設定していない場合は，設定して動作できるようにしてください。

次に，使用方法を示します。

- mkstup.bat のあるフォルダ Src￥cc78k0r￥bat で，次のようにコマンド行で実行してください。

```
mkstup デバイス種別注
```

注　各デバイスのユーザーズ・マニュアル，または「CubeSuite+ 対応機能一覧」を参照してください。

次に，使用例を示します。

- 対象品種が uPD78F1166_A0 のときに使用するスタートアップ・ルーチンを作成します。

```
mkstup f1166a0
```

バッチ・ファイル mkstup.bat は，フォルダ bat と同じ階層のフォルダ lib の下にスタートアップ・ルーチンのオブジェクト・ファイルを上書きする形で格納します。

それぞれのフォルダには，オブジェクト・ファイルをリンクするときに必要なスタートアップ・ルーチンが出力されます。

次に，lib に作成されるオブジェクト・ファイル名を示します。

lib ── s0rm.rel
s0rmb.rel
s0rme.rel
s0rml.rel
s0rmlb.rel
s0rmle.rel
s0rl.rel
s0rlb.rel
s0rle.rel
s0rll.rel
s0rllb.rel
s0rlle.rel

7.4　スタートアップ・ルーチン

この節では，スタートアップ・ルーチンについて説明します。

スタートアップ・ルーチンは，ユーザが作成した C ソース・プログラムを実行させるために必要な準備を行います。ユーザのプログラムとリンクさせることにより，目的を果たすロード・モジュール・ファイルを作成することができます。

(1) 機能

メモリの初期化，システムへ組み込むための ROM 化処理，C ソース・プログラムの起動，終了処理などを行います。

- メモリの初期化

  saddr 領域の初期化，スタック領域の初期化，初期値なし外部変数の初期化を行います。

- ROM 化処理

  C ソース・プログラム中で定義された外部変数，スタティック変数，sreg 変数の初期値は，ROM に配置されます。しかし，ROM に配置されたままでは，変数の値を書き換えることができません。そのため，ROM に配置された初期値を RAM にコピーする必要があります。この処理を ROM 化処理といい，プログラムを ROM に書き込んだとき，マイコン上で動作できるようにします。

（2）構成

スタートアップ・ルーチンに関連するプログラムとその構成を次に示します。

図 7―1　スタートアップ・ルーチンに関連するプログラムとその構成

注 1. 標準ライブラリを使用する場合は，ライブラリに関する処理が最初に行われます。スタートアップ・ルーチン・ソース・ファイルの名前の最後に n がないものは，標準ライブラリに関する処理があり，n が付いたファイルは処理が省かれています。

2. hdwinit 関数は，周辺装置（sfr）の初期設定をする関数としてユーザが必要に応じて作成する関数です。hdwinit 関数を作成することにより，初期設定のタイミングを早くすることができます（main 関数の中でも初期設定は可能です）。ユーザが hdwinit 関数を作成しない場合は，何もせずにリターンします。

cstart.asm，と cstartn.asm は，ほぼ同じ内容です。

上記のファイルの違いを次に示します。

スタートアップ・ルーチンの種類	ライブラリ処理の有無
cstart.asm	あり
cstartn.asm	なし

（3）スタートアップ・ルーチンの使い分け

CA78K0R が提供している各ソース・ファイルに対応したオブジェクト・ファイル名を次に示します。

ファイルの種類	ソース・ファイル	オブジェクト・ファイル
スタートアップ・ルーチン	cstart*.asm **注 1，2**	s0r*.rel **注 2，3，4**
ROM 化ファイル	rom.asm	ライブラリに含まれます。

注 1. *：標準ライブラリを使用しない場合は“n”が付きます。使用する場合は文字は付きません。

2. *：ブート領域用の場合は“b”，フラッシュ領域用の場合は“e”が付きます。

3. *：標準ライブラリの固定領域を使用する場合は“l”が付きます。

4. *：スモール・モデル，ミディアム・モデルの場合は“m”，ラージ・モデルの場合は“l”が付きます。
スモール・モデル，ミディアム・モデルの場合でも，far 領域に変数を配置する場合は，“l”付きのスタートアップ・ルーチンを使用してください。

備考　rom.asm は，ROM 化処理でコピーされるデータの最終アドレスを示すラベルを定義しています。
rom.asm のオブジェクトは，ライブラリに含まれています。

7. 4. 1　前 処 理

サンプル・プログラム（cstart.asm）の前処理について説明します。

備考　cstart などは，先頭に _@ を付加した形式で呼び出されます。

```
        NAME    @cstart

$INCLUDE ( def.inc )                                                    ; (1)
$INCLUDE ( macro.inc )
                                                                        ; (2)
BRKSW     EQU   1     ; brk, sbrk, calloc, free, malloc, realloc function use
EXITSW    EQU   1     ; exit, atexit function use
RANDSW    EQU   1     ; rand, srand  function use
DIVSW     EQU   1     ; div           function use
LDIVSW    EQU   1     ; ldiv          function use
FLOATSW   EQU   1     ; floating point variables use
STRTOKSW  EQU   1     ; strtok        function use

        PUBLIC  _@cstart, _@cend                                        ; (3)

$_IF ( BRKSW )
        PUBLIC  _@BRKADR, _@MEMTOP, _@MEMBTM
          :
$ENDIF
        EXTRN   _main, _@STBEG, _hdwinit, _@MAA                         ; (4)
$_IF ( EXITSW )
```

```
        EXTRN   _exit
$ENDIF
                                                                    ; (5)
        EXTRN   _?R_INIT, _?RLINIT, _?R_INIS, _?DATA, _?DATAL, _?DATS
@@DATA  DSEG    BASEP ; near                                        ; (6)

$_IF ( EXITSW )
_@FNCTBL :      DS    4 * 32
_@FNCENT :      DS    2
  :
_@MEMTOP :      DS    32
_@MEMBTM :
$ENDIF
```

(1) インクルード・ファイルの取り込み

def.inc　→　ライブラリ種別設定用

macro.inc　→　各種定型パターンについてのマクロ定義

(2) ライブラリ・スイッチ

コメントにある標準ライブラリを使用しない場合は，EQU 定義を 0 に修正することにより，使用しないライブラリの処理や，ライブラリ用に確保している領域を節約できます。デフォルトは，すべて使用する設定になっています（ライブラリ処理なしのスタートアップ・ルーチンには，この処理はありません）。

(3) シンボル定義

標準ライブラリ使用時に使用するシンボルを定義します。

(4) スタック解決用のシンボルの外部参照宣言

スタック解決用のパブリック・シンボル（_@STBEG）を外部参照宣言します。

_@STBEG は，スタック領域の最終アドレス +1 を値として持ちます。

_@STBEG は，リンカのスタック解決用シンボル生成オプション -s を指定することにより，自動生成されます。したがって，リンク時には -s オプションを必ず指定してください。この際，スタックに使用する領域名も指定してください。領域名を省略した場合は，RAM という領域を使用しますが，リンク・ディレクティブ・ファイルを作成することにより，スタック用領域を自由に配置することができます。メモリ・マップに関しては，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。

次にリンク・ディレクティブ・ファイルの例を示します。リンク・ディレクティブ・ファイルは，通常のエディタでユーザが作成するテキスト・ファイルです（記述方法に関する詳細については，「7.6　コーディング例」を参照してください）。

例　リンク時に -sSTACK を指定した場合

lk78k0r.dr（リンク・ディレクティブ・ファイル）を作成します。ターゲット・デバイスのメモリ・マップを参照して ROM，RAM はデフォルトで配置されるので，配置を変更しない場合は，指定する必要はありません。

リンク・ディレクティブについては，Smp78k0r￥CA78K0R フォルダの lk78k0r.dr を参考にしてください。

```
                    先頭アドレス    サイズ
                        ↓          ↓
memory  SDR     : ( 0xFFE20h, 0000098h )
memory  STACK   : ( 0xxxxxxh, 0xxxxxxh ) ← ここに先頭アドレスとサイズを指定し，
                                            -d リンカ・オプションで lk78k0r.dr を
                                            指定します（例：-dlk78k0r.dr）
merge @@INIS    : = SDR
merge @@DATS    : = SDR
merge @@BITS    : = SDR
```

（5）ROM 化処理用ラベルの外部参照宣言

ROM 化処理用ラベルは，後処理の部分で定義されます。

（6）標準ライブラリ用領域確保

標準ライブラリ使用時に使用するための領域を確保します。

7. 4. 2　初期設定

サンプル・プログラム（cstart.asm）の初期設定について説明します。

```
@@VECT00        CSEG    AT      0                                       ; (1)
        DW      _@cstart

@@LCODE CSEG    BASE
_@cstart :
        SEL     RB0                                                     ; (2)
        MOV     A, #_@MAA                                               ; (3)
        MOV1    CY, A.0
        MOV1    MAA, CY
        MOVW    SP, #LOWW _@STBEG       ; SP <-stack begin address      ; (4)
        CALL    !!_hdwinit                                              ; (5)
          :
$_IF ( BRKSW OR EXITSW OR RANDSW OR FLOATSW )
        CLRW    AX
$ENDIF
          :
```

（1）リセット・ベクタの設定

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・ルーチンの先頭アドレスを設定します。

```
@@VECT00        CSEG    AT      0000H
                DW      _@cstart
```

(2) レジスタ・バンクの設定

レジスタ・バンク RB0 をワーク・レジスタとして設定します。

(3) ミラー領域の設定

ミラー領域の設定を行います。

ミラー領域については，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。

(4) SP（スタック・ポインタ）の設定

スタック・ポインタに，_@STBEG を設定します。

_@STBEG は，リンカのスタック解決用シンボル生成オプション -s を指定することにより，自動生成されます。

(5) ハードウエア初期化関数呼び出し

hdwinit 関数は，周辺装置（SFR）の初期設定をする関数としてユーザが必要に応じて作成する関数です。この関数を作成することにより，ユーザの目的に合った初期設定が可能となります。

ユーザが hdwinit 関数を作成しない場合は，何もせずリターンします。

(6) ROM 化処理

cstart.asm の ROM 化処理について説明します。

```
; copy external variables having initial value
$_IF ( _ESCOPY )
        MOV     ES, #HIGHW _@R_INIT
$ENDIF
        MOVW    HL, #LOWW _@R_INIT
        MOVW    DE, #LOWW _@INIT
        BR      $LINIT2
LINIT1 :
$_IF ( _ESCOPY )
        MOV     A, ES:[HL]
$ELSE
        MOV     A, [HL]
$ENDIF
        MOV     [DE], A
        INCW    HL
        INCW    DE
LINIT2 :
        MOVW    AX, HL
        CMPW    AX, #LOWW _?R_INIT
        BNZ     $LINIT1
```

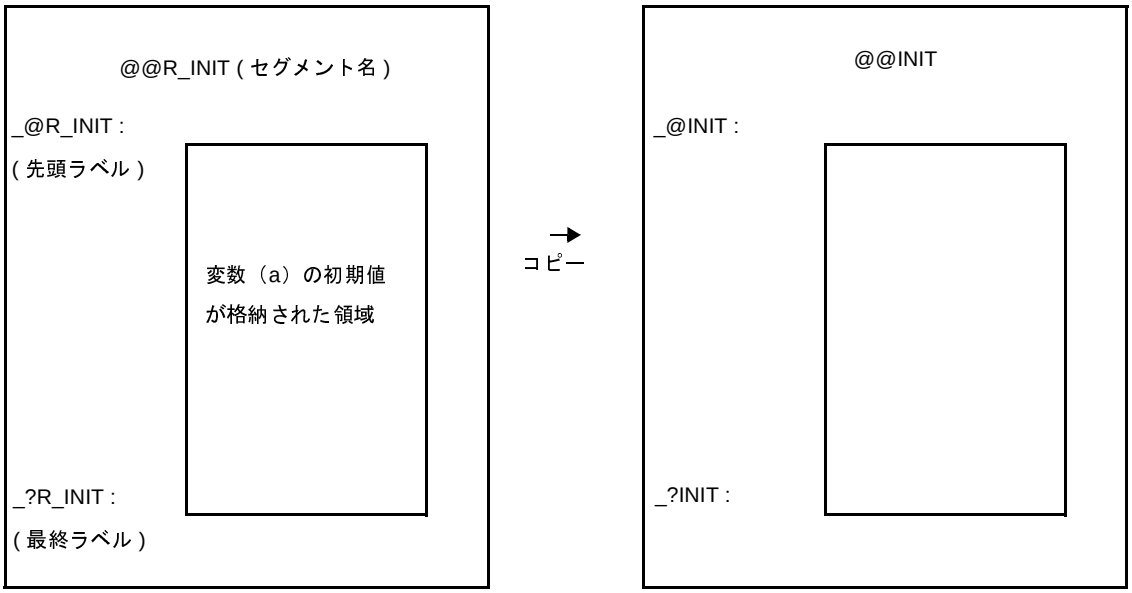
ROM 化処理では，ROM に配置された外部変数，sreg 変数の初期値を RAM にコピーします。処理される変数は，次の例に示すように（a）～（d）の 4 種類あります。

```
char    c = 1 ;                 （a）初期値あり外部変数
int     i ;                     （b）初期値なし外部変数注
__sreg  int     si = 0 ;        （c）初期値あり sreg 変数
__sreg  char    sc ;            （d）初期値なし sreg 変数注

void main ( void ) {
          :
}
```

注　初期値なし外部変数，sreg 変数はコピーせず，直接 RAM に 0 を入れます。

- （a）初期値あり外部変数の ROM 化処理を次に示します。
  変数（a）の初期値は，RL78,78K0R C コンパイラにより，ROM 上の @@R_INIT というセグメントに配置されます。
  ROM 化処理は，これらの値を RAM 上の @@INIT というセグメントにコピーします（変数（c）についても，同様な処理を行います）。

図 7―2　初期値あり外部変数の ROM 化処理



- @@R_INIT セグメントの先頭ラベル，最終ラベルは，_@R_INIT，_?R_INIT で，@@INIT セグメントの先頭ラベル，最終ラベルは，_@INIT，_?INIT で定義されています。

- 変数（b），（d）については，コピーではなく直接 RAM の決められたセグメントへ 0 を入れます。

（a），（c）の変数が配置される ROM 領域のセグメント名，および各セグメントでの初期値の先頭，最終ラベルを次に示します。

変数の種類	セグメント	先頭ラベル	最終ラベル
初期値あり外部変数（a） （near 領域に配置の場合）	@@R_INIT	_@R_INIT	_?R_INIT
初期値あり外部変数（a） （far 領域に配置の場合）	@@RLINIT	_@RLINIT	_?RLINIT
初期値あり sreg 変数（c）	@@R_INIS	_@R_INIS	_?R_INIS

（a）～（d）の変数が配置される RAM 領域のセグメント名，および各セグメントでの初期値の先頭，最終ラベルを次に示します。

変数の種類	セグメント	先頭ラベル	最終ラベル
初期値あり外部変数（a） （near 領域に配置の場合）	@@INIT	_@INIT	_?INIT
初期値あり外部変数（a） （far 領域に配置の場合）	@@INITL	_@INITL	_?INITL
初期値なし外部変数（b） （near 領域に配置の場合）	@@DATA	_@DATA	_?DATA
初期値なし外部変数（b） （far 領域に配置の場合）	@@DATAL	_@DATAL	_?DATAL
初期値あり sreg 変数（c）	@@INIS	_@INIS	_?INIS
初期値なし sreg 変数（d）	@@DATS	_@DATS	_?DATS

7. 4. 3　main 関数の起動と後処理

サンプル・プログラム（cstart.asm）の main 関数の起動と後処理について説明します。

```
            CALL    !!_main          ; main ( ) ;            ;  (1)
$_IF ( EXITSW )
            CLRW    AX
            CALL    !!_exit          ; exit ( 0 ) ;          ;  (2)
$ENDIF
            BR      $$
;
_@cend :
                                                            ;  (3)
@@R_INIT    CSEG    UNIT64KP
_@R_INIT :
@@RLINIT    CSEG    UNIT64KP
_@RLINIT :
```

```
@@R_INIS    CSEG    UNIT64KP
_@R_INIS :
@@INIT      DSEG    BASEP
_@INIT :
@@INITL     DSEG    UNIT64KP
_@INITL :
@@DATA      DSEG    BASEP
_@DATA :
@@DATAL     DSEG    UNIT64KP
_@DATAL :
@@INIS      DSEG    SADDRP
_@INIS :
@@DATS      DSEG    SADDRP
_@DATS :
@@CALT      CSEG    CALLT0
@@CNST      CSEG    MIRRORP
@@CNSTL     CSEG    PAGE64KP
@@BITS      BSEG
;
            END
```

（1）main 関数の起動

main 関数を呼び出します。

（2）exit 関数の起動

exit 処理が必要な場合は，exit 関数を呼び出します。

（3）ROM 化処理で使用するセグメント，ラベルの定義

ROM 化処理で，（1）～（4）の変数（「（6）ROM 化処理」を参照）ごとに，使用するセグメント，ラベルを定義します。セグメントは，各変数の初期値を格納する領域を示します。ラベルは，各セグメントの先頭アドレスを示します。

ROM 化用ファイル rom.asm について説明します。rom.asm のリロケータブル・オブジェクト・ファイルはライブラリの中に入っています。

```
        NAME    @rom
;
        PUBLIC  _?R_INIT, _?RLINIT, _?R_INIS
        PUBLIC  _?INIT, _?INITL, _?DATA, _?DATAL, _?INIS, _?DATS
;
@@R_INIT        CSEG    UNIT64KP                        ; （4）
_?R_INIT :
@@RLINIT        CSEG    UNIT64KP
_?RLINIT :
```

```
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG    BASEP
_?INIT :
@@INITL         DSEG    UNIT64KP
_?INITL :
@@DATA          DSEG    BASEP
_?DATA :
@@DATAL         DSEG    UNIT64KP
_?DATAL :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS :
;
        END
```

（4）ROM 化処理で使用するラベルの定義

ROM 化処理で，（1）～（4）の変数（「（6）ROM 化処理」を参照）ごとに，使用するラベルを定義します。これらのラベルは，各変数の初期値を格納するセグメントの最終アドレスを示します。

ユーザ・ライブラリが複数存在し，さらにそれぞれのユーザ・ライブラリの属するオブジェクト・モジュール・ファイル間にて相互参照が存在する場合に，CA78K0R に含まれる終端モジュール（rom.asm）のモジュール名“@rom”，“@rome”は変更しないでください。

変更した場合は，最後にリンクされない場合があります。

7.5　フラッシュ領域用スタートアップ・ルーチンでの ROM 化処理

フラッシュ用スタートアップ・ルーチンでは，通常のスタートアップ・ルーチンと次の点が異なります。

表 7―4　初期化データの ROM 領域のセクション

変数の種類	セグメント	先頭ラベル	終端ラベル
初期値あり外部変数（a） （near 領域に配置の場合）	@ER_INIT CSEG UNIT64KP	E@R_INIT	E?R_INIT
初期値あり外部変数（a） （far 領域に配置の場合）	@ERLINIT CSEG UNIT64KP	E@RLINIT	E?RLINIT
初期値あり sreg 変数（c）	@ER_INIS CSEG UNIT64KP	E@R_INIS	E?R_INIS

表 7―5　コピー先の RAM 領域のセクション

変数の種類	セグメント	先頭ラベル	終端ラベル
初期値あり外部変数（a） （near 領域に配置の場合）	@EINIT DSEG BASEP	E@INIT	E?INIT

変数の種類	セグメント	先頭ラベル	終端ラベル
初期値あり外部変数（a） （far 領域に配置の場合）	@EINITL DSEG UNIT64KP	E@INITL	E?INITL
初期値なし外部変数（b） （near 領域に配置の場合）	@EDATA DSEG BASEP	E@DATA	E?DATA
初期値なし外部変数（b） （far 領域に配置の場合）	@EDATAL DSEG UNIT64KP	E@DATAL	E?DATAL
初期値あり sreg 変数（c）	@EINIS DSEG SADDRP	E@INIS	E?INIS
初期値なし sreg 変数（d）	@EDATS DSEG SADDRP	E@DATS	E?DATS

- スタートアップ・ルーチンでは，ROM 領域，RAM 領域の各セグメントの先頭としてそれぞれに次のラベルを付けます。

  E@R_INIT，E@R_INIS，E@INIT，E@DATA，E@INIS，E@DATS，E@INITL，E@DATAL

  ラージ・モデルの場合，または far 領域に変数を配置する場合，さらに次のラベルを付けます。

  E@RLINIT，E@INITL，E@DATAL
- 終端モジュールでは，ROM 領域，RAM 領域の各セグメントの終端としてそれぞれに次のラベルを付けます。

  E?R_INIT，E?R_INIS，E?INIT，E?DATA，E?INIS，E?DATS，E?RLINIT，E?INITL，E?DATAL
- スタートアップ・ルーチンは ROM 領域の各セグメントの先頭ラベルのアドレスから，終端ラベルのアドレス - 1 までの内容を RAM 領域の各セグメントの先頭ラベルのアドレスからの領域にコピーします。
- E@DATA から E?DATA まで，E@DATS から E?DATS まで，0 を埋め込みます。
- ラージ・モデルの場合，または far 領域に変数を配置する場合，さらに E@DATAL から E?DATAL まで，0 を埋め込みます。

7.6　コーディング例

CA78K0R が提供しているスタートアップ・ルーチンは，実際に使用するターゲット・システムに合わせて修正できます。ここでは，これらのファイルを修正する場合のポイントについて説明します。

7.6.1　スタートアップ・ルーチンの修正ポイント

スタートアップ・ルーチン・ソース・ファイルの修正ポイントについて説明します。修正後は，修正したソース・ファイル（cstart*.asm）をフォルダ Src￥cc78k0r￥bat にあるバッチ・ファイル mkstup.bat を用いて，アセンブルしてください（*：英数字）。

（1）ライブラリ関数で使われるシンボル

次の表で示されるライブラリ関数を使わないのであれば，スタートアップ・ルーチン（cstart.asm）中の各関数に対応するシンボルは削除することができます。ただし，exit 関数は，スタートアップ・ルーチンで使用されるので，_@FNCTBL，_@FNCENT を削除することはできません（exit 関数も削除する場合は，それらのシンボルも削除することができます）。使用しないライブラリ関数のシンボルなどについては，ライブラリ・スイッチを変更することで削除することができます。

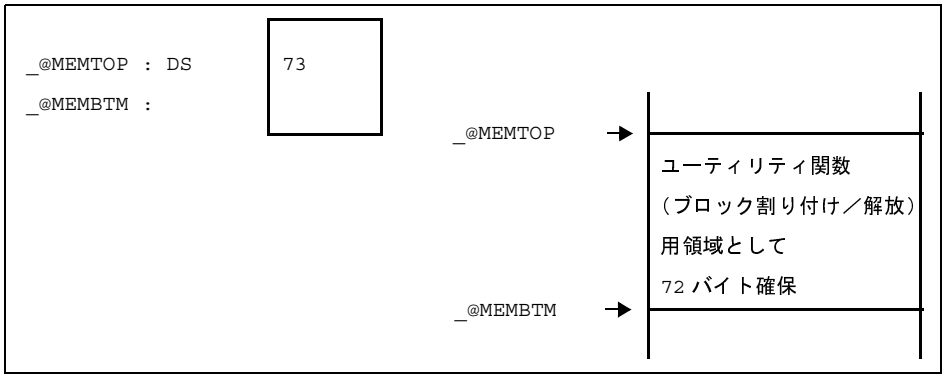
ライブラリ関数名	使われるシンボル
brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR
exit atexit	_@FNCTBL _@FNCENT
rand srand	_@SEED
div	_@DIVR
ldiv	_@LDIVR
strtok	_@TOKPTR
atof strtod strtol strtoul 数学関数 浮動小数点ランタイム・ライブラリ	_errno

（2）ユーティリティ関数（ブロック割り付け／解放）で使われる領域

ユーティリティ関数（ブロック割り付け／解放）で使われる領域サイズをユーザが定義する場合は，次の例のように設定します。

例　ユーティリティ関数（ブロック割り付け／解放）用として，72 バイト確保したい場合，スタートアップ・ルーチンの初期設定を次のように修正してください。

図 7―3　スタートアップ・ルーチンの初期設定例



スタートアップ・ルーチンに指定する数値は，確保したい領域サイズに 1 バイトを加えた値としてください。この例では，スタートアップ・ルーチンで 73 バイト確保していますが，実際にユーティリティ関数では 72 バイトまで確保可能となります。
指定したサイズが大きすぎる場合，RAM 領域に入りきらず，リンク時にエラーとなることがあります。このような場合には，次のように指定するサイズを小さくするか，リンク・ディレクティブ・ファイルを修正して回避してください。リンク・ディレクティブ・ファイルを修正する場合は，「5. 3. 2　コンパイラを使用する場合」を参照してください。

例　指定するサイズを小さくする場合

```
_@MEMTOP : DS    72    → 40 に変更
```

7. 6. 2　RTOS を使用する場合

RI78V4，および RL78,78K0R C コンパイラは，初期化処理ルーチン（アセンブラ形式）のサンプルをそれぞれ提供しています。したがって，RI78V4 と RL78,78K0R C コンパイラを併用する場合には，双方の初期化処理ルーチンの修正が必要となります。

第 8 章 ROM 化

ROM 化とは，初期値あり外部変数などの初期値を ROM に配置しておき，システム実行時に RAM にコピーする処理です。

CA78K0R は，プログラムの ROM 化処理付きのスタートアップ・ルーチンを提供しているので，スタートアップ時の ROM 化処理などを記述する手間が省けます。

スタートアップ・ルーチンの内容については，「7. 4 スタートアップ・ルーチン」を参照してください。

次に，プログラムの ROM 化を行う方法について説明します。

ROM 化時には，スタートアップ・ルーチン，オブジェクト・モジュール・ファイルとライブラリをリンクします。スタートアップ・ルーチンは，オブジェクト・プログラムの初期化を行います。

（1）s0r*.rel

スタートアップ・ルーチン（ROM 化対応）です。

初期化データのコピー・ルーチンを含み，初期化データの開始を示します。スタート・アドレスには，“_@cstart”というラベル（シンボル）が付けられます。

（2）cl0r*.lib

CA78K0R に添付されているライブラリです。

このライブラリ・ファイルの中には，次のものが含まれています。

- ランタイム・ライブラリ

  ランタイム・ライブラリ名は，シンボルの先頭に“@@”が付加されます。ただし，特殊ライブラリ cprep，cdisp には先頭に“_@”が付加されます。

- 標準ライブラリ

  標準ライブラリ名は，シンボルの先頭に“_”が付加されます。

（3）*.lib

ユーザ作成のライブラリです。

シンボルの先頭に“_”が付加されます。

注意 CA78K0R は，何種類かのスタートアップ・ルーチン，およびライブラリを提供しています。スタートアップ・ルーチンについては，「第 7 章 スタートアップ」を参照してください。ライブラリについては「7. 2. 2 フォルダ lib の内容」を参照してください。

第 9 章　コンパイラとアセンブラの相互参照

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，CA78K0R を使用し，次の順序で説明します。

- 引数／自動変数のアクセス方法
- 戻り値の格納方法
- アセンブリ言語から C 言語ルーチンの呼び出し
- C 言語からアセンブリ言語ルーチンの呼び出し
- C 言語で定義した変数をアセンブリ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 言語側で参照する方法
- C 言語ルーチンとアセンブラ関数間の呼び出しの注意事項

9. 1　引数／自動変数のアクセス方法

引数，および自動変数の割り当てについては，「3. 3. 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。

スタックに積まれた引数／自動変数をアクセスする際のベース・ポインタは，HL レジスタを使用します。

9. 2　戻り値の格納方法

「3. 3. 1　戻 り 値」を参照してください。

9. 3　アセンブリ言語から C 言語ルーチンの呼び出し

ここでは，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を説明します。

9. 3. 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。

（a）C のワーク・レジスタ（AX，BC，DE）を退避する

（b）引数をスタックに積む

（c）C 言語ルーチンをコールする

（d）引数のバイト数分スタック・ポインタ（SP）の値を修正する

（e）C 言語ルーチンの戻り値（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

```
$PROCESSOR ( F1166A0 )

        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）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。

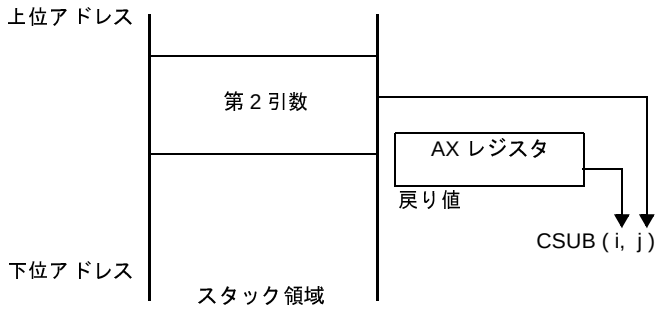
なお，HL レジスタについては，C 言語側で使用している場合，常に C 言語側で退避されます。

（2）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，次のようになります。

図 9—1　引数の受け渡し



（3）C 言語ルーチンのコール

C 言語ルーチンの呼び出しは，CALL 命令で行います。C 言語ルーチンが callt 関数の場合，“callt” 命令，callf 関数の場合，“callf” 命令で呼び出します。

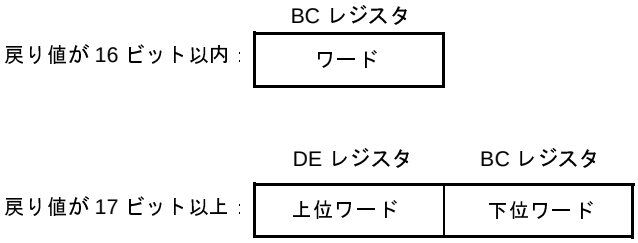
（4）スタック・ポインタ（SP）の復帰

引数を積んだバイト数分，スタックを復帰します。

（5）戻り値（BC，DE）の参照

C 言語からの戻り値は，次のように返されます。

図 9―2　戻り値の参照



9. 4　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，デフォルトの例を示します。

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

9. 4. 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 ( void ) {
        int     i, j ;
        long    l ;

        l = 0x54321 ;
        i = 1 ;
        j = FUNC ( i, l ) ;             /* 関数コール */
}
```

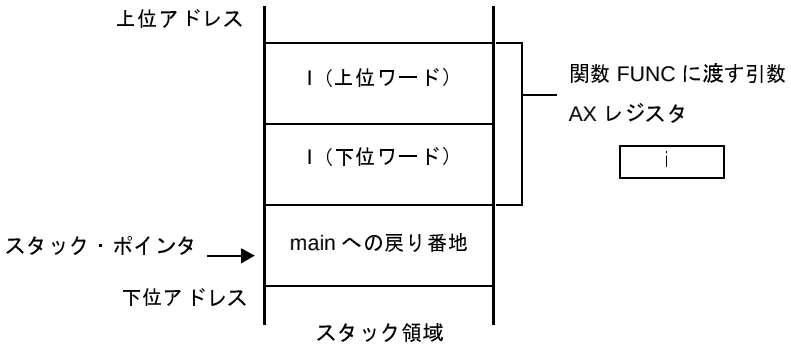
このプログラム例で，実行時に行われるプログラム間のインタフェースと制御の流れを次に示します。

（a）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

（b）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 9—3　関数呼び出し直後のスタック



9. 4. 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，次の手順で処理を行います。

（1）ベース・ポインタ，レジスタ変数用 saddr 領域を退避する

（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）戻り値をセットする

（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 ( F1166A0 )

        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2

@@DATA  DSEG    BASEP
_DT1 :  DS      ( 2 )
_DT2 :  DS      ( 4 )

@@CODE  CSEG
_FUNC :
        PUSH    HL              ; save base pointer      (1)
        PUSH    AX
        MOVW    HL, SP          ; copy stack pointer     (2)
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; move 1st argument ( i )
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]    ; arg2
        MOVW    !_DT2, AX       ; move 2nd argument ( l )
        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，“_”を付加したラベルを記述します。C ソース中で記述した関数名と同じ名前になります。

ラ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 saddr 領域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，レジスタ変数用 saddr 領域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。

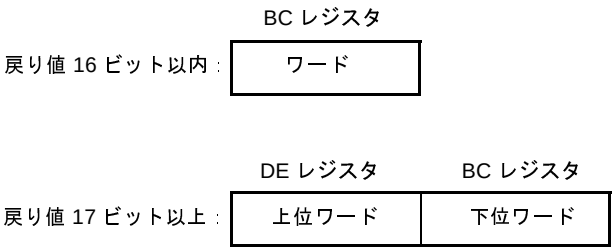
(3) 関数 FUNC 本来の処理を行う

（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）戻り値のセット

戻り値がある場合，戻り値を“BC”，“DE”レジスタへセットします。戻り値がない場合，セットする必要はありません。

図 9―4　戻り値のセット

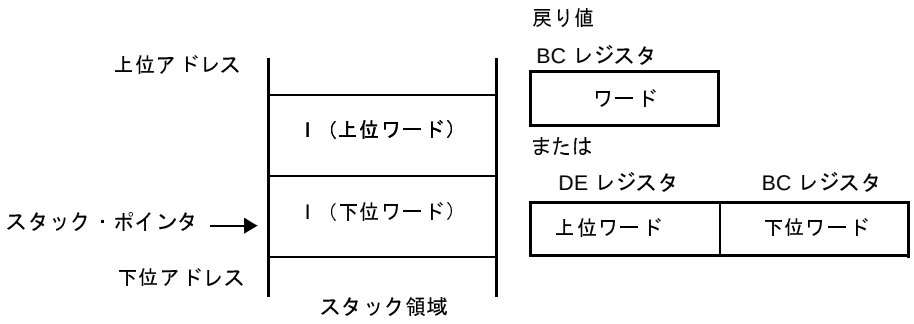


（5）レジスタの復帰

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 9―5　関数 main へのリターン



9.5　C 言語で定義した変数をアセンブリ言語側で参照する方法

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“_”（アンダースコア）を付けます。

C 言語のプログラム例を以下に示します。

```
extern void     subf ( void ) ;

char    c = 0 ;
int     i = 0 ;

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

アセンブラでは，次のように行います。

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

9.6　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を以下に示します。

```
extern char     c ;
extern int      i ;

void    subf ( void ) {
        c = ' A ' ;
        i = 4 ;
}
```

RL78,78K0R アセンブラでは，次のように行います。

```
        NAME    ASMSUB

        PUBLIC  _i
        PUBLIC  _c

ABC     DSEG    BASEP
_i :    DW      0
_c :    DB      0
        END
```

9. 7　C 言語ルーチンとアセンブラ関数間の呼び出しの注意事項

- “_”（アンダースコア）

RL78,78K0R 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“_”（アンダースコア，ASCII コード“5FH”）を付けます。

次の C プログラム例で，“b = FUNC (a, c) ;”は，“_FUNC という外部名を参照する”と翻訳されます。

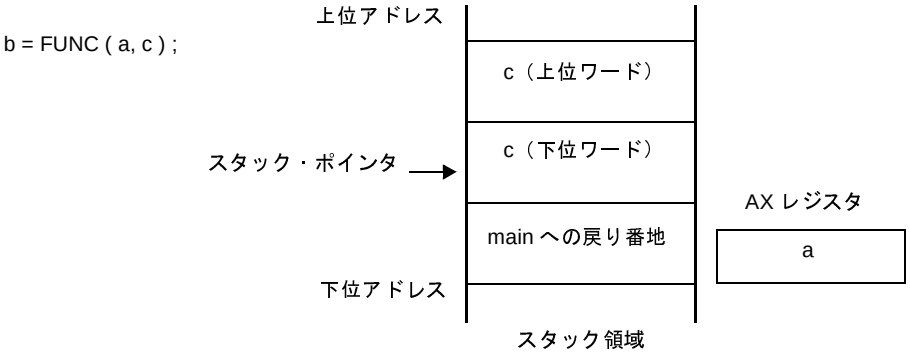
```
extern int      FUNC ( int, long ) ;    /* 関数プロトタイプ */

void main ( void ) {
        int     a,  b ;
        long    c ;

        c = 0x54321 ;
        a = c ;
        b = FUNC ( a, c ) ;             /* 関数コール */
}
```

RL78,78K0R アセンブラでは，ルーチン名を“_FUNC”と記述します。

- 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上位アドレスから下位アドレス方向に積まれます。

図 9―6　スタックに積む引数の配置



第 10 章　注意事項

この章では，コーディングを行う際の注意事項を示します。

（1）漢字コード種別

SJIS，または EUC コードを含むソースを Windows 上で使用するときは，環境変数 LANG78，または“コメント中の漢字コード”オプションを指定してください。

指定した漢字コードとソース中に含まれる漢字コードが異なる場合，ビルド時にエラーになる，またはソースの一部を誤ってコメントとみなして正しくビルドされない場合があります。

（2）インクルード・ファイル

インクルード・ファイル内で，関数を定義し（宣言を除きます），C ソース中で展開することはできません。インクルード・ファイル内で定義を行うと，ソース・デバッグ時に正しく定義行が表示されないなどの弊害があります。

（3）アセンブラ・ソースを出力して使用する場合

C ソース・プログラム中に，#asm ブロック，または __asm 文などのアセンブリ言語による記述がある場合，ロード・モジュール・ファイル作成手順は，コンパイル，アセンブル，リンクの順になります。

アセンブリ言語による記述がある場合などのように，RL78,78K0RC コンパイラで直接オブジェクトを出力せずに，いったんアセンブラ・ソースを出力し，アセンブルして使用する場合には，次の点に注意してください。

- #asm ブロック（#asm から #endasm で囲まれた部分），および __asm 文中で，シンボルを定義する必要があるときには，?L の文字列で始まるシンボル（たとえば，?L@01，?L@sym など）を使用してください。ただし，このシンボルを外部定義（PUBLIC 宣言）しないでください。また，#asm ブロック，および __asm 文中で，セグメントを定義することはできません。?L の文字列で始まるシンボルを使用しない場合，アセンブル時に致命的エラー（F2114）が出力されます。
- C ソースで extern されている変数を #asm ブロック内で使用している場合，他の C 記述部分で参照がないと EXTRN が生成されず，リンク・エラーとなるため，C で参照されない場合は，#asm ブロック内で EXTRN してください。
- #pragma section 指令でセグメント名を変更する場合，ソース・ファイル名のプライマリ名と同名のセグメント名を指定しないでください。アセンブル時にエラー（F2106）が出力されます。

（4）リンク・ディレクティブ・ファイルの作成について

RL78,78K0RC コンパイラで作成したオブジェクトをリンクする際に，ターゲット・デバイスの ROM/RAM 領域以外の領域を使用する場合，または任意のアドレスに指定してコードやデータを配置させたい場合は，リンク・ディレクティブ・ファイルを作成し，リンク時にリンカ・オプション -d で指定してください。

リンク・ディレクティブ・ファイルの作成方法については，「第 5 章　リンク・ディレクティブ仕様」，および CA78K0R に添付されている lk78k0r.dr（smp78k0r フォルダ以下）を参照してください。

例 ある C ソース・ファイルの初期値なし外部変数（sreg 変数を除く）を外部メモリに配置させたい場合

（a）C ソースの先頭で初期値なし外部変数用セクション名を変更する

```
#pragma section @@DATAL EXTDATA
    :
```

注意 変更されたセグメントの初期化，および ROM 化はスタートアップ・ルーチンを変更して行うようにしてください。

（b）リンク・ディレクティブ・ファイル lk78k0r.dr を作成する

```
memory  EXTRAM  : ( 040000H, 1000H )
merge   EXTDATA : = EXTRAM
```

リンク・ディレクティブ・ファイル作成時には，次の点に注意してください。

- リンク時に，スタック解決用シンボル生成指定オプション -s を使用する場合には，スタック領域をリンク・ディレクティブ・ファイルの memory ディレクティブで確保し，確保したスタック領域名を明示的に指定することをお勧めします。領域名を省略した場合は，スタック領域として RAM 領域内（SFR 領域以外）が使用されます。

例 リンク・ディレクティブ・ファイル lk78k0r.dr に追加した場合

```
memory  EXTRAM  : ( 040000H, 1000H )
memory  STK     : ( 0FB000H, 100H )
merge   EXTDATA : = EXTRAM
```

コマンド・ラインは，次のようになります。

```
C>lk78k0r s0rml.rel prime.rel -bcl0rm.lib -sSTK -dlk78k0r.dr
```

- 定義しているメモリ領域でリンクすると次のようなリンク・エラーが出力されることがあります。

```
RA78K0R error E3206 : Segment 'xxx' can't allocate to memory-ignored.
```

定義しているメモリ領域では，領域不足のために，指摘されたセグメントを配置することができないためです。
対処方法は，大きく分けて次の 3 つの手順になります。

- 配置できないセグメントのサイズを調べる（.map ファイル参照）。
  ただし，エラーで指摘されたセグメントの種類により，次のようにセグメントのサイズを調べる方法が異なります。

- コンパイル時に自動生成されるセグメントのとき
  リンクし作成されたマップ・ファイルによりセグメントのサイズを調べます。
- ユーザが作成したセグメントのとき
  アセンブル・リスト・ファイル（.prn）により配置されなかったセグメントのサイズを調べます。

- 上記で調べたセグメントのサイズをもとに，ディレクティブ・ファイルでセグメントが配置されている領域のサイズを拡大します。

- ディレクティブ・ファイル指定オプション -d を指定してリンクします。

（5）va_start マクロ使用時

関数により第 1 引数のオフセットが異なるため，stdarg.h に定義されている va_start マクロの動作は保証されません。

以下のようにマクロを使い分けてください。

- 第 1 引数を指定する場合は，va_starttop マクロを使用してください。

（6）スタートアップ・ルーチン，ライブラリについて

- スタートアップ・ルーチン，ライブラリは，使用している実行形式ファイル（cc78k0r.exe）と同じバージョンで提供されているものを使用してください。
- 浮動小数点対応 sprintf，vprintf，vsprintf において，"%f"，"%e"，"%E"，"%g"，"%G" 指定の変換結果の精度以下の値を切り捨ててしまいます。また，"%g"，"%G" 指定の変換結果が精度以上であっても "%f" 変換してしまいます。
  浮動小数点対応 sscanf，scanf において，"%f"，"%e"，"%E"，"%g"，"%G" 指定時に 1 文字も有効な文字を読み込まなかった場合 +0 を変換結果とし，" ± " だけの場合± 0 を変換結果とします。

（7）ROM 化を行う場合について

ROM 化とは，初期値あり外部変数などの初期値を ROM に配置しておき，システム実行時に RAM にコピーする処理です。RL78,78K0RC コンパイラでは，デフォルトで ROM 化用にコードを生成します。したがって，リンク時に ROM 化処理を含むスタートアップ・ルーチンとリンクする必要があります。

スモール・モデル，ミディアム・モデル用スタートアップ・ルーチンは，far 領域の ROM 化処理を含んでいません。__far 修飾子などで，変数を far 領域に配置した場合は，ラージ・モデル用のスタートアップ・ルーチンを使用してください。

CA78K0R が提供するスタートアップ・ルーチンには次のものがあり，すべて ROM 化処理を含んでいます。フラッシュ・メモリのセルフ書き換えモードを使用する場合は，「（3）スタートアップ・ルーチンの使い分け」参照してください。

C 標準ライブラリ用の領域を使用しない場合	s0rm.rel，s0rl.rel
C 標準ライブラリ用の領域を使用する場合	s0rml.rel，s0rll.rel

使用例を以下に示します。

なお，-s オプションは，スタック・シンボル（_@STBEG，_@STEND）自動生成オプションです。

```
C>lk78k0r s0rl.rel sample.rel -s -bcl0rxm.lib -bcl0rm.lib -osample.lmf
```

sample.rel	ユーザ・プログラムのオブジェクト・モジュール・ファイル
s0rl.rel	スタートアップ・ルーチン
cl0rxm.lib	乗算器使用ライブラリ
cl0rdm.lib	乗除算器使用ライブラリ
cl0rm.lib	ランタイム・ライブラリ，標準ライブラリ

注意 1. 必ずスタートアップ・ルーチンを最初にリンクしてください。

2. ユーザがライブラリを作成する場合は，CA78K0R が提供するライブラリとは分けて作成し，リンク時に CA78K0R のライブラリよりも前に指定するようにしてください。

3. CA78K0R のライブラリに，ユーザ関数を追加しないでください。

4. 浮動小数点ライブラリ（cl0r*f.lib）を使用する場合は，通常のライブラリ（cl0r*.lib）と両方リンクする必要があります。

例 浮動小数点対応の sprintf，sscanf，printf，scanf，vprintf，vsprintf を使用する場合

```
-bmylib.lib -bcl0rmf.lib -bcl0rm.lib
```

例 浮動小数点未対応の sprintf，sscanf，printf，scanf，vprintf，vsprintf を使用する場合

```
-bmylib.lib -bcl0rm.lib -bcl0rmf.lib
```

（8）プロトタイプ宣言

関数プロトタイプ宣言において，関数の型指定がない場合，エラー（E0301，E0701）となります。

```
f ( void ) ;    /* E0301 : Syntax error */
                /* E0701 : External definition syntax */
```

このような場合は，関数の型を記述してください。

```
int     f ( void ) ;
```

（9）エラー・メッセージ出力

関数外で，行頭のキーワードにスペル・ミスがある場合，エラー行の表示位置がずれたり，不適当なエラーを出す場合があります。

```
extren int      i ;             /* extern のスペルミス。ここでエラーにならない。*/
/* comment */
void            f ( void ) ;
[EOF]                           /* E0712 などのエラー */
```

（10）前処理指令中のコメント記述

前処理指令の記述において，前処理指令の前や途中，関数形式マクロの並びにコメントを記述すると，エラー（E0803，E0814，E0821 など）となります。

```
/* com1 */      #pragma         sfr                     /* E0803 */
/* com2 */      #define         ONE     1               /* E0803 */
#define         /* com3 */      TWO     2               /* E0814 */
#ifdef          /* com4 */      ANSI_C                  /* E0814 */

/* com5 */      #endif
#define         SUB ( p1, /* com6 */ p2 ) p2 = p1       /* E0821 */
```

このような場合は，前処理指令の後にコメントを記述してください。

```
#pragma         sfr                                     /* com1 */
#define         ONE     1                               /* com2 */
#define         TWO     2                               /* com3 */
#ifdef          ANSI_C                                  /* com4 */

#endif                                                  /* com5 */
#define         SUB ( p1, p2 ) p2 = p1                  /* com6 */
```

（11）構造体／共用体／ enum のタグ使用

関数プロトタイプ宣言で，（構造体，共用体，enum の）タグを定義する前に使用すると，（a）の条件を満たす場合はワーニング，（b），（c）の条件を満たす場合はエラーとなります。

（a）引数宣言で，そのタグを使用して，構造体，共用体へのポインタを定義すると，関数の呼び出し時にワーニング（W0411，W0412，W0510）となります。

```
void    func ( int, struct st ) ;

        struct  st {
                char    memb1 ;
                char    memb2 ;
        } st[ ] = {
                { 1, ' a ' }, { 2, ' b ' }
        } ;

void    caller ( void ) {
                        /* W0510 Pointer mismatch */
        func ( sizeof ( st ) / sizeof ( st[0] ), st ) ;
}
```

(b) 戻り値型宣言と引数宣言で，そのタグを使用して，構造体，共用体，enum 型を指定すると，エラー（E0737）となります。

```
                        /* E0737 Undeclared structure/union/enum tag */
void    func1 ( int, struct st ) ;
                        /* E0737 Undeclared structure/union/enum tag */
struct  st      func2 ( int ) ;
struct  st {
        char    memb1 ;
        char    memb2 ;
} ;
```

このような場合は，構造体，共用体，enum のタグの定義を先に行ってください。

(c) 構造体のメンバ宣言で，const，volatile などの型修飾をした自己参照構造体メンバを 2 つ以上指定すると，エラー (E0719) となります。

```
struct  st {
        const   struct  st*     a ;
        const   struct  st*     b ;
};
```

このような場合は，const，volatile などの型修飾をメンバの型宣言のあとに記述してください。

```
struct  st {
        struct  st      const*  a ;
        struct  st      const*  b ;
};
```

(12) 関数内の配列／構造体／共用体の初期化

関数内で，静的変数のアドレス，定数，文字列以外を用いた，配列／構造体／共用体の初期化を行うことができません。

```
void    f ( void ) ;
void    f ( void ) {
        char    *p, *p1, *p2 ;
        char    *ca[3] = { p, p1, p2 } ;   /* エラー ( E0750 ) */
}
```

このような場合は，代入文を記述して，初期化の代わりとしてください。

```
void    f ( void ) ;
void    f ( void ) {
        char    *ca[3] ;
        char    *p, *p1, *p2 ;
        ca[0] = p ;  ca[1] = p1 ; ca[2] = p2 ;
}
```

（13）構造体を返す関数

関数が構造体そのものを返す場合，戻り値を返す処理中に割り込みが発生し，割り込み処理中に同じ関数の呼び出しがあると，割り込み処理終了後に戻り値が不正となります。

```
struct  str {
        char    c ;
        int     i ;
        long    l ;
} st ;

struct  str     func ( ) {
        /* 割り込み発生 */
          :
}

void main ( ) {
        st = func ( ) ; /* 割り込み発生 */
}
```

上記の処理中，割り込み先で func 関数が呼ばれた場合，st が破壊される可能性があります。

（14）メモリ初期化疑似命令

メモリ初期化疑似命令 DB，DW，DG をデータ・セグメント（DSEG）で記述した場合，オブジェクト・コードは出力されますが，オブジェクト・コンバータでワーニング（W4301）になります。これは，ROM 領域（コード領域）以外のアドレスにコードが存在するためです。

この状態で ROM コード発注（アクロス処理，テープ・アウトと呼ばれている作業です）を行うと，エラーになります。

（15）メモリ・ディレクティブ

各デバイスのデフォルトのメモリ領域名は，消去できません。

使用しないデフォルトのメモリ領域名のサイズは，0 にしてください。

ただし，セグメントによってはデフォルトの領域に割り付けられるものもあるため，領域名を変更する際には注意してください。

なお，デフォルトのメモリ領域名については，各デバイスのユーザーズ・マニュアルを参照してください。

（16）セグメント名

セグメント名を記述する場合，ソース・ファイル名のプライマリ名と同名のセグメント名を記述しないでください。アセンブル時に，アボート・エラー F2106 になります。

（17）SFR 名の EQU 定義

EQU 疑似命令のオペランドには SFR 名を指定することができますが，saddr 領域外の SFR の名前を PUBLIC に指定した場合，アセンブル・エラーとなります。

（18）セクションの開始アドレス指定

#pragma section 指令で開始アドレスを指定したセクションのサイズは，常に偶数となります。

（19）ビット・フィールド

signed 型のビット・フィールドを，符号無しのビット・フィールドとして処理します。

（20）入出力関数の標準ライブラリの出力変換

関数 printf，sprintf，vprintf，vsprintf の出力変換で，以下のような場合に動作が不正となります。

（a）変換指定子“d, i, o, u, x，および X”に対して，“.2”のように精度を指定した場合に，0 フラグを無視しません。

例

```
#include <stdio.h>
void func ( )
{
        printf("%04.2d\n", 77);
}
```

（不正な動作）“0077”となります

（正しい動作）“77”となります

（b）変換指定子“g,G”に対して，指定した精度 +1 を精度とします。

例

```
#include <stdio.h>
void func ( )
{
        printf("%.2g", 12.3456789);
}
```

（不正な動作）“12.3”となります

（正しい動作）“12”となります

（21）int/short 型の最小値 -32768 のサイズ

int/short 型の最小値 -32768 のサイズが 4 となります。（-32767-1）と記述してください。

例

```
int     x;
void func ( )
{
        x = sizeof(-32768);
}
```

（不正な動作）x の値が 4 となります

（正しい動作）x の値が 2 となります

（22）関数定義の識別子の型

関数定義の識別子の型に対して実引数拡張をしていないため，仮引数型と関数定義の識別子の型と適合せずに，E0747 エラーとなります。

仮引数型と関数定義の識別子の型を合わせてください。

例

```
int     fn_char ( int );
int     fn_char ( c )
char    c;
{
        return  98;
}
```

（23）関数定義の識別子並び

関数定義の識別子並びで，宣言していない仮引数を int 型とせず E0706 エラーとなります。

関数定義の仮引数はすべて宣言してください。

例

```
void func ( x1, x2, f, x3, lp, fp )
int     (*fp)( );
long    *lp;
float   f;
{
        :
}
```

（24）# 演算子

以下のいずれかの条件で正しく展開できません。

（a）（条件１）# 演算子で「""」を正しく展開できずに，コンパイル時にエラーとなります。

例 条件 1 の例

```
#include <string.h>
#define str (  a ) ( # a )
int     x;
void func ( )
{
        if (strcmp(str('"'), "'\"'") == 0) x++;
}
```

（不正な動作）コンパイル時にエラーとなります

（正しい動作）if (strcmp(("'\"") , ("'\"") == 0) x++; となります

（b）（条件２）# 演算子とネストを含むマクロを正しく展開できません。

例 条件 2 の例

```
#define str ( a ) #a
#define xstr ( a ) str ( a )
#define EXP 1
char    *p;
void func ( )
{
        p = xstr(12EEXP);
}
```

（不正な動作）“p = ("12E1");” となります。

（正しい動作）“p = ("12EEXP") ;” となります。

（25）前処理指令 #line のデバッグ情報

前処理指令 #line を記述した時に，アセンブラ・ソース中のデバッグ情報が不正となります。また，このアセンブラ・ソースをアセンブルするとエラー（E2201）となります。

例

```
#include <stdio.h>
void main ( void )
{
        int     a;
        #line 1 "test_line"
        a = 3;
        printf("__FILE__ = %s, __LINE__ = %d\n",__FILE__,__LINE__);
}
```

以下のいずれかの方法で回避してください。

- オブジェクト・モジュール・ファイルを使用する。
- デバッグ情報を出力せずにアセンブラ・ソースを使用する。

（26）変数／関数情報ファイル生成ツール

callt 関数になることで，最適化のサブルーチンのパターンの対象にならず，コードが増える場合があります。変数／関数情報ファイルから，callt 関数になる対象の関数をコメントにしてください。

（27）RTOS 用システムコール

C ソース中に #pragma rtos_task を記述していない場合，ext_tsk を RTOS 用システムコールと認識しないため，RTOS 用割り込みハンドラから ext_tsk を呼び出しても，エラー「E0778: Cannot call ext_tsk in interrupt function」が出力されません。

以下のいずれかの方法で回避してください。

- C ソース中に #pragma rtos_task を記述して，タスクを使用することを明示してください。
- RTOS 割り込みハンドラからは，ext_tsk の呼び出しを記述しないでください。

（28）変数／関数情報ファイルのプロトタイプ宣言

-ma オプション指定時，仮引数の型宣言がない関数を呼び出し，引数に「変数／関数情報ファイルで callt 配置を指定した関数のアドレス」を記述すると，関数インタフェースが整合せずに不正動作となる場合があります。

例

```
int     func_c ( )              /* callt in .vfi */
{
        return 0;
}
void func ( )
{
        func2 ( func_c ) ;      /* W0553 */
}
int     func2 ( int ( *p ) ( void ) )
{
        return 1;
}

・変数／関数情報ファイル
[callt]
        func_c,,,,
```

上記条件を満たす場合は，コンパイラがワーニング W0553 を出力します。関数呼び出しのプロトタイプ宣言を，仮引数の型宣言を含めて記述してください。または，関数の引数に記述した関数に対して，変数／関数情報ファイルの callt 指定をコメントアウトしてください。

（29）変数／関数情報ファイル使用時の #pragma section 指令

-ma オプション指定時，#pragma section 指令を AT 開始アドレス付きで指定したセクション中の関数や変数を，変数／関数情報ファイルで callt テーブル，saddr 領域に配置すると，不正動作する場合があります。

例

```
#pragma section @@DATA @FCDATA AT 0FCF00H
#define dni1 ( * ( int * ) 0xfcf00 )
int     __near ni1;                             /* sreg in .vfi */
__sreg  int     x1, x2;
void func ( )
{
        x1 = ni1;
        x2 = dni1;
}
void main ( )
{
        ni1 = 0x10;
        func ( );
}

・変数／関数情報ファイル
[sreg]
        ni1,,,
```

このプログラムは，変数 x1，x2 の値が両方とも 0x10 となることを期待していますが，変数／関数情報ファイルによって，変数 ni1 が saddr 領域（0xffe20 ～）に割り当たると，変数 x1 の値は，ni1 の値である 0x10，変数 x2 の値は，0xfcf00 番地の値となり，意図した動作になりません。

#pragma section 指令を AT 開始アドレス付きで指定したセクション中の関数や変数は，VF78K0R では sreg，callt 指定の対象外としています。変数／関数情報ファイルを修正する際には，上記関数や変数を callt テーブル，saddr 領域に配置する指定をしないでください。

（30）変数／関数情報ファイル使用時のワーニング出力

-ma オプション指定時，変数／関数情報ファイルにより callt テーブルに配置した関数のアドレスを扱う際，ワーニングが出力される場合があります。

例 ミディアム／ラージ・モデルの場合

```
void f1 ( void ( *fp ) ( ) )
{
}
void f2 ( void )
{
}
void ( *fp1 ) ( void ) ;
void ( *fp2 ) ( void ) ;
void func ( void )
{
        f1 ( f2 ) ;                     /* W0510: Pointer mismatch in function */
        f1 ( ( void ( * ) ( ) ) f2 ) ;  /* キャストすれば OK */
        fp1 = f2;                       /* W0416: Illegal type and size (far/near)
                                           pointer combination */
        fp2 = ( void ( * ) ( ) ) f2;    /* キャストすれば OK */
}

・変数／関数情報ファイル
[callt]
        f2,,,,
```

動作上は問題ありません。

ワーニング出力を抑制したい場合は，関数ポインタを扱う処理でキャストしてください。

（31）-ma オプション指定

-ma オプションで指定するファイルのフォルダ名またはファイル名に ','（カンマ）が含まれている場合，コンパイラはカンマを区切り子と判断するため適切にファイルを認識できません。

カンマをフォルダ名およびファイル名に含めないでください。

（32）strtod 関数ライブラリ

strtod 関数に渡す文字列に，指数部が 3 桁以上の浮動小数点数を渡すと，オーバーフローの処理になりません。

例

```
char    *endptr;
double  result;
result = strtod ( "-5E+2000", &endptr );        /* 指数部を勝手にキャストする */
```

float,double 型ともに，CA78K0R で記述できる範囲は 1.17549435E-38F ～ 3.40282347E+38F ですが，CA78K0R の strtod は指数部を 2 桁しか読み込んでいないため，上記の記述だと "-5E+20" と読み込んでしまいます。

strtod 関数に渡す文字列の指数部は，表現できる値を記述してください。

（33）関数ポインタ±オフセットのアドレスの最上位

関数ポインタをデータ・ポインタにキャストし，オフセットを付加して 64Kbyte 境界をまたいでアクセスした場合に，最上位アドレスが正しく設定されない場合があります。

例

```
extern void vg ( ), void ng ( );
void func ( void )
{
        :
}
void main ( )
{
        unsigned long   *p = ( unsigned long * ) func - 1;
        if ( p == ( ( unsigned long * ) func -1 ) )
                vg ( );
        else
                ng ( );
}
```

データは 64Kbyte 境界をまたいで配置できないため，64Kbyte 境界をまたいでデータ参照ができません。

（34）_rcopy 関数

スタートアップ・ルーチンで呼び出している hdwinit 関数の中で，_rcopy 関数を呼び出さないようにしてください。

正常動作しない場合があります。

（35）@EBASE セグメント

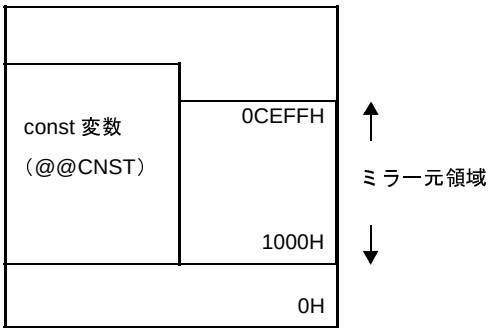
@EBASE セグメントに対して，リンカで E3206 エラーが出た場合は，ディレクティブ・ファイルで，@EBASE セグメントをフラッシュ側の ROM 領域に配置するように指定してください。

（36）ミラー元領域への const 変数の配置

const 変数等用の @@CNST セクションを以下のようにディレクティブで配置指定した場合，@@CNST セクションがミラー元領域をオーバーしても，ディレクティブに従って配置します。オーバーした領域の変数には far 修飾子を付けてください。

```
----------*.dr----------------------
MEMORY ROM       :(0H, 1000H)
MEMORY ROMX      :(1000H, 3F000H)
MERGE @@CNST     := ROMX
```

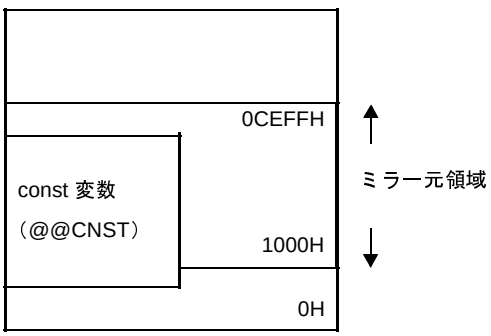
UPD78F166 の例 (MAA=0)



下図のように const 変数を配置する場合，ミラー元領域を外れた const 変数に対しては far 修飾子を付けてください。また，以下のように @@CNSTL をディレクティブ指定してください。

```
----------*.dr----------------------
MERGE @@CNSTL    : AT (0F00H)
```

UPD78F166 の例 (MAA=0)



（37）オンチップ・デバッグ領域の配置

オンチップ・デバッグ領域のセグメントが配置できない場合，以下のような配置エラーが出力されます。

```
RA78K0R error E3212: Default segment can't allocate to memory - ignored
                     Segment '?CSEGOB0' at C0H-4H
```

```
RA78K0R error E3212: Default segment can't allocate to memory - ignored
                     Segment '?OCDSTAD' at CEH-AH
```

```
RA78K0R error E3212: Default segment can't allocate to memory - ignored
                     Segment '??OCDROM' at FC00H-200H
```

オンチップ・デバッグ機能を使用する場合，オンチップ・デバッグ用のモニタ領域が配置できるように，リンク・ディレクティブにてオンチップ・デバッグ用の領域を空けてください。

（38）ブート - フラッシュの再リンク機能

再リンク機能を使用する際には，入力するロード・モジュール・ファイル（LMF）内のセグメントに対してマージ・ディレクティブ指定は行わないでください。

（39）#pragma section 指令

以下の２つ条件を満たすときに，エラーになる場合があります。

- #pragma section 指令で，@@CNST あるいは @ECNST セクションをアドレス指定している。
- 配列の参照において，オフセット部分が「符号無し変数ー定数」の 記述であり，「配列の先頭アドレス」から定数分を引いた値がミラー領域の範囲外である。

例

```
unsigned char   uc;
unsigned int    ui = 0x3;

#pragma section @@CNST CNEAR AT 1000H
const unsigned char cuc = 10, cuc2 = 20, uca[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};


void func()
{
          uc = *((uca - 3) + ui);          /* ERROR */
}
```

以下の様に式を分割してください。

```
unsigned char   uc, *tmp;

/*  uc = *((uca - 3) + ui);     ERROR */

        tmp = uca - 3;
        uc = *(tmp + ui);
```

（40）データ・フラッシュ領域

（a）データ・フラッシュ領域へのデータの配置方法

データ・フラッシュ領域へのデータの配置方法は以下の様にしてください。

例 R5F100LE の例

アセンブラ・ソース

```
PUBLIC  data_flash

FLDAT   DSEG    AT 0F1000H
data_flash:     DB 11H, 22H, 33H, 44H

END
```

リンク・ディレクティブ・ファイル

```
MEMORY DATFL    (0F1000H, 1000H)
```

（b）データ・フラッシュ領域に関する注意事項

CA78K0R はデータ・フラッシュ領域に対応したコード出力を行っていません。

アセンブラによる記述でアクセスしてください。

C ソースで記述する場合，8 ビット・データとしてアクセスしてください。

（41）外部バス・インタフェース

外部バス・インタフェースを搭載したデバイスの外部メモリには対応していません。

8 ビット・バス・モードで使用する場合は，アセンブラによる記述でアクセスしてください。

C ソースで記述する場合，8 ビット・データとしてアクセスしてください。また，標準ライブラリを使用してアクセスすることも出来ません。

（42）再リンク

再リンク機能で「変数／関数配置オプションの設定」を有効にした場合，フラッシュ領域のプロジェクトで，意図しない関数が callt 関数の対象になりリンク時にエラーになる場合があります。

この場合は，「変数／関数配置オプションの設定」を無効にしてください。

（43）乗除算器／積和演算器使用ライブラリ

乗算ライブラリはマイコン・リセット状態での MDUC レジスタの状態で演算を行います。

乗算以外の除算，および積和演算ライブラリは，ライブラリ，または組み込み関数中で，MDUC レジスタをそれぞれのモードに設定し，演算終了時にはデフォルト（マイコン・リセット状態）の乗算モードに戻します。

したがって，プログラム中で MDUC レジスタを変更することは推奨していません。

MDUC レジスタを変更する場合は，乗算前に必ず MDUC レジスタを 00H（デフォルト状態）にしてください。

MDUC レジスタを変更する場合は，割り込みを禁止した上で，変更，およびデフォルト状態への復帰を行うようにしてください。

（44）空ソースのコンパイル

以下のソースをコンパイルするとエラー（E0301，E0701）となります。

- 空ソース
- コメントのみのソース
- 前処理指令により，すべてのコードが無効となったソース

このような場合は，ダミーのプロトタイプ宣言を記述してください。

```
extern void     dummy(void);
```

（45）## 演算子

演算子を使用している場合に，正しく展開できずに，コンパイル時にエラーとなる場合があります。

例

```
#define NUM(a)  0x##a
```

この例の場合は，0x##a を 0x0##a に変えることでエラーを回避できます。

付録 A　ROM 化プロセッサ

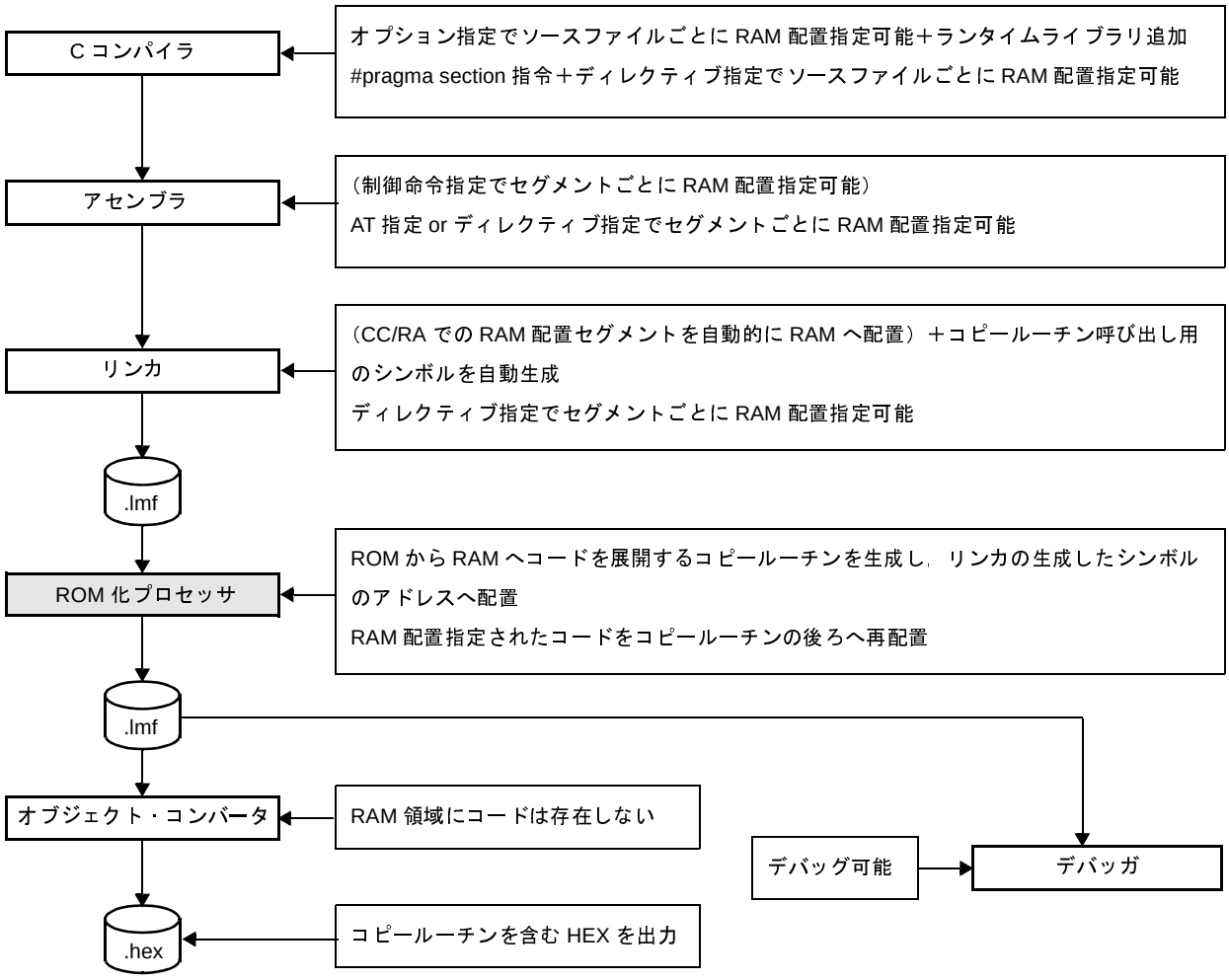
ここでは，ROM 化プロセッサを使用した ROM 化について説明します。

A. 1　概　　要

ROM 化プロセッサは，プログラムの一部を RAM 領域に配置し実行するために，C ソースを変更することなく RAM 配置指定を可能にします。

全体の流れを次に示します。

図 A―1　ROM 化用オブジェクトの作成



A. 2　ROM 化用ロード・モジュールの作成手順

ここでは，ROM 化用ロード・モジュールの作成手順を説明します。

（1）コピー関数の呼び出し

プログラム上で，

```
int     _rcopy(int) ;
```

と，プロトタイプ宣言した上で，

```
_rcopy ( n ) ;
```

とコピー関数を呼び出すことで，ROM 化したセグメント番号 *n* のセグメントが RAM に展開されます。

（2）ROM 化対象セグメント／コピー関数の配置指定

ROM 化対象セグメント[注1]もコピー関数[注2]を配置するアドレスも自動的に決定されます。

リンク時に，ROM 化対象セグメントがあれば，自動的にコピー関数を配置するアドレスを決定し，そのアドレスにシンボル“__rcopy”が定義されます。

（a）コピー関数を配置するアドレスを直接指定

リンク時に，ROM 化対象セグメントがあり，コピー関数のアドレス指定オプション（-rc）にて指定されたアドレスに ROM 化処理に必要な空き領域があれば，そのアドレスにシンボル“__rcopy”が定義されます。

（b）ROM 化対象を直接指定

リンク時に，ROM 化領域指定オプション（-ra）で指定した範囲に ROM 化対象セグメントがあれば，自動的にコピー関数を配置するアドレスを決定し，そのアドレスにシンボル“__rcopy”が定義されます。

注 1. 対象領域はデバイスファイルにて定義された内部 RAM 領域です。

2. コピー関数は，_rcopy.rel ファイルから読み込み，出力ファイルにリンクされます。

付録 B　ウインドウ・リファレンス

ここでは，コーディングに関するウインドウ／パネル／ダイアログについて説明します。

B. 1　説　　明

以下に，コーディングに関するウインドウ／パネル／ダイアログの一覧を示します。

表 B—1　ウインドウ／パネル／ダイアログ一覧

ウインドウ／パネル／ダイアログ名	機能概要
エディタ パネル	ファイルの表示／編集
ファイル・エンコードの選択 ダイアログ	ファイル・エンコードの選択
ブックマーク ダイアログ	ブックマークの表示／削除
指定行へのジャンプ ダイアログ	指定した行にキャレットを移動
Print Preview ウインドウ	印刷する前のソース・ファイルのプレビュー
ファイルを開く ダイアログ	オープンするファイルの選択

エディタ パネル

ファイルの表示／編集を行います。

自動的にファイルのエンコードと改行コードを判別してオープンし，保存の際は元のフォーマットで保存します。ただし，ファイル・エンコードの選択 ダイアログによりエンコードを指定してオープンすることができます。また，ファイルの保存設定 ダイアログでエンコードと改行コードを指定した場合は，それに従って保存します。

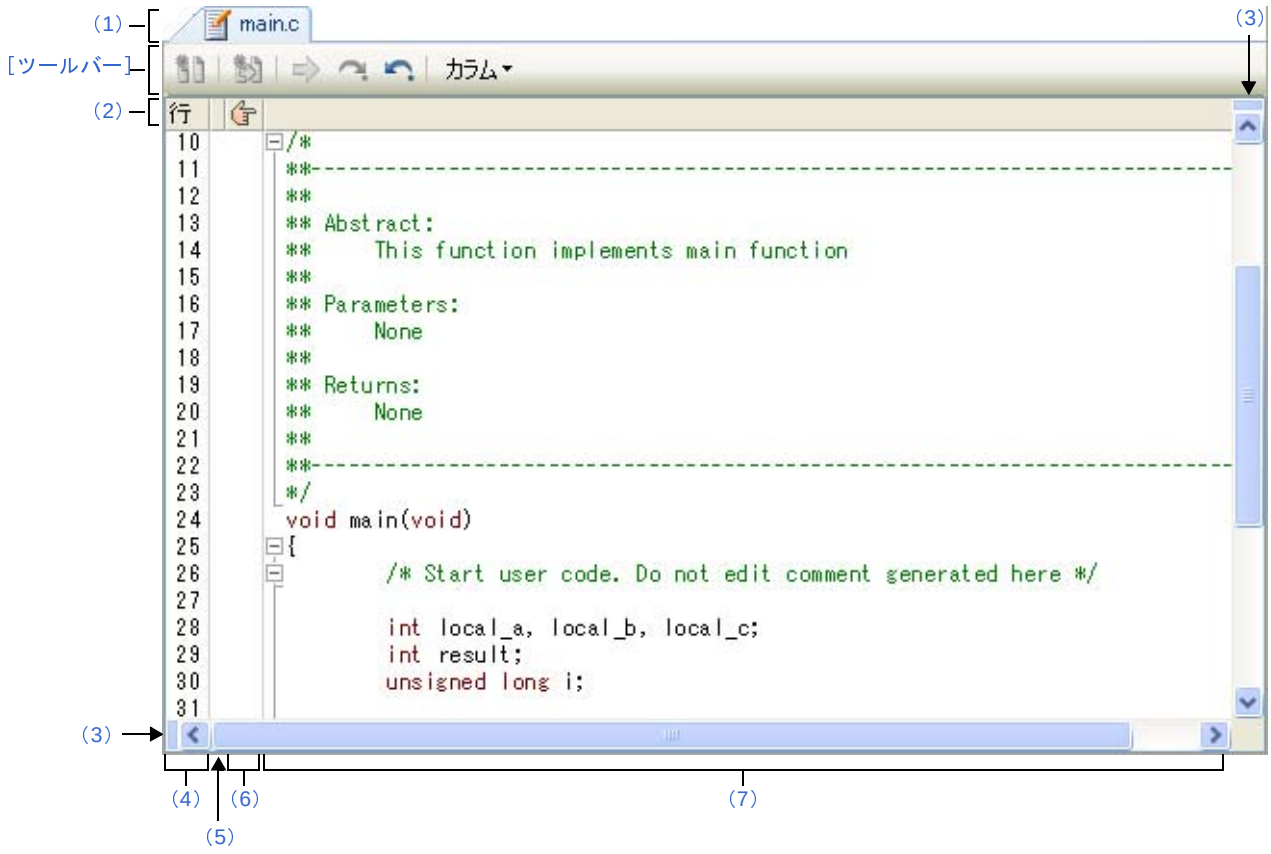
本パネルは複数オープンすることができます（最大個数：100 個）。

注意 1.　プロジェクトをクローズすると，該当プロジェクト内で登録されているファイルをオープンしているすべてのエディタ パネルがクローズします。

2.　プロジェクトからファイルの登録を外すと，該当ファイルをオープンしているエディタ パネルがクローズします。

備考　ツールバーの 100% ，または［Ctrl］キーを押下しながらマウス・ホイールを前後方に動かすことにより，本パネルの表示を拡大／縮小することができます。

図 B―1　エディタ パネル



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［ツールバー］
- ［[ファイル］メニュー（エディタ パネル専用部分）］
- ［[編集］メニュー（エディタ パネル専用部分）］
- ［[ウインドウ］メニュー（エディタ パネル専用部分）］
- ［コンテキスト・メニュー］

［オープン方法］

- プロジェクト・ツリー パネルにおいて，ファイルをダブルクリック
- プロジェクト・ツリー パネルにおいて，ファイルを選択したのち，コンテキスト・メニューの［開く］を選択
- プロジェクト・ツリー パネルにおいて，ファイルを選択したのち，コンテキスト・メニューの［内部エディタで開く ...］を選択
- プロジェクト・ツリー パネルにおいて，コンテキスト・メニューの［追加］→［新しいファイルを追加］を選択したのち，テキスト・ファイル／ソース・ファイルを作成

［各エリアの説明］

（1）タイトルバー

オープンしているテキスト・ファイル／ソース・ファイルのファイル名を表示します。

なお，ファイル名の末尾に表示するマークの意味は次のとおりです。

マーク	意味
*	テキスト・ファイルをオープンしたのち，編集している場合に表示します。
(編集不可)	書き込み禁止状態のテキスト・ファイルをオープンしている場合に表示します。

（2）カラム・ヘッダ

エディタ パネルの各列のタイトルを表示します（マウス・カーソルを重ねることによりタイトル名をホップ・アップ表示します）。

表示	タイトル名	説明
行	行	行番号を表示します（「(4) 行番号エリア」参照）。
（表示なし）	選択	編集状況に応じた色表示を行います（「(5) 選択エリア」参照）。
	メイン	ブックマークを表示します（「(6) メイン エリア」参照）。

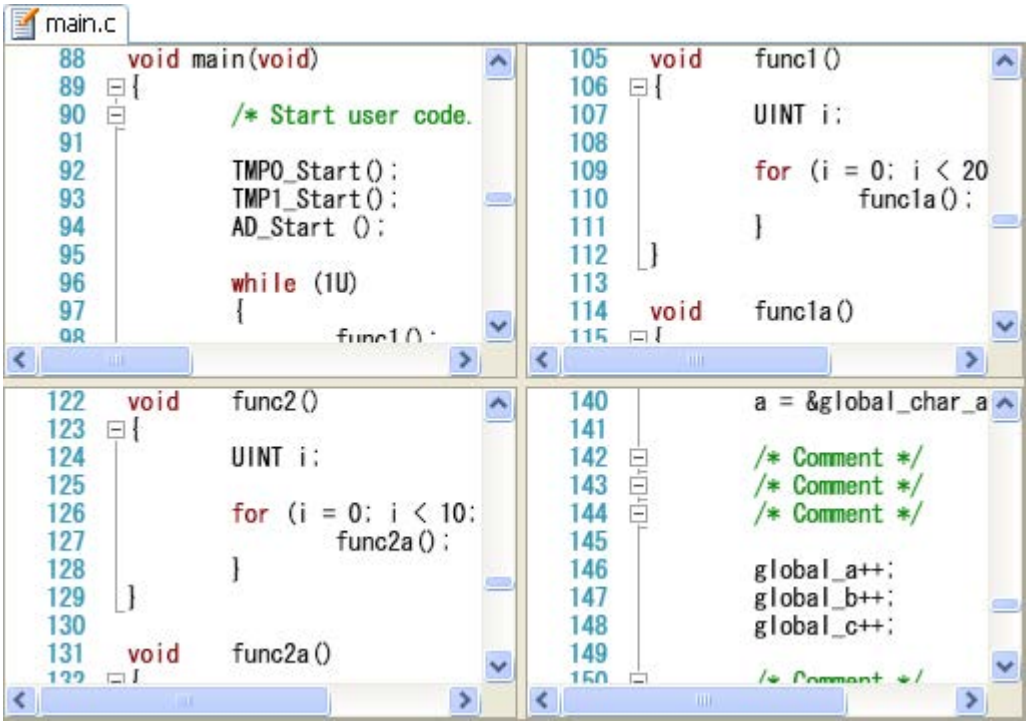
備考　カラム・ヘッダは，ツールバーの設定により，表示／非表示を切り替えることができます。

(3) 分割バー

縦と横の分割バーを使うことにより，エディタ パネルを分割して表示することができます。分割の上限は，縦 2 分割，横 2 分割までです。

- 分割表示するには，分割バーを下方向／右方向の目的の位置までドラッグします。または，分割バーをダブルクリックします。
- 分割表示を解除するには，分割バーをダブルクリックします。

図 B—2 エディタ パネル（縦横 2 分割表示した場合）

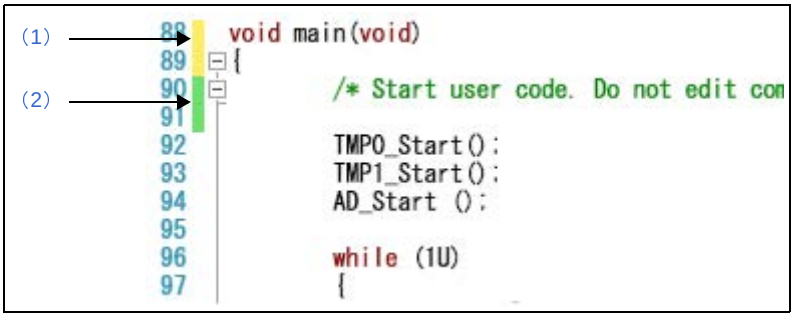


(4) 行番号エリア

表示しているテキスト・ファイル／ソース・ファイルの行番号を表示します。

(5) 選択エリア

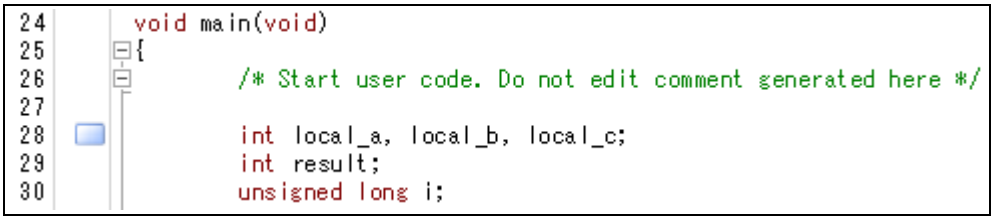
行の編集状態に応じた色表示を次のように行います。



(1)		新規または変更したが保存されていない。
(2)		新規または変更後，保存済み。 パネルをクローズしたのち，再度該当ソース・ファイルを表示するとこのマークは消失します。

(6) メイン エリア

登録したブックマーク（ ）を表示します。



(7) 文字列エリア

テキスト・ファイル／ソース・ファイルの文字列の表示／編集を行います。

本エリアは，次の機能を備えています。

(a) 文字列の編集

キーボードより，IME などの日本語入力システムを使用した文字列を入力することができます。

また，編集機能を充実させるための様々なショートカットキーを使用することができます。

備考 オプション ダイアログの設定により，次の項目をカスタマイズすることができます。

- 表示フォント
- タブ幅
- 空白記号の表示／非表示
- 予約語／コメントの色分け

(b) コードのアウトライン表示

ソース・コード・ブロックの展開／折りたたみ表示を行い，現在編集中，またはデバッグ中のコード領域に集中して作業することができます。使用できるファイルの種類は，C ソース・ファイル，および .h ファイルです。

展開／折りたたみを行うには，それぞれ文字列エリアの左にあるプラス／マイナス記号をクリックします。

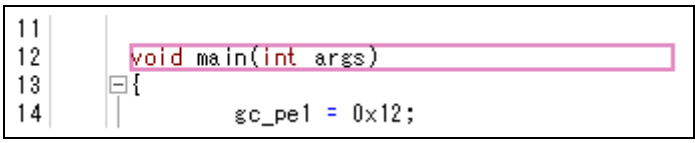
展開／折りたたみ可能なソース・コード・ブロックの種類を次に示します。

左中かっこと右中かっこ（“{” と “}”）	{ ... }
複数行のコメント（“/*” と “*/”）	/** /
プリプロセッサ文（“if”，“elif”，“else”，“endif”）	#if [Preprocessor block] #elif [Preprocessor block] #else [Preprocessor block] #endif

(c) カレント行の強調表示

オプション ダイアログの［全般 - テキスト・エディタ］カテゴリ内の［カレント行のハイライト表示］チェック・ボックスをチェックすることにより，現在キャレットのある行を四角い枠で強調表示します（枠の色は同ダイアログの［全般 - フォントと色］カテゴリの強調色に依存）。

図 B―3　カレント行の強調表示



(d) 括弧の強調表示

キャレット位置の括弧と，それに対応する括弧を強調表示します。

強調表示の対象となる括弧は次のとおりです。

ファイルの種類	対応する括弧
C 言語ファイル Python 言語ファイル	(と)，{ と }，[と]
HTML 言語ファイル XML 言語ファイル	< と >

備考　対応する括弧を表示する際に，コメント内の括弧，および文字定数／文字列／文字列定数内の括弧を考慮しません。そのため，これらが存在する場合では，実際に対応する括弧とは異なる括弧が強調表示されることがあります。

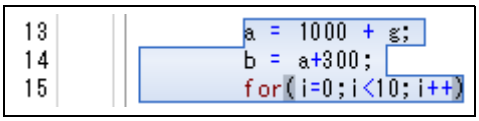
(e) 複数行選択とブロック選択

次の操作により，複数行選択，またはブロック選択を行うことができます。

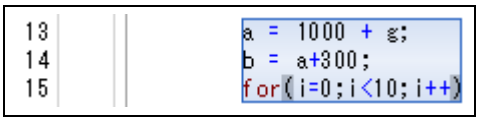
- 複数行選択を行う場合：
  - 左マウス・ボタンでドラッグ
  - ［Shift］キーを押下しながら，［←］／［→］／［↑］／［↓］ボタンを押下
- ブロック選択を行う場合：
  - ［Alt］キーを押下しながら，左マウス・ボタンでドラッグ
  - ［Alt］+［Shift］キーを押下しながら，［←］／［→］／［↑］／［↓］ボタンを押下

図 B―4　複数行選択とブロック選択

【複数行選択】



【ブロック選択】



注意 ブックマーク情報は，選択内容に含まれません。

備考 選択した内容は，［編集］メニューの［切り取り］／［コピー］／［貼り付け］／［削除］の対象となります。

(f) 関数へジャンプ

選択している文字列，またはキャレット位置の単語を関数名と判断し，該当する関数へジャンプすることができます。

ソース・テキスト上で，対象関数にキャレットを移動したのち，コンテキスト・メニューの［関数へジャンプ］を選択してください。

なお，この機能は，次の条件を満たしている場合のみ有効となります。

- アクティブ・プロジェクトの種類が“アプリケーション”である。
- 対象がアクティブ・プロジェクト内の関数である（デバッグ・ツールと切断している場合は，スタティック関数へのジャンプはできません）。
- シンボル情報を持つファイルが［ダウンロードするファイル］に指定されている（ロード・モジュール・ファイル以外の場合，シンボル情報をダウンロードする設定が必要です）。
  ただし，デバッグ・ツールと切断している場合は，［ダウンロードするファイル］の１番目に指定されている。

(g) タグ・ジャンプ

現在キャレットのある行にファイル名／行／桁の情報がある場合，［表示］メニュー→［タグ・ジャンプ］，またはコンテキスト・メニューの［タグ・ジャンプ］を選択することにより，該当ファイルを新たなエディタ パネルにオープンし，該当行／該当桁へジャンプすることができます（該当ファイルがすでにオープンしている場合は，そのエディタ パネルにジャンプ）。

(h) ブックマークの登録

ブックマーク・ツールバーの  ボタンをクリック，またはこのエリアのコンテキスト・メニューの［ブックマーク］→［ブックマークの挿入 / 削除］を選択することにより，現在キャレットのある行にブックマークを登録することができます。

(i) ファイルの監視機能

CubeSuite+ 以外によって，現在表示しているファイルの内容が変更（リネーム／削除を含む）された場合，ファイルを更新するか否かのメッセージを表示し，どちらかを選択することができます。

[ツールバー]

	このパネルの表示モードとして，通常表示モード（デフォルト）と混合表示モードを切り替えます。 ただし，デバッグ・ツールと接続中で，かつダウンロードしたソース・ファイルを表示している場合のみ有効となります。
	ステップ実行を行う際の単位として，ソース・レベル（デフォルト）と命令レベルを切り替えます。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合のみ有効となります。
	現在の PC 位置を表示します。 ただし，デバッグ・ツールと接続中の場合のみ有効となります。
	[コンテキスト・メニュー] → [ジャンプ前の位置へ戻る] を実行する前の位置へ進みます。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合は無効となります。
	[コンテキスト・メニュー] → [関数へジャンプ] を実行する前の位置へ戻ります。 ただし，デバッグ・ツールと接続中で，かつ混合表示モードの場合は無効となります。
カラム	このパネルで表示するカラム，またはマークの表示／非表示を切り替える次の項目を表示します。チェックを外すことにより非表示となります（デフォルトではすべての項目がチェックされています）。 なお，この設定は，すべてのエディタ パネルに反映します。
行	行番号エリアの表示／非表示を切替えます。
選択	選択エリアの表示／非表示を切替えます。
メイン	メイン エリアの表示／非表示を切替えます。
カラム・ヘッダ	カラム・ヘッダの表示／非表示を切替えます。

[[ファイル] メニュー（エディタ パネル専用部分）]

エディタ パネル専用の [ファイル] メニューは次のとおりです（その他の項目は共通）。

*ファイル名*を閉じる	現在編集しているエディタ パネルをクローズします。 なお，パネルの内容を保存していない場合は，確認メッセージを表示します。
*ファイル名*を保存	現在編集しているエディタ パネルの内容を上書き保存します。 なお，ファイルを一度も保存していない，またはファイルが書き込み禁止の場合は，[名前を付けて *ファイル名*を保存 ...] の選択と同等の動作となります。
名前を付けて *ファイル名*を保存 ...	現在編集しているエディタ パネルの内容を新規保存するために，名前を付けて保存 ダイアログをオープンします。
*ファイル名*の保存設定 ...	現在編集しているエディタ パネルでオープンしているファイルのエンコードと改行コードを変更するために，ファイルの保存設定 ダイアログをオープンします。
印刷 ...	現在編集しているエディタ パネルの内容を印刷するために，Windows の印刷用 ダイアログをオープンします。
印刷プレビュー	印刷するファイル内容のプレビューを行うために，Print Preview ウインドウをオープンします。

[[編集]メニュー（エディタ パネル専用部分）]

エディタ パネル専用の［編集］メニューは次のとおりです（その他の項目はすべて無効）。

元に戻す	前回行った操作をキャンセルし，文字とキャレット位置を元に戻します（最大 100 回まで）。
やり直し	前回行った［元に戻す］の操作をキャンセルし，文字とキャレット位置を元に戻します。
切り取り	選択範囲の文字列を切り取り，クリップ・ボードにコピーします。 何も選択されていない場合は，その行を切り取ります。
コピー	選択範囲の文字列をクリップ・ボードにコピーします。 何も選択されていない場合は，その行をコピーします。
貼り付け	クリップ・ボードにコピーしている文字列をキャレット位置に，挿入モードの場合は挿入し，上書きモードの場合は上書きします。 ただし，クリップ・ボードの内容を文字列として認識できない場合は無効となります。 また，現在のソース・ファイルのモードはステータスバーに表示されます。
削除	キャレット位置の文字を１文字削除します。 ただし，範囲選択している場合は，選択しているの文字列を削除します。
すべて選択	現在編集中のテキストの先頭から最終までを選択状態にします。
検索 ...	検索・置換 ダイアログを［クイック検索］タブが選択状態でオープンします。
置換 ...	検索・置換 ダイアログを［クイック置換］タブが選択状態でオープンします。
移動 ...	指定した行へキャレットを移動するため，指定行へのジャンプ ダイアログ をオープンします。
ブックマーク	ブックマークに関するカスケード・メニューを表示します。
ブックマークの挿入 / 削除	キャレット位置の行にブックマークを登録します。 すでにブックマークを登録している場合は，そのブックマークを削除します。
次のブックマーク	アクティブなエディタ パネルにおいて，次に登録したブックマーク位置にキャレットを移動します。 最後のブックマークへ移動後は，最初に登録したブックマークの行へ移動します。
前のブックマーク	アクティブなエディタ パネルにおいて，前に登録したブックマーク位置にキャレットを移動します。 最初のブックマークへ移動後は，最後に登録したブックマークの行へ移動します。
すべてのブックマークの削除	アクティブなエディタ パネルにおいて，登録しているブックマークをすべて削除します。
ブックマークをリスト表示する ...	登録しているブックマークをリスト表示するブックマーク ダイアログをオープンします。
コードのアウトライン	ソース・ファイルのコードの展開／折りたたみ表示を行うためのカスケード・メニューを表示します（「(b) コードのアウトライン表示」参照）。
定義を折りたたむ	関数定義など，実装ブロックとして登録されているすべてのノードを折りたたみます。
アウトラインを切り替える	折りたたまれた部分で，カーソルが置かれている最も内側のアウトライン部分の現在の状態を切り替えます。
すべてのアウトラインを切り替える	すべてのノードの状態を切り替え，すべて同じ状態（展開または折りたたみ）に設定します。折りたたまれているノードと展開されたノードが混在している場合，すべてを展開します。

アウトラインを中止する	コードのアウトラインを中止します。現在の ソース・ファイルからすべてのアウトライン情報を削除します。
自動アウトラインを開始する	コードの自動アウトラインを開始します。サポートしているソース・ファイルのアウトライン情報を自動的に表示します。
高度な設定	エディタ パネルに関する高度な操作を行うためのカスケード・メニューを表示します。
行インデントを増やす	現在カーソルのある行のインデントをタブ１個分増やします。
行インデントを減らす	現在カーソルのある行のインデントをタブ１個分減らします。
行コメントを削除する	現在カーソルのある行の先頭から，言語（C++ など）に応じた行コメントの区切り記号の最初のセットを削除します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
行コメントを付ける	現在カーソルのある行の先頭に，言語（C++ など）に応じた行コメントの区切り記号を設定します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
タブをスペースに変換する	現在カーソルのある行のすべてのタブをスペースに変換します。
スペースをタブに変換する	現在カーソルのある行の連続したスペースの一組をタブに変換します。ただし，そのスペースの各組がタブ１個以上の幅に等しい場合に限ります。
選択行をタブ化する	現在の行をタブ化します。行の先頭にある（テキストの前の）すべてのスペースを可能な限りタブに変換します。
選択行を非タブ化する	現在の行を非タブ化します。行の先頭にある（テキストの前の）すべてのタブをスペースに変換します。
大文字にする	選択しているすべての文字を大文字に変換します。
小文字にする	選択しているすべての文字を小文字に変換します。
大文字 / 小文字を切り替える	選択しているすべての文字を，大文字または小文字に切り替えます。
先頭を大文字にする	選択しているすべての単語の先頭文字を大文字に変換します。
前後の空白を削除する	カーソル位置の前後にある余分な空白を削除し，空白文字を１個だけ残します。カーソルが単語内にある場合，または前後に空白文字がない場合，何も行いません。
末尾の空白を削除する	カーソルのある行で，最後の非空白文字の後にある空白を削除します。
行を削除する	現在カーソルのある行を完全に削除します。
行をコピーする	現在カーソルのある行をコピーして，その直後に挿入します。
空白行を削除する	カーソルのある行が空である場合，または空白文字しかない場合，その行を削除します。

[[ウインドウ] メニュー（エディタ パネル専用部分）]

エディタ パネル専用の［ウインドウ］メニューは次のとおりです（その他の項目は共通）。

分割	アクティブのエディタ パネルを水平方向に分割します。 分割の対象は，アクティブのエディタ パネルのみで，他のパネルは分割されません。分割の上限は，２分割までです。
分割の解除	エディタ パネルの分割表示を解除します。

[コンテキスト・メニュー]

【タイトルバー・エリア】

閉じる	選択しているパネルを閉じます。
このタブ以外すべて閉じる	選択しているパネルと同じパネル表示エリアに表示されているパネルを，選択しているパネルのみ残し，すべて閉じます。
ファイル名 の保存	ファイルの内容を保存します。
完全パスのコピー	ファイルの絶対パスをクリップ・ボードにコピーします。
含んでいるフォルダを開く	テキスト・ファイルが保存されているフォルダをエクスプローラで開きます。
新しい水平タブグループ	アクティブなパネルの表示領域を水平方向に均等に 2 分割して，新たなタブ・グループを表示します。新たなタブ・グループには，アクティブなパネルが 1 つだけ入ります。 分割の上限は，4 分割までです。 以下の場合は，この項目は表示されません。 - タブ・グループにパネルが 1 つしか開いていない - 垂直方向にタブ・グループが分割されている - タブ・グループが 4 分割されている
新しい垂直タブグループ	アクティブなパネルの表示領域を垂直方向に均等に 2 分割して，新たなタブ・グループを表示します。新たなタブ・グループには，アクティブなパネルが 1 つだけ入ります。 分割の上限は，4 分割までです。 以下の場合は，この項目は表示されません。 - タブ・グループにパネルが 1 つしか開いていない - 水平方向にタブ・グループが分割されている - タブ・グループが 4 分割されている
次のタブグループへ移動	表示領域を水平方向に分割している場合，選択しているパネルを表示しているタブ・グループの下側のタブ・グループに移動します。 表示領域を垂直方向に分割している場合，選択しているパネルを表示しているタブ・グループの右側のタブ・グループに移動します。 移動する側にタブ・グループがない場合は，この項目は表示されません。
前のタブグループへ移動	表示領域を水平方向に分割している場合，選択しているパネルを表示しているタブ・グループの上側のタブ・グループに移動します。 表示領域を垂直方向に分割している場合，選択しているパネルを表示しているタブ・グループの左側のタブ・グループに移動します。 移動する側にタブ・グループがない場合は，この項目は表示されません。

【文字列エリア】

切り取り	選択範囲の文字列を切り取り，クリップ・ボードにコピーします。 何も選択されていない場合は，その行を切り取ります。
コピー	選択範囲の文字列をクリップ・ボードにコピーします。 何も選択されていない場合は，その行をコピーします。

貼り付け	クリップ・ボードにコピーされている文字列をキャレット位置に，挿入モードの場合は挿入し，上書きモードの場合は上書きします。 ただし，クリップ・ボードの内容を文字列として認識できない場合は無効となります。 また，現在のソース・ファイルのモードはステータスバーに表示されます。
検索 ...	検索・置換 ダイアログを［クイック検索］タブが選択状態でオープンします。
移動 ...	指定した行へキャレットを移動するため，指定行へのジャンプ ダイアログをオープンします。
ジャンプ先の位置へ進む	［ジャンプ前の位置へ戻る］を実行する前の位置へ進みます。
ジャンプ前の位置へ戻る	［関数へジャンプ］を実行する前の位置へ戻ります。
関数へジャンプ	選択している文字列，またはキャレット位置の単語を関数と判断し，該当する関数へジャンプします。（「(f) 関数へジャンプ」参照）
タグ・ジャンプ	キャレットのある行にファイル名／行／桁の情報がある場合，該当するファイルの該当行／該当桁へジャンプします（「(g) タグ・ジャンプ」参照）。
ブックマーク	ブックマークに関するカスケード・メニューを表示します。
ブックマークの挿入 / 削除	キャレット位置の行にブックマークを登録します。 すでにブックマークを登録している場合は，そのブックマークを削除します。
次のブックマーク	アクティブなエディタ パネルにおいて，次に登録したブックマーク位置にキャレットを移動します。 最後のブックマークへ移動後は，最初に登録したブックマークの行へ移動します。
前のブックマーク	アクティブなエディタ パネルにおいて，前に登録したブックマーク位置にキャレットを移動します。 最初のブックマークへ移動後は，最後に登録したブックマークの行へ移動します。
すべてのブックマークの削除	アクティブなエディタ パネルにおいて，登録しているブックマークをすべて削除します。
ブックマークをリスト表示する ...	登録しているブックマークをリスト表示するブックマーク ダイアログをオープンします。
高度な設定	エディタ パネルに関する高度な操作を行うためのカスケード・メニューを表示します。
行インデントを増やす	現在カーソルのある行のインデントをタブ１個分増やします。
行インデントを減らす	現在カーソルのある行のインデントをタブ１個分減らします。
行コメントを削除する	現在カーソルのある行の先頭から，言語（C++ など）に応じた行コメントの区切り記号の最初のセットを削除します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
行コメントを付ける	現在カーソルのある行の先頭に，言語（C++ など）に応じた行コメントの区切り記号を設定します。現在のソース・ファイルが行コメントの区切り記号が指定されている言語（C++ など）を使用している場合のみ使用できます。
タブをスペースに変換する	現在カーソルのある行のすべてのタブをスペースに変換します。
スペースをタブに変換する	現在カーソルのある行の連続したスペースの一組をタブに変換します。ただし，そのスペースの各組がタブ１個以上の幅に等しい場合に限ります。
選択行をタブ化する	現在の行をタブ化します。行の先頭にある（テキストの前の）すべてのスペースを可能な限りタブに変換します。
選択行を非タブ化する	現在の行を非タブ化します。行の先頭にある（テキストの前の）すべてのタブをスペースに変換します。

大文字にする	選択しているすべての文字を大文字に変換します。
小文字にする	選択しているすべての文字を小文字に変換します。
大文字 / 小文字を切り替える	選択しているすべての文字を，大文字または小文字に切り替えます。
先頭を大文字にする	選択しているすべての単語の先頭文字を大文字に変換します。
前後の空白を削除する	カーソル位置の前後にある余分な空白を削除し，空白文字を 1 個だけ残します。カーソルが単語内にある場合，または前後に空白文字がない場合，何も行いません。
末尾の空白を削除する	カーソルのある行で，最後の非空白文字の後にある空白を削除します。
行を削除する	現在カーソルのある行を完全に削除します。
行をコピーする	現在カーソルのある行をコピーして，その直後に挿入します。
空白行を削除する	カーソルのある行が空である場合，または空白文字しかない場合，その行を削除します。

ファイル・エンコードの選択 ダイアログ

ファイル・エンコードの選択を行います。

備考　タイトルバーには，設定対象ファイルの名前を表示します。

図 B―5　ファイル・エンコードの選択 ダイアログ

ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［機能ボタン］

［オープン方法］

- ［ファイル］メニュー→［エンコードを指定して開く ...］を選択してファイルを開く ダイアログをオープン→ダイアログ上で［開く］ボタンをクリック

［各エリアの説明］

(1)［利用可能なエンコード］エリア

設定するエンコードを本エリアより選択します。

デフォルトでは，選択したファイルのエンコードが選択されています。

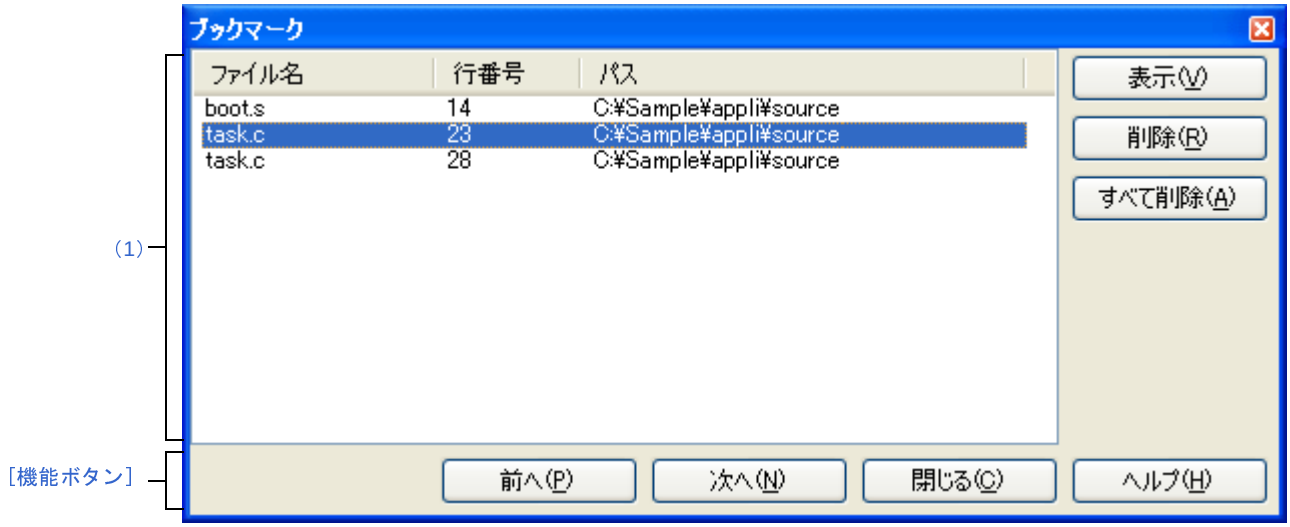
［機能ボタン］

ボタン	機能
OK	指定したファイル・エンコードを使用し，ファイルを開く ダイアログで選択したファイルをオープンします。
キャンセル	ファイルを開く ダイアログで選択したファイルをオープンせずに，このダイアログをクローズします。
ヘルプ	このダイアログのヘルプを表示します。

ブックマーク ダイアログ

ブックマーク設定位置の表示や，ブックマークの削除を行います。

図 B―6　ブックマーク ダイアログ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［機能ボタン］

［オープン方法］

- ツールバーの  をクリック
- ［編集］メニュー→［ブックマーク］→［ブックマークをリスト表示する ...］を選択
- エディタ パネルにおいて，コンテキスト・メニュー→［ブックマーク］→［ブックマークをリスト表示する ...］を選択

［各エリアの説明］

(1) ブックマーク・リスト表示エリア

登録しているブックマークをリスト表示します。

表示は，［ファイル名］のアルファベット順で表示します。同じファイル中のブックマークについては，行番号順に表示します。

エディタ パネルにブックマークが追加された場合，ブックマーク情報が追加されます。

ブックマーク・リスト表示エリアにおいて，行をダブルクリックすることにより，該当ブックマーク位置にキャレットを移動します。

(a)［ファイル名］

ブックマーク登録しているファイル名（パスなし）を表示します。

(b)［行番号］

ブックマーク登録している行番号を表示します。

(c)［パス］

ブックマーク登録しているファイルのパスを表示します。

(d) ボタン

表示	選択しているブックマーク位置に，キャレットを移動します。 ただし，ブックマークが選択されていない場合，複数個のブックマークを選択している場，またはブックマークが登録されていない場合は，無効となります。
削除	選択しているブックマークを削除します。複数個のブックマークを選択している場合は，選択しているすべてのブックマークを削除します。 ただし，ブックマークが選択されていない場合，またはブックマークが登録されていない場合は，無効となります。
すべてを削除	登録しているすべてのブックマークを削除します。 ブックマークが登録されていない場合は，無効となります。

注意 エディタ パネルをクローズしても，登録されていたブックマークは削除されません。ただし，ファイルを新規作成後に一度も保存していないエディタ パネルをクローズした場合，登録されていたブックマークは削除されます。

［機能ボタン］

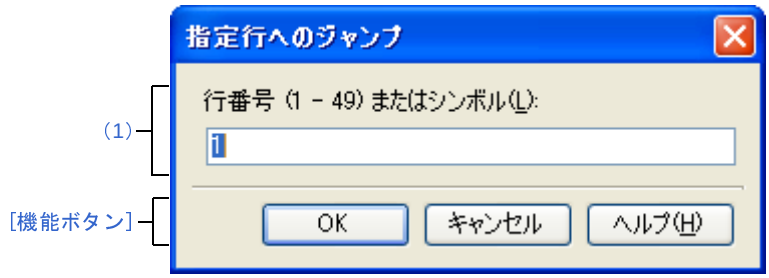
ボタン	機能
前へ	選択しているブックマークの前のブックマーク位置に，キャレットを移動します。 以下の場合は，無効となります。 - 先頭行に表示しているブックマークを選択している - ブックマークが選択されていない - 複数個のブックマークを選択している - ブックマークが登録されていない - ブックマークがひとつしか登録されていない
次へ	選択しているブックマークの次のブックマーク位置に，キャレットを移動します。 以下の場合は，無効となります。 - 最終行に表示しているブックマークを選択している - ブックマークが選択されていない - 複数個のブックマークを選択している - ブックマークが登録されていない - ブックマークがひとつしか登録されていない

ボタン	機能
閉じる	本ダイアログをクローズします。
ヘルプ	このダイアログのヘルプを表示します。

指定行へのジャンプ ダイアログ

指定した行番号／シンボル／アドレスにキャレットを移動します。

図 B—7　指定行へのジャンプ ダイアログ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［機能ボタン］

［オープン方法］

- ［編集］メニュー→［移動 ...］を選択
- エディタ パネルにおいて，コンテキスト・メニューの［移動 ...］を選択

［各エリアの説明］

(1) ［行番号（*有効な行の範囲*）］またはシンボル］エリア

“（*有効な行の範囲*）”には，現在のファイルの有効な行の範囲が表示されます。

キャレットを移動したい行番号／シンボル／アドレスを指定します。

デフォルトでは，エディタ パネル上の現在のキャレット位置の行番号が表示されます。

備考 1. シンボル（関数名，および変数名）を指定する場合は，ビルドの実行が完了している必要があります。

また，クロス・リファレンス情報を出力する必要があります。使用するビルド・ツールのプロパティ パネルにおいて，クロス・リファレンス情報を出力する設定（［共通オプション］タブ→［出力ファイルの種類と場所］カテゴリ→［クロス・リファレンス情報を出力する］プロパティ→［はい (-Xcref)］）を選択したのち，ビルドを実行してください。

2. アドレスを指定する場合は，ビルドの実行が完了している必要があります。

アドレス値は“0x”，または“0X”で始まる 16 進数のみ指定可能です。10 進数は行番号とみなします。

［機能ボタン］

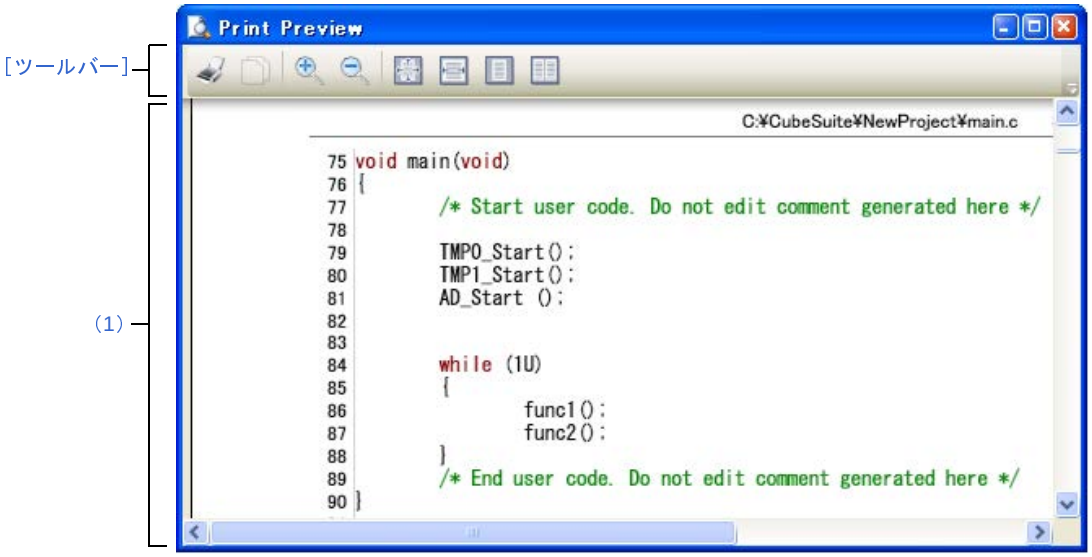
ボタン	機能
OK	指定したソース行の先頭にキャレットを移動します。
キャンセル	移動を無効とし，このダイアログをクローズします。
ヘルプ	このダイアログのヘルプを表示します。

Print Preview ウインドウ

印刷をする前に，現在エディタ パネルで表示されているファイルのプレビューを行います。

備考　［Ctrl］キーを押下しながらマウス・ホイールを前後方に動かすことにより，本ウインドウの表示を拡大／縮小することができます。

図 B—8　Print Preview ウインドウ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［ツールバー］
- ［コンテキスト・メニュー］

［オープン方法］

- エディタ パネルにフォーカスがある状態で，［ファイル］メニュー→［印刷プレビュー］を選択

［各エリアの説明］

(1) プレビュー エリア

印刷イメージをプレビュー表示します。

ヘッダ／フッダ部に，ファイル名（絶対パス）／ページ番号を表示します。

デバッグ・ツールと切断時，デバッグ・ツールと接続時（通常モード），またはデバッグ・ツールと接続時（混合表示モード）により表示するカラムが異なります。

ただし，エディタ パネルにおいて，非表示に設定しているカラムは表示されません（印刷されません）。

なお，アウトライン設定をしている場合，折りたたんだ状態を表示するマーク（「(b) コードのアウトライン表示」参照）とともに，たたまれている行の内容も表示します。

[ツールバー]

	印刷プレビュー表示しているアクティブなエディタ パネルの内容を印刷するために，Windows で用意されている，印刷 ダイアログをオープンします。
	選択範囲をクリップボードにコピーします。
	表示サイズを拡大します。
	表示サイズを縮小します。
	100% の倍率で表示します（デフォルト）。
	ページ幅で表示します。
	1 ページ全体を表示します。
	見開き 2 ページを表示します。

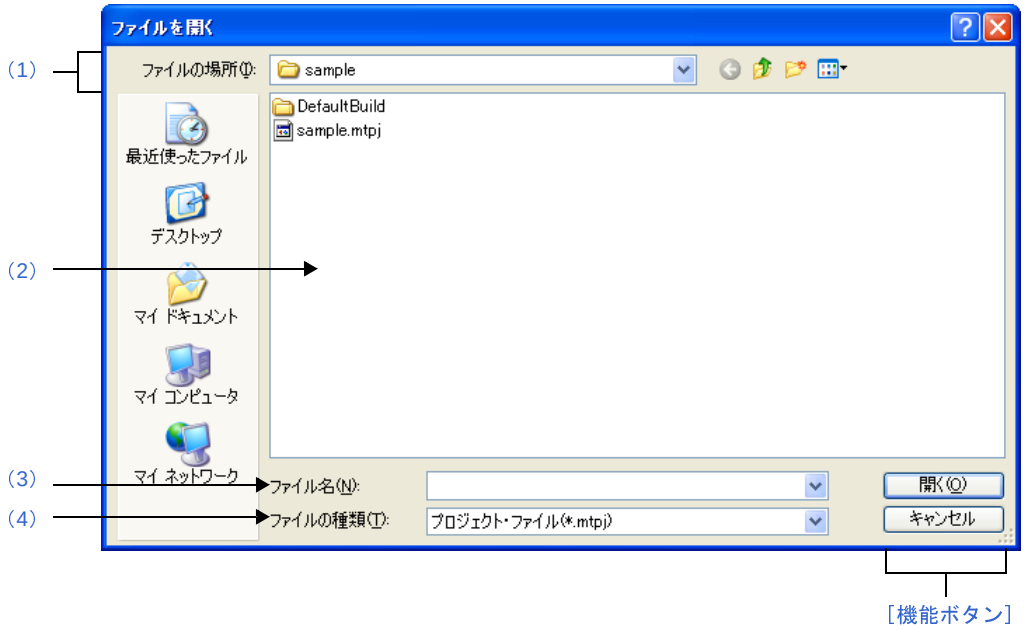
[コンテキスト・メニュー]

ズームの拡大	表示サイズを拡大します。
ズームの縮小	表示サイズを縮小します。

ファイルを開く ダイアログ

オープンするファイルの選択を行います。

図 B―9　ファイルを開く ダイアログ



ここでは，次の項目について説明します。

- ［オープン方法］
- ［各エリアの説明］
- ［機能ボタン］

［オープン方法］

- ［ファイル］メニュー→［ファイルを開く ...］，または［エンコードを指定して開く ...］を選択

［各エリアの説明］

(1)［ファイルの場所］エリア

オープンするファイルが存在するフォルダを選択します。

初回は “C:￥Documents and Settings￥*ユーザ名*￥My Documents”，2 回目以降は前回選択したフォルダが，デフォルトで選択されます。

(2) ファイルの一覧エリア

［ファイルの場所］，および［ファイルの種類］で選択した条件に合致するファイルの一覧を表示します。

(3)［ファイル名］エリア

オープンするファイルの名前を指定します。

(4)［ファイルの種類］エリア

オープンするファイルの種類（ファイル・タイプ）を選択します。

すべてのファイル (*.*)	すべての形式
プロジェクト・ファイル (*.mtpj)	プロジェクト・ファイル
e2 studio 用プロジェクト・ファイル (*.rcpc)	e^2 studio 用プロジェクト・ファイル
CubeSuite 用プロジェクト・ファイル (*.cspj)	CubeSuite 用プロジェクト・ファイル
HEW 用ワークスペース・ファイル (*.hws)	HEW 用ワークスペース・ファイル
HEW 用プロジェクト・ファイル (*.hwp)	HEW 用プロジェクト・ファイル
PM+ 用ワークスペース・ファイル (*.prw)	PM+ 用ワークスペース・ファイル
PM+ 用プロジェクト・ファイル (*.prj)	PM+ 用プロジェクト・ファイル
C ソース・ファイル (*.c)	C ソース・ファイル
ヘッダ・ファイル (*.h, *.inc)	ヘッダ・ファイル
テキスト・ファイル (*.txt)	テキスト形式

［機能ボタン］

ボタン	機能
開く	- ［ファイル］メニュー→［ファイルを開く ...］からオープンした場合 指定したファイルをオープンします。 - ［ファイル］メニュー→［エンコードを指定して開く ...］からオープンした場合 ファイル・エンコードの選択 ダイアログをオープンします。
キャンセル	本ダイアログをクローズします。

付録C 索 引

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2013.12.01	―	初版発行

CubeSuite+ V2.01.00　ユーザーズマニュアル
RL78,78K0Rコーディング編

発行年月日　2013年 12月 1日　Rev.1.00

発行　　ルネサス エレクトロニクス株式会社
〒211-8668　神奈川県川崎市中原区下沼部 1753

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/contact/

Colophon 2.0

CubeSuite+ V2.01.00

RENESAS

ルネサス エレクトロニクス株式会社

R20UT2774JJ0100