

# ビジュアルメモリ プログラマーズガイド

Visual Memory Programmer's Guide Revision 1.00

---



# Dreamcast™

# はじめに

弊社のハードウェア用アプリケーション開発にご協力いただき誠にありがとうございます。  
本書は、ビジュアルメモリに実装されているカスタムチップのコンパイラなどの開発ツールと、チップの命令を解説したマニュアルです。

ビジュアルメモリには、三洋電機社製の LC86870 ( カスタム版 ) が搭載されています。このチップは、さまざまな用途用の組み込みマイコンであり、その総称を LC86K シリーズと呼びます。

ビジュアルメモリのハードウェア詳細仕様を解説した『ビジュアルメモリ ハードウェアマニュアル』をご覧の上、本書を参考にしてコーディングを行なってください。

## 本書の内容

### インストール編

『Katana SDK』に添付のビジュアルメモリ用開発ツールをインストールする手順についてを解説しています。

### コンパイラ編

コンパイラ“ M86K.EXE ”のコマンドラインオプションやエラーメッセージなどを説明しています。

### リンケージローダ編

リンカ“ L86K.EXE ”のコマンドラインオプションやエラーメッセージなどを説明しています。

### ライブラリマネージャ編

ライブラリマネージャ“ LIB86K.EXE ”のコマンドラインオプションやエラーメッセージなどを説明しています。

### Eva to Hex 編

リンカが出力するファイルは、拡張子が“ .EVA ”というファイルです。このファイルをビジュアルメモリシミュレータや、ビジュアルメモリの実行ファイル形式に変更するために Eva to Hex ファイル変換ユーティリティ“ E2H86K.EXE ”を利用します。

ここでは、このツールのコマンドオプションを説明しています。

### MAKE 編

プログラム保守ユーティリティ“ MAKE.EXE ”を用いると、更新されたソースファイルのみを自動的にコンパイルし、新しいオブジェクトを作成するなど、面倒なコンパイル処理をバッチ処理的に行なうことが可能です。

ここでは、その処理手続きを記述するファイルの作成方法から、MAKE の使い方についてを説明しています。

## コーディング編

“ M86K.EXE ”アセンブラの文法、アセンブラ疑似命令リファレンスにあわせ、LC86870 の命令セットリファレンスを掲載しています。

## ご注意

本書は『LC86000 ユーザーズマニュアル』(三洋電機株式会社発行)を元にビジュアルメモリに特化した部分を株式会社セガ・エンタープライゼスが加筆した『ビジュアルメモリ プログラミングマニュアル Revision 1.00』を元に作成しました。

CPU の仕様、および本書に記載されている事項は、将来予告なしに変更することがあります。

ビジュアルメモリおよびマニュアルを運用した結果の影響については、いっさい責任を負いかねますのでご了承ください。

## 商標

ドリームキャスト、ビジュアルメモリは、株式会社セガ・エンタープライゼスの登録商標です。

その他、本文中に記載する製品名は、一般に開発メーカーの商標または登録商標です。なお、本文中では TM よび ( R ) マークは明記しておりません。

## 改版履歴

1998 年 8 月 31 日 初版発行

Copyright ( C ) 1998 三洋電機株式会社

Copyright ( C ) 1998 株式会社セガ・エンタープライゼス

編集・製作 株式会社アスキー AAP 書籍編集部

# 他のマニュアルとの関係

## ビジュアルメモリ関連のマニュアル

### 『ビジュアルメモリ ハードウェアマニュアル』

ビジュアルメモリのハードウェアとシステム BIOS の仕様をまとめた技術資料です。巻頭の概要編には、ビジュアルメモリのスペックを簡単にまとめたものを掲載しています。

ゲームデザイン担当の方は概要編を、プログラム担当の方は全般をご覧ください。

### 『ビジュアルメモリ プログラマーズガイド』

ビジュアルメモリに搭載されている三洋電機社製 LC86700 用のアセンブラ、リンカ、ライブラリマネージャ、ビジュアルメモリ用の実行ファイルを作成する“ E2H86K.EXE ”のインストールから使い方までをまとめたマニュアルです。また、LC86700 の命令セットやアセンブラの文法についても解説しています。

### 『ビジュアルメモリ チュートリアル』

ビジュアルメモリ用アプリケーションを開発するための手順、Katana またはコンピュータからビジュアルメモリにアプリケーションを転送する方法などを解説しています。また、詳細なコメント付きサンプルプログラムを掲載しています。

### 『ビジュアルメモリシミュレータガイド』

LC86700 用の実行ファイルをフェッチし、ビジュアルメモリの動作をソフトウェアにてエミュレーションする「ビジュアルメモリシミュレータ」を解説しています。

インストール方法から、使い方までをまとめたマニュアルです。

# 制限事項

開発ツールはすべて MS-DOS 用のプログラムです。

ビジュアルメモリシミュレータを除き、ビジュアルメモリ関連のツールはすべて MS-DOS 用のプログラムです。

# テクニカルサポートご案内

開発にあたって技術的なご質問や装置の不具合、またマニュアルの不備や装置の故障などがございましたら、下記連絡先までご一報ください。

株式会社セガ・エンタープライゼス  
テクニカルサポートセンター  
〒144-8531 東京都大田区羽田 1-2-12  
電 話：03-5736-7355  
F A X：03-5736-5357  
E-mail：katana@sft.sega.co.jp

# contents

目次

はじめに .....	2
他のマニュアルとの関係 .....	4
制限事項 .....	5
テクニカルサポートご案内 .....	6

## 第1部

## インストール編

21

### 第1章

### セットアップ

23

1.1 ファイルのコピー .....	23
--------------------	----

### 第2章

### 環境変数の設定

27

2.1 開発ツールで使用する環境変数 .....	27
2.1.1 環境変数の設定 .....	28

## 第3章

## アセンブルファイルの指定

31

- 3.1 ファイル名の指定 ..... 31
- 3.2 コマンドラインによる引数の指定方法 ..... 31
- 3.3 プロンプトによる引数の指定方法 ..... 32

## 第4章

## オプションスイッチ

35

- I 識別子(大文字/小文字)を区別しない指定 ..... 35
- D デバッグ情報の出力指定 ..... 35
- J 分岐命令の最適化をしない指定 ..... 35
- N 著作権表示を抑制する指定 ..... 35
- R 予約語ファイルの指定 ..... 36
- P 作業用バッファサイズの指定 ..... 36
- ? オプション一覧の表示 ..... 36

## 第5章

## 環境変数と予約語ファイル

37

- 5.1 環境変数 ..... 37
- 5.2 予約語ファイル ..... 38

## 第6章

### エラー一覧

39

- 6.1 警告 ..... 39
- 6.2 エラー ..... 41
- 6.3 致命的なエラー ..... 47

## 第7章

### リストファイルの書式

51

## 第3部

### リンケージローダ編

55

## 第8章

### リンクのファイル指定

57

- 8.1 ファイル名の指定 ..... 57
- 8.2 コマンドラインによる引数の指定方法 ..... 57
- 8.3 プロンプトによる引数の指定方法 ..... 59
- 8.4 リンク時に参照されるファイル ..... 60

-B=bank number	
LC86800シリーズフラッシュメモリ用 HEXファイルの作成 .....	61
-C=address	
CSEGローディングアドレスの指定方法 .....	61
-D=address	
DSEGローディングアドレスの指定方法 .....	61
-E DSEGのアドレス多重定義を許す .....	62
-I 大文字と小文字を区別しない .....	62
-P ローディングマップの作成 .....	62
-L ローカルシンボルリストの作成 .....	62
-W オペランドデータに関する警告メッセージの表示指定 .....	63
-A -F -O -R	
CSEG FREE ブロックの最適化ローディング .....	63
-S シンボルのソート処理に関する指定 .....	64

10.1 -A オプション .....	67
10.2 -A -F オプション .....	68
10.3 -A -O オプション .....	68
10.4 -A -R オプション .....	69

11.1 致命的なエラー .....	71
11.2 致命的ではないエラー .....	72

## 第 4 部

# ライブラリマネージャ編

73

## 第 12 章

# プログラムの起動

75

12.1	ファイル名の指定 .....	75
12.2	コマンドラインによる引数の指定方法 .....	75
12.2.1	オプション指定 .....	77
12.2.2	コマンドライン実行例 .....	77
12.3	プロンプトによる操作 .....	78
12.3.1	プロンプトラインの拡張 .....	78
12.3.2	デフォルトの応答 .....	78

## 第 13 章

# エラー一覧

79

## 第 14 章

# クロスリファレンス

81

第 15 章

プログラムの起動

85

- 15.1 ファイル名の指定 ..... 85
- 15.2 引数の指定方法 ..... 85
- 15.3 オプション指定 ..... 86
  - /I 変換中の表示情報を抑止 ..... 86

第 16 章

エラー一覧

89

- 16.1 致命的なエラー ..... 89

## 第 6 部

## MAKE 編

91

### 第 17 章

## MAKE の概要

93

17.1	MAKE の実行 .....	93
17.1.1	ビルドの優先順位 .....	94
17.1.2	コマンドラインオプション .....	94
	/E 外部マクロを優先する .....	94
	/I 終了コードを無視して最後まで実行 .....	94
	/N コマンドを実行せず経過のみを表示 .....	94
	/R ルールファイルの読み込みを禁止 .....	94
	/S コマンドの表示を抑止 .....	94
	/? ヘルプの表示 .....	95
17.2	MAKE 記述ファイル .....	95
17.2.1	記述ブロック .....	95
17.2.2	マクロ .....	97
17.2.3	ディレクティブ .....	98
17.3	推論規則 .....	99
17.3.1	メイクルールファイル .....	100

18.1	ステートメント .....	103
18.2	ラベル名およびシンボル名 .....	103
18.3	コメント .....	104
18.4	演算子 .....	104
18.5	数値定数 .....	104
18.6	文字定数 .....	105
18.7	文字列定数 .....	106
18.8	特殊シンボル .....	106

ORG .....	111
WORLD .....	112
CSEG .....	113
DSEG .....	114
END .....	115
PUBLIC .....	116
EXTERN .....	118
OTHER_SIDE_SYMBOL .....	119
EQU .....	120
SET .....	121
DB .....	122
DW .....	123
DC .....	124
DS .....	125
MACRO .....	126
REPT .....	129
IRP .....	130
IRPC .....	131
ENDM .....	132
EXITM .....	133
LOCAL .....	134
IFDEF .....	136
IFNDEF .....	137
IFB .....	138
IFNB .....	139
IFE .....	140
IFNE .....	141
IFIDN .....	142
IFDIF .....	143
ELSE .....	144
ENDIF .....	145
.PRINTX .....	146
.LIST .....	147
.XLIST .....	148
.MACRO .....	149
.XMACRO .....	150
.IF .....	151

.XIF .....	152
INCLUDE .....	153
TITLE .....	154
PAGE .....	155
CHIP .....	156
COMMENT .....	157
WIDTH .....	158
BANK .....	159
CHANGE .....	160
RADIX .....	161
JMPO .....	162
BRO .....	163
CALLO .....	164
BZO .....	165
BNZO .....	166
BPO .....	167
BPCO .....	168
BNO .....	169
DBNZO .....	170
BEO .....	171
BNEO .....	172

## 第20章

# LC86K 命令概要

173

20.1 命令概要 .....	173
20.1.1 算術演算命令 .....	173
20.1.2 論理演算命令 .....	174
20.1.3 データ転送命令 .....	174
20.1.4 ジャンプ命令 .....	174
20.1.5 条件分岐命令 .....	174
20.1.6 サブルーチン命令 .....	174
20.1.7 ビット操作命令 .....	175
20.1.8 その他の命令 .....	175
20.1.9 マクロ命令 .....	175
20.1.10 アドレッシング .....	175
20.1.11 プログラムメモリのアドレッシング .....	175
20.1.12 RAMと特殊機能レジスタ SFR のアドレッシング .....	177

<b>算術演算命令</b> .....	<b>182</b>
ADD # i8 .....	182
ADD d9 .....	183
ADD @ Rj .....	184
ADDC # i8 .....	185
ADDC d9 .....	186
ADDC @ Rj .....	187
SUB # i8 .....	188
SUB d9 .....	189
SUB @ Rj .....	190
SUBC # i8 .....	191
SUBC d9 .....	192
SUBC @ Rj .....	193
INC d9 .....	194
INC @ Rj .....	195
DEC d9 .....	196
DEC @ Rj .....	197
MUL .....	198
DIV .....	199
<b>論理演算命令</b> .....	<b>200</b>
AND # i8 .....	200
AND d9 .....	201
AND @ Rj .....	202
OR # i8 .....	203
OR d9 .....	204
OR @ Rj .....	205
XOR # i8 .....	206
XOR d9 .....	207
XOR @ Rj .....	208
ROL .....	209
ROLC .....	210
ROR .....	211
RORC .....	212

<b>データ転送命令</b> .....	<b>213</b>
LD d9 .....	213
LD @ Rj .....	214
ST d9 .....	215
ST @ Rj .....	216
MOV # i8 d9 .....	217
MOV # i8 @ Rj .....	218
LDC .....	219
PUSH d9 .....	220
POP d9 .....	221
XCH d9 .....	222
XCH @ Rj .....	223
 <b>ジャンプ命令</b> .....	 <b>224</b>
JMP a12 .....	224
JMPF a16 .....	225
BR r8 .....	226
BRF r16 .....	227
 <b>条件分岐命令</b> .....	 <b>228</b>
BZ r8 .....	228
BNZ r8 .....	229
BP d9 b3 r8 .....	230
BPC d9 b3 r8 .....	231
BN d9 b3 r8 .....	232
DBNZ d9 r8 .....	233
DBNZ @ Rj r8 .....	234
BE # i8 r8 .....	236
BE d9 r8 .....	237
BE @ Rj ,# i8 r8 .....	238
BNE # i8 r8 .....	239
BNE d9 r8 .....	240
BNE @ Rj ,# i8 r8 .....	241
 <b>サブルーチン命令</b> .....	 <b>242</b>
CALL a12 .....	242
CALLF a16 .....	243
CALLR r16 .....	244
RET .....	245
RETI .....	246

ビット操作命令 .....	247
CLR1 d9 b3 .....	247
SET1 d9 b3 .....	248
NOT1 d9 b3 .....	249
その他の命令 .....	250
NOP .....	250
マクロ命令 .....	251
CHANGE .....	251

## 第22章

## LC68K 命令セット一覧

253





---

## 第 1 部

---

# インストール編

---

---

ここでは、アセンブラ、リンカなどのツール類のインストールについてを説明します。ビジュアルメモリシミュレータのインストールについては、『ビジュアルメモリシミュレータガイド』を参照してください。



## 第 1 章

## セットアップ

CD-ROM の『KatanaSDK』に納められているビジュアルメモリ関連のツール類をインストールします。

## 注意

ビジュアルメモリシミュレータのインストールについては、『ビジュアルメモリシミュレータガイド』を参照してください。

## 1.1 ファイルのコピー

インストーラは特に用意されておりませんので、SDK からファイルをコピーします。

## 注意

ここでは、『KatanaSDK SET5 リリース JR5』を元に解説します。

- ① ツール類をインストールするフォルダ (ディレクトリ) を新規に作成します。  
ここでは、C ドライブの“ ¥DC\_DEV¥VM ”というフォルダにインストールするものとします。



ツール関連のフォルダは階層化されていますので、Windows のエクスプローラなどを用いると便利です。

- ② 『KatanaSDK』を CD-ROM ドライブにセットし、“ ¥VMS¥VMSTOOL¥ ”のファイルを表示させます。



- ③ “ LC86K ”フォルダを①で作成したフォルダにコピーします。



これで“ C : ¥DC\_DEV¥VM¥LC86K ”フォルダ以下に、ツール関連のファイルがすべてコピーされます。

- ④ “ AUTOEXEC.BAT ”などで環境変数“ PATH ”を設定します。

□

```
SET PATH=%PATH%;C:\¥DC_DEV¥VM¥LC86K
```

```
[]
```

SET PATH = % PATH % ; とすることで、現在設定されている PATH に “ ; C : ¥DC\_DEV¥VM¥LC86K ” を追加しています。

たとえば、環境変数 PATH に “ C : ¥dosbin ; C : ¥bintmp ” が設定されている場合、先の SET PATH=%PATH%;C:\¥DC\_DEV¥VM¥LC86K を実行すると、環境変数 PATH は “ C : ¥dosbin ; C : ¥bintmp ; C : ¥DC\_DEV¥VM¥LC86K ” となります。

これでツールのインストールは終わりです。

参照

PATH 以外にも、設定すると便利な環境変数があります。詳細については、次章を参照してください。



## 第 2 章

## 環境変数の設定

## 2.1 開発ツールで使用する環境変数

L86K シリーズの開発支援ツールでは、次の環境変数を使用します。

## PATH

検索パスを定義します。既に定義されている PATH に加えた形式で定義します。

名称	検索ファイル
M86K	PATH に記述されたディレクトリ中から予約語ファイル“ M86KRSVD.RWD ”を検索します。
L86K	PATH に記述されたディレクトリ中から予約語定義シンボルファイル“ LC86L.LIB ”を検索します。
CGR86K	PATH に記述されたディレクトリ中からデフォルトのキャラクタジェネレータのデータファイル“ DEFAULT.CGR ”を検索します。

## CHIPNAME

処理対象のチップ名（または、シリーズ名）を定義します。

ビジュアルメモリ用アプリケーションを開発する場合は、LC868700 としてください。

名称	内 容
M86K	アセンブル対象のチップ名を定義します。CHIP 疑似命令（第 7 部「コーディング編」、第 2 章「アセンブラ疑似命令」の「2.42 CHIP アセンブル対象チップの定義」を参照）がソースプログラムに記述されていると無視されます。この環境変数はチップ疑似命令の記述がないソースのアセンブル時に参照されます。チップ名中の ROM サイズ部分（チップ名の下 2 桁）を 00 とすると ROM サイズのチェックを行わず、64K 実装としてアセンブルします。
SU86K	オプションデータを作成対象のチップ名を定義します。チップ名中の ROM サイズ部分（チップ名の下 2 桁）は無視されます。
CGR86K	キャラクタジェネレータデータファイルの作成対象のチップ名を定義します。チップ名中の ROM サイズ部分（チップ名の下 2 桁）は無視されます。

チップ名の下 2 桁の数字が 00 の場合、実際のチップとしては存在していません。これらのチップ名称は、最大 ROM 容量を持つものとして、便宜的に定義されているものです。自分で作成したプログラムの大きさを知るためにアセンブルする場合、このチップ名を使用すると有用です。

## M86KRSVDFILE

予約語ファイルの名称を定義します。

名称	検索ファイル
M86K	予約語ファイルの格納先のディレクトリおよび、ファイル名を定義します。この環境変数が定義されていない場合には“ M86KRSVD.RWD ”がデフォルトのファイル名として使用され、PATH に記述された順序で“ M86KRSVD.RWD ”を検索します。

## M86KWORKFILE

作業ファイルの名称を定義します。

名称	検索ファイル
M86K	M86K コンパイラがアセンブル作業の途中で動的に割り付ける作業用メモリがメインメモリに入り切らなくなり、EMS メモリが実装されていないかあるいはそれも使い果たしてしまった場合に、一種の拡張メモリとして使用される作業ファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。

## TMP

作業ファイルを格納するディレクトリを定義します。

名称	検索ファイル
M86K	M86K が作業ファイル（「M86KWORKFILE」を参照）を作る必要があって、かつ環境変数 M86KWORKFILE が定義されていないければ、この環境変数で指定されるディレクトリに作業ファイルを作ります。またこの環境変数も定義されていないければ、カレントディレクトリに作業ファイルを作ります。いずれの場合も、ファイル名は“ M86KWORK.TMP ”に固定されています。

### 2.1.1 環境変数の設定

MS-DOS で環境変数の設定を行うには SET コマンドを使用します。SET コマンドの詳細は、MS-DOS のマニュアルやヘルプを参照してください。

**例：デフォルトのチップ名を LC868700 に設定する。**

```
A> SET CHIPNAME=LC868700↵
```



---

## 第 2 部

---

# コンパイラ編

---

ここでは、コンパイラである“ M86K.EXE ”のコマンドラインオプションや、エラーメッセージなどについてを説明します。

アセンブラの文法やアセンブラ疑似命令、LC86870 の命令セットについては「コーディング編」を参照してください。



## 第 3 章

# アセンブルファイルの指定

M86K を起動し、その実行に必要な情報を M86K に引き渡すには 2 種類の方法があります。

( 1 ) コマンドラインにてすべての情報を M86K に引き渡す方法

( 2 ) M86K が表示するプロンプトに回答してすべての情報を引き渡す方法

M86K を強制的に終了させるには、次のキーを使います。

機種名	キー
PC/AT 互換機の場合	<input type="button" value="Ctrl"/> + <input type="button" value="C"/> キーまたは <input type="button" value="Ctrl"/> + <input type="button" value="PauseBreak"/> キー
PC-9800 シリーズの場合	<input type="button" value="CTRL"/> + <input type="button" value="C"/> キーまたは <input type="button" value="STOP"/> キー

## 3.1 ファイル名の指定

M86K を起動するコマンドライン上で指定するファイル名、または M86K のコマンドプロンプトに対する応答として与えるファイル名には、大文字と小文字をどのように組み合わせ使用してもかまいません。たとえば、次の 3 種類のファイル名は同等のものとしてあつかわれます。

```
sample.asm  
SAmpLE.ASM  
SAMPLE.asM
```

また、ファイル名を拡張子なしで指定する場合、M86K は次のデフォルトのファイル名拡張子を使用します。

ファイル形式	デフォルトの拡張子
ソースファイル	.ASM
オブジェクトファイル	.OBJ
リストファイル	.LST
クロスリファレンスファイル	.CRF
エラーファイル	.ERR

## 3.2 コマンドラインによる引数の指定方法

M86K [ option ] [ source ], [ object ], [ list ], [ cross ], [ error ]

#### option フィールド

「第4章 オプションスイッチ」で説明されているアセンブラオプションを指定します。option を指定する場合、他のフィールドより前に指定します。

#### source フィールド

アセンブルするソースファイルの名前を指定します。ファイル名拡張子を省略した場合はデフォルトの拡張子“ .ASM ”を補給してファイルを探します。ファイル名拡張子も含めて指定されている場合は、そちらを優先します。いずれの場合もドライブ名およびパス名を含めて指定することができます。

#### object フィールド

アセンブル結果であるオブジェクトファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。このファイル名全体を省略すると、ソースファイル名の拡張子を“ .OBJ ”に換えたものをオブジェクトファイルとします。既に同一の名前のファイルが存在する場合には上書きされます。

#### list フィールド

アセンブル結果のリストを出力するファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。このファイル名全体を省略すると、リストファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

#### cross フィールド

アセンブル対象にしたソースファイル中のシンボルの相互参照リストファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。このファイル名全体を省略するとシンボルの相互参照リストファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

#### error フィールド

アセンブルの結果検出されたエラーメッセージを格納するファイル（エラーファイル）の名前を指定します。ドライブ名およびパス名を含めて指定することができます。このファイル名全体を省略するとエラーファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

#### 例

```
A> M86K MAIN.ASM,MAIN,,TEST.CXX ↵
```

カレントディレクトリに存在する“ MAIN.ASM ”をソースファイルとしてアセンブルを開始します。オブジェクトは“ MAIN.OBJ ”に書き込まれ、リストは生成されず、クロスリファレンスは“ TEST.CXX ”に書き込まれます。

## 3.3 プロンプトによる引数の指定方法

アセンブラの起動時にファイル名を指定せずにコマンドを入力します。その後、アセンブラの出力するプロンプトにしたがって、それぞれのファイル名を入力します。

```
prompt M86K [option]↵
SANYO (R) LC86K series Macro Assembler Version X.XX
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.
Source filename[.ASM]:____
Object filename[.OBJ]:____
Source listing [NUL.LST]:____
Cross reference[NUL.CRF]:____
Error messages [NUL.ERR]:____
```

## option フィールド

「第4章 オプションスイッチ」で説明されているアセンブラオプションを指定します。

### source filename

アセンブルするソースファイルの名前を指定します。ファイル名拡張子を省略した場合はデフォルトの拡張子“ .ASM ”を補給してファイルを探します。ファイル名拡張子も含めて指定されている場合は、そちらを優先します。いずれの場合もドライブ名およびパス名を含めて指定することができます。また、このファイル名を省略することはできません。↵キーのみが入力された場合には、再度ソースファイル名の入力をうながすようになっています。このとき入力を中止するには、次のキーを使います。

機種名	キー
PC/AT 互換機の場合	<span>Ctrl</span> + <span>C</span> キーまたは <span>Ctrl</span> + <span>PauseBreak</span> キー
PC-9800 シリーズの場合	<span>CTRL</span> + <span>C</span> キーまたは <span>STOP</span> キー

### Object Filename

アセンブル結果であるオブジェクトファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。この指定を省略する（↵のみを入力する）と、ソースファイル名の拡張子を“ .OBJ ”に換えたものをオブジェクトファイルとします。既に同一の名前のファイルが存在する場合には上書きされます。

### List Filename

アセンブル結果のリストを出力するファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。この指定を省略する（↵のみを入力する）と、リストファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

### Cross reference

アセンブル対象にしたソースファイル中のシンボルの相互参照リストファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。この指定を省略する（↵のみを入力する）と、シンボルの相互参照リストファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

## Error messages

アセンブルの結果検出されたエラーメッセージを格納するファイル（エラーファイル）の名前を指定します。ドライブ名およびパス名を含めて指定することができます。この指定を省略する（`/`のみを入力する）と、エラーファイルを作成しません。また、既に同一の名前のファイルが存在する場合には上書きされます。

## 第 4 章

## オプションスイッチ

この章では、M86K 自体の動作を指定し制御するために、アセンブラオプションをどのように使用するかを説明します。すべてのオプションは、アセンブラオプション文字である「-」または「/」で始まります。いずれの場合も、各オプションを指定する文字に大文字 / 小文字の区別はありません。たとえば、i と - I は同じ意味に解釈されます。

## -I

識別子 ( 大文字 / 小文字 ) を区別しない指定

このスイッチが指定されると、アセンブラはユーザー定義の識別子 ( ラベル、マクロ名、シンボル ) において、アルファベットの太文字と小文字を区別しないようになります。このスイッチが指定されていないければ、同一英文字の太文字と小文字は区別されます。また、このスイッチの効果はユーザー定義の識別子のみに限定され、二重モニタや SFR には適用されません。

## -D

デバッグ情報の出力指定

このスイッチが指定されると、アセンブラはオブジェクトファイル中のシンボル情報とソース行情報を出力しません。この情報がないオブジェクトのデバッグにおいて、ソースラインモードでデバッグを行うことはできません。このスイッチが指定されていないければ、両者はオブジェクトファイル中に出力され、ソースラインモードでデバッグを行うことができます。

## -J

分岐命令の最適化をしない指定

最適化を要する擬似命令 ( JMPO , CALLO , BRO ) を含むソースをアセンブルする際にこのスイッチが指定されていると、最適化動作を抑制することができます。結果としてこれらの擬似命令はいずれもジャンプ先にかかわらず 3 バイト長の命令として解釈されます。最適化を要する擬似命令を含む際にこのスイッチが指定されていないければ、最適化動作が行われます。最適化を要する擬似命令がない場合、このスイッチがあってもなくてもアセンブラの動作は同一です。

## -N

著作権表示を抑制する指定

このスイッチが指定されていると、アセンブラは起動時に著作権などの表示を行わなくなります。make などのユーティリティからアセンブラを起動する際に、エラーメッセージが他の必ずしも必要ではない表示によって見づらくなるのを防ぐために使用します。

---

## -R

### 予約語ファイルの指定

---

このスイッチの次の文字からそれに続く最初の空白文字の直前までを、予約語ファイルの名前としてアセンブラに指示します。たとえば、次のように指示します

```
m86k -rm86krsvd.rwd source.asm,,source.lst
```

このようにした場合“ M86KRSVD.RWD ”が予約語ファイル名となります。この指定は環境変数「 M86KRSVDFILE 」による指定より優先します。

---

## -P

### 作業用バッファサイズの指定

---

このスイッチに続いて数値が指定されている場合、その値をアセンブラ内部の作業用バッファの大きさとして採用します。作業用バッファとは、アセンブラがマクロの登録や展開を行う場合に処理速度を向上させるために使用するメモリ領域で、アセンブラの起動時にメインメモリ上に確保されます。デフォルトの大きさは 4096 バイトで、一般的なソースプログラムの場合、不足することはまず考えられません。しかし、万一このバッファの大きさが不足すると、アセンブラは次のエラーメッセージを表示します。

```
no more PARAMETER buffer (123) 45
```

アセンブラは、このエラーメッセージを表示して処理を中止します（メッセージ末尾の 2 つの数字は内部情報で、場合によって変わります）。万一このようなエラーメッセージが表示された場合は、このスイッチを用いてより大きなサイズを指定し、再度アセンブルを行ってください。たとえば、次のようにします。

```
m86k -p8192 source.asm
```

このようにした場合、作業用バッファの大きさは 8192 バイトになります。指定できるのは 10 進数のみで、必ずスイッチ文字である P の直後に空白などを入れずに指定します。また、このスイッチのみが指定されている場合など、数値が見当たらない場合は、バッファの大きさは変更を受けず、デフォルトの 4096 になります。

---

## -?

### オプション一覧の表示

---

このスイッチが指定されていると、アセンブラは次のような使用可能なオプションの一覧を表示し、その後実行を終了します。このスイッチが指定されていると、他にどのような指定があったとしても、オプション一覧の表示後、実行を終了するので注意してください。

```
Usage: m86k [option] source,[object],[list],[xref]
option:
  /D      do not make local symbol table and source line
          attributes in object file
  /I      ignore case for user defined symbol
  /J      do not try to optimize
  /N      skip displaying copyright message
  /Psize  parameter buffer size in decimal
  /Rfile  read 'file' as reserved word file
```

# 環境変数と予約語ファイル

## 5.1 環境変数

MS-DOS で環境変数の設定を行うには SET コマンドを使用します。SET コマンドの詳細は、MS-DOS のマニュアルやヘルプを参照してください。

**例：デフォルトのチップ名を LC868700 に設定する。**

```
A> SET CHIPNAME=LC868700 ↵
```

M86K では、次の環境変数を必要に応じて参照します。

### PATH

予約語ファイルの検索パスとして使用しています。予約語ファイル、およびその検索アルゴリズムについては「5.2 予約語ファイル」を参照してください。

### CHIPNAME

アセンブル対象のチップ名を定義します。CHIP 疑似命令がソースプログラムに記述されていると無視されます。ただし、CHIP 疑似命令で指定されたチップ名と本環境変数でのそれが一致しない場合は、警告メッセージを発生します。CHIP 疑似命令の記述がないソースのアセンブル時に参照されます。

### M86KRSVDFILE

予約語ファイルの格納先のディレクトリ、およびファイル名を定義します。なお、この環境変数で指定されるファイル名にデフォルトのファイル名拡張子はありません。ドライブ名とパス名を必要に応じて、ファイル名と拡張子は必ず指定してください。

### M86KWORKFILE

M86K がアセンブル作業の途中で動的に割り付ける作業用メモリがメインメモリに入りきらなくなり、EMS メモリが実装されていないか、あるいはそれも使い果たしてしまった場合に、一種の拡張メモリとして使用される作業ファイルの名前を指定します。ドライブ名およびパス名を含めて指定することができます。

### TMP

M86K が作業ファイル（前述の「M86KWORKFILE」を参照）を作る必要があって、かつ環境変数 M86KWORKFILE が定義されていない場合は、この環境変数で指定されるディレクトリに作業ファイルを作ります。また、この環境変数も定義されていなければ、カレントディレクトリに作業ファイルを作ります。いずれの場合も、ファイル名は“M86KWORK.TMP”

に固定されています。

## 5.2 予約語ファイル

予約語ファイルとは、M86K がその起動時に必ず読み込むファイルで、アセンブル対象とするチップに関するさまざまな情報（RAM/ROM の大きさ、SFR のモニタリングなど）を含んでいます。M86K は、この予約語ファイルなしには正常に動作しません。M86K はその起動時に、次のような手順で予約語ファイルを探します。

- (1) アセンブラオプション - R によってファイル名が明示的に指定されている場合、そのファイルを読み込みます。そのファイルが存在しない、または読み込みが許可されない場合はエラーとなります。
- (2) 環境変数 M86KRSVDFILE が定義されていれば、それによって指定されているファイルを読み込みます。そのファイルが存在しない、または読み込みが許可されない場合はエラーとなります。
- (3) "M86K.EXE" があるディレクトリに "M86KRSVD.RWD" という名前のファイルがあり、読み込みが許可されている場合は、そのファイルを読み込みます。
- (4) カレントディレクトリに "M86KRSVD.RWD" という名前のファイルがあり、読み込みが許可されている場合は、そのファイルを読み込みます。
- (5) 環境変数 PATH に指定されているディレクトリを順番に検索し、最初に見つかった読み込み可能な "M86KRSVD.RWD" という名前のファイルを読み込みます。

上記検索を順番に行っても予約語ファイルが見つからなかった場合は、エラーとして M86K の実行を中止します。通常は、予約語ファイルは "M86K.EXE" と同じディレクトリに格納しておきます。なお、このファイルの内容は M86K を正常に動作させる上で不可欠なものであり、削除または改変にともなう M86K の不具合については責を負いかねます。そのため、このファイルについては書き込み保護を指定しておくことを強くお勧めします。

## 第 6 章

## エラー一覧

M86K が検出するエラーには、致命的なエラー、エラー、警告の 3 つのレベルがあります。致命的なエラーが検出された場合、M86K はその時点で即時に実行を中止します。「作業用バッファが不足している」などの問題がこのレベルに該当します。これに対してエラーが検出された場合には、その時点で実行中のパス（パス 1 またはパス 2）が完了した時点で実行を中止します。いわゆる「文法エラー」がこれにあたります。さらに警告の場合には、オペランドの値が許されている範囲を越えている、といった一概にはエラーといえないレベルの問題であるため、M86K は実行を中止しません。

致命的なエラーが検出された場合、M86K は出力するように指示されているすべてのファイルを生成しません。これに対してパス 1 でエラーが検出された場合には、出力するように指示されているすべてのファイルを生成しませんが、パス 2 で検出された場合にはリストファイルだけは生成します（生成することを指定されていた場合）。次にエラー表示の書式を示します。

```
filename(linenum): source line  
error message
```

## 例

```
sample.asm(54): LD      xyz  
xyz: undefine symbol
```

シンボル xyz は未定義です。

## 6.1 警告

M86K が検出する可能性のある警告レベルのメッセージとその意味を次に示します。なお、メッセージ中の???は、場合によって変わる部分を表します。

??? : ビット number exceeds limits

ビット操作命令において、操作対象ビットの指定が許される範囲を越えています。

absolute expression expected

アセンブル時点で値が確定するような式が必要です。

address beyond zero

ORG 命令のオペランドにおいて、指定された値が負です。

address exceeds limits

ORG 命令のオペランドにおいて、指定された値が ROM の容量を越えています。

address exceeds ROM size

アセンブルされた命令のアドレスが ROM の容量を越えました。

chip name is different from one specified by CHIPNAME ( ??? ).

CHIP 命令のオペランドが環境変数で指定されているものと食い違ってしています。

END in included file

INCLUDE 疑似命令にて指定されたソースファイル中に END 疑似命令が現れました。

ENDF without FUNCTION

ファンクションの定義中でないのに ENDF が現われました。

ENDM without MACRO

マクロの定義中でないのに ENDM が現われました。

EXITM outside MACRO

マクロの定義中でないのに EXITM が現われました。

function code buffer overflow

ファンクション定義の内容が大き過ぎてバッファに入りません。

illegal combination of attributes : ???

2 項演算子の両辺の属性 ( 帰属するバンクやセグメント ) が一致しません。

illegal style expression

SET または EQU のオペランドの式が不当な形式です。

JMP/CALL placed at the end of memory block ( FREE )

アドレスの下位 12 ビットが 0FFEh または 0FFFh の時に JMP または CALL 命令が現われました。セグメントの配置モードが「FREE」のためリンクの結果によっては問題ない場合もありますが、当該セグメントがメモリバウンダリの先頭から配置された場合はリンクでエラーとなります。

Jump address is out of range ( FREE )

ジャンプ先のアドレスがメモリバウンダリの外にあります。セグメントの配置モードが「FREE」のためリンクの結果によっては問題ない場合もありますが、当該セグメントがメモリバウンダリの先頭から配置された場合はリンクでエラーとなります。

LOCAL outside MACRO

マクロの定義中でないのに LOCAL が現われました。

macro name in expression

演算式中にマクロとして登録されているシンボルが現われました。

macro name required

マクロの定義にもかかわらずマクロの名前がありません。

no character in string

文字列定数中に文字が見当たりません。

page width must be 72 ~ 132 : ???

WIDTH 命令のオペランドは 72 以上 132 以下でなければなりません。

public ??? not defined

パブリック宣言されているシンボルの値が定義されていません。

SET conflicts with PUBLIC

パブリック宣言されたシンボルに SET を使って値を再設定しようとしています。

symbol name required

PUBLIC , EXTERN , OTHER\_SIDE\_SYMBOL のオペランドにシンボルがありません。

undefined symbol in expression

演算式の中に未定義シンボルがあります ( パス 2 でのみ検出されます )。

value is out of range

値が許される範囲にありません ( 「 許される範囲 」 はオペランドによって変わります )。

zero divide : ??? modulo 0

演算子 MOD の右辺が 0 です。

zero divided : ??? / 0

演算子 / の右辺が 0 です。

## 6.2 エラー

M86K が検出する可能性のあるエラーレベルのメッセージとその意味を次に示します。  
なお、メッセージ中の???は、場合によって変わる部分を表します。

??? : 2 , 8 , 10 or 16 required

疑似命令 RADIX のオペランドとして指定できる値は 2、8、10、または 16 のいずれかです。

??? : constant required

数値定数が見当たりません。

??? : duplicated label

ラベルが重複しています。

??? : duplicated symbol

シンボルが重複しています。

??? : illegal character in numeric constant

数値定数の中に不当な文字があります。

??? : no such chip in the table

CHIP 命令で指定されたシンボルが予約語ファイル中に見当たりません。

??? : open error

ファイルオープン時にエラーを検出しました。

??? : undefined symbol

未定義シンボルが参照されています。

??? : radix violation

指定された基数に含まれない文字が数値定数の中にあります。

???H, ??? : out of internal RAM area

データセグメントのアドレス割り付けが許容される範囲を越えています。

' not seen

文字定数の右側の「 ' 」が見当たりません。

: ' not seen

EXTERN のオペランドでセグメントを明示する書式の場合にセグメントとシンボルを区切る「 : 」が見当たりません。

0x??? : RAM address exceeds limits

データセグメントのアドレス割り付けが許容される範囲を超えています。

address duplicated

DS 疑似命令で指定された RAM 上のアドレス領域が重複しています。

address exceeds absolute limits

アセンブルされた命令のアドレスが 65535 を超えました。

bank number should be 0~15

バンク番号は 0~15 でなければなりません。

**Branch address beyond zero**

ブランチ先アドレスに0番地(当該コードセグメントの先頭)より小さい値が指定されています。

**Branch address exceeds limits**

ブランチ先アドレスがROMの容量を越えています。

**CSEG conflicts with WORLD EXTERNAL\_DATA**

WORLD EXTERNAL\_DATA とセグメントを指定する疑似命令とは同一ソースファイル中には記述できません。

**CSEG isn't allowed in macro**

マクロの定義中にセグメントを指定する疑似命令は記述できません。

**DS must be in DSEG**

DS 疑似命令はデータセグメントでのみ指定可能です。

**DSEG conflicts with WORLD EXTERNAL\_DATA**

WORLD EXTERNAL\_DATA とセグメントを指定する疑似命令とは同一ソースファイル中には記述できません。

**DSEG isn't allowed in macro**

マクロの定義中にセグメントを指定する疑似命令は記述できません。

**ELSE without IFxxx**

条件アセンブル用の疑似命令 ELSE に対応する IFxxx が見当たりません。

**ENDF not seen**

ファンクション定義の終了を宣言する ENDF 疑似命令が見当たりません。

**ENDIF without IFxxx**

条件アセンブル用の疑似命令 ENDF に対応する IFxxx が見当たりません。

**ENDM not seen**

マクロ定義の終了を宣言する ENDM 疑似命令が見当たりません。

**external symbol can't be public**

外部シンボルがパブリック宣言されています。

**Hardware configuration violation**

指定されたチップにはインプリメントされていない機能に対応する命令(CHANGE 命令など)です。

identifier expected

マクロ定義の仮引数リストか EXTERN のオペランドに識別子以外のものがあります。

illegal character in ??? constant

指定された基数では許されない文字が数値定数内にあります。

illegal character in binary constant

指定された基数では許されない文字が数値定数内にあります。

illegal symbol type

PUBLIC 宣言しようとしているシンボルの型が不当です。

illegal word in external list

EXTERN のオペランドでの文法エラーです。

instructions can 't be in DSEG

データセグメントに DS 以外の命令が記述されています。

JMP/CALL placed at the end of memory block (INBLOCK)

アドレスの下位 12 ビットが 0FFEh または 0FFFh の時に JMP または CALL 命令が現われました。セグメントの配置モードが「INBLOCK」のためリンカでエラーとなります。

Jump address beyond zero

ジャンプ先アドレスに 0 番地 (当該コードセグメントの先頭) より小さい値が指定されています。

Jump address exceeds limits

ジャンプ先アドレスが ROM の容量を越えています。

Jump address is out of range (INBLOCK)

ジャンプ先のアドレスがメモリバウンダリの外にあります。セグメントの配置モードが「INBLOCK」のためリンカでエラーとなります。

local symbol can 't be public

ローカルシンボルがパブリック宣言されています。

lost SET symbol

SET 疑似命令で定義されたシンボルがパス 2 で見つからなくなりました。アセンブラ内部のエラーの疑いがあります。

macro can 't be public

マクロがパブリック宣言されています。

maximum nesting of macro is 10

マクロのネストレベルの最大は 10 までです。

Multiple WORLD specified

同一ソースファイルに複数の WORLD 疑似命令が記述されています。

name required for macro

マクロの定義に名前が見当たりません。

no room for source line attribute object

ソース行属性 ( デバッガ用の情報 ) を格納するためのメモリがなくなりました。

no value for EXT

CHANGE 命令が使用されているのに SFR にレジスタ EXT が見当たりません。

not the symbol defined by SET

SET で定義されたのではないシンボルに対して SET で値を再設定しようとしています。

operand exceeds limits

REPT マクロ疑似命令の繰り返し回数指定が 1 ~ 65535 の範囲にありません。

ORG isn 't allowed in macro

マクロの中に ORG 疑似命令は記述できません。

other-side symbol isn 't allowed

ここでは OTHER\_SIDE\_SYMBOL で宣言されたシンボルは指定できません。

other-side symbol isn 't allowed here

ここでは OTHER\_SIDE\_SYMBOL で宣言されたシンボルは指定できません。

other-side symbol or absolute constant is required

OTHER\_SIDE\_SYMBOL で宣言されたシンボルまたは定数が必要です。

positive value required

負の値は使用できません。

public ??? not defined

PUBLIC 宣言されたシンボルを定義するものがありませんでした。このエラーは、PUBLIC 宣言されただけで値の定義も参照もないシンボルの場合には「警告」レベルのエラー、値の定義はないが参照されている場合は「エラー」レベルのエラーとなります。

string is too long

文字列定数の長さが制限 ( 255 文字まで ) を超えました。

symbol name required

SET, EQU の左側にシンボルがありません。

symbol not defined

パス 2 で PUBLIC, EXTERN, OTHER\_SIDE\_SYMBOL のオペランドに指定されたシンボルが見当たりません。アセンブラ内部のエラーの疑いがあります。

syntax error

文法エラーが検出されました。

syntax error near ???

???の近辺で文法エラーが検出されました。

too complexed expression for an operand

1 つのオペランドに記述された演算式が複雑すぎて解析できませんでした。

too many CHIP pseudo operation

1 つのソースファイル内に複数の CHIP 疑似命令が記述されています。

too nested if-statements

条件アセンブル用の疑似命令がネストレベル 10 を超えました。

unbalanced conditional assembling controllers

条件アセンブルによる読み飛ばしの途中でソースファイルの終りが現われました。

unbalanced IF statement

条件アセンブルによる読み飛ばしの途中でソースファイルの終りが現われました。

unexpected end of file in string

文字列定数の途中でソースファイルの終りが現われました。

unexpected end of line in string

文字列定数の途中で行の終りが現われました。

unexpected EOF in conditional assembling

条件アセンブルによる読み飛ばしの途中でソースファイルの終りが現われました。

unexpected terminator ??? in conditional assembling

条件アセンブルによる読み飛ばしの途中で構文解析ルーチンが異常終了しました。アセンブラ内部のエラーの疑いがあります。

unmatched ELSE in skipping

条件アセンブルによる読み飛ばしの途中でソースファイルの終りが現われました。

unmatched ENDIF

条件アセンブルによる読み飛ばしの途中でソースファイルの終りが現われました。

WORLD conflicts xSEG

WORLD EXTERNAL\_DATA とセグメントを指定する疑似命令とは同一ソースファイル中には記述できません。

## 6.3 致命的なエラー

M86K が検出する可能性のある致命的なエラーのレベルのメッセージとその意味を次に示します。なお、メッセージ中の???は、場合によって変わる部分を表します。

??? (???): chip name not seen

??? (???): chip name not seen.

??? (???): decimal value required

??? (???): hex-value and reserved-word are required

??? (???): no chip name list

??? (???): no reserved word seen

??? (???): ROM size not seen

??? (???): too many chip names

??? (???): ??? : unknown chip name

??? (???): ??? : unknown flag

予約語ファイル中の文法エラーです。

??? : illegal file name

指定されたファイル名中に不当な文字があります。

??? : no such chip in the table

環境変数 CHIPNAME で指定されたチップ名が予約語ファイル中に見当たりません。

??? : no such user

~user で指定されるユーザー名が見当たりません。

??? : open error

指定されたファイルのオープンに失敗しました。

??? : unknown flag

不当なアセンブラオプションが指定されています。

??? : unreadable

指定されたファイルが読み込めません。

EMM v3.2 or later is required ( v??? found )

EMS ドライバのバージョンが古く、対応できません。v3.2 以降のドライバが必要です。

EMS allocation ( ??? pages ) was failed

EMS メモリの割り当てに失敗しました。

EMS deallocation was failed

EMS メモリの開放時にエラーを検出しました。

flushing error in workfile

作業ファイルの掃き出し時にエラーを検出しました ( ディスクの空き領域がなくなったなど )

Getting EMM version was failed

Getting EMS status was failed

Getting free page count on EMS is failed

Getting physical page frame address was failed

EMS ドライバのバージョンのチェックなど、EMS メモリの初期化の段階でエラーを検出しました。

making temp. name for ??? failed

出力ファイルに仮の名前を与える際にエラーを検出しました。

Neither CHIP pseudo operation nor CHIPNAME environment variable were defined. Further execution aborted.

CHIP 疑似命令も環境変数 CHIPNAME によるチップの指定もなく、アセンブル対象チップが特定できないため、以後の動作を中止します。

no more MAIN memory ( ??? ) ???

必ずメインメモリに割り当てなければならない領域があるにもかかわらず、これ以上割り当てられるメインメモリがありません。

no more memory ( ??? )

動的に割り当てられるメモリ ( メインメモリ、EMS メモリ、作業ファイル ) がありません。

no more NODE buffer ( ??? ) ???

演算式の解析に用いる作業領域が不足しています。

no more PARAMETER buffer ( ??? ) ???

マクロの定義や呼び出しの際の引数リストを処理するための作業領域が不足しています。

no reserved word file available.

予約語ファイルの読み込みに失敗しました。

no room for file : ???

ディスクがいっぱいでファイルの書き込みができません。

Pxxxx must be less than 65536

パラメータバッファの大きさは 65535 以下を指定してください。

read error in workfile ( ??? )

作業ファイルの読み込み時にエラーが発生しました。

removing ??? failed

エラーを検出したために、途中まで作成した出力ファイルを削除しようとしたが失敗しました。

renaming ??? ==> ??? failed

仮の名前を与えて作成した出力ファイルを正式な名前に変更しようとしたときにエラーが発生しました。変更後の名前と同じ名前のファイルが既にある場合、書き込み禁止になっている場合に生じるエラーです。

too many file names

コマンド行に 5 つ以上のファイル名が指定されています。

too many nested include files

ネストしているファイルのインクルードが許される限度 ( レベル 10 ) を越えました。

unlinking work file is failed

作業ファイルを削除するときにエラーを検出しました。

workfile ??? : already exist

workfile ??? : open error

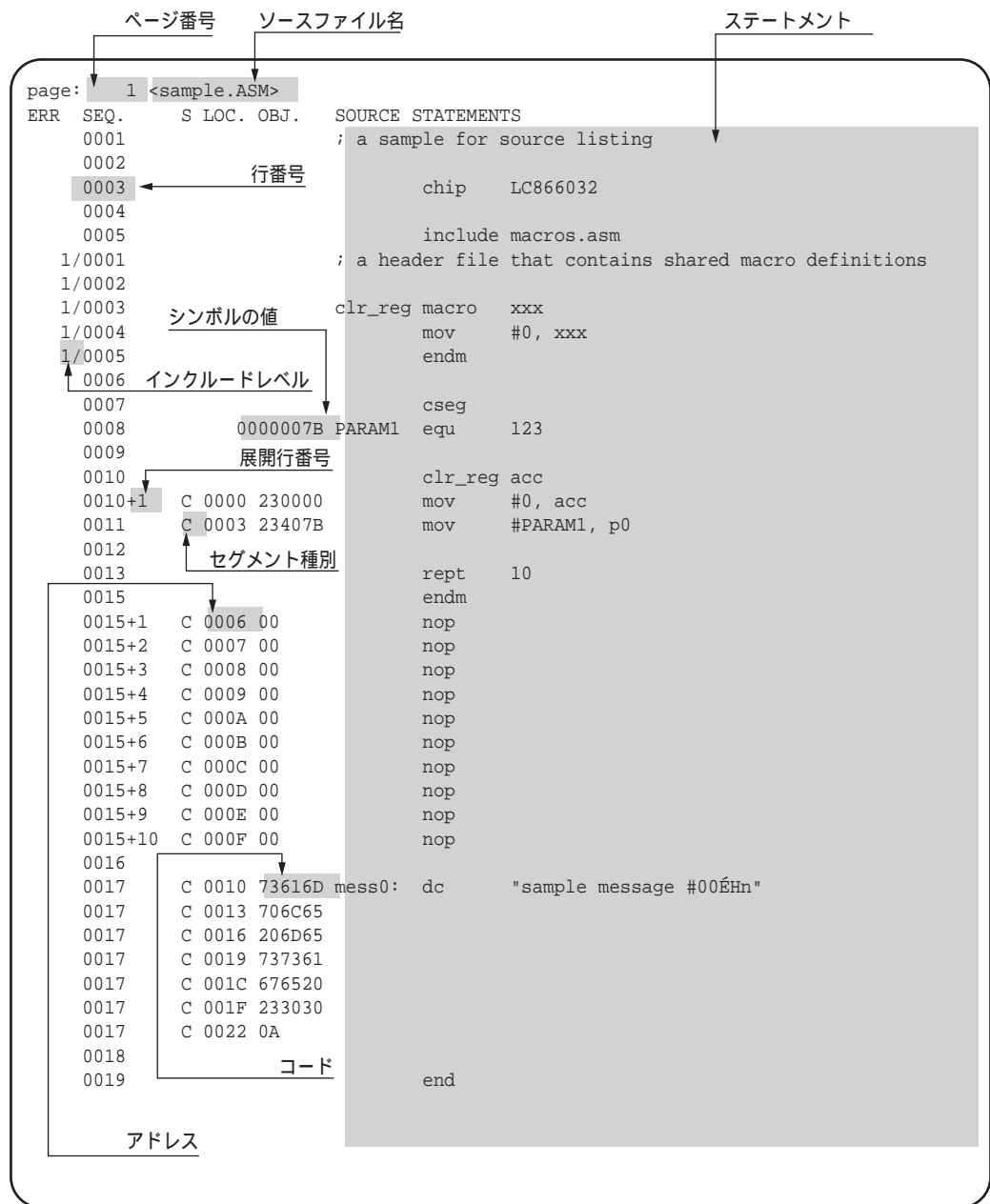
作業ファイルを新たに作成するときにエラーを検出しました。



## 第 7 章

## リストファイルの書式

M86K が生成するリストファイルは、次に示すような形式をしています。これは基本的にソースファイルの内容をそのままに、左側に行番号や機械語のコードを表示したものです。また、行や桁の配置は、紙に印刷されることを念頭に置いた書式设计になっています。すなわち、1 ページは 60 行（ヘッダ部を含みます）、1 行は 132 文字（ソース行がそれより長い場合は折り返します）で構成され、左半分には桁位置を固定して各種の情報が表示されるようになっています。また、ソース行に含まれている水平タブは、スペースに変換されて見掛け上の位置が変化しないようになっています。



## ヘッダ部

各ページごとにその先頭に現われ、とじ代のための空白行とページ番号、ソースファイル名、およびリスト本体の各桁位置の見出しからなっています。

## ページ番号

1 から始まる通し番号です。

## ソースファイル名

アセンブルすることを指定されたソースファイルの名前が表示されます。指定時にドライブ名やパス名が付随していた場合には、それらも含めて表示されます。

## ステートメント

ソースファイルの内容、マクロ呼び出しがあってリスト出力が抑制されていない場合にはその展開結果、インクルードファイルが読み込まれている場合はその展開結果を表示します。

## 行番号

ソースファイルにおける行番号 (10 進) です。コード部が複数行に渡る場合など、ソースファイルの 1 行がリストファイルの 2 行以上になる場合には、同一の行番号が繰り返されることとなります。

## インクルードレベル

インクルードファイルのネストレベルを表示します。ソースファイルの内容を表示している行では、この部分は現われません。ソースファイルから直接インクルードされているファイルの内容を表示している行ではこのレベルは 1 になります。そのファイルからさらにインクルードされているファイルではこの数字が 2 になります。行番号との間の / は区切り記号です。

## シンボルの値

シンボルに値が設定されるときには、設定された値がアセンブル時点で確定するものであればその値が 16 進 8 桁で表示されます。値が確定しない場合には表示しません。

## 展開行番号

マクロ呼び出し (繰り返しマクロを含む) の結果生成された、元々のソースファイルにはない行であることを示す番号です。展開された順番に 1 からの通し番号が割り当てられます。1 つのマクロ呼び出しが終り、別のマクロ呼び出しが現われると再び 1 からの通し番号が割り当てられます。

## セグメント種別

対応する行が CSEG または DSEG にコードを発生する場合、セグメントの種別を表す 1 文字が表示されます。C (大文字の C) が CSEG INBLOCK、c (小文字の c) が CSEG FREE、D が DSEG を表します。

## アドレス

対応する行が CSEG または DSEG にコードを発生する場合、そのコードの第 1 バイト目が位置するアドレスを 16 進 4 桁で表示します。ただし、ここでのアドレスとは、該当セグメントの先頭からのオフセットです。

## コード

対応するソース行をアセンブルした結果 ROM に書き込むべきコードが発生する場合、そのコードを 16 進 2 桁 × バイト数で表示します。コードは 1 行当たり最大 3 バイト分表示され、最も左側の 2 桁が最も値の小さいアドレスに対応するバイトです。コードの長さが 3 バイトを越える場合には、同一行番号を持つ行を新たに生成して表示を続行します。





---

## 第 3 部

---

# リンケージ ローダ編

---

---

ここでは、リンカである“ L86K.EXE ”のコマンドラインオプションや、エラーメッセージなどについてを説明します。



## 第 8 章

# リンクのファイル指定

L86K を起動し、その実行に必要な情報を L86K に引き渡すには次の 2 種類の方法があります。

(1) コマンドラインにてすべての情報を L86K に引き渡す方法

(2) L86K が表示するプロンプトに回答してすべての情報を引き渡す方法

また、L86K を強制的に終了させるには起動の方法にかかわらず、次のキーを使います。

機種名	キー
PC/AT 互換機の場合	<b>Ctrl</b> + <b>C</b> キーまたは <b>Ctrl</b> + <b>PauseBreak</b> キー
PC-9800 シリーズの場合	<b>CTRL</b> + <b>C</b> キーまたは <b>STOP</b> キー

## 8.1 ファイル名の指定

L86K を起動するコマンドライン上で指定するファイル名、または L86K のコマンドプロンプトに対する応答として与えるファイル名には、大文字と小文字をどのように組み合わせ使用してもかまいません。たとえば、次の 3 種類のファイル名は同等のものとしてあつかわれます。

```
sample.obj
SAmp1E.OBJ
SAMPLE.OBJ
```

また、ファイル名を拡張子なしで指定する場合、L86K は次のデフォルトのファイル名拡張子を使用します。

ファイル形式	デフォルトの拡張子
オブジェクトファイル	.OBJ
実行ファイル	.EVA
ライブラリファイル	.LIB
オプションファイル	.OPT
フォントファイル	.CGR
フラッシュメモリ用データファイル	.Hnn

nn はオプション - B で指定する数値です

## 8.2 コマンドラインによる引数の指定方法

```
L86K [ option ] objectfiles [ , [ evafile ] [ , [ libraryfile ] ] ] ; ]
```

### option フィールド

第2章「リンケージローダーオプションの指定」で説明されているリンケージローダオプションを指定します。option を指定する場合、任意のフィールドの前に指定します。

### objectfiles フィールド

リンクするオブジェクトの名前、リンク開始アドレス、およびライブラリの名前を指定します。少なくとも1つのファイル名は必要です。複数のファイル名を指定する際には、ファイル名とファイル名を空白記号（スペース）で区切ります。1行で収まらない場合には行の最後にプラス記号（+）を記述します。オブジェクトファイルを示す拡張子“.OBJ”を省略したファイル名を省略すると自動的に“.OBJ”を拡張子とします。ファイル名の拡張子に“.LIB”を指定するとライブラリを含めてリンクすることになります。

### evafile フィールド

EVA86K上で実行可能な（ダウンロードする）ファイルの名前を指定します。指定しない場合には objectfiles フィールドの先頭で指定したファイル名の拡張子を“.EVA”に変更したものを evafile フィールドで指定したことになります。

#### 注意

EVA86K は、コンピュータで LC86K シリーズをエミュレーションする装置です。  
ビジュアルメモリでは、ビジュアルメモリシミュレータに EVA86K 用のデータを HEX 形式に変更し読み込ませ実行させます。

### Libraryfile フィールド

ライブラリの名前を指定します。ライブラリを必要としない場合には指定する必要はありません。

#### 注意

ビジュアルメモリ SDK に“ DUMMY.OBJ ”が添付されている場合は、このファイルをリンクしてください。

### 例

```
A> L86K MAIN SUB0 SUB1,TEST,TEST.LIB
```

オブジェクトモジュール“ MAIN.OBJ ” SUB0.OBJ ” SUB1.OBJ ” MAIN.OPT ” MAIN.CGR ”をリンクします。このとき、“ MAIN.OBJ ” SUB0.OBJ ” SUB1.OBJ ”を結合した際に未定義のシンボルが存在する場合には、“ TEST.LIB ”から未定義となったシンボルと同一のシンボルを探し出し、そのシンボルを含むモジュールをリンクします。

## 8.3 プロンプトによる引数の指定方法

リンケージローダへの引数をプロンプトに回答して指定する場合には、コマンドラインでは次のコマンドを入力します。

```
L86K [ option ]
```

L86K は次の行を 1 行ずつ表示することによって、必要な入力をうながします。

```
SANYO (R) LC86K series Linkage Loader Version 4.00
Copyright (c) SANYO Electric Co., Ltd. 1989-1995. All rights reserved.
Object modules.OBJ]:
EVA filenamebasefilename.EVA]:
Libraries.LIB]:
Option filenamebasefilename.OPT]:
Font filenamebasefilename.CGR]:
```

L86K は、それぞれのプロンプトに回答されるまで、次の行を表示しません。「3.1 ファイル名の指定」では、これらのプロンプトに対してファイル名を指定するときの規則が説明されています。

L86K コマンドの Option filename および、Font filename を除くプロンプトに対する応答は、L86K コマンドライン上のフィールドに相当します。次にこれらの対応を示します。

参照

L86K コマンドラインについては「3.2 コマンドラインによる引数の指定方法」を参照してください。

プロンプト	コマンドラインフィールド
Object modules	<i>objectfiles</i>
EVA filename	<i>evafile</i>
Libraries	<i>libraryfiles</i>

L86K プロンプトを使用した場合、上記の 4 項目を応答後、次の 2 項目の入力をうながします。

### Option filename

evafile の対象チップに対応するオプションファイル名を入力します。

### Font filename

フォントファイル名を入力します。

応答ラインにタイプする最後の文字がプラス記号 ( + ) である場合、プロンプトは次の行に移り、応答をタイプし続けることができます。この場合、プラス記号は、完全なファイルまたはライブラリ名、パス名、またはドライブ名の最後になくてもなりません。

注意

ビジュアルメモリでは、このファイルを指定する必要はありません。

## デフォルトの応答

カレントのプロンプトに対してデフォルトの応答を選ぶ場合、ファイル名を指定せずに( にも入力せずに ) ↵ キーを押してください。次のプロンプトが表示されます。

カレントのプロンプトと残りの全プロンプトに対してデフォルト応答を選ぶ場合、セミコロン( ; ) をタイプし、すぐに ↵ キーを押してください。セミコロンを入力するとそのリンクセッションでは残りのプロンプトのどれに対しても応答することができません。デフォルトの応答を用いたいとき、または時間を節約するためには、このオプションを使ってください。しかし、Object modules プロンプトにはデフォルトの応答がないので、このプロンプトに対してセミコロンを入力することはできません。

次に L86K プロンプトに対するデフォルトを示します。

プロンプト	デフォルト
EVA filename	Object modules プロンプトに指定される最初のオブジェクトファイル名です。“ .OBJ ” 拡張子は“ .EVA ” 拡張子に置き換えられます。
Libraries	ライブラリの検索は行ないません。
Option filename	EVA filename プロンプトに指定されるファイル名です。“ .EVA ” 拡張子は“ .OPT ” 拡張子に置き換えられます。
Font filename	Option filename プロンプトに指定されるファイル名です。“ .OPT ” 拡張子は“ .CGR ” 拡張子に置き換えられます。

## 8.4 リンク時に参照されるファイル

L86K は、リンク時に次のファイルを必ず参照します。

### 注意

ビジュアルメモリは、フラッシュメモリの読み書きを行なうシステム BIOS が ROM に実装されています。したがって、下記を特に考慮する必要はありません。

LC86K.LIB

“ LC86K.LIB ” には、システムに関する情報が格納されています。L86K は、リンク時に “ LC86K.LIB ” からリンク対象 CPU に対応するシステム情報を取り出して EVA ファイルに格納します。

また、リンク時には次のオプションファイルを参照します。

LCnn00.OPT ... nn は該当機種名に対応する2桁の数値

“ LC86K.LIB ” および “ LIBnn00.OPT ” は “ L86K.EXE ” が格納されているディレクトリと同じディレクトリ、または環境変数 PATH にて設定されたディレクトリに存在していなければいけません。

## 第 9 章

## オプションスイッチ

この章では、L86K によって行なわれるタスクを指定し制御するために、リンケージローダオプションをどのように使用するかを説明します。オプションは、リンケージローダオプション文字である「 / 」または「 - 」で始まります。

## -B=bank number

LC86800 シリーズフラッシュメモリ用 HEX ファイルの作成

- B オプションは、LC86800 シリーズにおけるフラッシュメモリ ( WORLD EXTER...NAL\_DATA ) のバンク番号を指定します。bank number には 16 進数 ( 1 ~ FF ) を記述します。ここで指定したバンク番号はデータファイルの拡張子になります。

### 注意

ビジュアルメモリでアプリケーションに解放されているフラッシュメモリは、バンク 0 のみです。  
このオプションは指定しないでください。

### 例

```
A> L86K /B=1 SAMPLE;
```

この場合、作成されるデータファイルは“ SAMPLE.H01 ”となります。

## -C=address

CSEG ローディングアドレスの指定方法

- C オプションは、このオプションの直後に記述されたオブジェクトモジュールに対して有効で、コードセグメント部のローディングアドレスを指定します。address には 16 進数を記述します。

このオプションを省略した場合、L86K はそのオブジェクトモジュールのコードセグメント部を任意の位置にロードします。

## -D=address

DSEG ローディングアドレスの指定方法

- D オプションは、このオプションの直後に記述されたオブジェクトモジュールに対して有効で、データセグメント部のローディングアドレスを指定します。address には 16 進数を記述します。

このオプションを省略した場合、L86K はそのオブジェクトモジュールのデータセグメント部を任意の位置にロードします。

---

## -E

### DSEG のアドレス多重定義を許す

- E オプションを指定した場合、DSEG で複数のシンボルを同一アドレスに定義してもエラーとしません。

---

## -I

### 大文字と小文字を区別しない

- デフォルトでは、L86K は大文字と小文字を区別していますが、この - I オプションを指定すると大文字と小文字を区別しくくなります。

---

## -P

### ローディングマップの作成

- P オプションは、リンク結果のマップ (各セグメントごとのリンク状況やパブリックシンボルの配置状態のリスト) を書き込んだファイルを作成します。このマップファイルのファイル名は、コマンドラインあるいはプロンプトにおける EVA file フィールドにて指定した名前の拡張子を ".MAP" に置き換えたものとなります。ただし、リンクの続行が不可能となるような致命的なエラーが発生した場合は、マップファイルは作成されません。

ビジュアルメモリシミュレータの逆アセンブル機能を利用し、アドレスのラベルを表示させる場合には、マップファイルが必要になります。

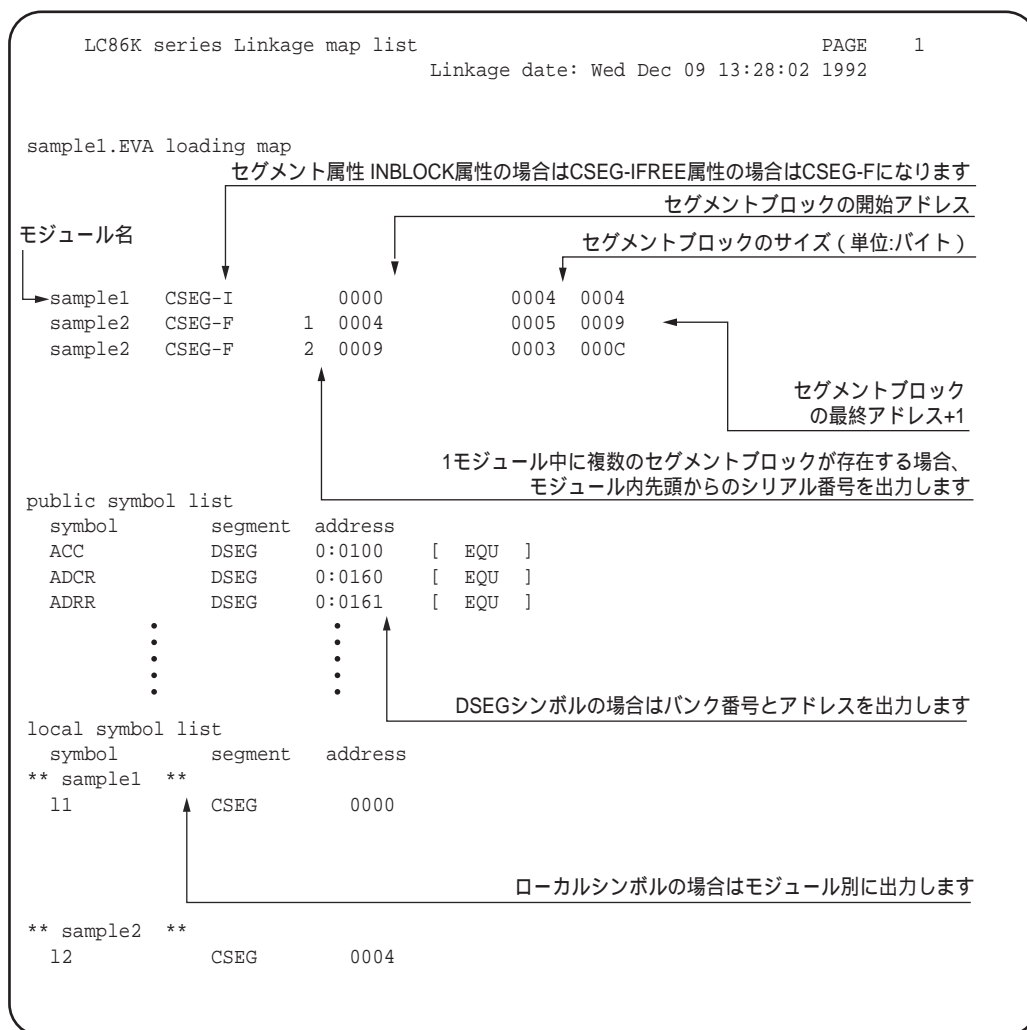
---

## -L

### ローカルシンボルリストの作成

- L オプションは - P オプションと組み合わせたときのみ有効で、マップファイルに各モジュールごとのローカルシンボルのリストを付加します。

ビジュアルメモリシミュレータの逆アセンブル機能を利用し、アドレスのラベルを表示させる場合には、マップファイルが必要になります。



## -W

### オペランドデータに関する警告メッセージの表示指定

- W オプションは、JMP などのオペランドに記述された値が有効範囲 ( JMP ならばオペランドに許される値は 12 ビット分ですから 0 ~ 4095 です ) 以外である場合に警告メッセージを表示します ( 命令コードに格納される値は有効ビットのみで、オーバーフロー部分のビットは切り捨てられます )。

## -A , -F , -O , -R

### CSEG FREE ブロックの最適化ローディング

通常 L86K は、オブジェクトモジュール中に記述された CSEG 部および、コマンドライン上で指定された順序にしたがってリンク ( 割り付け ) をしますが、このとき L86K は実行可能ファイルのセグメントデータ ( コードセグメント ) を 4096 バイト境界にしたがって境界合わせを行いません。このため、コードセグメント空間にいくつかの空き領域が生じる場合があります。

L86K には、上記のような空き領域の発生を最小に抑え、メモリを効率良く使用できるように各セグメントブロックを最適な位置に割り付けるために 4 種類の配置機能があります。

次に各ローディング方法について記述します。

#### -A オプション

- A オプションはリンク対象のすべてのコードセグメントブロックをサイズの大きい順にローディングします。このとき 4096 バイト境界にかかるセグメントブロックが INBLOCK 属性の場合は境界合わせを行ない、FREE 属性の場合は境界合わせを行わずそのまま配置します。

#### -F オプション

- F オプションは - A と組み合わせたときのみ有効で、INBLOCK 属性のコードセグメントブロックをコマンドライン上で指定された順序にしたがってリンクした後、前述の理由により生じた空き領域に対して FREE 属性のコードセグメントブロックを割り付けるようにします ( FREE 属性のコードセグメントブロックを割り付ける領域がない場合は最終アドレスよりサイズの大きい順に割り付けます )。

#### -O オプション

- O オプションは - A と組み合わせたときのみ有効で、まず INBLOCK 属性のコードセグメントをサイズの大きい順にリンクした後、空き領域に対して FREE 属性のコードセグメントブロックを割り付けるようにします ( FREE 属性のコードセグメントブロックを割り付ける領域がない場合は最終アドレスよりサイズの大きい順に割り付けます )。

#### -R オプション

- R オプションは - A と組み合わせたときのみ有効で、まず INBLOCK 属性のコードセグメントをサイズの大きい順にリンクした後、空き領域に対して FREE 属性のコードセグメントブロックを割り付けるようにします。このとき、2 つの連続した 4096 バイト空間にそれぞれ空き領域があれば後方の INBLOCK 属性のコードセグメントを再配置して前後の空き領域を結合し、その領域に対して FREE 属性のコードセグメントを割り付けます ( FREE 属性のコードセグメントブロックを割り付ける領域がない場合は最終アドレスよりサイズの大きい順に割り付けます )。

---

## -S

### シンボルのソート処理に関する指定

---

すべてのリンク対象オブジェクトファイル中 ( “ LC86K.LIB ” 中の該当 SFR 定義も含む ) のパブリックシンボル定義数の合計が 8192 個を超える場合、またはリンク対象オブジェクトファイル中のローカルシンボル定義数が 8192 個を超える場合、シンボルソート処理に関する処理速度が低下するため、ソート処理の経過を表わすメッセージを表示して処理を続行します。

```
Public(Local) symbol table: Sorting .. nn / nn blocks  
Public(Local) symbol table: Sorting(merge) .. nn %
```

しかし、- S オプションを指定した場合はシンボルソート処理を行わずに次のメッセージを表示してリンク処理を中断します。

```
* * Link Error ,Public symbol table overflow :nn symbols
```

このとき表示されるシンボル数は 8192 個を超える個数です。



## 第 10 章

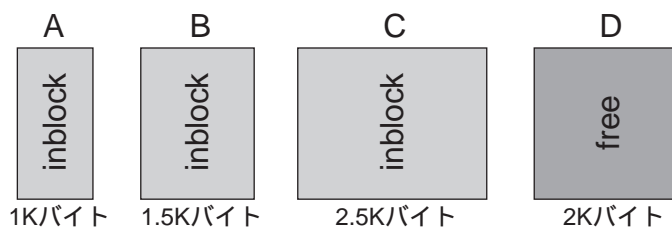
# オブジェクトの配置

「第 9 章 オプションスイッチ」の「-A, -F, -O, -R CSEG FREE ブロックの最適化ローディング」での説明のように、リンク時に最適化の指定をするとオブジェクトの配置が通常のリンク結果と比べ異なります。最適化ローディングには、4 種類の方法がありますが、次に各最適化指定時のオブジェクト配置について説明します。

## 10.1 -A オプション

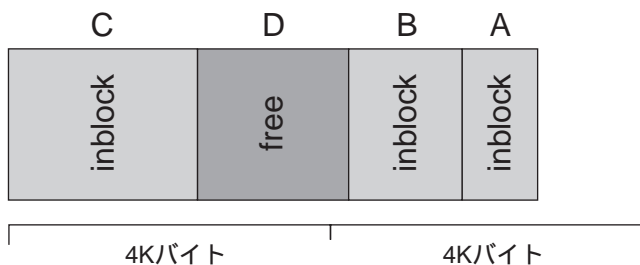
CSEG には INBLOCK (4096 バイトの境界内に配置する) 指定と FREE (4096 バイトの境界に関係なく配置する) 指定の 2 とおりの配置指定がありますが、-A を指定した場合は INBLOCK/FREE に関わらずサイズの大きいものから順に最適な位置へ配置されます (このとき INBLOCK 指定のセグメントは 4K バイト境界に対して境界合わせを行ない、FREE 指定のセグメントは境界合わせを行ないません)。

たとえば、次のようなオブジェクト群があります。

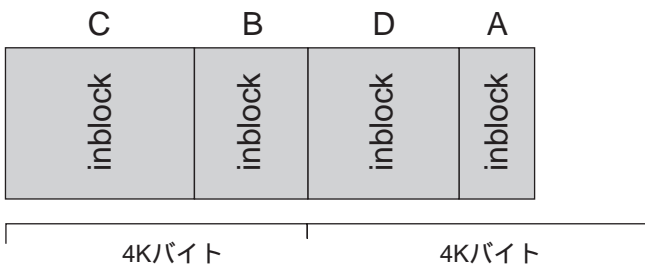


これを -A を指定してリンクを行なうと INBLOCK/FREE 指定のセグメントブロックをサイズの大きい順に最適な位置へ配置されますので次のようになります。

L86K -A A B C D;



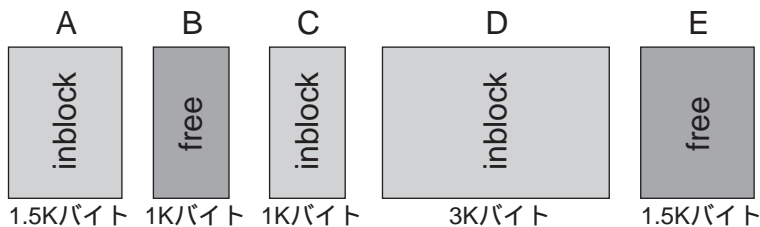
この例の場合、4096 バイトの境界線上に配置されるセグメントが FREE 属性であるためオブジェクト D に対する境界合わせは行なわれません。オブジェクト D が INBLOCK 属性だった場合は次のようになります。



## 10.2 -A -F オプション

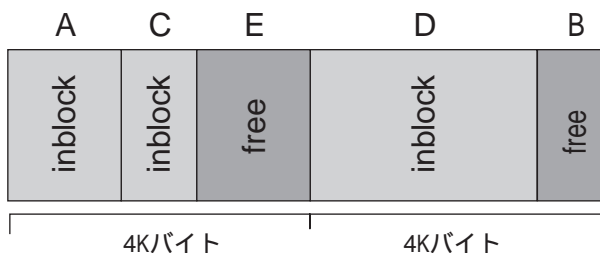
-A-F を指定した場合は、INBLOCK 属性のセグメントブロックをコマンドライン記述順に配置された後、FREE 属性のセグメントブロックをサイズの大きい順に最適な位置に配置されます。

次のようなオブジェクト群があります。



これを-A-F を指定してリンクを行なうと、コマンドライン順に A、C、D が配置され (D は 4096 バイトの境界線に合わせられます)、E、B の順に最適な位置へ配置されます。

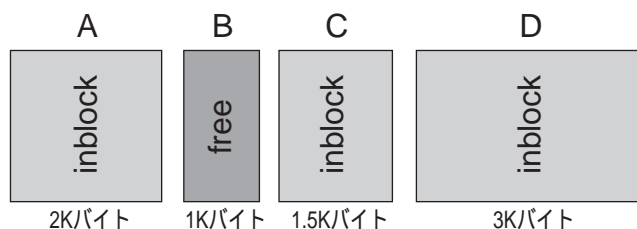
```
L86K -A-F A B C D E;
```



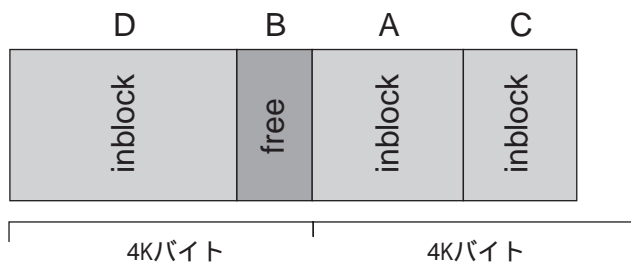
## 10.3 -A -O オプション

-A-O を指定した場合は、INBLOCK 属性のセグメントブロックをサイズの大きいものから最適な位置に配置され、その後 FREE 属性のセグメントブロックをサイズの大きいものから順に最適な位置へ配置されます。

次のようなオブジェクト群があります。



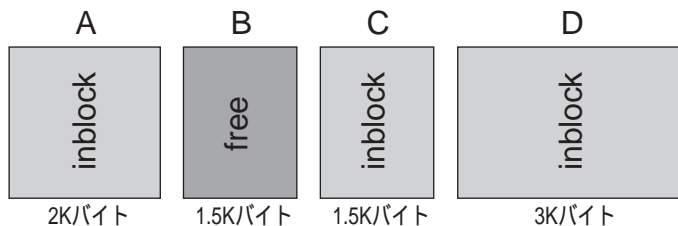
これを -A-O を指定してリンクを行なうと、D、A、C の順に最適な位置へ配置された後、B を最適な位置へ配置されます。



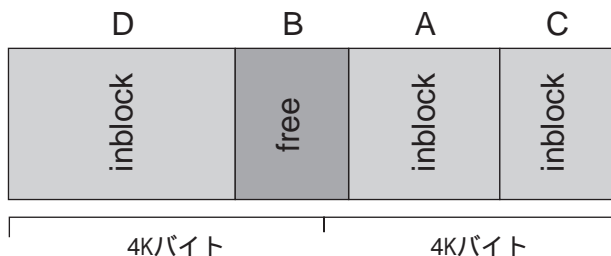
## 10.4 -A -R オプション

-A-R を指定した場合は、INBLOCK 属性のセグメントブロックをサイズの大きいものから順に最適な位置に配置された後、連続する 2 つの 4096 バイト境界内の後方のセグメントを再配置して前後の空き領域を結合し、そこへ FREE 属性のセグメントブロックが配置されます。

次のようなオブジェクト群があります。



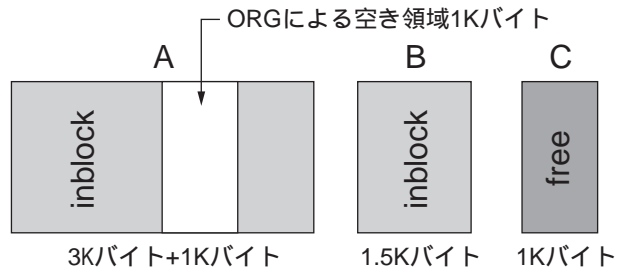
これを -A-R を指定してリンクを行なうと、D、A、C の順に配置された後、A、C をそれぞれ境界内後方に再配置し、D/A 間の空き領域に B が配置されます。



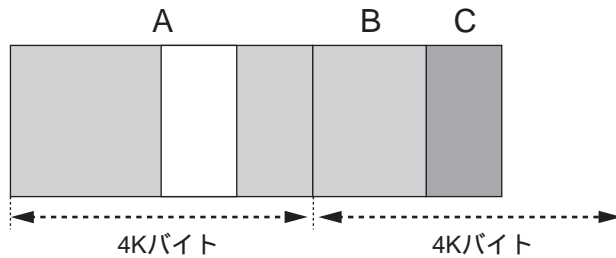
また、すべての最適化においてオブジェクト中に ORG 疑似命令により空き領域が生じる場合、その領域もセグメントブロックの配置対象領域となります。

この場合の例を次に示します。

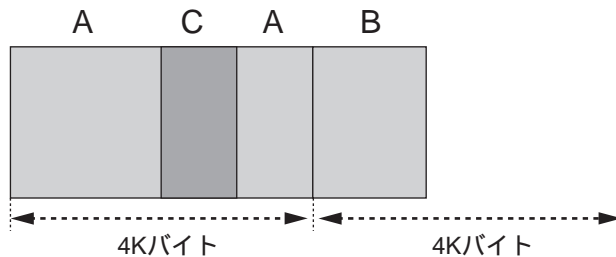
次のようなオブジェクト群があります



これを最適化をしない通常のリンクを行なうと次のようになります。



これを最適化の指定をしてリンクを行なうと A なるセグメントブロック中に ORG による空き領域があるので、次のようになります。



また、最適化指定と合わせてローディングアドレス指定（-C オプション）を行なった場合で、そのファイルの先頭セグメントが FREE 指定だった場合は、そのセグメントブロックについてのみローディングアドレス指定にしたいがいます（以降の FREE 指定ブロックは最適化の対象となります）。

## 11.1 致命的なエラー

リンク実行中致命的なエラーが検出された場合、L86K はディスプレイ上にメッセージを表示し処理を中断します。次に L86K が表示するエラーメッセージを示します。

### Chip name unmatched

異なるチップのオブジェクトモジュールをリンクしようとしてしました。

### Data file specified

EVA ファイル作成にデータファイルを指定しました。

### Data segment size exceeds

RAM サイズを越えて DSEG オブジェクトをリンクしようとしてしました。

### External undefine symbol

外部参照シンボルが見つかりません。

### Illegal bank number specified

フラッシュメモリのバンク番号の指定に誤りがあります。

### Illegal file format

指定したファイルが LC86K シリーズのものではありません。

### Illegal option specified

不当なオプション指定があります。

### Internal module not specified

リンクで内部プログラムファイルの指定がありません。

### Loading address multiple assignment

同一のアドレスに異なるオブジェクトを割り付けようとしてしました。

### No such file or directory

指定したファイルが存在しません。

#### Program file specified

フラッシュメモリ用データファイル(バンク1)の作成にプログラムファイルを指定しました。

#### Public symbol multiple define

パブリックシンボルが多重定義です。

#### Segment size exceeds

セグメントサイズを越えてリンクしようとしてしました。

#### WORLD attribute unmatched

WORLD INTERNAL や WORLD EXTERNAL の属性を持つプログラムファイルと WORLD EXTERNAL\_DATA の属性を持つデータファイルが混在しています。

## 11.2 致命的ではないエラー

リンク実行中致命的でないエラーが検出された場合、ディスプレイ上にメッセージを表示し処理を続行します。次に L86K が表示する警告メッセージを示します。

#### Cannot access file : LC86K.LIB

予約語を登録してあるライブラリ“ LC86K.LIB ”が存在しません。“ LC86K.LIB ”はチップごとの予約語が登録してあるライブラリで、カレントもしくは環境変数 PATH に設定されているディレクトリに存在する必要があります。

#### Module not in library

“ LC86K.LIB ”中に対象チップの予約語が登録されていません。

#### Operand data overflow

オペランドフィールドに記述した値が所定の範囲(ステートメントにより異なります)を越えています。

#### Operand data type mismatch

オペランドフィールドに不当なセグメントのシンボルを指定しています。



---

## 第 4 部

---

# ライブラリ マネージャ編

---

ここでは、ライブラリマネージャ(ライブラリアン)である“LIB86K.EXE”のコマンドラインオプションや、エラーメッセージなどについてを説明します。

---



## 第 12 章

# プログラムの起動

LIB86K を起動し、その実行に必要な情報を LIB86K に引き渡すには、次の 2 種類の方法があります。

( 1 ) コマンドラインにてすべての情報を LIB86K に引き渡す方法

( 2 ) LIB86K が表示するプロンプトに応答してすべての情報を引き渡す方法

また、LIB86K を強制的に終了させるには起動の方法にかかわらず、次のキーを使います。

機種名	キー
PC/AT 互換機の場合	<b>Ctrl</b> + <b>C</b> キーまたは <b>Ctrl</b> + <b>PauseBreak</b> キー
PC-9800 シリーズの場合	<b>CTRL</b> + <b>C</b> キーまたは <b>STOP</b> キー

## 12.1 ファイル名の指定

LIB86K を起動するコマンドライン上で指定するファイル名、または LIB86K のコマンドプロンプトに対する応答として与えるファイル名には、大文字と小文字をどのように組み合わせ使用してもかまいません。たとえば、次の 3 種類のファイル名は同等のものとしてあつかわれます。

```
sample.obj
SAmpLE.OBJ
SAMPLE.OBJ
```

また、ファイル名を拡張子なしで指定する場合、LIB86K は次のデフォルトのファイル名拡張子を使用します。

ファイル形式	デフォルトの拡張子
ライブラリファイル	.LIB
オブジェクトファイル	.OBJ
リストファイル	なし

## 12.2 コマンドラインによる引数の指定方法

```
LIB86K [ option ] oldlibrary [ commands ][ , [ listfile ][ , [ newlibrary ] ] ] ; ]
```

## option フィールド

*option* フィールドに指定できるオプションは / ? のみです。

## oldlibrary フィールド

処理対象となるライブラリファイルを指定します。このフィールドを省略することはできません。ライブラリファイルの拡張子が “.LIB ” の場合には省略することができます。しかし、ユーザーのライブラリファイル拡張子が “.LIB ” 以外の場合には、拡張子を省略することはできません。このライブラリファイルにはデフォルトは存在しないため指定されなかった場合には、エラーメッセージを出力します。また、存在しないファイル名を指定した場合には次のプロンプトを表示します。

```
Library file does not exist. Create? (y/n)
```

新規ライブラリを作成する場合は、Y を入力します。Y 以外を入力すると処理を中断して OS に戻ります。

既存のライブラリファイルにセミコロンを付けただけのものを入力すると、ライブラリの整合性のチェックが行われます。整合性のチェックとは、ライブラリ中のすべてのモジュールが使用できるかどうかのチェックです。もし、エラーが存在する場合には、エラーメッセージを表示されます。

## commands フィールド

*commands* には、+ , - , - + , \* , - \* のようなコマンド記号があり、これらを入力してプログラムの動作指示を行うことができます。指定できるオブジェクトファイル名やモジュール名は、コマンド 1 つにつき 1 つ指定して複数の操作を行うことができます。コマンドを省略するとライブラリファイルに対して変更を行いません。

コマンド	意味
+	: モジュール追加のコマンド記号です。コマンド記号の直後に記述されたオブジェクトファイル中のモジュールを対象として <i>oldlibrary</i> の最後に追加します。ライブラリ同士の結合を行うことはできません。
-	: モジュール削除のコマンド記号です。 <i>oldlibrary</i> よりコマンド記号の直後に指定されたモジュールを削除します。
- +	: モジュール置き換えのコマンド記号です。コマンド記号の直後に記述されたオブジェクトファイル中のモジュールを対象として <i>oldlibrary</i> 中のモジュールを置き換えます。モジュールの置き換えは今まであったモジュールを削除して、ライブラリの最後に同名のモジュールを追加します。
*	: モジュールコピーのコマンド記号です。コマンドの直後に記述されたモジュールを <i>oldlibrary</i> より探し、同名のオブジェクトファイルに内容を書き込みます。コピーされたモジュールは、 <i>oldlibrary</i> 内にも残ります。
- *	: モジュール移動のコマンド記号です。コマンド記号の直後に記述されたモジュールを <i>oldlibrary</i> より探し、同名のオブジェクトファイルに内容を書き込みます。モジュールコピーと同じ操作を行いますが、書き出されたモジュールが <i>oldlibrary</i> 内に残りません。

### listfile フィールド

*listfile* には、ライブラリ中のパブリックシンボルリスト、外部参照シンボルリスト、モジュール名リストを出力するファイルを指定します。省略した場合は、標準出力にデータを表示します。

### newlibrary フィールド

*newlibrary* には、出力対象となるライブラリ名を指定します。指定のない場合、処理以前の *oldlibrary* の拡張子を “.BAK ”に変更し、*oldlibrary* を *newlibrary* としてデータを保存します。

## 12.2.1 オプション指定

/ ? オプションを指定するとヘルプメッセージが画面に表示されます。

## 12.2.2 コマンドライン実行例

### 例 1

```
LIB86K HOME-+ROM;
```

この例では、HOME というライブラリより ROM というモジュールを削除して、ライブラリの最後にオブジェクトファイル“ ROM.OBJ ”を追加します。

### 例 2

```
LIB86K HOME-ROM+ROM;  
LIB86K HOME+ROM-ROM;
```

例 1 の場合は、HOME というライブラリより ROM というモジュールを削除してから“ ROM.OBJ ”というオブジェクトファイルを追加します。しかし、例 2 の場合は、HOME に“ ROM.OBJ ”を追加してから ROM モジュールを削除します。したがって、上段の場合ライブラリ内に ROM モジュールが残りますが、下段の場合ライブラリ内には残りません。これは、コマンド記号が指定された順序で処理を行っているからです。

### 例 3

```
LIB86K HOME,LCROSS.PUB
```

“ HOME.LIB ”の整合性のチェックを行った後、“ LCROSS.PUB ”という名前のクロスリファレンスファイルを生成します。

### 例 4

```
LIB86K FIRST -*STUFF*MORE,,SECOND
```

モジュール STUFF をライブラリ“ FIRST.LIB ”から“ STUFF.OBJ ”というファイルに書き出し、モジュール STUFF をライブラリより削除します。モジュール MORE は、ライ

ブラリから“ MORE.OBJ ”に書き出され、ライブラリ内に残ります。書き換えられたライブラリは、“ SECOND.LIB ”となり“ FIRST.LIB ”より STUFF モジュールだけが削除されたライブラリとなります。

## 12.3 プロンプトによる操作

コマンドラインより次のように LIB86K のみを入力すると、プロンプトにしたがって各フィールドを1つずつ入力することができます。

```
LIB86K [ option ]
```

このとき LIB86K は、次のフィールドを1つずつ表示します。

```
Library name:
Operations:
List file:
Output library:
```

LIB86K はプロンプトを1つ表示すると、ユーザーからの入力を待ちます。入力が完了すると次のプロンプトを表示して再び入力待ちとなります。

プロンプトに対する応答は、コマンドラインの各フィールドに対応しています。次の表は、コマンドラインのフィールドとの対応を表しています。

プロンプト	コマンドラインフィールド
Library name	<i>oldlibrary</i> フィールドに相当します。ファイル名の後にセミコロン( ; )を入力するとライブラリの整合性をチェックします。
Operations	<i>command</i> フィールドに相当します。
List file	<i>listfile</i> フィールドに相当します。
Output library	<i>newlibrary</i> フィールドに相当します。

### 12.3.1 プロンプトラインの拡張

Operations プロンプトにおいて、入力の最後でアンパサンド( & )を入力すると、Operations プロンプトをもう1度表示し、さらに処理を追加指定することができます。

### 12.3.2 デフォルトの応答

Library name プロンプト以外の項目では、デフォルトの値が設定されます。デフォルトの設定は、プロンプトに対してセミコロンまたは ↵ キーを入力することで行うことができます。次に各プロンプトに対応したデフォルトの値を示します。

プロンプト	デフォルト値
Operations	ライブラリファイルに対して何も変更を行いません。
List file	リストファイルの出力先に標準出力を選択します。リストファイルの生成は行いません。
Output library	出力するライブラリファイル名をオリジナルと同一名にします。

この章では、エラーメッセージとその意味について記述します。

#### cannot access file

LIB86K が指定されたファイルをオープンすることができません。

#### cannot create new library

ディスクまたはルートディレクトリがいっぱいであるか、ライブラリファイルがすでに読み込み専用でプロテクトされて存在しています。

#### cannot rename old library

“.BAK ”バージョンが読み込み専用でプロテクトされているため、LIB86K が古いライブラリを“.BAK ”拡張子を持つように名前を付け直すことができません。

#### comma or newline missing

コマンドラインで、カンマまたは $\backslash$ があるべきところに存在しません。

#### error Reading from library

LIB86K が指定されたライブラリファイルよりデータをリードすることができません。

#### error writing to new library

ディスクまたはルートディレクトリがいっぱいです。

#### insufficient memory

LIB86K 実行に必要なメモリを確保することができません。

#### interrupted by user

ユーザーからのキー操作によって LIB86K を中断しました。

#### invalid library header

入力ライブラリファイルが、無効な形式を持っています。

#### module not in library ; ignored

置き換えを指定されたモジュールが、ライブラリ中に存在しません。

#### output-library specification ignored

新しいライブラリ名に加え、出力ライブラリが指定されています。

syntax error : illegal file specification

マイナス記号( - )のようなコマンドオペレータが、モジュール名なしで指定されています。

この章では、クロスリファレンスリストのフォーマットについて記述します。

LC86K series Library Analysis List

PAGE 1

Tue Feb 18 13:56:12 1992

Number of Module count: 2 Library create date: Wed Oct 16 15:34:53 1991  
Library update date: Tue Feb 18 10:55:23 1992

Including Modules: 1 2

-----  
Module name: 1 Source name: 1.ASM  
Assembler name: SASM 1.0 Assembly date: Tue Oct 22 15:54:43 1991

Target chip name: LC868700

Including Public symbols:

Including External symbols:

test sample label1

-----  
Module name: 2 Source name: 2.ASM  
Assembler name: SASM 1.0 Assembly date: Tue Oct 22 15:54:43 1991

Target chip name: LC868700

Including Public symbols:

Including External symbols:

label1 label2 label3





---

## 第 5 部

---

# Eva to Hex 編

---

---

ここでは、リンカによって出力された EVA 形式のファイルをビジュアルメモリやビジュアルメモリシミュレータに読み込ませる HEX 形式のファイルに変換する“ E2H86K.EXE ”のコマンドラインオプションや、エラーメッセージなどについてを説明します。



## 第 15 章

# プログラムの起動

## 15.1 ファイル名の指定

E2H86K コマンドライン上で指定するファイル名には、大文字と小文字をどのように組み合わせ使用してもかまいません。たとえば、次の 3 種類のファイル名は同等のものとしてあつかわれます。

```
sample.eva
SAmp1E.EVA
SAMPLE.EVA
```

また、ファイル名を拡張子なしで指定する場合、E2H86K は次のデフォルトのファイル名拡張子を使用します。

ファイル形式	デフォルトの拡張子
EVA ファイル	.EVA
HEX ファイル	.HEX

## 15.2 引数の指定方法

```
E2H86K [ option ] EVA_filename [ HEX_filename ]
```

### option フィールド

「15.3 オプション指定」で説明されているオプションを指定します。option はコマンド名の次に記述します。

### EVA\_filename フィールド

デバッグの終了したファイル( 拡張子が“ .EVA ”であるファイル ) の名前を指定します ( 以下、EVA ファイル )。

### HEX\_filename フィールド

Intellec HEX フォーマットのファイルの名前を指定します ( 以下、HEX ファイル )。HEX\_filename が省略された場合には、EVA\_filename と同じファイル名になります。フラッシュメモリ用データファイルを変換すると、拡張子が“ .H00 ”になります。

**注意**

プロンプトによる操作はできません。

**起動例 1**

```
A>E2H86K PROG012
```

EVA ファイル                      HEX ファイル    ,    フラッシュメモリ用 HEX ファイル  
“ PROG012.EVA ”                      “ PROG012.HEX ”, “ PROG012.H00 ”

**起動例 2**

```
A>E2H86K
```

次のメッセージ (簡易ヘルプ) を表示します。

```
SANYO LC86000 Series EVA-file to HEX-file generator V1.00A  
Copyright (C) SANYO Electric Co.,Ltd. 1992
```

```
Usage: e2h86k options] EVA_filename HEX_filename]  
Options: /I ... information on/off (default: on )
```

## 15.3 オプション指定

オプションは「 / 」で始まります。

**注意**

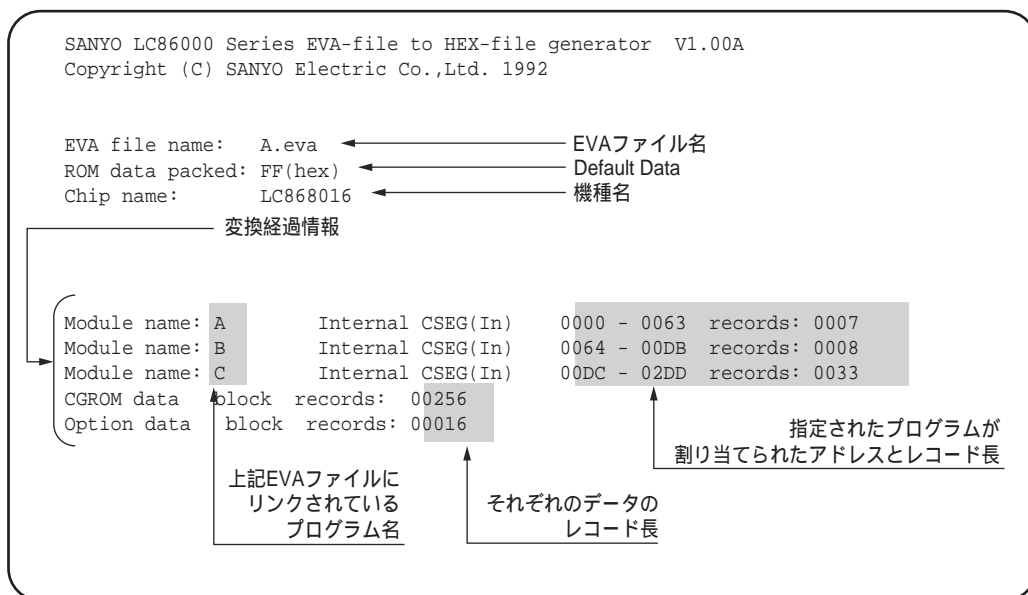
「 - 」は使用できません。

### /I

変換中の表示情報を抑止

/I オプションは、EVA ファイルを HEX ファイルに変換している間にディスプレイに変換経過情報が表示されないようにします。このオプションを指定しないと、変換中に変換経過情報が表示されます。

## 表示例





## 16.1 致命的なエラー

E2H86K 実行中に致命的なエラーが検出された場合、E2H86K はディスプレイ上にメッセージを表示し処理を中断します。次に E2H86K が表示するエラーメッセージを示します。

エラーメッセージ : Fatal error : .....messages.....

‘ filename ’ File not close.  
filename ファイルをクローズできません。

‘ filename ’ File not create.  
filename ファイルを作成できません。

‘ filename ’ File not open.  
filename ファイルが見つかりません。

‘ filename ’ not EVA file format.  
filename ファイルは EVA フォーマットファイルではありません。

‘ filename ’ user disk full.  
filename を書き込み中にディスクがいっぱいになりました。

Chipname undefined.  
EVA ファイル内の機種名が間違っています。

ROM size over. ( ROM size : XXXX )  
プログラムサイズが ROM サイズを超えています。

Tablename allocation error.  
メモリ不足のため、tablename のメモリ確保に失敗しました。





---

## 第 6 部

---

# MAKE 編

---

---

ここでは、更新されたソースファイルなどをチェックし、必要最低限なビルド作業を行なわせる「プログラム保守ユーティリティ」"MAKE.EXE" のコマンドラインオプションや、エラーメッセージなどについてを説明します。



## 第 17 章

## MAKE の概要

MAKE は、プログラム開発を自動化します。MAKE は、ソースファイル (ASM)、オブジェクトファイル (OBJ)、オプションファイル (OPT)、CGROM ファイル (CGR) などが更新された場合、実行可能ファイル (EVA) を自動的に更新します。

MAKE を実行するには、MAKE が必要とする情報を含むファイル (MAKE ファイル) を必要とします。MAKE ファイルは、さまざまな命令で構成されるテキストファイルであり、この命令にしたがってプログラムをビルドします。これらの命令は、記述ブロック、マクロ、ディレクティブ、推論規則で構成されます。記述ブロックには、ターゲットとその依存ファイル、ターゲットをビルドするためのコマンドが記述されます。MAKE は、ターゲットのタイムスタンプと依存ファイルのタイムスタンプを比較し、ターゲットの最終更新日時以降に依存ファイルが更新されている場合、記述ブロックのコマンドを実行しターゲットをビルドします。

## 17.1 MAKE の実行

MAKE を起動するには、次のコマンドを入力します。

```
MAKE [options] [/f makefile] [/x errorfile] [targets]
```

### options フィールド

*options* フィールドには、MAKE のオプションを指定します。MAKE のオプションについては「17.1.2 コマンドラインオプション」を参照してください。

### makefile

*makefile* には、MAKE ファイルの名称を指定します。この場合 ' / f ' と ' *makefile* ' の間にはスペースが必要です。MAKE ファイル名が ' MAKEFILE ' の場合、ファイル名を省略できます。

### errorfile

*errorfile* には、エラー出力ファイルの名称を指定します。この場合 ' / x ' および ' *errorfile* ' の間にはスペースが必要です。通常、エラー出力は、ディスプレイですがこのオプションを指定することにより、ファイルへの出力ができます。

### targets フィールド

*targets* フィールドには、ビルドするターゲットを指定します。コマンドラインで指定されたターゲットがビルドされます。ターゲットの指定がなく、.TARGET ディレクティブの指定もない場合には、MAKE ファイル中の最初に指定されているターゲットがビルドされます。

---

## 17.1.1 ビルドの優先順位

---

MAKE によるビルド処理は、次の優先順位にしたがいます。

- (1) /f オプションを指定した場合、MAKE は指定された MAKE ファイルをカレントディレクトリまたは指定のディレクトリより検索します。ファイルが見つからない場合は、処理を中止します。
- (2) /f オプションを指定しない場合は、MAKE はファイル名が 'MAKEFILE' という MAKE ファイルをカレントディレクトリで探します。
- (3) /r オプションを指定しない場合、MAKE はメイクルールファイルをカレントディレクトリより探します。カレントディレクトリにない場合は、MAKE 本体のあるディレクトリを探します。メイクルールファイルが見つからない場合は、処理を中止します。

---

## 17.1.2 コマンドラインオプション

---

MAKE の処理を制御するために、次のオプションがサポートされています。オプションは、大文字でも小文字でもかまいません。また、指定する場合は、オプション名の前に ' / ' を付加します。

---

### /E

外部マクロを優先する

マクロ参照において外部マクロを優先します。デフォルトは、内部マクロ優先です。

---

### /I

終了コードを無視して最後まで実行

MAKE ファイルに記述されているコマンドからの終了コードを無視します。エラーが発生しても MAKE は処理を中断せず MAKE ファイルの最後まで実行します。MAKE ファイル中で部分的に終了コードを無視するには、ハイフン ( - ) コマンド修飾子または IGNORE ディレクティブを使用します。

---

### /N

コマンドを実行せず経過のみを表示

MAKE ファイルのビルド時に実行されるコマンドを表示するのみで、コマンドの実行は行いません。MAKE ファイルのデバッグや更新すべき古いターゲットファイルをチェックする場合などに有効です。

---

### /R

ルールファイルの読み込みを禁止

このオプションを指定すると、MAKE はメイクルールファイルを読み込みません。デフォルトは、読み込みとなっています。

---

### /S

コマンドの表示を抑止

MAKE ファイルに記述されているコマンドを表示しません。MAKE ファイル中で部分的

にコマンド表示を禁止するには、アットマーク ( @ ) コマンド修飾子または SILENT ディレクティブを使用します。

## /? ヘルプの表示

MAKE のコマンドライン構文規則の要約を表示します。

## 17.2 MAKE 記述ファイル

MAKE 記述ファイルは、テキストファイル形式です。作成する場合は、テキストエディタを使用します。通常、MAKE ファイル名は MAKEFILE としますが、複数の MAKE ファイルが存在する場合など、ユニークな名称を付加することができます。MAKE 記述ファイルは、記述ブロック、マクロ、推論規則、およびディレクティブで構成されています。

### 17.2.1 記述ブロック

記述ブロックは、MAKE ファイルの中心となる部分です。次に MAKE の記述ブロックを示します。

ターゲット部	:	依存部	
sample.eva	:	sample.obj sample.opt sample.cgr	依存記述行
186k/p	:	sample;	コマンド記述ブロック
copy	:	sample.eva c:¥myprog¥	コマンド記述ブロック

#### 依存記述行

記述ブロックは、「依存記述行」で始まります。依存記述行には、コロン ( : ) で区切られた 2 つの部分があります。コロンの左側の部分を「ターゲット」と呼び、更新されるべきファイルを記述します。コロンの右側の部分を「ソース」と呼び、ターゲットを更新するためのソースになるファイルを記述します。上記の例では、“ sample.eva ”がターゲット、“ sample.obj ” “ sample.opt ” sample.cgr ”がソースとなります。依存記述行の先頭には、空白 ( スペース またはタブ ) を置いてはいけません。また、ターゲットおよびソースは、それぞれ空白で区切るにより、複数個記述することができます。

依存記述行は、「ターゲットがソースよりも古いか、ターゲットが存在しない場合に、更新処理を行う」ことを表します。ただし、ソースが指定されていない場合は、例外として更新処理が行われます。依存記述ブロックが、複数ブロック記述されている場合は、MAKE ファイル中で 1 番最初に現れる依存記述部のターゲットを最終ターゲットとするので、EVA ファイルに関する依存記述ブロックは最初に記述するようにしてください ( MAKE 起動時にターゲットの指定が可能 )。最終ターゲットは、.TARGET ディレクティブにより、デフォルトターゲットファイルの拡張子を設定できます。依存記述行の記述が長い場合は、各行の終わりに円記号 ( ¥ ) を付加すると、MAKE は次の行を継続行と解釈します。

#### 継続行の使用例

```
sample.eva : sample.obj ¥
            sample.opt ¥
```

```

        sample.cgr
    186k/p sample;
sample.obj : sample.asm
    m86k sample;

```

## コマンドブロック

コマンド行は、依存記述行の直後に記述します。依存記述行に続く先頭に空白の置かれた行の集まりを、コマンドブロックと呼びます。MAKE は、先頭の空白の有無によって依存記述行とコマンド行を区別します。コマンド行には、依存記述部のターゲットを更新するためのコマンドを記述します。コマンドブロックが複数行からなる場合は、上から順に 1 行ずつ実行されます。コマンド行では、DOS コマンドが記述できます。また、内部コマンド ( dir など ) も記述できます。長いコマンド行は、各行の終わりに円記号 ( ¥ ) を付加すると、MAKE は次の行を継続行と解釈します。MAKE は、継続行を 1 コマンド行として DOS に渡しますが DOS のコマンド行の長さの制限 ( 最大 127 文字 ) を超えることはできません。

### 注意

MAKE 実行中は、MAKE 本体およびワークエリアとして 100KB 程度消費するので、コマンド実行時、メモリ不足が発生する可能性があります。

## コマンド修飾子

コマンド修飾子を用いることにより、コマンドブロックの制御をより細かく制御可能になります。1 つのコマンドに複数のコマンド修飾子を使用できます。コマンド修飾子は、修飾するコマンドの前に付加します。

### @ command

MAKE が実行中のコマンドのコマンド行をディスプレイに表示しません。ただし、コマンドの結果 ( コマンドが出力する ) は、表示されます。関連する機能として、/ S オプションスイッチ、.SILENT ディレクティブがあります。

/ S オプションは、MAKE ファイル全体に対してコマンド表示を禁止します。  
.SILENT ディレクティブは、コマンド非表示モードになります。

### -command

コマンドのエラーチェックをしません。MAKE は、コマンドが返す終了コードが、0 以外の時、処理を中断します。- 修飾子を用いると、MAKE はそのコマンド行の終了コードを無視します。関連する機能として、/ I オプションスイッチ、.IGNORE ディレクティブがあります。

/ I オプションは、MAKE ファイル全体に対して、コマンドの終了コードを無視します。  
.IGNORE ディレクティブは、コマンドの終了コードを無視するモードになります。

## コマンド修飾子の使用例

```
sample.eva : sample.obj subr.obj sample.opt sample.cgr
            @echo sample.eva creating now.          エコー表示
            186k/p sample+subr;
sample.obj : sample.asm
            -m86k sample;                          アセンブルエラーを無視
subr.obj   : subr.asm
            -m86k subr;                             アセンブルエラーを無視
```

## 擬似ターゲット

擬似ターゲットは、ターゲットの指定の1つですが、ファイルを指定するのではなく単にラベルという感覚でターゲットを指定することをいいます。この場合 MAKE は、擬似ターゲットを、常に存在しない更新の必要なファイルと解釈します。擬似ターゲットとして指定する場合は、カレントディレクトリに同一名のファイルが、存在しないことを確認してください。

### 擬似ターゲットの使用例

```
all : copy sample1.eva sample2.eva
sample1.eva : sample1.obj sample1.opt sample1.cgr
            m86k/p sample1;
sample2.eva : sample2.obj sample2.opt sample2.cgr
            m86k/p sample2;
copy : copy sampl?.eva c:¥old_prog¥
```

上記の例では、MAKE 起動時のコマンドパラメータで、all が指定またはターゲットの指定がすべて省略された時、擬似ターゲット all は“ sample1.eva ” sample2.eva ”の両方をビルドします。擬似ターゲット copy は“ sample1.exe ” sample2.eva ”を更新する前に、c:¥old\_prog へコピーします。

## 17.2.2 マクロ

マクロは、MAKE ファイル中の文字列を別の文字列に置換する機能です。C 言語における # define と同等の機能といえます。マクロには、ユーザマクロと組み込みマクロがあります。ユーザによって定義されるマクロをユーザマクロと言います。また、既に組み込まれているマクロのことを組み込みマクロと言います。

### ユーザマクロ定義

新しくマクロを定義するには、次の構文に基づいて行います。

```
macroname=string
```

macroname には、英数字とアンダースコア ( \_ ) を任意に組み合わせた文字列を指定します。macroname の最大長は、255 文字です。マクロ名の大文字と小文字は区別されません。たとえば、MacroName と MACRONAME は同じマクロとみなします。また、macroname には、マクロ参照を含むことができます。その場合、参照されるマクロは、それ以前の行で、定義されていない限りなりません。

string は、任意の長さの文字列を指定できますが、マクロ定義は 1 行で完結しなければなりません。円記号 ( ¥ ) による継続行指定が可能です。また、長さ 0 の文字列も指定できます。この場合、このマクロを参照しても、置換される文字列はないので、文字列の消去などに利用できます。同一名のマクロが 2 か所以上で定義されている場合は、最後に定義されたものが有効となります。

## 内部マクロと外部マクロ

ユーザー定義マクロには、MAKE ファイル内で定義参照される「内部マクロ」と、MS-DOS シェル機能が、サポートしている環境変数を用いた「外部マクロ」があります。いずれも、書式は同じです。内部マクロと外部マクロの優先度は、デフォルトで内部マクロ優先ですが、/ E オプションを指定することにより、外部マクロを優先することができます。

## ユーザーマクロ参照

マクロを参照するには、ドル記号 ( \$ ) の後にカッコで囲み、マクロ名を指定します。ただし、マクロ名が 1 文字の場合は、カッコで囲む必要はありません。

`$( macroname )` または `$c`

未定義のマクロ名が参照された場合、置換される文字列はありません。

### 組み込みマクロの参照

MAKE には、ファイル名を表す組み込みマクロが用意されています。

<code>\$@</code>	依存記述ブロックのターゲットファイル名 ( パス名、ベース名、拡張子名 )
<code>\$*</code>	依存記述ブロックのターゲットファイルのパス名とベース名 ( パス名、ベース名 )
<code>\$?</code>	依存記述ブロックのターゲットファイルよりも、新しいすべての依存ファイル

### マクロ設定および参照例

ASM = m86k	# LC86000 シリーズ アセンブラ
LINK = 186k/p	# LC86000 シリーズ リンカ
all : sample.eva	擬似ターゲットの使用
sample.obj : *.asm	ターゲットのベース名参照
\$(ASM) \$*;	アセンブラ コマンドの参照
sample.eva : *.obj *.opt *.cgr	ターゲットのベース名参照
\$(LINK) \$*;	リンカ コマンドの参照

## 17.2.3 ディレクティブ

ディレクティブは、記述ブロックの外側で、しかも行の先頭に記述します。MAKE には、次の 3 つのディレクティブがあります。

`.IGNORE : { + | - }`

MAKE ファイルのコマンド実行により、起動されたプログラムが返す終了コードを無視するモードを変更します。プラス記号 ( + ) を指定した時、終了コードを無視するモード、マイナス記号 ( - ) を指定した時、終了コードを有効にするモードとなります。デフォルト

では、終了コードが 0 以外の時、MAKE は処理を中断します。1 つのコマンドに対してのみ終了コードを無視するには、マイナス記号修飾子を使用します。MAKE ファイル全体に対して終了コードを無視するには、/ I オプションを使用します。

**.SILENT : { + | - }**

MAKE ファイルのコマンドを実行する時、そのコマンドを表示しないモードにします。プラス記号 ( + ) を指定した時、コマンドを表示しないモード、マイナス記号 ( - ) を指定した時、コマンドを表示するモードとなります。デフォルトでは、実行するコマンドを表示します。1 つのコマンドに対してのみ表示を禁止するには、アットマーク記号 ( @ ) 修飾子を使用します。MAKE ファイル全体に対して表示を禁止するには、/ S オプションを使用します。

**.DEFAULT :**

MAKE では、コマンドブロックの記述がない記述ブロックでは、推論規則が機能し対応する生成規則のコマンドを実行します。この時、生成規則がない場合は、コマンドブロックは空欄となりますが、MAKE ファイルに DEFAULT で始まる行によって、コマンド列を記述すると、その時に実行すべきコマンドを指定することができます。

```
.DEFAULT
    commands
```

**.TARGET : suffixs**

MAKE では、最終ターゲット ( ビルドしたいターゲット ) をコマンドラインで指定できますが、省略した場合は、最初の記述ブロックのターゲットを最終ターゲットとしますが、この TARGET ディレクティブにより、デフォルト最終ターゲットの拡張子を指定することで、ビルドしたいファイルの拡張子を最終ターゲットに指定できます。suffix は、ピリオド ( . ) を含めて 4 文字までです。また、suffix は、スペースで区切ることで複数指定できます。

## 17.3 推論規則

推論規則は、ある拡張子を持ったファイルから、別の拡張子を持ったファイルを作る時の方法を定義したファイルです。MAKE は、この推論規則にしたがって、ターゲットを更新するためのコマンドを用意し、ターゲットの依存関係を推論します。この推論規則を用いることにより、MAKE ファイルの記述が簡略化できます。推論規則は、MAKE ファイルまたはメイクルールファイル “ MAKERULE.DEF ” に記述します。

各記述ブロックの依存ファイルについて、その依存ファイルをターゲットとする記述ブロックがあるか調べ、ない場合には推論します。推論の条件は、次のようになります。

- ( 1 ) その依存ファイルに関する依存記述部がないこと。
- ( 2 ) その依存ファイルを作成するための生成規則があること。
- ( 3 ) その依存ファイルを作成するための依存ファイルがあること。

これらの条件が満足できた場合、MAKE は次のように記述ブロックを追加します。

- ( 1 ) 記述ブロックがない場合は、記述ブロックを追加する。
- ( 2 ) 依存ファイルとして “ basename.sss ” を付加する ( sss : ソースファイルの拡張子 ) 。
- ( 3 ) コマンド列として生成規則にあるコマンドを付加する。

### 17.3.1 メイクルールファイル

メイクルールファイル“MAKERULE.DEF”は、MAKE が生成規則を推論するための規則を記述したもので、次の形式で表します。

```
.sss.ttt:
    commands
```

1 行目で 2 つのファイルの拡張子を指定します。“sss”は、依存ファイルの拡張子を、“ttt”は、ターゲットファイルの拡張子を指定します。拡張子には、大文字と小文字の区別がありません。“sss”のピリオド(.)は、行の先頭でなければなりません。続く行は、コマンドブロックであり、記述内容は MAKE ファイルと同じです。つまり、アセンブラソース“basename.asm”からオブジェクトファイル“basename.obj”を生成するための規則は“.asm.obj”となります。

#### メイクルールファイルの記述例

```
# *****
# ***
# ***    internal generate rule for EVA86000 utility make.    ***
# ***    definition for M86K                                  ***
# ***
# *****
ASM = m86k

.ASM.OBJ:
    $ (ASM) $*;

.TARGET: .EVA .HEX                デフォルト最終ターゲットの指定
.DEFAULT:
    @echo -----
    @echo ??? Undefined build commands ???
    @echo -----
# end of makerule.def
```

#### メイクルールファイルを利用した MAKE ファイルの記述例

```
ASM = m86k                # LC86000 シリーズ アセンブラ
LINK = 186k/p              # LC86000 シリーズ リンカ

all : sample.eva

sample.obj : *.asm

sample.eva : *.obj *.opt *.cgr    アセンブラ コマンドの記述を省略
    $ (LINK) $*;
```



---

## 第 7 部

---

# コーディング編

---

---

ここでは、アセンブラの文法やアセンブラ疑似命令、LC86K の命令セットについて説明しています。



## 第 18 章

# アセンブラの文法

ソースファイルは、1 行につき（行末の CR、LF を含んで）511 文字までの長さの文字列です。また、ソースプログラム内で定義されるシンボル（ラベルやマクロの名前など）以外の英文字には、大文字 / 小文字の区別はありません。たとえば、Nop と nop は共に NOP 命令を表すニモニックとして認識されます。また、ラベルなどのシンボルも、アセンブラオプション - i が指定されていれば大文字 / 小文字の区別をしなくなります。

## 18.1 ステートメント

ステートメントは、アセンブル時に作成すべきオブジェクトコードを定義するニモニック、オペランド、およびコメントを組み合わせたものです。ソースコードの 1 行が、1 個のニモニックに相当します。また、複数行にまたがるステートメントは認められません。各ステートメントは、次に示す 4 種類のフィールドから構成されます。

```
[ label: ] [ operation ] [ operand ] [ ;comment ]
```

フィールド	目的
<i>label</i>	ステートメントを他のステートメントから名前でアクセスするための、ステートメントに付けるラベルです。必ずコロン ( : ) で区切られます。
<i>operation</i>	ステートメントの動作を指定します。
<i>operand</i>	ステートメントの動作対象となるデータを定義します。
<i>comment</i>	ステートメントに対する説明であり、アセンブルに対して何も影響を与えません。

### 注意

[ ] で示したデータは省略可能なものを示します。

## 18.2 ラベル名およびシンボル名

ラベル名およびシンボル名は、1 文字以上の任意の長さの文字列です。ただし、最初の 32 文字のみが有効になります。使用可能な文字は、次のとおりです。

A ~ Z a ~ z 0 ~ 9 \$ ? \_ @ .

また、ラベル名およびシンボル名の先頭は、英字、「\_」、「.」、または「@」で始まらなければなりません。アセンブラオプション - i が指定されていなければ、大文字と小文字は区別されます。また、ラベル名は必ずコロン ( : ) で区切られます。

## 18.3 コメント

コメントは、セミコロン「;」で始まり、改行で終了します。

## 18.4 演算子

M86K で使用可能な演算子およびそれらの優先順位は次のとおりです。NOT などの英文字のみの演算子では、大文字 / 小文字の区別はされません。NOT も not も同じ演算子として認識されます。

演算子	説明	優先順位
NOT	1の補数	1
HIGH	上位バイト	
LOW	下位バイト	
*	乗算	2
/	除算	
MOD	剰余	
+	加算	3
-	減算	
SHR	右シフト	4
SHL	左シフト	
LAND	論理積	5
LOR	論理和	
LXOR	排他的論理和	
EQ	等しい	6
NE	等しくない	
LT	より小さい	
LE	より小さいか等しいか	
GT	より大きい	
GE	より大きいか等しい	

## 18.5 数値定数

M86K では2進、8進、10進、16進の4種類の基数の数値定数を記述することができます。数値定数を記述する方法には“123H”のように基数を明示する書式と、デフォルトの基数を指示するための疑似命令 RADIX によってあらかじめ基数を指定しておく方法の2つがあります。基数を明示する書式で記述された数値定数の値は、デフォルトの基数に優先して指定された基数で変換されます。一方“123”のように基数を明示しない数値定数は、疑似命令 RADIX で指定された基数で値に変換されます。疑似命令 RADIX による基数の指定がない場合のデフォルトの基数は10です。

いずれの場合も記述された定数は、アセンブラ内部では32ビットで処理されます。また、数値定数など最終的に数値定数に帰着する演算式の値が命令のオペランドにイミディエイト

データとして書き込まれる場合、そのオペランドに必要なだけのビット数が格納され、残りの上位ビットは捨てられます。

### 基数を明示した数値定数の書式

基数を明示した書式の数値定数の解釈

基数	書式	例		
2	%に続く1つ以上の0~1	%01111011	%1111111	%0000010000000000
	1つ以上の0~1に続くB*	01111011B	11111111B	0000010000000000B
	1つ以上の0~1に続く.B	01111011.B	11111111.B	0000010000000000.B
8	1つ以上の0~7に続く.O	273.O	377.O	2000.O
10	1つ以上の0~9に続く.D	123.D	255.D	1024.D
16	\$に続く1つ以上の0~9,a~f,A~F	\$7B	\$FF	\$0400H
	先頭の0~9個以上の0~9,a~f,A~Fに続くH	7BH	0FFH	0400H
	先頭の0~9個以上の0~9,a~f,A~Fに続く.H	7B.H	0FF.H	0400.H

基数を指定する文字 BODH には大文字 / 小文字の区別はありません。これはアセンブラオプション - i には影響されません。

#### 注意

この書式に関しては RADIX による設定の影響を受けます。詳しくは、次の表を参照してください。

### 基数を明示しない書式の数値定数の解釈

基数を明示しない書式の数値定数の解釈

書式	例	RADIXで指定された基数の値			
		2	8	10	16
1つ以上の0~1	0101	5 <sub>10</sub>	65 <sub>10</sub>	101 <sub>10</sub>	257 <sub>10</sub>
1つ以上の0~7	123	エラー	83 <sub>10</sub>	123 <sub>10</sub>	291 <sub>10</sub>
1つ以上の0~9	789	エラー	エラー	7891 <sub>10</sub>	1929 <sub>10</sub>
1つ以上の0~1に続く	101B	5 <sub>10</sub>	5 <sub>10</sub>	5 <sub>10</sub>	4123 <sub>10</sub>
先頭の0~9と0個以上の0~9, a~f, A~F	OFF	エラー	エラー	エラー	255 <sub>10</sub>

## 18.6 文字定数

引用符 ( ' ) で囲まれた文字は文字定数としてあつかわれます。文字定数は一種の数値定数で、指定された文字の ASCII コードがその値となります。印刷可能なすべての ASCII 文

字が記述できるほか、次のような書式で任意のコードを記述することも可能です。なお、引用符の中に複数の文字があった場合は文字定数ではなく文字列定数(「18.7 文字列定数」を参照)としてあつかうので注意してください。

#### 文字・文字列定数中のコード入力 of 書式

書式	コード(16進)	備考
¥n	0A	改行
¥r	0D	復帰
¥t	09	水平タブ
¥b	08	バックスペース
¥f	0C	紙送り
¥ "	22	ダブルクォート
¥ '	27	シングルクォート
¥¥	5C	円記号またはバックslash
¥ooo		ooo は3桁までの8進数
¥xhh		hh は2桁までの16進数

例1: ADD # 'A '

例2: DB 'A ', '¥012 ', 'C '

例3: DB 'ABC '

#### 注意

例3の場合、'ABC 'は文字列定数と見なされ、DB のオペランドとしてはエラーになります。

## 18.7 文字列定数

二重引用符( ")で囲まれた文字列(1文字以上の長さ)または引用符( ')で囲まれた文字列(2文字以上の長さ)は文字列定数としてあつかわれます。文字列定数は、疑似命令 DC および PRINTX のオペランドとして記述することができます。文字列の中には、印刷可能なすべての ASCII 文字が記述できるほか、「18.6 文字定数」で述べた書式により任意のコードを記述することも可能です。

#### 例

DC "This is a sample string with special codes ¥007¥r¥n"

## 18.8 特殊シンボル

アスタリスク( \*)をオペランドで使用した場合、記述箇所のロケーション値を表します。

### 例 1

命令の記述されている箇所より 6 バイト前方を示す場合には次のように記述します。

BR       \*-6

### 例 2

命令の記述されている箇所より 12 バイト後方を示す場合には次のように記述します。

BR       \*\*+12



## 第 19 章

# アセンブラ疑似命令

疑似命令は、通常の命令（ADD や MOV などの LC86K 自身の動作を指示する命令）とは異なり、アセンブラに対して種々の指示や定義を行うもので、疑似命令単体では機械語を発生しません（JMPO などの最適化用の疑似命令と CHANGE 疑似命令はこの限りではありません）。多くの場合、通常の命令との組み合わせで使用します。

分類	疑似命令	機能
リンク制御	ORG WORLD CSEG DSEG END PUBLIC EXTERN OTHER_SIDE_SYMBOL	オリジンの指定 コードを格納する ROM の選択 コードセグメントの指定 データセグメントの指定 プログラムの終了 外部定義名の指定 外部参照名の指定 CHANGE 命令用飛先ラベルの宣言
シンボル定義	EQU SET	値の割当 一時的な値の割当
データ定義	DB DW DC DS	バイト定義 ワード定義 文字列定義 データ領域（RAM）の確保
マクロ制御	MACRO REPT IRP IRPC ENDM EXITM LOCAL	マクロ定義 繰り返しマクロ 連続マクロ 文字列マクロ マクロ定義の終了 マクロ展開の中断 ローカルラベルの定義
条件付きアセンブル	IFDEF IFNDEF IFB IFNB IFE IFNE IFIDN IFDIF ELSE ENDIF .PRINTX .LIST .XLIST .MACRO .XMACRO .IF .XIF	定義済みならばアセンブルする 定義済みでなければアセンブルする オペランドが空白ならばアセンブルする オペランドが空白でなければアセンブルする 式の値が 0 ならばアセンブルする 式の値が 0 でなければアセンブルする 2 つの文字列が等しければアセンブルする 2 つの文字列が異なればアセンブルする 上記の IF の条件の逆の条件でアセンブルする 条件付きアセンブルの終了 アセンブル中のディスプレイ上への表示 リストの出力 リストの出力の中断 マクロ展開の出力 マクロ展開の出力の中断 条件スキップの出力 条件スキップの出力の中断

その他	INCLUDE TITLE PAGE CHIP COMMENT WIDTH BANK CHANGE RADIX	ファイルの読み込み リストタイトルの指定 改ページ アセンブル対象チップの定義 オブジェクトファイルへのコメント出力 リストファイルの桁数を指定 RAM 領域のバンク指定 フラッシュメモリ / ROM をまたぐジャンプ デフォルトの基数の指定
最適化	JMPO BRO CALLO BZO BNZO BPO BPCO BNO DBNZO BEO BNEO	最適な JMP 命令の発生 最適な BR 命令の発生 最適な CALL 命令の発生 アドレスエラーの発生しない BZ 命令の発生 アドレスエラーの発生しない BNZ 命令の発生 アドレスエラーの発生しない BP 命令の発生 アドレスエラーの発生しない BPC 命令の発生 アドレスエラーの発生しない BN 命令の発生 アドレスエラーの発生しない DBNZ 命令の発生 アドレスエラーの発生しない BE 命令の発生 アドレスエラーの発生しない BNE 命令の発生

# ORG

オリジンの指定

## 書式

ORG *expression*

## 説明

ORG 疑似命令は、プログラムメモリ（フラッシュメモリ）のアドレス指定を *expression* の値から開始させます。expression は数値定数か、アセンブル時点で値の確定する式でなければなりません。

## 例

```
page:      1 <org.ASM>
ERR  SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                                ;a sample program for ORG
0002                                chip    lc866032
0003                                extern  wait1s
0004                                dseg
0005      D 0000      min1:  ds      1
0006      D 0001      min0:  ds      1
0007                                cseg
0008                                org     0h
0009      C 0000 6201'  label1: inc     min0
0010      C 0002 0201'          ld      min0
0011      C 0004 A13C          sub     #60
0012      C 0006 900311        bzo     label2
0012      C 0009 F600
0013      C 000B 210200'        jmpf    label3
0014                                org     100h
0015      C 0100 6200'  label2: inc     min1
0016      C 0102 220100'        mov     #00,min0
0017      C 0105 210200'        jmpf    label3
0018                                org     200h
0019      C 0200 100000' label3: CALLr   wait1s
0020      C 0203 210000'        jmpf    label1
0021                                end
```

# WORLD

## コードを格納する ROM の選択

### 書式

WORLD *selection*

### 説明

アセンブルされたコードが格納されるべき ROM を指定します。この疑似命令は、ターゲットチップが LC86800 シリーズのときにのみ意味を持ちます。*selection* に指定できるのは、次の 3 種類です。

INTERNAL	チップ内蔵の ROM に格納されます。
EXTERNAL	フラッシュメモリのバンク 0 に格納されます。
EXTERNAL_DATA	フラッシュメモリのバンク 1 に格納されます。

### 注意

ビジュアルメモリでは、必ず EXTERNAL を指定してください。その他の指定を行なうとデータの破壊や誤動作の原因になります。

1 つのファイルに複数の WORLD 疑似命令があると、エラーとみなします。また、LC86800 シリーズ以外のチップが指定され、かつ WORLD 疑似命令で INTERNAL 以外が選択されると、エラーとなります。

# CSEG

## コードセグメントの開始宣言

### 書式

CSEG *mode*

### 説明

プログラムコードが格納されるセグメントの開始をアセンブラに知らせます。また、*mode* を記述しない場合、または *mode* に INBLOCK を記述すると 4K バウンダリに配置することを示します。*mode* に FREE と記述した場合には 4K バウンダリに関係なく配置することを示します。

### 例

```

page:      1 <cseg.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for CSEG
0002                               chip    lc864024
0003                               extern wait1s
0004                               dseg
0005      D 0000      min1:  ds    1
0006      D 0001      min0:  ds    1
0007                               cseg  inblock
0008      C 0000 6201'  label1: inc    min0
0009      C 0002 0201'          ld     min0
0010      C 0004 A13C          sub    #60
0011      C 0006 900311        bzo     label2
0012      C 000B 210000'        jmpf   label3
0013                               cseg  free
0014      c 0000 6200'  label2: inc    min1
0015      c 0002 220100'        mov    #00,min0
0016      c 0005 210000'        jmpf   label3
0017                               cseg
0018      C 0000 100000' label3: CALLr  wait1s
0019      C 0003 210000'        jmpf   label1
0020                               end

```

各セグメントの  
始まりでローカルアドレスは0リセットされます

独立したセグメント

# DSEG

## データセグメントの開始宣言

データメモリ上に領域の確保を開始することをアセンブラに知らせます。

### 例

```
page:      1 <dseg.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for CSEG
0002                               chip    lc864024
0003                               extern  waitls
0004                               cseg    inblock
0005  C 0000 6201' label1: inc    min0
0006  C 0002 0201'         ld     min0
0007  C 0004 A13C         sub     #60
0008  C 0006 900311       bzo     label2
0008  C 0009 0000
0009  C 000B 210000'      jmpf    label3
0010                               cseg    free
0011  c 0000 6200' label2: inc    min1
0012  c 0002 220100'      mov     #00,min0
0013  c 0005 210000'      jmpf    label3
0014                               cseg
0015  C 0000 100000' label3: CALLr  waitls
0016  C 0003 210000'      jmpf    label1
0017
0018                               dseg
0019  D 0000         min1: ds     1
0020  D 0001         min0: ds     1
0021                               end
```

コードセグメント

データセグメント

# END

プログラムの終了

## 書式

END

## 説明

ソースプログラムの終了を宣言します。アセンブラはこの命令を検出した時点で実行中のパスのアセンブル作業を終了するので、この命令のあとに有効なステートメントがあっても無視されます。

## 例

```
; a sample program for END
```

```
chip    lc866032
cseg
mov     #20h, 01h
mov     #10h, 00h
ld      00h
add     0fh
end
```

```
inc     00h
inc     01h
ld      01h
```



疑似命令ENDより後はアセンブルされません

```
page:    1 <end.ASM>
```

ERR	SEQ.	S LOC.	OBJ.	SOURCE STATEMENTS
	0001			; a sample program for END
	0002			chip    lc866032
	0003			cseg
	0004	C 0000	220120	mov     #20h, 01h
	0005	C 0003	220010	mov     #10h, 00h
	0006	C 0006	0200	ld      00h
	0007	C 0008	820F	add     0fh
	0008			end

# PUBLIC

外部定義名の指定

## 書式

```
PUBLIC  symbol  { , symbol }
```

## 説明

PUBLIC 疑似命令は、そのソースプログラムで定義した *symbol* を他のソースファイルから参照できるように宣言します。

## 例

```
page:      1 <extern.ASM>
ERR  SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for EXTERN
0002                                chip    lc866032
0003                                extern label1, label2
0004
0005                                cseg inblock
0006      C 0000 200000' CALLf  label1
0007
0008      C 0003 200000' start: CALLf  label2
0009      C 0006 0303      ld      c
0010      C 0008 90F9      bnz     start
0011
0012      C 000A A300      sub     a
0013
0014                                end
```

他のソースファイル上で値が定義されるシンボルを参照する場合は、あらかじめEXTERNで宣言しておく必要があります。

他のソースファイル上で値が定義されるシンボルを参照する場合は、あらかじめEXTERNで宣言しておく必要があります。

```
ERR  SEQ.  S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for PUBLIC
0002                                chip    lc866032
0003                                public label1, label2
0004
0005                                cseg    inblock
0006      C 0000 220000' label1: mov    #00, data1
0007      C 0003 23033C      mov    #60, c
0008      C 0006 A0          ret
0009
0010      C 0007 6200' label2: inc     data1
0011      C 0009 0200'      ld      data1
0012      C 000B 410A05      bne     #10, label3
0013      C 000E 220000'      mov    #00, data1
0014      C 0011 6201'      inc     data2
0015
0016      C 0013 7303 label3: dec     c
0017      C 0015 A0          ret
0018
0019                                dseg
0020      D 0000      data1: ds      1
0021      D 0001      data2: ds      1
0022
0023                                end
```

**注意**

他のソースファイル上で値が定義されるシンボルを参照する場合は、あらかじめ EXTERN で宣言しておく必要があります。

**注意**

他のソースファイルから参照されるシンボルは、PUBLIC で宣言して外部から「見える」ようにしておく必要があります。

```

page:      1 <public.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for PUBLIC
0002                                chip    lc866032
0003                                public  label1, label2
0004
0005                                cseg    inblock
0006      C 0000 220000' label1: mov     #00, data1
0007      C 0003 23033C          mov     #60, c
0008      C 0006 A0              ret
0009
0010      C 0007 6200'  label2: inc     data1
0011      C 0009 0200'          ld      data1
0012      C 000B 410A05          bne     #10, label3
0013      C 000E 220000'          mov     #00, data1
0014      C 0011 6201'          inc     data2
0015
0016      C 0013 7303  label3: dec     c
0017      C 0015 A0              ret
0018
0019                                dseg
0020      D 0000          data1: ds      1
0021      D 0001          data2: ds      1
0022
0023                                end

```

PUBLIC と EXTERN の組み合わせによって、他のソースファイル上で値が定義されるシンボルであっても参照することができるようになります。

# EXTERN

外部参照名の指定

## 書式

```
EXTERN  [ segmanet : ] symbol { [ segment : ] symbol }
```

## 説明

EXTERN 疑似命令は他のプログラムで指定された *symbol* を参照する場合に使用します。フォーマットの中の *segmanet* の指定では、CSEG , DSEG の内のいずれかのセグメントの指定が可能です。何も指定しない場合には、コードセグメント CSEG が使用されたものとみなします。

### 参照

例については「第 19 章 アセンブラ疑似命令」の「PUBLIC 外部定義名の指定」を参照してください。

# OTHER\_SIDE\_SYMBOL

CHANGE 命令用ジャンプ先ラベルの宣言

## 書式

```
OTHER_SIDE_SYMBOL  label  { , label }
```

## 説明

LC86800 シリーズにおいて ROM とフラッシュメモリの切り換えを行う CHANGE 命令のオペランドとして指定されるアドレスラベルを宣言します。宣言されるラベルは一種の外部シンボルですが、ROM に格納されるコードを記述したソースファイルではフラッシュメモリ上のラベルが宣言される（フラッシュメモリに格納されるコードを記述したソースファイルでは ROM 上のラベル）点が異なります。なお、この疑似命令は LC86800 シリーズ専用であり、それ以外の場合にはエラーとなります。

### 参照

例については「第 19 章 アセンブラ疑似命令」の「CHANGE フラッシュメモリ / ROM をまたぐジャンプ」を参照してください。

# EQU

## 値の割当

### 書式

*symbolname* EQU *expression*

### 説明

EQU 疑似命令は、*symbolname* に *expression* の値を割り当てます。EQU 疑似命令で定義されたシンボルは再定義することはできません。EQU 疑似命令を効果的に使用すると、コンスタントなデータに対して視覚的な意味付けが行え、メンテナンス時の作業効率向上が図れます。

### 例

定義された値が計算可能な場合は、  
その値を表示（16進）します

値を定義するシンボルとEQUの間にコロン（:）は記述しません

```
page:      1 <equ.ASM>
ERR SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for EQU
0002                               chip    lc866032
0003
0004      00000064 loop_max      equ    100
0005      00000001 mode_a       equ    1
0006      00000002 mode_b       equ    2
0007      00000003 mode_c       equ    3
0008
0009                               cseg    inblock
0010      C 0000 220000'         mov     #00, loop_ctr
0011
0012      C 0003 230201 label1:  mov     #mode_a, b
0013      C 0006 0818'          CALL    subl
0014      C 0008 230202         mov     #mode_b, b
0015      C 000B 0818'          CALL    subl
0016      C 000D 230303         mov     #mode_c, c
0017      C 0010 6200'          inc     loop_ctr
0018      C 0012 0200'          ld      loop_ctr
0019      C 0014 4164EC         bne     #loop_max, label1
0020      C 0017 A0             ret
0021
0022      C 0018 0302  sub1:  ld      b
0023      C 001A 310107         be      #mode_a, suj10
0024      C 001D 310208         be      #mode_b, suj11
0025      C 0020 310309         be      #mode_c, suj12
0026      C 0023 A0             suj0:  ret
0027
0028      C 0024 1201'  suj10:  st      data_a
0029      C 0026 01FB         br      suj0
0030      C 0028 1202'  suj11:  st      data_b
0031      C 002A 01F7         br      suj0
0032      C 002C 1203'  suj12:  st      data_c
0033      C 002E 01F3         br      suj0
0034
0035                               dseg
0036      D 0000      loop_ctr:  ds      1
0037      D 0001      data_a:    ds      1
0038      D 0002      data_b:    ds      1
0039      D 0003      data_c:    ds      1
0040
0041                               end
```

任意の演算式が記述可能です

# SET

一時的な値の割当

## 書式

*symbolname* SET *expression*

## 説明

SET 疑似命令は、*symbolname* に *expression* の値を割り当てます。SET 疑似命令で定義されたシンボルは、SET によって再定義することができます。また、この疑似命令で値を設定されたシンボルを、PUBLIC 宣言したり EQU で再定義することはできません。

## 例

```

page:      1 <set.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                ; a sample program for SEY
0002                chip   lc866032
0003                cseg   inblock
0004
0005      00000000 dd      set    0
0006
0007      C 0000 220000'      mov    #dd,zz+dd
0008
0009      C 0003 6300          inc    a
0010      C 0005 6302          inc    b
0011
0012      00000001 dd      set    dd+1
0013
0014      C 0007 220101'      mov    #dd,zz+dd
0015
0016      C 000A 7300          dec    a
0017      C 000C 7302          dec    b
0018
0019                dseg
0020      D 0000          zz:    ds    2
0021
0022                end

```

定義された値が計算可能な場合は、その値を表示 (16進) します

値を定義するシンボルとSETの間にコロンの ( : ) は記述しません

値を定義しようとしているシンボルを含む、任意の演算式が記述可能です

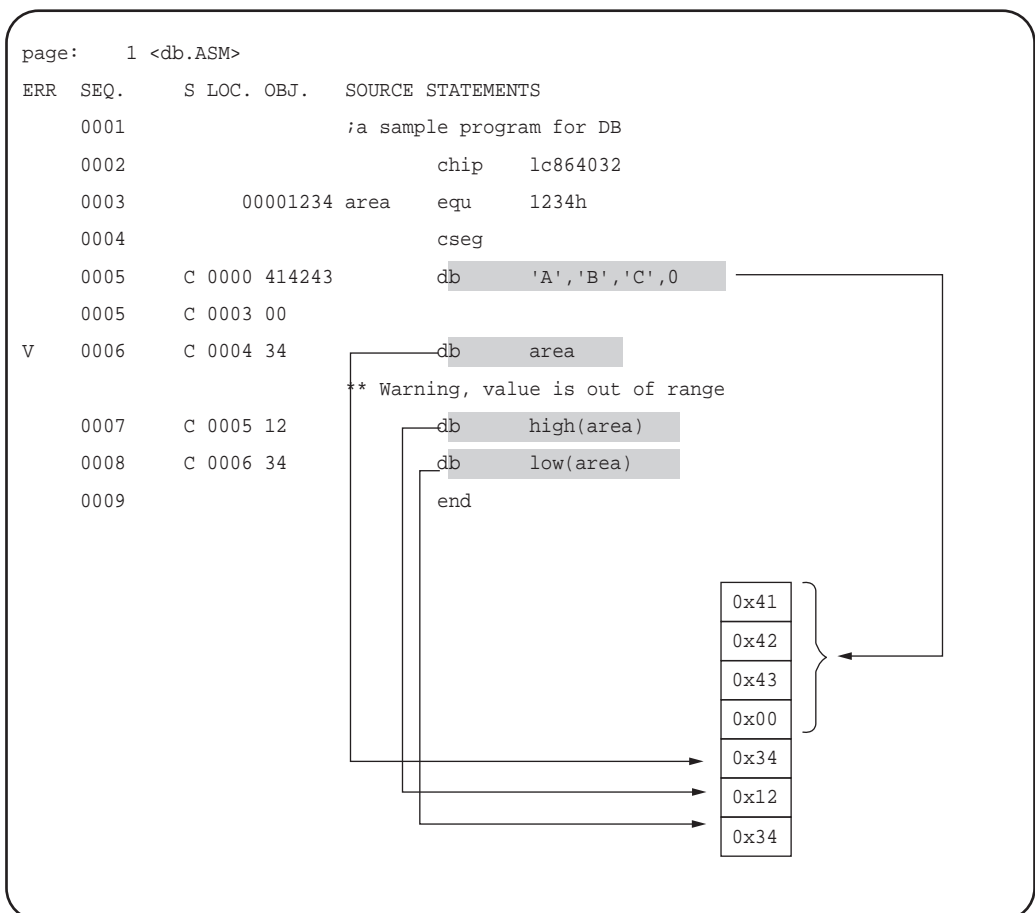
## 書式

```
labelname DB expression { , expression }
```

## 説明

DB 疑似命令は、オペランドの *expression* に対応した 8 ビットデータがプログラムメモリ（ROM）上に格納されます。オペランドは 1 つだけでなく、カンマ（,）で区切って複数個記述することができます。オペランドが 2 つ以上の場合は左から順番に評価され、アドレスが増加する方向へ順番に格納されます。また、カンマとカンマの間に何も記述しない場合は 0 を記述したことと同じになります。

## 例



上記例の「db area」は、シンボル `area` の持つ値が 16 ビットの幅であるために、アセンブル時に「value is out of range」のエラー（警告レベル）が発生します。ただし、オブジェクトには下位 8 ビットの値が出力されます。

# DW

## ワードデータの定義

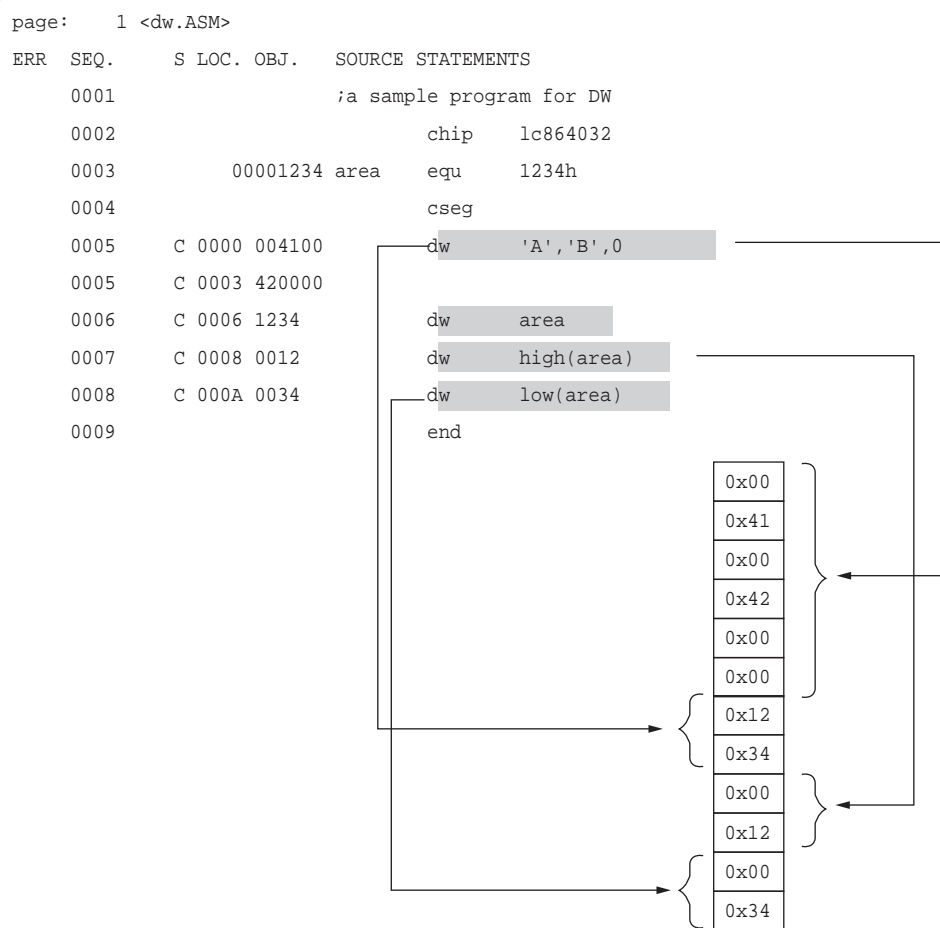
### 書式

*labelname* DW *expression* { , *expression* }

### 説明

DW 疑似命令は、オペランドの *expression* に対応した 16 ビットデータをプログラムメモリ (ROM) 上に格納する場合に使用します。上位バイトが先に格納され、下位バイトが次 (アドレスが 1 つ大きい) に格納されます。オペランドは 1 つだけでなく、カンマ ( , ) で区切って複数個記述することができます。オペランドが 2 つ以上の場合は連続した領域に格納されます。カンマとカンマの間に何も記述しない場合は 0 を指定したことになることです。

### 例



8ビットの値をDW疑似命令で確保すると、16ビットの上位8ビットは必ず0になります。

# DC

## 文字列データの定義

### 書式

*labelname* DC *"string"*

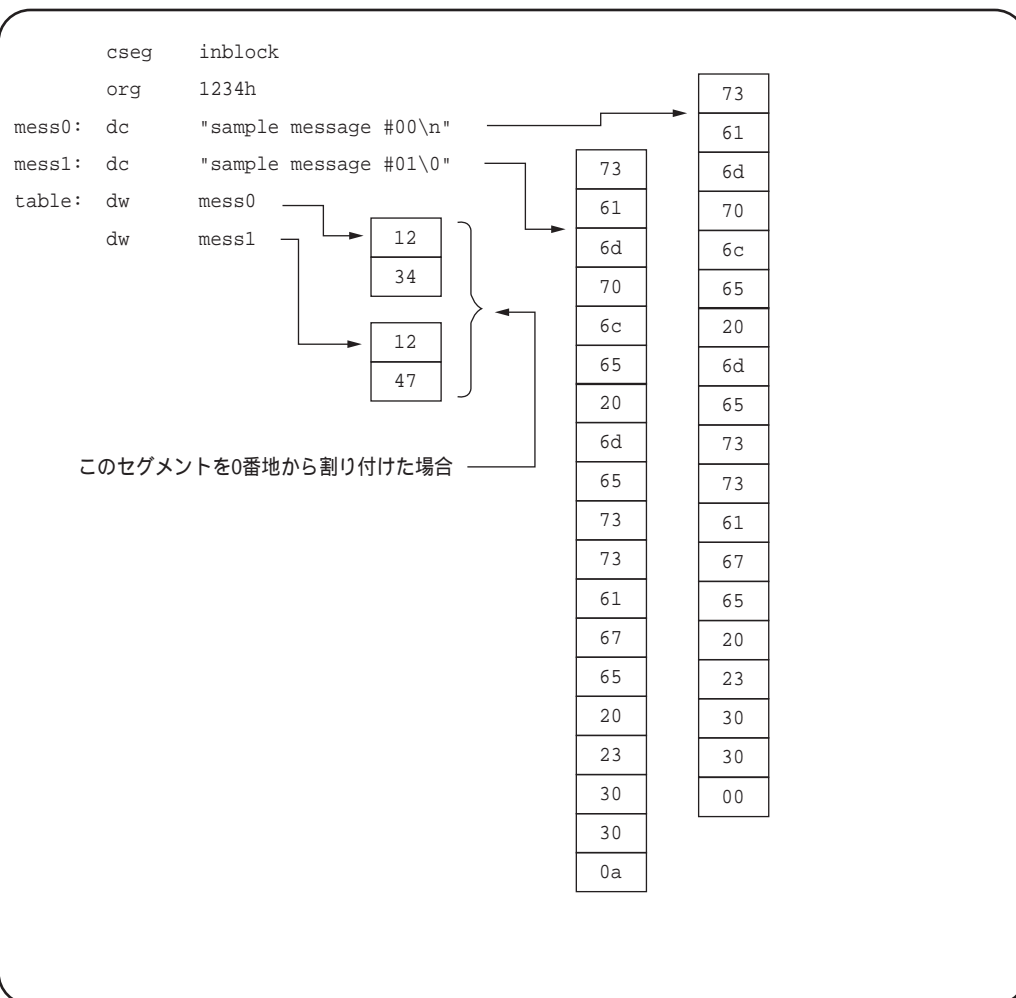
### 説明

*string*(文字列定数)の内容をASCIIコードの値として順番にプログラムメモリ(ROM)領域に格納します。

#### 参照

文字列定数については「18.7 文字列定数」を参照してください。

### 例



# DS

## バイト領域の定義

### 書式

*labelname* DS *absolute\_expression*

### 説明

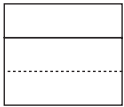
DS 疑似命令は、*absolute\_expression* の値で指定したバイト数だけの領域をデータメモリ ( RAM ) の中に確保します。*absolute\_expression* の記述は、絶対式 ( 値が決定している式 ) のみが許されます。この疑似命令は DSEG 疑似命令の記述のあとでないと使用することができません。

### 例

```

page:      1 <ds.ASM>
ERR  SEQ.    S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for DS
0002                                chip    lc864032
0003                                dseg
0004      D 0000      area0: ds      1
0005      D 0001      area1: ds      2
0006                                cseg    inblock
0007      C 0000 0200' start: ld      area0
0008      C 0002 1201'      st      area1
0009      C 0004 1202'      st      area1+1
0010                                end
                                dseg
                                → area0:
                                → area1:

```



上記例はarea0なる名称で1バイト領域を確保し、area1なる名称でarea0の直後に2バイト連続で領域を確保します。

# MACRO

## マクロ定義

### 書式

```
name  MACRO  parameter{ , parameter }
```

### 説明

MACRO 疑似命令はマクロ定義を行います。MACRO 疑似命令以降、ENDM 疑似命令までのステートメントがマクロ定義の本体となります。*name* は定義したマクロを呼び出す、すなわち *name* をマクロ定義の本体で置き換えるために必要であるため、必ず指定してください。これに対して *parameters* は仮引数リストであり、定義されたマクロの内容に応じて指定してください。

#### 注意

なお、マクロ中で他のマクロを呼び出す場合や、IFB などのように、< , > が必要な疑似命令を使用する場合には、ネストの段数分だけの < , > が必要です。詳しくは「第 19 章 アセンブラ疑似命令」の「EXITM マクロ展開の中断」を参照してください。

## 例

```
_push macro
    push    acc
    push    c
    push    b
endm
```

← acc,c,bの各レジスタをスタックに格納します

```
_pop macro
    pop     b
    pop     c
    pop     acc
endm
```

← スタックに格納されている値をb,c,accの順で取り出します

```
_shl macro    count
    ifne      count
        rept  count
            rol    c
        endm
    else
        .printx  "logical shift count is zero !!\007"
    endif
endm
```

↑ 引数で指定された数の左シフトを行うコードを発生します  
ただし、引数が0ならばシフトを行うコードを発生しません

```
cseg
start: _push
       _shl    0
       _shl    2
       _shl    1
```

← ソースプログラム上の記述です

```

0027          start: _push
0027+1 C 0000 6100      push    acc
0027+2 C 0002 6103      push    c
0027+3 C 0004 6102      push    b
0028          _shl     0
0028+1          ifne    0
0028+2          rept    0
0028+3          rolc
0028+4          endm
0028+5          else
0028+6          .printx  "logical shift count is ze
0028+7          endif
0029          _shl     2
0029+1          ifne    2
0029+2          rept    2
0029+4          endm
0029+1 C 0006 F0          rolc
0029+2 C 0007 F0          rolc
0029+5          else
0029+6          .printx  "logical shift count is ze
0029+7          endif
0030          _shl     1
0030+1          ifne    1
0030+2          rept    1
0030+4          endm
0030+1 C 0008 F0          rolc
0030+5          else
0030+6          .printx  "logical shift count is ze
0030+7          endif
0031          _pop
0031+1 C 0009 7102      pop     b
0031+2 C 000B 7103      pop     c

```

# REPT

繰り返しマクロ

## 書式

REPT *count*

## 説明

REPT 疑似命令は、ENDM 命令までのステートメントを *count* で指定された回数だけ繰り返す場合に使用します。*count* には 1 以上 65535 以下の整数が指定できます。

## 例

次の例では、プログラムの存在しない部分をすべて NOP で埋めます (256 バウンダリの場合)。

```

page:      1 <rept.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                               ; a sample program for REPT
0002                               chip    1c864024
0003                               cseg    inblock
0004      C 0000 230000 start:  mov     #0,acc
0005      C 0003 1200          st       00h
0006      C 0005 6300          inc      acc
0007      C 0007 1201          st       01h
0008      C 0009 6300          inc      acc
0009      C 000B 1202          st       02h
0010      C 000D A0      last:  ret
0011                               rept    255-(last-start)
0013                               endm
0013+1      C 000E 00          nop
0013+2      C 000F 00          nop
0013+3      C 0010 00          nop
0013+240    C 00FD 00          nop
0013+241    C 00FE 00          nop
0013+242    C 00FF 00          nop
0014                               end
  
```

マクロ定義本体は表示されません

展開されたステートメント

# IRP

連続マクロ

## 書式

IRP *parameter* , *argument*{ , *argument* }...

## 説明

IRP 疑似命令は、ENDM 疑似命令までのステートメントを *argument* の数だけ繰り返します。繰り返しにおいて、ステートメント中にパラメータが現れるたびに *argument* 内の各項目が、順次 *parameter* と置き換えられます。

## 例

```
_push macro
    irp    reg_name,acc,b,psw,c
        push    reg_name
    endm
endm

_pop macro
    irp    reg_name,c,psw,b,acc
        push    reg_name
    endm
endm
```

```
0016
0017                _push
0017+1                irp    reg_name,acc,b,psw,c
0017+3                endm
0017+1    C 0000 6100                push    acc
0017+2    C 0002 6102                push    b
0017+3    C 0004 6101                push    psw
0017+4    C 0006 6103                push    c
0018                _pop
0018+1                irp    reg_name,c,psw,b,acc
0018+3                endm
0018+1    C 0008 6103                push    c
0018+2    C 000A 6101                push    psw
0018+3    C 000C 6102                push    b
0018+4    C 000E 6100                push    acc
```

# IRPC

文字列マクロ

## 書式

IRPC *parameter* , *string*

## 説明

IRPC 疑似命令は、ENDM 命令までのステートメントを *string* の文字の数だけ繰り返します。*string* は文字列定数とは異なり、引用符などで囲みません。また、円記号 ( ¥ ) で始まる任意コードの入力もできません。

ステートメント中のすべての *parameter* が *string* 内の 1 つの文字と置き換えられ、この置き換えが *string* の文字の数だけ繰り返されます。

## 例

```
; a sample program for IRPC
chip    lc866032
dseg
irpc    x,01234567
buf&x:  ds    2
endm
end
```

↑ 仮引数  
↑ 実引数

↑ 仮引数が実引数の1文字ずつで置き換えられる  
識別子の一部に仮引数がある場合の区切り文字

展開結果

```
page:    1 <irpc.ASM>
ERR  SEQ.    S LOC. OBJ.    SOURCE STATEMENTS
0001                                ; a sample program for IRPC
0002                                chip    lc866032
0003                                dseg
0004                                irpc    x,01234567
0006                                endm
0006+1  D 0000    buf0:    ds    2
0006+2  D 0002    buf1:    ds    2
0006+3  D 0004    buf2:    ds    2
0006+4  D 0006    buf3:    ds    2
0006+5  D 0008    buf4:    ds    2
0006+6  D 000A    buf5:    ds    2
0006+7  D 000C    buf6:    ds    2
0006+8  D 000E    buf7:    ds    2
```

# ENDM

マクロ定義の終了

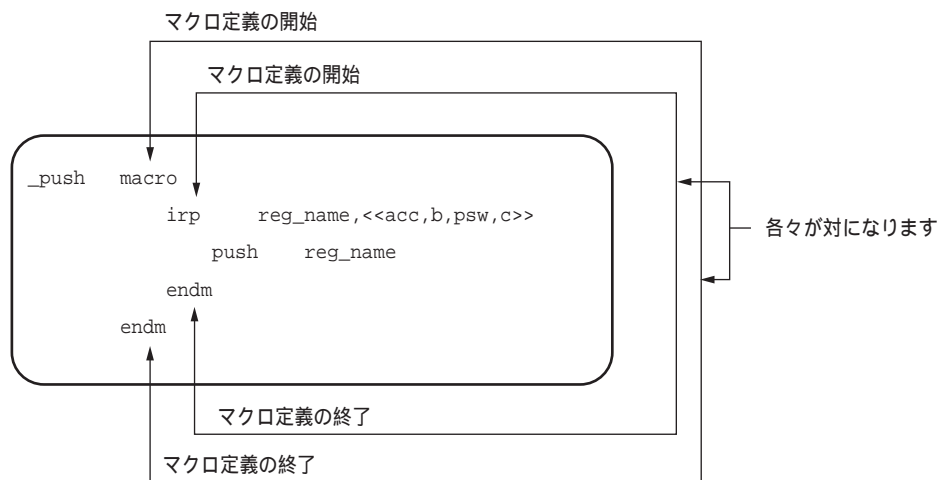
## 書式

ENDM

## 説明

マクロ定義文の終了を宣言します。

## 例



# EXITM

マクロ展開の中断

## 書式

EXITM

## 説明

EXITM 疑似命令は、マクロ展開を中断します。条件アセンブル用の疑似命令と組み合わせ、同一のマクロで与える実引数によって異なる展開結果を得るときなどに使用します。

## 例

マクロの展開時に1組の<>が除去されるので、二重の<>が必要です

```

page:      1 <exitm.ASM>
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001          ; a sample program for EXITM
0002          chip      LC866032
0003  rpush  macro  al,a2,a3,a4
0004          ifb      <<a1>>
0005                  .printx "not enough argument"
0006          exitm
0007          endif
0008          ifnb     <<a2>>
0009                  push      a1
0010                  push      a2
0011                  push      a3
0012                  push      a4
0013          endif
0014          endm
0015          cseg      inblock
0016  rpush  acc,b,psw,c
0016+1      ifb      <acc>
0016+2          .printx "not enough argument"
0016+3      exitm
0016+4      endif
0016+5      ifnb     <b>
0016+6  C 0000 6100      push      acc
0016+7  C 0002 6102      push      b
0016+8  C 0004 6101      push      psw
0016+9  C 0006 6103      push      c
0016+10     endif
0017  rpush
0017+1      ifb      <>
0017+2          .printx "not enough argument"
0017+3      exitm
0018      end

```

1つ目の実引数が与えられているので、この部分がアセンブルされます

2つ目の実引数がないのでこの部分を展開し、EXITMを認識して展開を中止します

---

# LOCAL

ローカルラベルの定義

---

## 書式

```
LOCAL  name{ , name }
```

## 説明

LOCAL 疑似命令は、マクロ定義の内部のみで使用可能なラベルの宣言に使用します。LOCAL 疑似命令で宣言された *name* がマクロ展開中に現われると、マクロアセンブラはその *name* をほかと競合しないユニークな名前に置き換えます。

## 例

```
; a sample program for LOCAL
      chip    lc864008
b_ne  macro  val,dst
      local  skip
      be     val,skip
      bro    dst

skip:
      endm

cseg
      b_ne    #0, over
org    200h
over:  b_ne    #0, under
      nop
under:  nop
      end
```

上記の例は BRO 疑似命令を用いて、分岐先に応じた命令語の自動発生を行う BNEO マクロ命令を定義、および使用した例です。次にそのアセンブル結果を示します。

```

page:      1 <local.ASM>
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001                      ; a sample program for LOCAL
0002                      chip    lc864008
0003                      b_ne    macro val,dst
0004                      local   skip
0005                      be      val,skip
0006                      bro     dst
0007                      skip:
0008                      endm
0009
0010                      cseg
0011                      b_ne    #0, over
0011+1                      local  _L0000000L_
0011+2  C 0000 310003      be      #0,_L0000000L_
0011+3  C 0003 11FB01      bro     over
0011+4                      _L0000000L_:
0012
0013                      org     200h
0014                      over:   b_ne    #0, under
0014+1                      local  _L0000001L_
0014+2  C 0200 310002      be      #0,_L0000001L_
0014+3  C 0203 0101      bro     under
0014+4                      _L0000001L_:
0015  C 0205 00          nop
0016  C 0206 00      under:  nop
0017                      end

```

LOCALで宣言された識別子はユニークな名前に置き換えられます

発生される名前は\_L#####L\_ (#####の部分は000000から始まる通し番号になります) という形式になります。

# IFDEF

定義済みならばアセンブルする

## 書式

IFDEF *symbol*

## 説明

IFDEF 疑似命令は *symbol* が既に定義されているときに、IFDEF 疑似命令以降の ELSE または ENDIF が現れるまでのソースプログラムをアセンブルします。

## 例

```
page:      1 <ifndef.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                               ; a sample program for IFDEF
0002                               chip    lc864024
0003           00000001 abc      equ     1
0004                               dseg
0005   D 0000           count: ds     1
0006
0007                               cseg    inblock
0008   C 0000 230010          mov     #10h, acc
0009                               ifdef   abc
0010   C 0003 8302              add     b
0011   C 0005 1200'              st      count
0012                               else
0013                               inc     acc
0014                               endif
0015   C 0007 A303              sub     c
0016                               ifdef   efg
0017                               add     count
0018                               endif
0019                               end
```

efgは定義されていないシンボルなので、  
この部分はアセンブルされません

abcは定義済みのシンボルなので、  
この部分がアセンブルされます

# IFDEF

未定義ならばアセンブルする

## 書式

IFDEF *symbol*

## 説明

IFDEF 疑似命令は *symbol* が未定義のときに、IFDEF 疑似命令以降、ELSE または ENDIF が現れるまでのソースプログラムをアセンブルします。

## 例

```
page:      1 <ifndef.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                ; a sample program for IFDEF
0002                chip   lc864024
0003          00000001 abc   equ   1
0004                dseg
0005  D 0000          count: ds   1
0006
0007                cseg   inblock
0008  C 0000 230010        mov   #10h, acc
0009                ifndef abc
0010                add    b
0011                st     count
0012                else
0013  C 0003 6300            inc    acc
0014                endif
0015  C 0005 A303          sub    c
0016                ifndef efg
0017  C 0007 8200'          add    count
0018                endif
0019                end
```

efgは定義されていないシンボルなので、  
この部分はアセンブルされません

abcは定義済みのシンボルなので、  
この部分がアセンブルされます

# IFB

オペランドが空白ならばアセンブルする

## 書式

IFB < argument >

## 説明

IFB 疑似命令は *argument* が空白のときに、IFB 疑似命令以降の ELSE または ENDIF が現れるまでのソ

## 例

```
page:      1 <ifb.ASM>
ERR  SEQ.    S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for IFB
0002                                chip    lc864016
0003      tifb  macro  arg
0004                                ifb    <<arg>>
0005                                inc     a
0006                                else
0007                                inc     b
0008                                endif
0009                                endm
0010
0011      tifb  xxx
0011+1     ifb    <xxx>
0011+2     inc     a
0011+3     else
0011+4     C 0000 6302      inc     b
0011+5     endif
0012      tifb
0012+1     ifb    <>
0012+2     C 0002 6300      inc     a
0012+3     else
0012+4     inc     b
0012+5     endif
0013      end
```

マクロの展開時に1つが  
取り除かれるので、二  
重の<>が必要です

IFBの引数が空なので、こちら  
側がアセンブルされます

IFBの引数が空ではないので、  
こちら側がアセンブルされます

# IFNB

オペランドが空白でなければアセンブルする

## 書式

IFNB < *argument* >

## 説明

IFNB 疑似命令は *argument* が空白でない場合に IFNB 疑似命令の以降の ELSE または ENDIF が現れるまでをアセンブルします。スペースやタブなどの文字も空白ではない文字としてあつかわれます。*argument* は < , > で囲む必要があります。

## 例

```

page:      1 <ifnb.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for IFNB
0002                                chip    lc864016
0003                                tifb    macro  arg
0004                                ifnb    <<arg>> ← マクロの展開時に1つが
0005                                inc     a      取り除かれるので、二
0006                                else
0007                                inc     b      重の<>が必要です
0008                                endif
0009                                endm
0010
0011                                tifb    xxx
0011+1                                ifnb    <xxx>
0011+2 C 0000 6300                                inc     a ←
0011+3                                else
0011+4                                inc     b
0011+5                                endif
0012                                tifb
0012+1                                ifnb    <>
0012+2                                inc     a
0012+3                                else
0012+4 C 0002 6302                                inc     b ←
0012+5                                endif
0013                                end

```

IFBの引数が空なので、こちら側がアセンブルされます

IFBの引数が空ではないので、こちら側がアセンブルされます

# IFE

式の値が 0 ならばアセンブルする

## 書式

IFE *expression*

## 説明

IFE 疑似命令は *expression* の値が 0 になるときに IFE 疑似命令以降の ELSE または EN... DIF が現れるまでのソースプログラムをアセンブルします。

## 例

```
page:      1 <ife.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for IFE
0002                                chip    lc866032
0003                                cseg
0004      00000003 aa      set     3
0005                                ife     aa-2
0006                                inc     70h
0007                                else
0008  C 0000 7270          dec     70h
0009                                endif
0010      00000002 aa      set     aa-1
0011                                ife     aa-2
0012  C 0002 6270          inc     70h
0013                                else
0014                                dec     70h
0015                                endif
0016                                end
```

式の値がゼロになるので、  
こちら側がアセンブルされます

式の値がゼロではないので、  
こちら側がアセンブルされます

# IFNE

式の値が 0 でなければアセンブルする

## 書式

IFNE *expression*

## 説明

IFNE 疑似命令は、*expression* の値が 0 でないときに IFNE 疑似命令以降の ELSE または ENDIF が現れるまでのソースプログラムをアセンブルします。

## 例

```

page:      1 <ifne.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for IFNE
0002                                chip    lc866032
0003                                cseg
0004      00000003 aa        set     3
0005                                ifne   aa-2
0006  C 0000 6270            inc     70h
0007                                else
0008                                dec     70h
0009                                endif
0010      00000002 aa        set     aa-1
0011                                ifne   aa-2
0012                                inc     70h
0013                                else
0014  C 0002 7270            dec     70h
0015                                endif
0016                                end

```

式の値がゼロになるので、  
こちら側がアセンブルされます

式の値がゼロではないので、  
こちら側がアセンブルされます

# IFIDN

2つの文字列が等しければアセンブルする

## 書式

IFIDN < string1 > , < string2 >

## 説明

IFIDN 疑似命令は、*string1* と *string2* が同じ場合に IFIDN 疑似命令以降の ELSE または ENDIF が現れるまでのソースプログラムをアセンブルします。*string1* と *string2* は < , > で囲む必要があります。< , > 内部のスペースやタブも含めて比較を行います。

## 例

```
page:      1 <ifidn.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                ; a sample program for IFIDN
0002                chip   lc866032
0003                cseg
0004      tifoldn  macro   arg1,arg2
0005                ifidn  <<arg1>>,<<arg2>>
0006                inc    a
0007                else
0008                dec    a
0009                endif
0010                endm
0011
0012                tifoldn same, same
0012+1              ifidn  <same>,<same>
0012+2  C 0000 6300      inc    a
0012+3              else
0012+4                dec    a
0012+5              endif
0013                tifoldn same, not_same
0013+1              ifidn  <same>,<not_same>
0013+2                inc    a
0013+3              else
0013+4  C 0002 7300      dec    a
0013+5              endif
0014                end
```

マクロの展開時に1つが  
取り除かれるので、二  
重の<>が必要です

2つの文字列が異なっているので、  
こちら側がアセンブルされます

2つの文字列が等しいので、  
こちら側がアセンブルされます

# IFDIF

2つの文字列が異なればアセンブルする

## 書式

IFDIF < string1 > , < string2 >

## 説明

IFDIF 疑似命令は、*string1* と *string2* が異なるときに IFDIF 疑似命令以降、ELSE または ENDIF が現れるまでのソースプログラムをアセンブルします。*string1* と *string2* は < , > で囲む必要があります。< , > 内部のスペースやタブも含めて比較を行います。

## 例

```

page:      1 <ifdif.ASM>
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001                      ; a sample program for IFDIF
0002                      chip    lc866032
0003                      cseg
0004      tifidn  macro  arg1,arg2
0005                      ifdif   <<arg1>>,<<arg2>>
0006                      inc     a
0007                      else
0008                      dec     a
0009                      endif
0010                      endm
0011
0012                      tifidn  same, same
0012+1                      ifdif  <same>,<same>
0012+2                      inc     a
0012+3                      else
0012+4      C 0000 7300                      dec     a
0012+5                      endif
0013                      tifidn  same, not_same
0013+1                      ifdif  <same>,<not_same>
0013+2      C 0002 6300                      inc     a
0013+3                      else
0013+4                      dec     a
0013+5                      endif
0014                      end

```

マクロの展開時に1つが  
取り除かれるので、二  
重の<>が必要です

2つの文字列が等しくないので、  
こちら側がアセンブルされます

2つの文字列が等しいので、  
こちら側がアセンブルされます

---

# ELSE

IF の条件の逆でアセンブルする

---

## 書式

ELSE

## 説明

ELSE 疑似命令は、先行する IF 条件命令の反対の条件で ENDIF が現れるまでのソースプログラムをアセンブルします。



例については「第 19 章 アセンブラ疑似命令」の「IFDEF 定義済みならばアセンブルする」などを参照してください。

# ENDIF

条件付きアセンブルの終了

## 書式

ENDIF

## 説明

条件アセンブルの終了を宣言します。

### 参照

例については「第 19 章 アセンブラ疑似命令」「IFDEF 定義済みならばアセンブルする」などを参照してください。

# .PRINTX

アセンブル中のディスプレイ上への表示

## 書式

`.PRINTX    "string "`

## 説明

.PRINTX 疑似命令は、アセンブル途中に文字列定数 *string* の内容をディスプレイ上に出します。

### 参照

文字列定数については「18.7 文字列定数」を参照してください。

## 例

### ソースプログラム

```
; a sample program for .PRINTX
chip    lc866000
switch equ    1
.printx "Start"

cseg    inblock
.printx "..CSEG"
ld      count
add     b
st      data1

ifdef   switch
.printx "Condition#1"
inc     data1
else
.printx "Condition#2"
dec     data1
endif

dseg
.printx "..DSEG"
count:  ds    1
data1:  ds    1
.printx "End"
end
```

IFDEF疑似命令により、この部分はアセンブルされませんので、対応する出力も現われません

### 画面上の表示

SANYO (R) LC86K series Macro As  
Copyright (c) SANYO Electric Co

Pass 1 .....

Start

..CSEG

Condition#1

..DSEG

End

Source file:    pprintx

Chip name:     LC866000

ROM size:      64K bytes

RAM size:      384 bytes

XRAM size:     128 bytes

Pass 2 .....

Start

..CSEG

Condition#1

..DSEG

End

# .LIST

リストの出力

## 書式

.LIST

## 説明

.LIST 疑似命令は、.XLIST 疑似命令によりリストファイルへの出力を中断していた状態を解除します。

## 例

```
; a sample program for LIST
chip    lc866200
cseg    inblock
mov     #00, count
ld      count
add     #10h
st      b
```

.XLISTのある行以降はリストファイルに現われなくなります。ただし行番号はカウントされ続けますので、ずれることはありません

```
.xlist
abc     equ    10h
        dseg
count:  ds     4
```

```
.list
```

```
cseg    inblock
ld      b
sub     #abc
st      count
end
```

.LIST以降は中断していたリストファイルへの出力が再開されます

```
page:    1 <plist.ASM>
```

ERR	SEQ.	S	LOC.	OBJ.	SOURCE STATEMENTS
	0001				; a sample program for LIST
	0002				chip    lc866200
	0003				cseg    inblock
	0004	C 0000	220000'		mov     #00, count
	0005	C 0003	0200'		ld      count
	0006	C 0005	8110		add     #10h
	0007	C 0007	1302		st      b
	0008				
	0014				.list
	0015				cseg    inblock
	0016	C 0000	0302		ld      b
	0017	C 0002	A110'		sub     #abc
	0018	C 0004	1200'		st      count
	0019				end

---

# .XLIST

リストの出力中断

---

## 書式

.XLIST

## 説明

.XLIST 疑似命令は、リストファイルに出力中にリストファイルへの出力を中断します。

### 参照

例については「第 19 章 アセン疑似命令」の「.LIST リストの出力」を参照してください。

# .MACRO

マクロ展開の出力

## 書式

.MACRO

## 説明

.MACRO 疑似命令は、マクロコール実行時にマクロ本体を展開して、リストファイルに出力します。

## 例

```

; a sample program for .MACRO
    chip    lc866200
t.mac  macro
    inc     a
    inc     b
    endm
cseg   inblock
    t.mac
    .xmacro
    t.mac
    .macro
    t.mac
    end
page:  1 <pmacro.ASM>
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for .MACRO
0002                                chip    lc866200
0003                                t.mac  macro
0004                                inc     a
0005                                inc     b
0006                                endm
0007
0008                                cseg   inblock
0009                                t.mac
0009+1  C 0000 6300          inc     a
0009+2  C 0002 6302          inc     b
0010                                .xmacro
0011                                t.mac
0012                                .macro
0013                                t.mac
0013+1  C 0008 6300          inc     a
0013+2  C 000A 6302          inc     b
0014                                end

```

.XMACROはマクロ展開の結果をリストファイルに出力しないようにします。展開されたステートメントが発生するコードも表示されなくなりますので注意してください。

.MACROはマクロ展開結果のリストファイルへの出力を再開します。

---

# .XMACRO

マクロ展開の出力中断

---

## 書式

.XMACRO

## 説明

.XMACRO 疑似命令は、マクロコール実行時にマクロ本体を展開してリストファイルに出力するのを一時中断します。



例については「第 19 章 アセンブラ疑似命令」の「.MACRO マクロ展開の出力」を参照してください。

# .IF

条件スキップの出力

## 書式

.IF

## 説明

.IF 疑似命令は、条件命令実行時にスキップされたソースプログラムのステートメントを展開し、リストファイルへ出力します。

## 例

```

; a sample program for .IF
chip    lc866200
t.if    macro    arg1
ifb     <<arg1>>
inc     a
else
inc     b
endif
endm
cseg    inblock
t.if
.xif
t.if    abc
.if
t.if    def
end

```

.XIFは条件アセンブル用の疑似命令によってスキップされたステートメントをリスト・ファイルに出力しないようにします。与えられた条件に一致し、スキップされなかった行は、この疑似命令に関係なくリストファイルに出力されます

.IFは条件アセンブル用の疑似命令によってスキップされたステートメントであってもリストファイルへは出力するようにします

```

page:    1 <pif.ASM>
ERR SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for .IF
0002                                chip    lc866200
0003
0004                                t.if    macro    arg1
0005                                ifb     <<arg1>>
0006                                inc     a
0007                                else
0008                                inc     b
0009                                endif
0010                                endm
0011                                cseg    inblock
0012                                t.if
0012+1                                ifb     <>
0012+2 C 0000 6300                                inc     a
0012+3                                else
0012+4                                inc     b
0012+5                                endif
0013                                .xif
0014                                t.if    abc
0014+1                                ifb     <abc>
0014+3 C 0002 6302                                inc     b
0014+5                                endif
0015                                .if
0016                                t.if    def
0016+1                                ifb     <def>
0016+2                                inc     a
0016+3                                else
0016+4 C 0004 6302                                inc     b
0016+5                                endif
0017                                end

```

---

# .XIF

## 条件スキップの出力中断

---

### 書式

.XIF

### 説明

.XIF 疑似命令は、条件命令実行時にスキップされたソースプログラムのステートメントをリストファイル上で展開しない状態にします。



#### 参照

例については「第 19 章 アセンブラ疑似命令」の「.IF 条件スキップの出力」を参照してください。

# INCLUDE

ファイルの読み込み

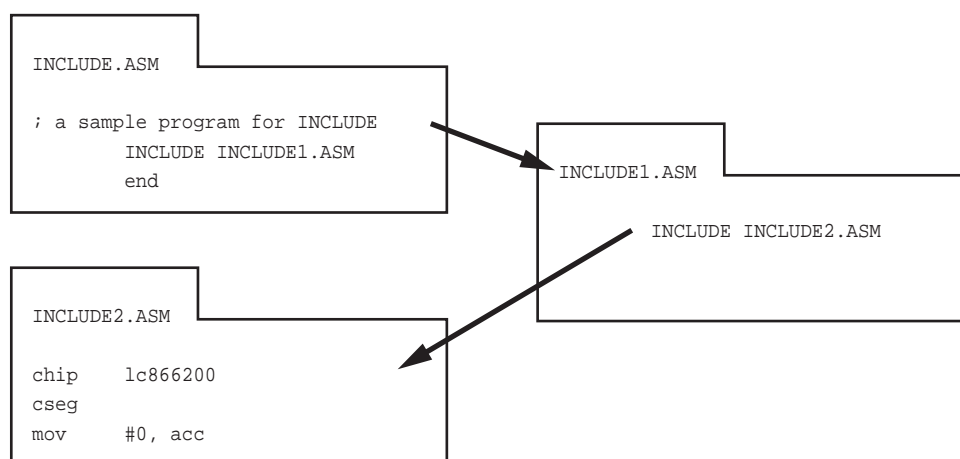
## 書式

```
INCLUDE filename
```

## 説明

INCLUDE 疑似命令は、ソースプログラムをアセンブル途中で *filename* で指定されたソースファイルを読み込み、アセンブルするための疑似命令です。*filename* は拡張子も含めて指定する必要があります。INCLUDE 疑似命令のネスティングは9レベルまで可能です。また、読み込んだファイル中に END 疑似命令があると、そこでアセンブルを終了します。

## 例



M86K INCLUDE INCLUDE INCLUDE

```

page:      1 <include.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                ; a sample program for INCLUDE
0002                INCLUDE INCLUDE1.ASM
1/0001                INCLUDE INCLUDE2.ASM
2/0001                chip    1c866200
2/0002                cseg
2/0003      C 0000 230000    mov     #0, acc
0003
  
```

↑ インクルードのネストレベルを示す表示

# TITLE

## リストタイトルの指定

### 書式

TITLE    *string*

### 説明

TITLE 疑似命令は、*string* をリストファイルの見出しに指定します。*string* は文字列定数とは異なり、引用符などで囲みません。また、円記号 ( ¥ ) で始まる任意コードの入力もできません。

### 例

すべてのページのこの部分にstringが表示されます

```
page:      1 <title.ASM> sample program's title for the listing
ERR  SEQ.      S LOC. OBJ.  SOURCE STATEMENTS
      0001              ; a sample program for TITLE
      0002                      TITLE  sample program's title for the listing
      0003                      chip   lc864024
      0004                      cseg
      0005      C 0000 00      nop
      0006                      end
```

# PAGE

改ページ

## 書式

PAGE

## 説明

PAGE 疑似命令は、リストファイルを出力中に強制的に改ページを実行します。改ページ文字は、本疑似命令の直前に挿入されます。

## 例

ソースファイル

```
; a sample program for PAGE
chip    lc866032
page
cseg
page
nop
page
end
```

リストファイル

```
page:    1 <page.ASM>
ERR SEQ.    S LOC. OBJ.    SOURCE STATEMENTS
0001                                ; a sample program for PAGE
0002                                chip    lc866032
```

```
page:    2 <page.ASM>
ERR SEQ.    S LOC. OBJ.    SOURCE STATEMENTS
0003                                page
0004                                cseg
```

```
page:    3 <page.ASM>
ERR SEQ.    S LOC. OBJ.    SOURCE STATEMENTS
0005                                page
0006    C 0000 00                                nop
```

```
page:    4 <page.ASM>
ERR SEQ.    S LOC. OBJ.    SOURCE STATEMENTS
0007                                page
0008                                end
```

---

# CHIP

## アセンブル対象チップの定義

---

### 書式

CHIP *chipname*

### 説明

CHIP 疑似命令は、アセンブラ本体にどのチップを対象としてアセンブルを行うかを知らせます。アセンブラは *chipname* により予約語の切り換え、およびメモリサイズのチェックを行ないます。この疑似命令は、1つのソースファイルの冒頭部分、他の命令や疑似命令に先立って1つ記述されます。本疑似命令が見当たらない場合、環境変数 CHIPNAME の値が参照されます。また、本疑似命令によって宣言されるチップの名前が環境変数 CHIPNAME の値と異なる場合、警告レベルのエラーとなります。

ビジュアルメモリ用アプリケーションを開発する場合は、chipname を LC868700 にしてください。

# COMMENT

オブジェクトファイルへのコメント

## 書式

COMMENT *comment\_string*

## 説明

COMMENT 疑似命令は、アセンブルされたオブジェクトに出力されるコメントを与えます。*comment\_string* は文字列定数とは異なり、引用符などで囲みません。また、円記号(¥)で始まる任意コードの入力もできません。*comment\_string* がオブジェクトファイルの 680 バイト目以降に格納されます。コメントとして使用できる文字数は最大 255 文字です。

## 例

ソースファイル

```
; a sample program for COMMENT
chip    lc866024
comment This is a comment string embedded into OBJ file
cseg
nop
end
```

オブジェクトファイルのダンプ (必要部分の抜粋)

文字数 (1バイト)

00000260	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00	.....
00000270	00 00 00 00 00 00 60 00 00-80 01 00 00 80 00 00 00	.....`.....
00000280	C6 92 40 2B 4D 38 36 4B-20 20 20 20 63 6F 6D 6D	**+M86K comm
00000290	65 6E 74 2E 41 53 4D 20-63 6F 6D 6D 65 6E 74 20	ent.ASM comment
000002A0	4C 43 38 36 36 30 32 34-30 54 68 69 73 20 69 73	LC8660240This is
000002B0	20 61 20 63 6F 6D 6D 65-6E 74 20 73 74 72 69 6E	a comment strin
000002C0	67 20 65 6D 62 65 64 64-65 64 20 69 6E 74 6F 20	g embedded into
000002D0	4F 42 4A 20 66 69 6C 65-00 00 01 01 00 01 00 05	OBJ file.....
000002E0	00 01 00 00 00 00 00 00-00 00 E0 00 00 00 00 C4	.....*
000002F0	00 00 00 00 C4 00 00 00-00 24 00 00 01 00 04 01	....*....\$.....
00000300	00 00 00 24	...\$

# WIDTH

## リストファイルの桁数指定

### 書式

WIDTH *number*

### 説明

WIDTH 疑似命令は、リストファイルの桁数 (1 行当りの文字数) を指定します。*number* には 72 以上 132 以下の数値が指定できますが、できるだけソースファイルの桁数 + 28 以上の値を指定することをお勧めします。また、本疑似命令は 1 つのソースファイルにいくつでも記述することができますが、通常はファイルの冒頭部分に 1 度だけ記述します。なお、本疑似命令がない場合のデフォルトのリストファイルの桁数は 132 です。

### 例

WIDTHはパス 1、パス 2 ともに評価されますが、リストファイル生成はパス 2 でのみ行われるため、パス 1 で行った最後のWIDTHの評価の結果が反映されて、この行はここで折り返されることになります

```
1 2 3 4 5 6 7 8
1234567890123456789012345678901234567890123456789012345678901234567890
page: 1 <width.ASM>
ERR SEQ. S LOC. OBJ. SOURCE STATEMENTS
0001 ; a sample program for WIDTH
0002 chip lc866200
0003 cseg ; this is a long line to indicate WIDTH's effect
0003 e WIDTH's effect
0004 WIDTH 72
0005 C 0000 00 nop ; this is also a long line to indicate WIDTH's effect
0005 to indicate WIDTH's effect
0006 WIDTH 78
0007 end
```

72文字目に「復帰改行」文字が挿入されますので、ここで折り返されます

# BANK

RAM 領域のバンク指定

## 書式

BANK *expression*

## 説明

BANK 疑似命令は、DSEG 疑似命令以降に記述される RAM 領域の DS 疑似命令で定義されるシンボルにバンク番号を与えます。

## 例

```

page:      1 <bank.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                                ; a sample program for BANK
0002                                chip    lc866032
0003                                cseg    inblock
0004
0005      C 0000 220000'          mov     #0,data1
0006
0007      C 0003 6200'            inc     data1
0008      C 0005 0200'            ld      data1
0009      C 0007 1201'            st      data2
0010
0011      C 0009 6200'            inc     dataa
0012      C 000B 0200'            ld      dataa
0013      C 000D 1202'            st      datac
0014
0015                                dseg
0016                                bank    0
0017      D 0000          data1:  ds      1
0018      D 0001          data2:  ds      1
0019      D 0002          data3:  ds      1
0020
0021                                bank    1
0022      D 0000          dataa:  ds      1
0023      D 0001          datab: ds      1
0024      D 0002          datac: ds      1
0025
0026                                end

```

bank 0

bank 1

これらのシンボルはバンク1に  
割り当てられます

これらのシンボルはバンク0に  
割り当てられます

# CHANGE

フラッシュメモリ / ROM をまたぐジャンプ

## 書式

CHANGE *symbol*

## 説明

LC86800 シリーズにおいて、フラッシュメモリ上のコードと ROM 上のコード（システム BIOS）で相互に実行を切り換える場合に使用する特殊なジャンプ命令です。オペランドとして指定できる *symbol* は、疑似命令 OTHER\_SIDE\_SYMBOL で宣言されたものに限ります。なお、この疑似命令は LC86800 シリーズ専用であり、それ以外の場合にはエラーとなります。

ビジュアルメモリでは、OS プログラムの呼び出しでこの命令を利用します。

参照

OS プログラムの呼び出し方については『ビジュアルメモリ ハードウェアマニュアル』の「システム BIOS 編」を参照してください。

## 例

```
page:      1 <change.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for CHANGE
0002                                chip    lc868032
0003                                other_side_symbol    far_away
0004
0005                                cseg
0006      C 0000 B80D21'      change  far_away
0006      C 0003 0000'
```

# RADIX

## デフォルトの基数の指定

### 書式

RADIX *expression*

### 説明

RADIX 疑似命令は、基数を明示しない書式で記述された数値定数を値に変換する際の基数を指定します。*expression* には値が 2, 8, 10, 16 のいずれかになるような定数式が記述できます。指定された基数は本疑似命令以降、次の RADIX 疑似命令による指定まで有効となります。また、本疑似命令による指定がない場合のデフォルトの基数は 10 です。

### 例

xxx	SET	10	デフォルトの基数は10により $10_{10}$ と解釈されます
	RADIX	16	
xxx	SET	10	基数が16に設定されたことにより $16_{10}$ と解釈されます
	RADIX	2	
xxx	SET	10	基数が2に設定されたことにより $2_{10}$ と解釈されます

# JMPO

最適な JMP 命令の発生

## 書式

JMPO *expression*

## 説明

JMPO 疑似命令は、*expression* と現在のロケーションを比較し、同一ブロック内へのジャンプ（アドレスの下位 12 ビット以外の部分が同一）であるならば JMP を発生し、そうでないか、あるいはジャンプ先アドレスが外部シンボルである場合のようにその値が特定できなければ JMPF を発生します。

## 例

同一メモリブロック内のときはJMP命令になります

```
page:      1 <jmpo.ASM>
ERR  SEQ.   S LOC. OBJ.  SOURCE STATEMENTS
0001                      ; a sample program for JMPO
0002                      chip    1c866032
0003                      cseg
0004  C 0000 2803'      jmpo    near ←
0005  C 0002 00          nop
0006  C 0003 00    near:  nop
0007  C 0004 211000'    jmpo    far ←
0008
0009                      org     1000h
0010  C 1000 00    far:    nop
0011                      end
```

異なるメモリブロックのときにはJMPF命令が発生します

# BRO

最適な BR 命令の発生

## 書式

BRO *expression*

## 説明

BRO 疑似命令は、*expression* と現在のロケーションを比較し、分岐先が + 127 ~ - 128 の範囲内にあるならば BR を発生し、+ 127 ~ - 128 の範囲外にあるならば BRF を発生します。

## 例

分岐先が+127~-128の範囲にあるのでBR命令になります

```
page:      1 <bro.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                                ; a sample program for BRO
0002                                chip    lc866032
0003                                cseg
0004      C 0000 0101      bro    near ←
0005      C 0002 00                                nop
0006      C 0003 00      near:   nop
0007      C 0004 11FA00     bro    far ←
0008
0009                                org     100h
0010      C 0100 00      far:    nop
0011                                end
```

分岐先が+127~-128の範囲外にあるのでBRF命令を発生します

# CALLO

最適な CALL 命令の発生

## 書式

CALLO *expression*

## 説明

CALLO 疑似命令は、*expression* と現在のロケーションを比較し、同一ブロック内へのジャンプ（アドレスの下位 12 ビット以外の部分が同一）であるならば CALL を発生し、そうでないか、あるいはジャンプ先アドレスが外部シンボルである場合のようにその値が特定できなければ CALLF を発生します。

## 例

同一メモリブロック内のときはCALL命令になります

```
page:      1 <CALLO.ASM>
ERR  SEQ.   S LOC. OBJ.   SOURCE STATEMENTS
0001                               ; a sample program for CALLO
0002                               chip    lc866032
0003                               cseg
0004   C 0000 0805'          CALLo  near ←
0005   C 0002 201000'        CALLo  far  ←
0006
0007   C 0005 00      near:  nop
0008   C 0006 A0              ret
0009
0010                               org     1000h
0011   C 1000 00      far:   nop
```

異なるメモリブロックのときにはCALLF命令を発生します

# BZO

アドレスエラーの発生しない BZ 命令の発生

## 書式

BZO *expression*

## 説明

BZO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のない BZ 命令に相当する命令コードを発生するマクロです。BZO マクロは BZ 命令の論理の反転である BNZ 命令と BRO 命令を使用しています。*expression* に分岐先を記述します。

## コード発生のマクロ

```
; *** Branch near relative address if accumulator is zero ***
bzo macro r8
    local _next_
    bnz _next_
    bro r8
_next_:
endm
```

---

# BNZO

アドレスエラーの発生しないBNZ命令の発生

---

## 書式

BNZO *expression*

## 説明

BNZO マクロ命令は、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のないBNZ命令に相当する命令コードを発生するマクロです。BNZO マクロはBNZ命令の論理の反転であるBZ命令とBRO命令を使用しています。*expression* に分岐先を記述します。

## コード発生のマクロ

```
; *** Branch near relative address if accumulator is not zero ***
bnzo macro r8
    local _next_
    bz _next_
    bro r8
_next_:
endm
```

# BPO

アドレスエラーの発生しない BP 命令の発生

## 書式

BPO *expression*

## 説明

BPO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のない BP 命令に相当する命令コードを発生するマクロです。BPO マクロは BP 命令と BR 命令、および BRO 命令を使用しています。*expression* に分岐先を記述します。

## コード発生のマクロ

```
; *** Branch near relative address if direct ビット is positive ***
bpo macro d9,b3,r8
    local _next_
    local _true_
    bp d9,b3,_true_
    br _next_
_true_: bro r8
_next_:
endm
```

---

# BPCO

アドレスエラーの発生しないBPC命令の発生

---

## 書式

*BPCO expression*

## 説明

BPCO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のないBPC命令に相当する命令コードを発生するマクロです。BPCO マクロはBPC命令とBR命令、およびBRO命令を使用しています。*expression* に分岐先を記述します。

## コード発生のマクロ

```
; *** Branch near relative address if direct ビット is positive,  
;      and clear ***  
bpc macro d9,b3,r8  
    local _next_  
    local _true_  
    bpc d9,b3,_true_  
    br _next_  
_true_: bro r8  
_next_:  
    endm
```

# BNO

アドレスエラーの発生しないBN命令の発生

## 書式

*BNO expression*

## 説明

BNO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のないBN命令に相当する命令コードを発生するマクロです。BNO マクロはBN命令とBR命令、およびBRO命令を使用しています。*expression* はBN命令と同じものを記述します。

## コード発生のマクロ

```
; *** Branch near relative address if direct ビット is negative ***
bno macro d9,b3,r8
    local _next_
    local _true_
    bn d9,b3,_true_
    br _next_
_true_: bro r8
_next_:
endm
```

---

# DBNZO

アドレスエラーの発生しない DBNZ 命令の発生

---

## 書式

DBNZO *expression*

## 説明

DBNZO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のない DBNZ 命令に相当する命令コードを発生するマクロです。DBNZO マクロは DBNZ 命令と BR 命令、および BRO 命令を使用しています。*expression* は DBNZ 命令と同じものを記述します。

## コード発生のマクロ

```
; *** Decrement direct バイト and branch near relative address
;      if direct バイト is not zero ***
dbnzo macro d9,r8
    local _next_
    local _true_
    dbnz d9,_true_
    br _next_
    _true_: bro r8
    _next_:
endm
```

# BEO

アドレスエラーの発生しないBE命令の発生

## 書式

BEO *expression*

## 説明

BE マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のないBE命令に相当する命令コードを発生するマクロです。BE マクロはBNE 命令と BRO 命令を使用しています。*expression* は BE 命令と同じものを記述します。

## コード発生のマクロ

```
; *** Compare immediate data or accumulator and branch
;      near relative address if equal ***
beo macro arg0,arg1,arg2
    local _next_
    local _txen_
    ifb <<arg2>>
        bne arg0,_next_
        bro arg1
    _next_:
    else
        bne arg0,arg1,_txen_
        bro arg2
    _txen_:
    endif
endm
```

---

# BNEO

アドレスエラーの発生しない BNE 命令の発生

---

## 書式

BNEO *expression*

## 説明

BNEO マクロは、同一ソース内の同一セグメントの分岐先とこの命令の置かれる位置との差分に制限のない BNE 命令に相当する命令コードを発生するマクロです。BNE マクロは BE 命令と BRO 命令を使用しています。*expression* は BNE 命令と同じものを記述します。

## コード発生のマクロ

```
; *** Compare immediate data or accumulator and branch
;      near relative address if equal ***
bneo macro arg0,arg1,arg2
    local _next_
    local _txen_
    ifb <<arg2>>
        be  arg0,_next_
        bro arg1
    _next_:
    else
        be  arg0,arg1,_txen_
        bro arg2
    _txen_:
    endif
endm
```

## 第 20 章

## LC86K 命令概要

ここでは、個々の命令を説明する前に、フラグの動作、アドレッシングについてを説明します。

## 20.1 命令概要

### 20.1.1 算術演算命令

算術演算命令は、アキュムレータを中心とした四則演算、インクリメント、またはデクリメントを行う命令です。四則演算の結果により、次のように CY, AC, OV が設定されます。

#### CY (キャリーフラグ)

演算命令	演算結果	CY
加算命令実行時	ビット7 (MSB) からの桁上がりが発生する場合	1
	ビット7 (MSB) からの桁上がりが発生しない場合	0
減算命令や比較命令実行時	ビット7 (MSB) のためのボローが必要な場合	1
	ビット7 (MSB) のためのボローが不要な場合	0
乗除算命令の実行時	-	0

#### AC (補助キャリーフラグ)

演算命令	演算結果	AC
加算命令実行時	ビット3からの桁上がりが発生する場合	1
	ビット3からの桁上がりが発生しない場合	0
減算命令実行時	ビット3のためのボローが必要な場合	1
	ビット3のためのボローが不要な場合	0

## OV (オーバーフローフラグ)

演算命令	演算結果	OV
加減算命令実行時	ビット7から桁上がりが発生し、ビット6から桁上がりが発生しない場合	1
	ビット6から桁上がりが発生し、ビット7から桁上がりが発生しない場合	1
	符号付き変数の加減算命令実行時にオーバーフロー・エラーが発生した場合	1
乗算命令実行時	上記以外の場合	0
除算命令実行時	積が256以上の場合	1
	積が255以下の場合	0
除算命令実行時	除数が0の場合	1
	除数が0以外の場合	0

### 20.1.2 論理演算命令

論理演算命令は、論理演算やローテートを行う命令です。RORC、ROLC 命令実行時には CY も影響を受けます。

### 20.1.3 データ転送命令

データ転送命令は、RAM、特殊機能レジスタ (SFR) へのデータの書き込み、読み込み、データの退避または交換を行います。

### 20.1.4 ジャンプ命令

ジャンプ命令は、無条件に目的の命令に制御を移す命令です。

### 20.1.5 条件分岐命令

条件分岐命令は、命令が指定する条件が満たされている (真) か、いない (偽) かを判定し、真ならば目的の番地へ分岐します。偽ならば分岐は行われず、次の命令の実行に移ります。

BE, BNE 命令は、8 ビットデータどうしの比較によって分岐する命令で、比較結果により次のように CY をセットまたはリセットします。

	オペランド			キャリー フラグ (CY)
	#i8, r8	d9, r8	@Rj, #i8, r8	
大小 関係	#i8 > (ACC)	(d9) > (ACC)	#i8 > ((Rj))	1
	#i8 = (ACC)	(d9) = (ACC)	#i8 = ((Rj))	0
	(d9) < (ACC)	(d9) < (ACC)	#i8 < ((Rj))	0

### 20.1.6 サブルーチン命令

サブルーチン命令は、目的の命令に制御を移すための無条件分岐を行います。分岐後、復帰命令 (RET、RETI) によって CALL 命令に続く命令に制御を戻すために、スタックにア

ドレスを格納します。スタックは RAM にあり、スタックポインタ ( SP ) によって指定します。サブルーチンのネスティングレベルに応じてスタックに使用する RAM 領域を確保しておく必要があります。

ビジュアルメモリのスタックは、RAM のバンク 0 に確保されます。また、ビジュアルメモリ初期化時にシステム BIOS が、80H に設定します。  
スタックは、80H から 0FFH へと上位へ向って消費されます。なお、内蔵時計機能もスタックを 20 バイト消費します。

## 20.1.7 ビット操作命令

ビット操作命令は、指定した RAM または特殊機能レジスタ ( SFR ) の内容をビット単位で操作する命令です。

## 20.1.8 その他の命令

何もせず 1 サイクルクロック消費する NOP 命令があります。

## 20.1.9 マクロ命令

専用の標準マクロ命令です。ROM 内システム BIOS とフラッシュメモリのアプリケーションプログラムの実行を切り換えます。

## 20.1.10 アドレッシング

フラッシュメモリや RAM、特殊機能レジスタ ( SFR ) のアドレッシングには、いくつかの方法があります。

## 20.1.11 プログラムメモリのアドレッシング

ジャンプ命令や分岐命令、サブルーチン命令では、ジャンプ ( 分岐 ) 先のプログラム ROM のアドレスを命令コードで指定します。この場合、次のアドレッシングによってアドレスが指定されます。

### r8 ( 8 ビット相対アドレッシング )

現在実行中の次の命令の先頭番地を中心として、- 128 ~ + 127 番地のアドレスの範囲内にジャンプ ( 分岐 ) が可能です。符号付きの 8 ビットデータで表されます。

[ 80H ~ 7FH : - 128 ~ + 127 ]

### r16 ( 16 ビット相対アドレッシング )

64K バイトのフラッシュメモリ空間内のどこにでもジャンプが可能です。符号なしの 16 ビットデータで表されます。

[ 0000H ~ FFFFH : + 0 ~ + 65535 ]

### a12 ( 12 ビット絶対アドレッシング )

現在実行中の次の命令の先頭番地 [ PC15 ~ PC00 ] の PC15 ~ PC12 ( カレントページ ) をそのままに、PC11 ~ PC00 を 12 ビットのアドレッシングデータ [ 000H ~ FFFH ] で置き換えます。ページ ( PC15 ~ PC12 ) 内のジャンプが可能です。

#### 注意

JMP 命令、CALL 命令がページの最終部分にある場合は、カレントページが変わるので注意が必要です。

### a16 ( 16 ビット絶対アドレッシング )

64K バイトのフラッシュメモリ空間内のどこにでもジャンプが可能です。16 ビットデータがそのままアドレスを表します。

[ 0000H ~ FFFFH : 0 ~ 65535 ]

### テーブルジャンプについて

スタックにジャンプ先のアドレスを設定しておき、RET 命令でその値を強制的にプログラムカウンタ ( PC ) に入れることによって、ジャンプすることができます。

例 1 では、1 行目でスタックポインタ ( SP ) を 09H に設定しています。ここで RET 命令を実行すると、RAM08H が上位バイト、RAM07H が下位バイトとしたアドレスへジャンプするので、2, 3 行目でジャンプ先アドレスを設定します。

ジャンプ先を PC = 0C13H としているので、2 行目では下位バイトの 13H を、3 行目では上位バイトの 0CH をセットしています。4 行目の RET 命令を実行すると、SP は 07H になり、0C13H へジャンプします。しかし、例 1 では SP 値が既知でなければならないので、通常は例 2 に示すように PUSH 命令を用います。

#### 例 1

```
MOV #09H, SP
MOV #13H, 07H
MOV #0CH, 08H
RET
```

#### 例 2

```
MOV #13H, ACC
PUSH ACC
MOV #0CH, ACC
PUSH ACC
RET
```

例 3 は、RAM70H のデータによって 00H ~ 7FH まで ( 128 とおり ) の分岐を行なっています。

1, 2 行目で分岐先の下位アドレスを、4 行目では上位アドレスを設定しています。6 行目の RET 命令を実行することによって、7, 8 行目のジャンプテーブルへ分岐し、さらにそこに記述された分岐先へジャンプします。

このような手法を「テーブルジャンプ」といい、条件によって複数のアドレスへ分岐する

場合に有効です。

### 例 3

```

A0: LD  070H
    ROL
    ADD #LOW(A1)
    PUSH ACC
    MOV #HIGH(A1),ACC
    PUSH ACC
    RET
;
    ORG 0C00H
A1: JMP B00      ジャンプテーブル

    JMP B7F
;
B00: XXXXXX

```

## 20.1.12 RAM と特殊機能レジスタ (SFR) のアドレッシング

### d9 (直接アドレッシング)

d8 ~ d0 の 9 ビットにより RAM または SFR を直接指定します。

アドレス 000H ~ 0FFH ..... RAM を指定します。

アドレス 100H ~ 1FFH ..... SFR を指定します。

### b3 (ビットアドレッシング)

ビット操作命令 (SET1, CLR1, NOT1) と BP, BPC, BN 命令では d9 (直接アドレッシング) と合わせて、3 ビットのビットアドレッシングデータを使用し、RAM または SFR の特定のビットを指定します。

	MSB							LSB
	2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>
ビットアドレッシングデータ .....	7	6	5	4	3	2	1	0
	(111)	(110)	(101)	(100)	(011)	(010)	(001)	(000)

### @ Rj (間接アドレッシング)

間接アドレッシングとは、特定の RAM に、実際にアクセスする RAM または SFR のアドレスをあらかじめ設定しておき、その特定の RAM をアクセスすることによって RAM または SFR を指定することです。

参照

間接アドレッシングについては『ビジュアルメモリハードウェアマニュアル』もあわせてご覧ください。

この特定の RAM を間接アドレスレジスタといい、@ R0, @ R1, @ R2, @ R3 で表します。間接アドレスレジスタは、2 ビットの間接アドレッシングデータ (j1, j0) を指定するこ

とにより@ R0～@ R3のうち1つが使用されます。

各 RAM バンクの先頭 16 バイト ( 00H～0FH ) には、次の表のとおりに 4 つの間接アドレスレジスタ群 ( バンク ) が割り付けられています。この RAM バンクは RAMBK0 ( PSW のビット 1 ) で設定します。また、間接アドレスレジスタのバンクは IRBK1 , 0 ( PSW のビット 4 , 3 ) で設定します。

なお、間接アドレッシング命令実行時には、間接アドレスレジスタや間接アドレスレジスタで指定される RAM として、IRBK1 , 0 と RAMBK0 で設定されている RAM バンク上の RAM が使用されます。リセット時は、IRBK0 , 1 共に '0' となり、また RAMBK0 も '0' となるので、@ R0 , @ R1 , @ R2 , @ R3 の絶対アドレスはそれぞれ RAM バンク 0 の 00H , 01H , 02H , 03H になります。

間接アドレスレジスタ ..... @ R3      @ R2      @ R1      @ R0

間接アドレッシングデータ ( j1, j0 ) ... ( 11 ) ( 10 ) ( 01 ) ( 00 )

間接アドレッシング レジスタ マップ

間接 アドレス レジスタ名	機能	バンク 0 ( IRBK1 = 0 ) ( IRBK0 = 0 )	バンク 1 ( IRBK1 = 0 ) ( IRBK0 = 1 )	バンク 2 ( IRBK1 = 1 ) ( IRBK0 = 0 )	バンク 3 ( IRBK1 = 1 ) ( IRBK0 = 1 )
@ R0	RAM アクセス	RAM 00H	RAM 04H	RAM 08H	RAM 0CH
@ R1	RAM アクセス	RAM 01H	RAM 05H	RAM 09H	RAM 0DH
@ R2	SFR アクセス	RAM 02H	RAM 06H	RAM 0AH	RAM 0EH
@ R3	SFR アクセス	RAM 03H	RAM 07H	RAM 0BH	RAM 0FH

## 間接アドレッシングの使用例

間接アドレスレジスタを用いた演算例です。

例 1 では、2 行目で RAM ( アドレス 00H ) にイミディエイトデータ 68H をセットしています。ここで間接アドレスレジスタとして RAM ( アドレス 00H ) を用いると、RAM ( アドレス 68H ) がアクセスされます。たとえば、3 行目では間接アドレスレジスタ ( @ R0 ) で指定される RAM ( アドレス 68H ) にイミディエイトデータ 10H をセットしています。

また、5 行目では間接アドレスレジスタ ( @ R0 ) で指定される RAM ( アドレス 68H ) とアキュムレータ ( ACC ) を加算しています。

### 例 1

```
MOV #055H,ACC
MOV #068H,00H
MOV #010H,@R0
ADD #015H
ADD @R0
```



次に、間接アドレッシングで SFR を指定した例を示します。

例 2 では 1, 2 行目で PSW のビット 4, 3 をクリアして、RAM00H~03H を間接アドレスレジスタとして使えるように設定しています。4 行目では RAM02H にイミディエイトデータ 02H をセットします。ここで間接アドレスレジスタとして RAM ( アドレス 02H ) を用いると、RAM ( アドレス 02H ) がアクセスされます。たとえば、5 行目では間接アドレスレジスタ ( @ R2 ) で指定される SFR ( アドレス 02H : B レジスタ ) にイミディエイトデータ 12H をセットしています。また、6 行目では間接アドレッシングされた B レジスタをインクリメントしています。

## 例 2

```
CLR1 PSW,4
CLR1 PSW,3
MOV #0ACH,ACC
MOV #002H,02H
MOV #012H,@R2
INC @R2
```

次に、PSW でバンクを換えて、間接アドレッシングで SFR を指定した例を示します。

例 3 では 1, 2 行目で PSW のバンクを 2 にセットして、RAM08H~0BH を間接アドレスレジスタとして使えるように設定しています。4 行目では、RAM0BH にイミディエイトデータ 02H をセットします。ここで間接アドレスレジスタとして RAM ( アドレス 0BH ) を用いると、RAM ( アドレス 02H ) がアクセスされます。たとえば、5 行目では間接アドレスレジスタ ( @ R2 ) で指定される SFR ( アドレス 02H : B レジスタ ) にイミディエイトデータ 12H をセットしています。また、6 行目では間接アドレッシングされた B レジスタをインクリメントしています。

## 例 3

```
SET1 PSW,4
CLR1 PSW,3
MOV #0ACH,ACC
MOV #002H,0BH
MOV #012H,@R2
INC @R2
```



第 21 章

# 命令セットリファレンス

LC68K 命令セットには、70 種類もの豊富な命令群があります。これらは、45 種類のオペコードに分類され、機能別には次の 8 種類に分類されます。

算術演算命令	ADD , ADDC , SUB , SUBC , INC , DEC , MUL , DIV
論理演算命令	AND , OR , XOR , ROL , ROLC , ROR , RORC
データ転送命令	LD , ST , MOV , LDC , PUSH , POP , XCH
ジャンプ命令	JMP , JMPF , BR , BRF
条件分岐命令	BZ , BNZ , BP , BPC , BN , DBNZ , BE , BNE
サブルーチン命令	CALL , CALLF , CALLR , RET , RETI
ビット操作命令	CLR1 , SET1 , NOT1
その他の命令	NOP
マクロ命令	CHANGE

# 算術演算命令

## ADD # i8

ADD immediate data to accumulator

命令コード	1 0 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 (81H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) + # i8
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

### 説明

アキュムレータ (ACC) の内容とイミディエイトデータ (i7~i0) を加算し、その結果を ACC に転送します。

### 例

		ACC	CY	AC	OV
MOV	#055H, ACC	55H	-	-	-
ADD	#013H	68H	0	0	0
ADD	#00AH	72H	0	1	0
ADD	#00FH	81H	0	1	1
ADD	#080H	01H	1	0	1

# ADD d9

ADD direct byte to accumulator

命令コード	1 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 ( 82H ~ 83H )
バイト数	2
サイクル数	1
機能	( ACC ) ( ACC ) + ( d9 )
影響を受けるフラグ	CY , AC , OV
割り込みの受付	可

## 説明

アキュムレータ ( ACC ) の内容と d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容を加算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H, ACC	55H	-	-	-	-
MOV	#068H, 023H	55H	68H	-	-	-
ADD	#00CH	61H	68H	0	1	0
ADD	023H	C9H	68H	0	0	1

## 例 2

		ACC	B	CY	AC	OV
MOV	#070H, ACC	70H	-	-	-	-
MOV	#095H, B	70H	95H	-	-	-
ADD	#002H	72H	95H	0	0	0
ADD	B	07H	95H	1	0	0

# ADD @Rj

ADD indirect byte to accumulator

命令コード	1 0 0 0 0 1j1j0 (84H~87H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) + ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容と j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容を加算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H, ACC	55H	-	-	-	-	-
MOV	#068H, 000H	55H	68H	-	-	-	-
MOV	#010H, @R0	55H	68H	10H	-	-	-
ADD	#015H	6AH	68H	10H	0	0	0
ADD	@R0	7AH	68H	10H	0	0	0

## 例 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH, ACC	AAH	-	-	-	-	-
MOV	#004H, 002H	AAH	04H	-	-	-	-
MOV	#055H, @R2	AAH	04H	55H	-	-	-
ADD	#001H	ABH	04H	55H	0	0	0
ADD	@R2	00H	04H	55H	1	1	0

# ADDC # i8

ADD immediate data and carry flag to accumulator

命令コード	1 0 0 1 0 0 0 1 i7i6i5i4i3i2i1i0 (91H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) + (CY) + # i8
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容とキャリーフラグ (CY) とイミディエイトデータ (i7~i0) を加算し、その結果を ACC に格納します。

## 例

		ACC	CY	AC	OV
MOV	#055H, ACC	55H	-	-	-
ADD	#013H	68H	0	0	0
ADDC	#00AH	72H	0	1	0
ADDC	#00FH	81H	0	1	1
ADDC	#080H	01H	1	0	1
ADDC	#001H	03H	0	0	0

# ADDC d9

ADD direct byte and carry flag to accumulator

命令コード	1 0 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (92H~93H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) + (CY) + (d9)
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容とキャリーフラグ (CY) と d8~d0 で指定される RAM、または特殊機能レジスタ (SFR) の内容を加算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H, ACC	55H	-	-	-	-
MOV	#068H, 023H	55H	68H	-	-	-
ADD	#00CH	61H	68H	0	1	0
ADDC	023H	C9H	68H	0	0	1
SET1	PSW, 7	C9H	68H	1	0	1
ADDC	023H	32H	68H	1	1	0

## 例 2

		ACC	B	CY	AC	OV
MOV	#070H, ACC	70H	-	-	-	-
MOV	#095H, B	70H	95H	-	-	-
ADD	#002H	72H	95H	0	0	0
ADDC	B	07H	95H	1	0	0
ADDC	B	9DH	95H	0	0	0

# ADDC @ Rj

ADD indirect byte and carry flag to accumulator

命令コード	1 0 0 1 0 1j1j0 (94H~97H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) + (CY) + ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容とキャリーフラグ (CY) と j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容を加算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H, ACC	55H	-	-	-	-	-
MOV	#068H, 000H	55H	68H	-	-	-	-
MOV	#010H, @R0	55H	68H	10H	-	-	-
ADD	#015H	6AH	68H	10H	0	0	0
ADDC	@R0	7AH	68H	10H	0	0	0
SET1	PSW, 7	7AH	68H	10H	1	0	0
ADDC	@R0	8BH	68H	10H	0	0	1

## 例 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH, ACC	AAH	-	-	-	-	-
MOV	#004H, 002H	AAH	04H	-	-	-	-
MOV	#055H, @R2	AAH	04H	55H	-	-	-
ADD	#001H	ABH	04H	55H	0	0	0
ADDC	@R2	00H	04H	55H	1	1	0
ADDC	@R2	56H	04H	55H	0	0	0

# SUB #i8

Subtract immediate data from accumulator

命令コード	1 0 1 0 0 0 0 1 i7i6i5i4i3i2i1i0 (A1H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) - #i8
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容からイミディエイトデータ (i7~i0) を減算し、その結果を ACC に転送します。

## 例

		ACC	CY	AC	OV
MOV	#055H, ACC	55H	-	-	-
SUB	#013H	42H	0	0	0
SUB	#003H	3FH	0	1	0
SUB	#03FH	00H	0	0	0
SUB	#002H	FEH	1	1	0

# SUB d9

Subtract direct byte from accumulator

命令コード	1 0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (A2H~A3H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) - (d9)
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容から d8~d0 で指定される RAM、または特殊機能レジスタ (SFR) の内容を減算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H, ACC	55H	-	-	-	-
MOV	#068H, 023H	55H	68H	-	-	-
SUB	#00CH	49H	68H	0	1	0
SUB	023H	E1H	68H	1	0	0

## 例 2

		ACC	RAM	CY	AC	OV
MOV	#080H, ACC	80H	-	-	-	-
MOV	#095H, B	80H	95H	-	-	-
SUB	#002H	7EH	95H	0	1	1
SUB	B	E9H	95H	1	0	1

# SUB @Rj

Subtract indirect byte from accumulator

命令コード	1 0 1 0 0 1j1j0 (A4H~A7H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) - ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容から j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容を減算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H, ACC	55H	-	-	-	-	-
MOV	#068H, 00H	55H	68H	-	-	-	-
MOV	#010H, @R0	55H	68H	10H	-	-	-
SUB	#016H	3FH	68H	10H	0	1	0
SUB	@R0	2FH	68H	10H	0	0	0

## 例 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH, ACC	AAH	-	-	-	-	-
MOV	#004H, 002H	AAH	04H	-	-	-	-
MOV	#0AAH, @R2	AAH	04H	AAH	-	-	-
SUB	#001H	A9H	04H	AAH	0	0	0
SUB	@R2	FFH	04H	AAH	1	1	0

# SUBC # i8

Subtract immediate data and carry flag from accumulator

命令コード	1 0 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 (B1H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) - (CY) - # i8
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容からキャリーフラグ (CY) 、イミディエイトデータ (i7~i0) を減算し、その結果を ACC に転送します。

## 例

		ACC	CY	AC	OV
MOV	#055H, ACC	55H	-	-	-
SUB	#013H	42H	0	0	0
SUBC	#003H	3FH	0	1	0
SUBC	#03FH	00H	0	0	0
SUBC	#002H	FEH	1	1	0
SUBC	#03EH	BFH	0	1	0

# SUBC d9

Subtract direct byte and carry flag from accumulator

命令コード	1 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (B2H~B3H)
バイト数	2
サイクル数	1
機能	(ACC) (ACC) - (CY) - (d9)
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容からキャリーフラグ (CY) と d8~d0 で指定される RAM、または特殊機能レジスタ (SFR) の内容を減算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	CY	AC	OV
			23H			
MOV	#055H, ACC	55H	-	-	-	-
MOV	#068H, 023H	55H	68H	-	-	-
SUB	#00CH	49H	68H	0	1	0
SUBC	023H	E1H	68H	1	0	0
SUBC	023H	78H	68H	0	1	1

## 例 2

		ACC	B	CY	AC	OV
MOV	#080H, ACC	80H	-	-	-	-
MOV	#095H, B	80H	95H	-	-	-
SUB	#002H	7EH	95H	0	1	1
SUBC	B	E9H	95H	1	0	1
SUBC	B	53H	95H	0	0	1

# SUBC @ Rj

Subtract indirect byte and carry flag from accumulator

命令コード	1 0 1 1 0 1j1j0 (B4H~B7H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) - (CY) - ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	CY, AC, OV
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容からキャリーフラグ (CY) と j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容を減算し、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM	CY	AC	OV
			00H	68H			
MOV	#055H, ACC	55H	-	-	-	-	-
MOV	#068H, 00H	55H	68H	-	-	-	-
MOV	#040H, @R0	55H	68H	40H	-	-	-
SUB	#016H	3FH	68H	40H	0	1	0
SUBC	@R0	FFH	68H	40H	1	0	0
SUBC	@R0	BEH	68H	40H	0	0	0

## 例 2

		ACC	RAM	TRL	CY	AC	OV
			02H				
MOV	#0AAH, ACC	AAH	-	-	-	-	-
MOV	#004H, 002H	AAH	04H	-	-	-	-
MOV	#0AAH, @R2	AAH	04H	AAH	-	-	-
SUB	#001H	A9H	04H	AAH	0	0	0
SUBC	@R2	FFH	04H	AAH	1	1	0
SUBC	@R2	54H	04H	AAH	0	0	0

# INC d9

Increment direct byte

命令コード	0 1 1 0 0 0 1d8	d7d6d5d4d3d2d1d0	( 62H ~ 63H )
バイト数	2		
サイクル数	1		
機能	( d9 )	( d9 ) + 1	
影響を受けるフラグ			
割り込みの受付	可		

## 説明

d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容をインクリメントします。

## 例 1

		ACC
MOV	#0FDH, ACC	FDH
INC	ACC	FEH
INC	ACC	FFH
INC	ACC	00H
INC	ACC	01H

## 例 2

		RAM
		7FH
MOV	#0FDH, 07FH	FDH
INC	07FH	FEH
INC	07FH	FFH
INC	07FH	00H
INC	07FH	01H

### 注意

- ・ CY , AC , OV は変化しません。
- ・ この命令をポート P0 ~ P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# INC @ Rj

Increment indirect byte

命令コード	0 1 1 0 0 1j1j0 ( 64H~67H )
バイト数	1
サイクル数	1
機能	(( Rj )) (( Rj )) + 1 j = 0 , 1 , 2 , 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ ( SFR ) の内容をインクリメントします。

## 例 1

		ACC	RAM
03H			
MOV	#000H,003H	-	00H
MOV	#0FDH,@R3	FDH	00H
INC	@R3	FEH	00H
INC	@R3	FFH	00H
INC	@R3	00H	00H

## 例 2

		RAM	RAM
		7FH	01H
MOV	#07FH,001H	-	7FH
MOV	#0FDH,@R1	FDH	7FH
INC	@R1	FEH	7FH
INC	@R1	FFH	7FH
INC	@R1	00H	7FH

### 注意

- ・ CY , AC , OV は変化しません。
- ・ この命令をポート P0~P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# DEC d9

Decrement direct byte

命令コード	0 1 1 1 0 0 1d8	d7d6d5d4d3d2d1d0	( 72H ~ 73H )
バイト数	2		
サイクル数	1		
機能	( d9 )	( d9 ) - 1	
影響を受けるフラグ			
割り込みの受付	可		

## 説明

d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容をデクリメントします。

## 例 1

		ACC
MOV	#002H, ACC	02H
DEC	ACC	01H
DEC	ACC	00H
DEC	ACC	FFH
DEC	ACC	FEH

## 例 2

		RAM
		7FH
MOV	#002H, 07FH	02H
DEC	07FH	01H
DEC	07FH	00H
DEC	07FH	FFH
DEC	07FH	FEH

### 注意

- ・ CY , AC , OV は変化しません。
- ・ この命令をポート P0 ~ P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# DEC @Rj

Decrement indirect byte

命令コード	0 1 1 1 0 1j1j0 (74H~77H)
バイト数	1
サイクル数	1
機能	((Rj)) ((Rj)) - 1 j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容をデクリメントします。

## 例 1

		ACC	RAM
			02H
MOV	#000H,002H	-	00H
MOV	#002H,@R2	02H	00H
DEC	@R2	01H	00H
DEC	@R2	00H	00H
DEC	@R2	FFH	00H

## 例 2

		RAM	RAM
			7FH 00H
MOV	#07FH,000H	-	7FH
MOV	#002H,@R0	02H	7FH
DEC	@R0	01H	7FH
DEC	@R0	00H	7FH
DEC	@R0	FFH	7FH

### 注意

- CY, AC, OV は変化しません。
- この命令をポート P0~P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# MUL

Multiply accumulator and c register times b register

命令コード	0 0 1 1 0 0 0 0 (30H)
バイト数	1
サイクル数	7
機能	(B)(ACC)(C) (ACC)(C) × (B)
影響を受けるフラグ	CY, OV
割り込みの受付	7 サイクル目で可

## 説明

アキュムレータ (ACC) と C レジスタ (C) で表される符号なし 16 ビットデータと B レジスタ (B) の符号なし 8 ビットデータを乗算します。24 ビットの演算結果のうち、下位 8 ビットを C に、中位 8 ビットを ACC に、上位 8 ビットを B に転送します。

演算の結果、B の内容が 0 ならばオーバーフローフラグ (OV) がリセットされ、B の内容が 0 でなければ OV がセットされ、キャリーフラグ (CY) は、必ずリセットされます。

## 例 1

		ACC	C	B	CY	AC	OV
MOV	#0C4H, PSW	-	-	-	1	1	1
MOV	#011H, ACC	11H	-	-	1	1	1
MOV	#023H, C	11H	23H	-	1	1	1
MOV	#052H, B	11H	23H	52H	1	1	1
MUL		7DH	36H	05H	0	1	1

## 例 2

		ACC	C	B	CY	AC	OV
MOV	#0C4H, PSW	-	-	-	1	1	1
MOV	#007H, ACC	07H	-	-	1	1	1
MOV	#005H, C	07H	05H	-	1	1	1
MOV	#010H, B	07H	05H	10H	1	1	1
MUL		70H	50H	00H	0	1	0

# DIV

Divide accumulator and c register by b register

命令コード	0 1 0 0 0 0 0 0 (40H)
バイト数	1
サイクル数	7
機能	(ACC)(C), mod(B) (ACC)(C) ÷ (B)
影響を受けるフラグ	CY, OV
割り込みの受付	7 サイクル目で可

## 説明

アキュムレータ (ACC) の内容 (上位バイト) と C レジスタ (C) の内容 (下位バイト) で表される 16 ビットのデータを B レジスタ (B) の内容 (符号なしの 8 ビットデータ) で除算します。演算結果の商を ACC (上位バイト) と C (下位バイト) に転送し、剰余を B に転送します。

### 注意

この命令で B レジスタの内容が 0 の場合、ACC には FFH のデータが設定され、オーバーフローフラグ (OV) がセットされます。一方、B レジスタの内容が 0 でない場合、OV はリセットされ、キャリーフラグ (CY) も必ずリセットされます。

## 例 1

		ACC	C	B	CY	AC	OV
MOV	#0C4H, PSW	-	-	-	1	1	1
MOV	#078H, ACC	79H	-	-	1	1	1
MOV	#005H, C	79H	05H	-	1	1	1
MOV	#007H, B	79H	05H	07H	1	1	1
DIV		11H	49H	06H	0	1	0

## 例 2

		ACC	C	B	CY	AC	OV	
MOV	#0C0H, PSW	-	-	-	1	1	0	
MOV	#007H, ACC	07H	-	-	1	1	0	
MOV	#010H, C	07H	10H	-	1	1	0	
MOV	#000H, B	07H	10H	00H	1	1	0	
DIV		FFH	10H	00H	0	1	1	エラー

# 論理演算命令

## AND # i8

AND immediate data to accumulator

命令コード	1 1 1 0 0 0 0 1	i7i6i5i4i3i2i1i0	(E1H)
バイト数	2		
サイクル数	1		
機能	(ACC)	(ACC)	# i8
影響を受けるフラグ			
割り込みの受付	可		

### 説明

アキュムレータ (ACC) の内容とイミディエイトデータ (i7~i0) の論理積をとり、その結果を ACC に転送します。

### 例 1

		ACC
MOV	#OFFH, ACC	FFH
AND	#0FAH	FAH
AND	#0AFH	AAH
AND	#00FH	0AH
AND	#0F0H	00H

### 例 2

		ACC
MOV	#OFFH, ACC	FFH
AND	#0FEH	FEH
AND	#0FDH	FCH
AND	#0FBH	F8H
AND	#0F7H	FOH
AND	#0EFH	EOH
AND	#0DFH	C9H
AND	#0BFH	80H
AND	#07FH	00H

# AND d9

AND direct byte to accumulator

命令コード	1 1 1 0 0 0 1d8	d7d6d5d4d3d2d1d0	( E2H ~ E3H )
バイト数	2		
サイクル数	1		
機能	( ACC )	( ACC )	( d9 )
影響を受けるフラグ			
割り込みの受付	可		

## 説明

アキュムレータ ( ACC ) の内容と d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容と論理積をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM
			23H
MOV	#0FFH, ACC	FFH	-
MOV	#055H, 023H	FFH	55H
AND	023H	55H	55H
MOV	#0AAH, 023H	55H	AAH
AND	023H	00H	AAH

## 例 2

		ACC	B
MOV	#0FFH, ACC	FFH	-
MOV	#0FEH, B	FFH	FEH
AND	B	FEH	FEH
MOV	#0FDH, B	FEH	FDH
AND	B	FCH	FDH
MOV	#0FBH, B	FCH	FBH
AND	B	F8H	FBH
MOV	#0F7H, B	F8H	F7H
AND	B	FOH	F7H

# AND @ Rj

AND indirect byte to accumulator

命令コード	1 1 1 0 0 1j1j0 (E4H~E7H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容と j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容と論理積をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM
			00H	68H
MOV	#0FFH, ACC	FFH	-	-
MOV	#068H, 000H	FFH	68H	-
MOV	#0F0H, @R0	FFH	68H	F0H
AND	@R0	F0H	68H	F0H
MOV	#00FH, @R0	F0H	68H	0FH
AND	@R0	00H	68H	0FH

## 例 2

		ACC	RAM	
			02H	TRL
MOV	#0FFH, ACC	FFH	-	-
MOV	#004H, 002H	FFH	04H	-
MOV	#0EFH, @R2	FFH	04H	EFH
AND	@R2	EFH	04H	EFH
MOV	#0DFH, @R2	EFH	04H	DFH
AND	@R2	CFH	04H	DFH

# OR # i8

OR immediate data to accumulator

命令コード            1 1 0 1 0 0 0 1    i7i6i5i4i3i2i1i0    ( D1H )  
 バイト数            2  
 サイクル数           1  
 機能                ( ACC )   ( ACC )   # i8  
 影響を受けるフラグ  
 割り込みの受付      可

## 説明

アキュムレータ ( ACC ) の内容とイミディエイトデータ ( i7 ~ i0 ) の論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC
MOV	#000H, ACC	00H
OR	#003H	03H
OR	#00CH	0FH
OR	#030H	3FH
OR	#0C0H	FFH

## 例 2

		ACC
MOV	#000H, ACC	00H
OR	#001H	01H
OR	#002H	03H
OR	#004H	07H
OR	#008H	0FH
OR	#010H	1FH
OR	#020H	3FH
OR	#040H	7FH
OR	#080H	FFH

# OR d9

OR direct byte to accumulator

命令コード	1 1 0 1 0 0 1d8	d7d6d5d4d3d2d1d0	( D2H ~ D3H )
バイト数	2		
サイクル数	1		
機能	( ACC )	( ACC )	( d9 )
影響を受けるフラグ			
割り込みの受付	可		

## 説明

アキュムレータ ( ACC ) の内容と d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容と論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM
			23H
MOV	#000H, ACC	00H	-
MOV	#055H, 023H	00H	55H
OR	023H	55H	55H
MOV	#0AAH, 023H	55H	AAH
OR	023H	FFH	AAH

## 例 2

		ACC	B
MOV	#000H, ACC	00H	-
MOV	#001H, B	00H	01H
OR	B	01H	01H
MOV	#002H, B	01H	02H
OR	B	03H	02H
MOV	#004H, B	03H	04H
OR	B	07H	04H
MOV	#008H, B	07H	08H
OR	B	0FH	08H

# OR @ Rj

OR indirect byte to accumulator

命令コード	1 1 0 1 0 1j1j0 (D4H~D7H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容と j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容と論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM
			00H	68H
MOV	#000H, ACC	00H	-	-
MOV	#068H, 000H	00H	68H	-
MOV	#0F0H, @R0	00H	68H	F0H
OR	@R0	F0H	68H	F0H
MOV	#000FH, @R0	F0H	68H	0FH
OR	@R0	FFH	68H	0FH

## 例 2

		ACC	RAM	TRL
		02H		
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 002H	AAH	04H	-
MOV	#005H, @R2	AAH	04H	05H
OR	@R2	AFH	04H	05H
MOV	#050H, @R2	AFH	04H	50H
OR	@R2	FFH	04H	50H

# XOR # i8

XOR immediate data to accumulator

命令コード	1 1 0 1 0 1j1j0 (D4H~D7H)
バイト数	1
サイクル数	1
機能	(ACC) (ACC) ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容とイミディエイトデータ (i7~i0) の排他的論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC
MOV	#000H, ACC	00H
XOR	#00FH	0FH
XOR	#0F0H	FFH
XOR	#00FH	F0H
XOR	#0F0H	00H

## 例 2

		ACC
MOV	#000H, ACC	00H
XOR	#001H	01H
XOR	#002H	03H
XOR	#004H	07H
XOR	#008H	0FH
XOR	#008H	07H
XOR	#004H	03H
XOR	#002H	01H
XOR	#001H	00H

# XOR d9

XOR direct byte to accumulator

命令コード	1 1 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 ( F2H ~ F3H )
バイト数	2
サイクル数	1
機能	( ACC ) ( ACC ) ( d9 )
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ ( ACC ) の内容と d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容と排他的論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM
			23H
MOV	#000H, ACC	00H	-
MOV	#055H, 023H	00H	55H
XOR	023H	55H	55H
MOV	#0FFH, 023H	55H	FFH
XOR	023H	AAH	FFH

## 例 2

		ACC	B
MOV	#0FFH, ACC	FFH	-
MOV	#010H, B	FFH	10H
XOR	B	EFH	10H
MOV	#020H, B	EFH	20H
XOR	B	CFH	20H
MOV	#040H, B	CFH	40H
XOR	B	8FH	40H
MOV	#080H, B	8FH	80H
XOR	B	0FH	80H

# XOR @ Rj

XOR indirect byte to accumulator

命令コード            1 1 1 1 0 1j1j0    ( F4H~F7H )  
バイト数                1  
サイクル数             1  
機能                    ( ACC )   ( ACC )   (( Rj ))   j = 0 , 1 , 2 , 3  
影響を受けるフラグ  
割り込みの受付        可

## 説明

アキュムレータ ( ACC ) の内容と j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ ( SFR ) の内容と排他的論理和をとり、その結果を ACC に転送します。

## 例 1

		ACC	RAM	RAM
			01H	68H
MOV	#000H, ACC	00H	-	-
MOV	#068H, 001H	00H	68H	-
MOV	#0F0H, @R1	00H	68H	F0H
XOR	@R1	F0H	68H	F0H
MOV	#0FFH, @R1	F0H	68H	FFH
XOR	@R1	0FH	68H	FFH

## 例 2

		ACC	RAM	TRL
			03H	
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 003H	AAH	04H	-
MOV	#0FFH, @R3	AAH	04H	FFH
XOR	@R3	55H	04H	FFH
XOR	@R3	AAH	04H	FFH
XOR	@R3	55H	04H	FFH
XOR	@R3	AAH	04H	FFH

# ROL

Rotate accumulator left

命令コード	1 1 1 0 0 0 0 0 (E0H)
バイト数	1
サイクル数	1
機能	A7 A6 A5 A4 A3 A2 A1 A0 (A7)
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) に格納されている 8 ビットデータを 1 ビットずつ左にローテートします。したがって、ACC のビット 7 のデータはビット 0 に移動します。

## 例 1

		ACC
MOV	#01H, ACC	01H 0000 0001B
ROL		02H 0000 0010B
ROL		04H 0000 0100B
ROL		08H 0000 1000B
ROL		10H 0001 0000B
ROL		20H 0010 0000B
ROL		40H 0100 0000B
ROL		80H 1000 0000B
ROL		01H 0000 0001B
MOV	#55H, ACC	55H 0101 0101B
ROL		AAH 1010 1010B
ROL		55H 0101 0101B
ROL		AAH 1010 1010B
ROL		55H 0101 0101B

# ROLC

Rotate accumulator left through the carry flag

命令コード	1 1 1 1 0 0 0 0 (F0H)
バイト数	1
サイクル数	1
機能	A7 A6 A5 A4 A3 A2 A1 A0 CY (A7)
影響を受けるフラグ	CY
割り込みの受付	可

## 説明

アキュムレータ (ACC) に格納されている 8 ビットデータをキャリーフラグ (CY) を含めて 1 ビットずつ左にローテートします。したがって、ACC のビット 7 のデータは CY に移動し、CY の内容はビット 0 に移動します。

## 例 1

		ACC		CY
MOV	#01H, ACC	01H	0000 0001B	-
SET1	PSW, 7	01H	0000 0001B	1
ROLC		03H	0000 0011B	0
ROLC		06H	0000 0110B	0
ROLC		0CH	0000 1100B	0
ROLC		11H	0001 1000B	0
ROLC		30H	0011 0000B	0
ROLC		60H	0110 0000B	0
ROLC		COH	1100 0000B	0
ROLC		80H	1000 0000B	1
ROLC		01H	0000 0001B	1
MOV	#55H, ACC	55H	0101 0101B	1
ROLC		ABH	1010 1011B	0
ROLC		56H	0101 0110B	1
ROLC		ADH	1010 1101B	0

# ROR

Rotate accumulator right

命令コード	1 1 0 0 0 0 0 0 (C0H)
バイト数	1
サイクル数	1
機能	(A0) A7 A6 A5 A4 A3 A2 A1 A0
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) に格納されている 8 ビットデータを 1 ビットずつ右にローテートします。したがって、ACC のビット 0 のデータはビット 7 に移動します。

## 例 1

		ACC		
MOV	#01H, ACC	01H	0000	0001B
ROR		80H	1000	0000B
ROR		40H	0100	0000B
ROR		20H	0010	0000B
ROR		10H	0001	0000B
ROR		08H	0000	1000B
ROR		04H	0000	0100B
ROR		02H	0000	0010B
ROR		01H	0000	0001B
MOV	#51H, ACC	51H	0101	0001B
ROR		A8H	1010	1000B
ROR		54H	0101	0100B
ROR		2AH	0010	1010B
ROR		15H	0001	0101B

# RORC

Rotate accumulator right through the carry flag

命令コード	1 1 0 1 0 0 0 0 (D0H)
バイト数	1
サイクル数	1
機能	(A0) CY A7 A6 A5 A4 A3 A2 A1 A0
影響を受けるフラグ	CY
割り込みの受付	可

## 説明

アキュムレータ (ACC) に格納されている 8 ビットデータをキャリーフラグ (CY) を含めて 1 ビットずつ右にローテートします。したがって、ACC のビット 0 のデータは CY に移動し、CY の内容はビット 7 に移動します。

## 例 1

		ACC		CY
MOV	#01H, ACC	01H	0000 0001B	-
SET1	PSW, 7	01H	0000 0001B	1
RORC		80H	1000 0000B	1
RORC		C0H	1100 0000B	0
RORC		60H	0110 0000B	0
RORC		30H	0011 0000B	0
RORC		18H	0001 1000B	0
RORC		0CH	0000 1100B	0
RORC		06H	0000 0110B	0
RORC		03H	0000 0011B	0
RORC		01H	0000 0001B	1
MOV	#55H, ACC	55H	0101 0101B	1
RORC		AAH	1010 1010B	1
RORC		D5H	1101 0101B	0
RORC		6AH	0110 1010B	1

# データ転送命令

## LD d9 Load direct byte to accumulator

命令コード	0 0 0 0 0 0 1d8 d8d7d6d5d4d3d2d1d0 ( 02H ~ 03H )
バイト数	2
サイクル数	1
機能	( ACC ) ( d9 )
影響を受けるフラグ	
割り込みの受付	可

### 説明

d8 ~ d0 で指定された RAM、または特殊機能レジスタ ( SFR ) の内容をアキュムレータ ( ACC ) に転送します。

### 例 1

		ACC	RAM	RAM
			70H	71H
MOV	#0FF, ACC	FFH	-	-
MOV	#055H, 070H	FFH	55H	-
MOV	#0AAH, 071H	FFH	55H	AAH
LD	070H	55H	55H	AAH
LD	071H	AAH	55H	AAH

### 例 2

		ACC	B	SP
MOV	#0FF, ACC	FFH	-	-
MOV	#0F0H, B	FFH	F0H	-
MOV	#00FH, SP	FFH	F0H	0FH
LD	B	F0H	F0H	0FH
LD	SP	0FH	F0H	0FH
LD	B	F0H	F0H	0FH

# LD @ Rj

Load indirect byte to accumulator

命令コード            0 0 0 0 0 1j1j0    ( 04H~07H )  
バイト数                1  
サイクル数             1  
機能                    ( ACC )    (( Rj ))    j = 0, 1, 2, 3  
影響を受けるフラグ  
割り込みの受付        可

## 説明

j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ ( SFR ) の内容をアキュムレータ ( ACC ) に転送します。

## 例 1

		ACC	RAM	RAM	RAM	RAM
			00H	01H	70H	7FH
MOV	#0FFH, ACC	FFH	-	-	-	-
MOV	#070H, 000H	FFH	70H	-	-	-
MOV	#07FH, 001H	FFH	70H	7FH	-	-
MOV	#0F0H, @R0	FFH	70H	7FH	F0H	-
MOV	#00FH, @R1	FFH	70H	7FH	F0H	0FH
LD	@R0	F0H	70H	7FH	F0H	0FH
LD	@R1	0FH	70H	7FH	F0H	0FH

## 例 2

		ACC	RAM	RAM	B	C
			02H	03H	102H	103H
MOV	#0FF, ACC	FFH	-	-	-	-
MOV	#004H, 002H	FFH	04H	-	-	-
MOV	#005H, 003H	FFH	04H	05H	-	-
MOV	#0AAH, @R2	FFH	04H	05H	AAH	-
MOV	#055H, @R3	FFH	04H	05H	AAH	55H
LD	@R2	AAH	04H	05H	AAH	55H
LD	@R3	55H	04H	05H	AAH	55H

# ST d9

Store direct byte to accumulator

命令コード	0 0 0 1 0 0 1d8	d7d6d5d4d3d2d1d0	( 12H ~ 13H )
バイト数	2		
サイクル数	1		
機能	( d9 )	( ACC )	
影響を受けるフラグ			
割り込みの受付	可		

## 説明

アキュムレータ ( ACC ) の内容を d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) に転送します。

## 例 1

		ACC	RAM	RAM
			70H	71H
MOV	#0FFH, ACC	FFH	-	-
MOV	#055H, 070H	FFH	55H	-
MOV	#0AAH, 071H	FFH	55H	AAH
ST	070H	FFH	FFH	AAH
MOV	#000H, ACC	00H	FFH	AAH
ST	071H	00H	FFH	00H

## 例 2

		ACC	B	SP
MOV	#012H, ACC	12H	-	-
MOV	#0F0H, B	12H	FOH	-
MOV	#00FH, SP	12H	FOH	0FH
ST	B	12H	12H	0FH
MOV	#034H, ACC	34H	12H	0FH
ST	SP	34H	12H	34H
ST	B	34H	34H	34H

# ST @ Rj

Store indirect byte to accumulator

命令コード            0 0 0 1 0 1j1j0    ( 14H~17H )  
バイト数                1  
サイクル数             1  
機能                    (( Rj ))    ( ACC )    j = 0 , 1 , 2 , 3  
影響を受けるフラグ  
割り込みの受付        可

## 説明

アキュムレータ ( ACC ) の内容を j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ ( SFR ) に転送します。

## 例 1

		ACC	RAM	RAM	RAM	RAM
			00H	01H	70H	7FH
MOV	#0FFH, ACC	FFH	-	-	-	-
MOV	#070H, 000H	FFH	70H	-	-	-
MOV	#07FH, 001H	FFH	70H	7FH	-	-
MOV	#0F0H, @R0	FFH	70H	7FH	F0H	-
MOV	#00FH, @R1	FFH	70H	7FH	F0H	0FH
ST	@R0	FFH	70H	7FH	FFH	0FH
ST	@R1	FFH	70H	7FH	FFH	FFH

## 例 2

		ACC	RAM	RAM	TRL	TRH
			02H	03H	104H	105H
MOV	#000H, ACC	00H	-	-	-	-
MOV	#004H, 002H	00H	04H	-	-	-
MOV	#005H, 003H	00H	04H	05H	-	-
MOV	#0AAH, @R2	00H	04H	05H	AAH	-
MOV	#055H, @R3	00H	04H	05H	AAH	55H
ST	@R2	00H	04H	05H	00H	55H
ST	@R3	00H	04H	05H	00H	00H

# MOV #i8,d9

Move immediate data to direct byte

命令コード	0 0 1 0 0 0 1d8	d7d6d5d4d3d2d1d0	i7i6i5i4i3i2i1i0	( 22H ~ 23H )
バイト数	3			
サイクル数	2			
機能	( d9 )	# i8		
影響を受けるフラグ				
割り込みの受付	2 サイクル目で可			

## 説明

イミディエイトデータ ( i7 ~ i0 ) を d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) に転送します。

## 例 1

		RAM	RAM	RAM	RAM
		00H	01H	02H	03H
MOV	#0FFH,000H	FFH	-	-	-
MOV	#0FEH,001H	FFH	FEH	-	-
MOV	#0FDH,002H	FFH	FEH	FDH	-
MOV	#0FCH,003H	FFH	FEH	FDH	FCH
MOV	#0FBH,003H	FFH	FEH	FDH	FBH
MOV	#0FAH,002H	FFH	FEH	FAH	FBH
MOV	#0F9H,001H	FFH	F9H	FAH	FBH
MOV	#0F8H,000H	F8H	F9H	FAH	FBH

## 例 2

		ACC	B	TRL
MOV	#0FFH,100H	FFH	-	-
MOV	#0FEH,102H	FFH	FEH	-
MOV	#0FDH,104H	FFH	FEH	FDH
MOV	#0FAH,104H	FFH	FEH	FAH
MOV	#0F9H,102H	FFH	F9H	FAH
MOV	#0F8H,100H	F8H	F9H	FAH

# MOV #i8 @Rj

Move immediate data to indirect byte

命令コード	0 0 1 0 0 1j1j0 i7i6i5i4i3i2i1i0 (24H~27H)
バイト数	2
サイクル数	1
機能	((Rj)) #i8 j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

イミディエイトデータ (i7~i0) を j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) に転送します。

## 例 1

		RAM	RAM	RAM	RAM
		00H	01H	7EH	7FH
MOV	#07FH,000H	7FH	-	-	-
MOV	#07EH,001H	7FH	7EH	-	-
MOV	#0FDH,@R0	7FH	7EH	-	FDH
MOV	#0FCH,@R1	7FH	7EH	FCH	FDH
MOV	#0FBH,@R0	7FH	7EH	FCH	FBH
MOV	#0FAH,@R1	7FH	7EH	FAH	FBH
MOV	#0F9H,@R0	7FH	7EH	FAH	F9H
MOV	#0F8H,@R1	7FH	7EH	F8H	F9H

## 例 2

		RAM	RAM	ACC	B
		02H	03H	100H	102H
MOV	#000H,002H	00H	-	-	-
MOV	#002H,003H	00H	02H	-	-
MOV	#0FDH,@R2	00H	02H	FDH	-
MOV	#0FCH,@R3	00H	02H	FDH	FCH
MOV	#0FBH,@R2	00H	02H	FBH	FCH
MOV	#0FAH,@R3	00H	02H	FBH	FAH

# LDC

Load code byte relative to TRR to accumulator

命令コード	1 1 0 0 0 0 0 1 (C1H)
バイト数	1
サイクル数	2
機能	(ACC) (BNK)((TRR) + (ACC)) [ROM]
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

テーブルリファレンスレジスタ (TRR) の内容とアキュムレータ (ACC) の内容を加算した値によって指定されるプログラムメモリ (ROM) の内容を ACC に転送します。ROM 内プログラム動作時とフラッシュメモリ内プログラム動作時で参照される ROM データは異なります。ROM 内プログラム動作時は ROM が、フラッシュメモリ内プログラム動作時は、フラッシュメモリのバンク 0 が参照されます。

LDC 命令では、フラッシュメモリのバンク 1 は参照できません。フラッシュメモリのバンク 1 を参照する場合は、ビジュアルメモリに用意されているシステム BIOS の OS プログラムを使用してください。



システム BIOS や OS プログラムについては『ビジュアルメモリ ハードウェアマニュアル』の「システム BIOS 編」を参照してください。

## 例

		ACC	TRR	TRR	TRR
			TRH	TRL	+ACC
MOV	#001H, TRH	-	01H	-	-
MOV	#023H, TRL	-	01H	23H	-
MOV	#000H, ACC	00H	01H	23H	0123H
LDC		30H	01H	23H	0153H
MOV	#001H, ACC	01H	01H	23H	0124H
LDC		FFH	01H	23H	0222H
MOV	#002H, ACC	02H	01H	23H	0125H
LDC		57H	01H	23H	017AH
MOV	#003H, ACC	03H	01H	23H	0126H
LDC		EAH	01H	23H	020DH

PC	ROM
0123H	30H
0124H	FFH
0125H	57H
0126H	EAH

# PUSH d9

Push direct byte to stack

命令コード	0 1 1 0 0 0 0d8 d7d6d5d4d3d2d1d0 ( 60H ~ 61H )
バイト数	2
サイクル数	2
機能	( SP ) ( SP ) + 1 [ ( SP ) ] ( d9 )
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

スタックポインタ ( SP ) がインクリメントされます。後に、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容を SP で指定される RAM に転送します。

## 例

		ACC	B	RAM	SP	RAM	RAM	RAM
				00H		20H	21H	22H
MOV	#0AAH, ACC	AAH	-	-	-	-	-	-
MOV	#055H, B	AAH	55H	-	-	-	-	-
MOV	#012H, 000H	AAH	55H	12H	-	-	-	-
MOV	#01FH, SP	AAH	55H	12H	1FH	-	-	-
PUSH	ACC	AAH	55H	12H	20H	AAH	-	-
PUSH	B	AAH	55H	12H	21H	AAH	55H	-
PUSH	000H	AAH	55H	12H	22H	AAH	55H	12H
POP	B	AAH	12H	12H	21H	AAH	55H	12H
POP	ACC	55H	12H	12H	20H	AAH	55H	12H
POP	000H	55H	12H	AAH	1FH	AAH	55H	12H

# POP d9

Pop direct byte from stack

命令コード	0 1 1 1 0 0 0d8 d7d6d5d4d3d2d1d0 ( 70H ~ 71H )
バイト数	2
サイクル数	2
機能	( d9 ) (( SP )) ( SP ) ( SP ) - 1
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

スタックポインタ ( SP ) で指定される RAM の内容が、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) に転送されます。後に、SP がデクリメントされます。

## 例

		ACC	B	TRL	SP	RAM 20H	RAM 21H	RAM 22H
MOV	#0AAH, ACC	AAH	-	-	-	-	-	-
MOV	#055H, B	AAH	55H	-	-	-	-	-
MOV	#012H, TRL	AAH	55H	12H	-	-	-	-
MOV	#01FH, SP	AAH	55H	12H	1FH	-	-	-
PUSH	ACC	AAH	55H	12H	20H	AAH	-	-
PUSH	B	AAH	55H	12H	21H	AAH	55H	-
PUSH	TRL	AAH	55H	12H	22H	AAH	55H	12H
POP	B	AAH	12H	12H	21H	AAH	55H	12H
POP	ACC	55H	12H	12H	20H	AAH	55H	12H
POP	TRL	55H	12H	AAH	1FH	AAH	55H	12H

# XCH d9

Exchange direct byte with accumulator

命令コード	1 1 0 0 0 0 1d8	d7d6d5d4d3d2d1d0	( C2H ~ C3H )
バイト数	2		
サイクル数	1		
機能	( ACC )	( d9 )	
影響を受けるフラグ			
割り込みの受付	可		

## 説明

アキュムレータ ( ACC ) の内容と d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の内容を交換します。

## 例 1

		ACC	RAM
			23H
MOV	#0FFH, ACC	FFH	-
MOV	#055H, 023H	FFH	55H
XCH	023H	55H	FFH
XCH	023H	FFH	55H
XCH	023H	55H	FFH
XCH	023H	FFH	55H

## 例 2

		ACC	B
MOV	#0FFH, ACC	FFH	-
MOV	#0FEH, B	FFH	FEH
XCH	B	FEH	FFH
XCH	B	FFH	FEH
XCH	B	FEH	FFH
XCH	B	FFH	FEH

# XCH @ Rj

Exchange indirect byte with accumulator

命令コード	1 1 0 0 0 1j1j0 (C4H~C7H)
バイト数	1
サイクル数	1
機能	(ACC) ((Rj)) j = 0, 1, 2, 3
影響を受けるフラグ	
割り込みの受付	可

## 説明

アキュムレータ (ACC) の内容と j1 ~ j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容を交換します。

## 例 1

		ACC	RAM	RAM
			01H	68H
MOV	#0FFH, ACC	FFH	-	-
MOV	#068H, 001H	FFH	68H	-
MOV	#0F0H, @R1	FFH	68H	FOH
XCH	@R1	FOH	68H	FFH
XCH	@R1	FFH	68H	FOH
XCH	@R1	FOH	68H	FFH
XCH	@R1	FFH	68H	FOH

## 例 2

		ACC	RAM	TRL
			03H	
MOV	#0AAH, ACC	AAH	-	-
MOV	#004H, 003H	AAH	04H	-
MOV	#055H, @R3	AAH	04H	55H
XCH	@R3	55H	04H	AAH
XCH	@R3	AAH	04H	55H
XCH	@R3	55H	04H	AAH

# ジャンプ命令

## JMP a12

Jump near absolute address

命令コード	0 0 1a11 1a10a9a8 a7a6a5a4a3a2a1a0 ( 28H ~ 2FH , 38H ~ 3FH )
バイト数	2
サイクル数	2
機能	( PC ) ( PC ) + 2 ( PC11 ~ 00 ) a12
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

### 説明

プログラムカウンタ ( PC ) を 2 回インクリメントします。後に、a11 ~ a0 のデータを PC のビット 11 ~ 00 に転送します。

### 例 1

ラベル LA の値は、0F0EH です。

		PC	命令コード
	NOP	OFFBH	00H
	NOP	OFFCH	00H
	JMP LA	OFFDH	3F0EH
LA:	INC ACC	0F0EH	6300H
	ROR	0F10H	C0H

### 例 2

ラベル LA の値は、1F0EH です。

		PC	命令コード
	NOP	OFFCH	00H
	NOP	OFFDH	00H
	JMP LA	OFFEH	3F0EH
LA:	INC ACC	1F0EH	6300H
	ROR	1F10H	C0H

# JMPF a16

Jump far absolute address

命令コード            0 0 1 0 0 0 0 1    a15a14a13a12a11a10a9a8    a7a6a5a4a3a2a1a0    ( 21H )  
 バイト数            3  
 サイクル数           2  
 機能                ( PC )    a16  
 影響を受けるフラグ  
 割り込みの受付      2 サイクル目で可

## 説明

a15 ~ a0 のデータをプログラムカウンタ ( PC ) に転送します。

## 例 1

ラベル LA の値は、0F0EH です。

		PC	命令コード
	NOP	OFFAH	00H
	NOP	OFFBH	00H
	JMPF      LA	OFFCH	210F0EH
LA:	INC      ACC	OF0EH	6300H
	ROR	OF10H	C0H

## 例 2

ラベル LA の値は、0F0EH です。

		PC	命令コード
	NOP	OFFCH	00H
	NOP	OFFDH	00H
	JMPF      LA	OFFEH	210F0EH
LA:	INC      ACC	OF0EH	6300H
	ROR	OF10H	C0H

# BR r8

Branch near relative address

命令コード	0 0 0 0 0 0 0 1	r7r6r5r4r3r2r1r0	(01H)
バイト数	2		
サイクル数	2		
機能	(PC)	(PC) + 2(PC)	(PC) + r8
影響を受けるフラグ			
割り込みの受付	2 サイクル目で可		

## 説明

プログラムカウンタ(PC)を2回インクリメントします。後に、r7～r0のデータをPCに加算し、その結果をPCに転送します。

## 例 1

ラベル LA の値は、0F5FH です。

		PC	命令コード
	NOP	0F1CH	00H
	NOP	0F1DH	00H
	BR LA	0F1EH	013FH
LA:	INC ACC	0F5FH	6300H
	ROR	0F61H	C0H

## 例 2

ラベル LA の値は、1F0EH です。

		PC	命令コード
	NOP	1F0CH	00H
	NOP	1F0DH	00H
LA:	INC ACC	1F0EH	6300H
	ROR	1F10H	C0H
	NOP	1F11H	00H
	NOP	1F12H	00H
	BR LA	1F13H	01F9H

# BRF r16

Branch far relative address

命令コード	0 0 0 1 0 0 0 1	r7r6r5r4r3r2r1r0	r15r14r13r12r11r10r9r8	(11H)
バイト数	3			
サイクル数	4			
機能	(PC) (PC) + 3 (PC) (PC) - 1 + r16			
影響を受けるフラグ				
割り込みの受付	4 サイクル目で可			

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、PC をデクリメントし、さらに r15 ~ r0 のデータを PC に加算し、その結果を PC に転送します。

## 例 1

ラベル LA の値は、105FH です。

		PC	命令コード
	NOP	0F1CH	00H
	NOP	0F1DH	00H
	BRF LA	0F1EH	113F01H
LA:	INC ACC	105FH	6300H
	ROR	1061H	C0H

## 例 2

ラベル LA の値は、1F0EH です。

		PC	命令コード
	NOP	1FFCH	00H
	NOP	1FFDH	00H
LA:	INC ACC	1F0EH	6300H
	ROR	1F10H	C0H
	NOP	1F11H	00H
	NOP	1F12H	00H
	BRF LA	1F13H	11F8FFH

# 条件分岐命令

## BZ r8

Branch near relative address if accumulator is zero

命令コード	1 0 0 0 0 0 0 0 r7r6r5r4r3r2r1r0 (80H)
バイト数	2
サイクル数	2
機能	(PC) (PC) + 2, if (ACC) = 0 then (PC) (PC) + r8
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

### 説明

プログラムカウンタ (PC) を 2 回インクリメントします。後に、アキュムレータ (ACC) の内容が 0 ならば、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。ACC の内容が 0 でない場合、次の命令を実行します。

### 例 1

BZ 命令実行時に ACC = 0 なので、ラベル LA に分岐します。

			PC	命令コード	ACC
	MOV	#000H, ACC	0F1BH	230000H	00H
	BZ	LA	0F1EH	803FH	00H
LA:	INC	ACC	0F5FH	6300H	01H
	ROR		0F61H	C0H	80H

### 例 2

BZ 命令実行時に ACC = 01H なので、次の命令の実行に移ります。

			PC	命令コード	ACC
	MOV	#001H, ACC	0F1BH	230001H	01H
	BZ	LA	0F1EH	803FH	01H
	DEC	ACC	0F20H	7300H	00H
	ROR		0F22H	C0H	00H
LA:	INC	ACC			

# BNZ r8

Branch near relative address if accumulator is not zero

命令コード	1 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 (90H)
バイト数	2
サイクル数	2
機能	(PC) (PC) + 2, if (ACC) 0 then (PC) (PC) + r8
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ (PC) を 2 回インクリメントします。後に、アキュムレータ (ACC) の内容が 0 でない場合、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。ACC の内容が 0 の場合、次の命令を実行します。

## 例 1

BNZ 命令実行時に ACC = 0 なので、ラベル LA に分岐します。

			PC	命令コード	ACC
	MOV	#001H, ACC	0F1BH	230001H	01H
BNZ	LA	0F1EH	903FH	01H	
LA:	INC	ACC	0F5FH	6300H	02H
	ROR		0F61H	C0H	01H

## 例 2

BNZ 命令実行時に ACC = 0 なので、次の命令の実行に移ります。

			PC	命令コード	ACC
	MOV	#000H, ACC	0F1BH	230000H	00H
	BNZ	LA	0F1EH	903FH	00H
	DEC	ACC	0F20H	7300H	FFH
	ROR		0F22H	C0H	FFH
LA:	INC	ACC			

# BP d9 b3 r8

Branch near relative address if direct ビット is positive

命令コード	0 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 ( 68H ~ 6FH , 78H ~ 7FH )
バイト数	3
サイクル数	2
機能	( PC ) ( PC ) + 3 , if ( d9 , b3 ) = 1 then ( PC ) ( PC ) + r8
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ ( PC ) を 3 回インクリメントします。後に、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットがセットされている場合、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。d8 ~ d0 で指定される RAM、または SFR の b2 ~ b0 で指定されるビットがリセットされている場合、次の命令を実行します。

## 例 1

BP 命令実行時に B のビット 0 が 1 なので、ラベル LA に分岐します。

		PC	命令コード	B
	MOV #001H, B	0F1AH	230201H	01H
	BP B, 0, LA	0F1DH	78023FH	01H
LA:	INC B	0F5FH	6302H	02H
	NOP	0F61H	00H	02H

## 例 2

BP 命令実行時に ACC のビット 0 が 0 なので、次の命令の実行に移ります。

		PC	命令コード	ACC
	MOV #080H, ACC	0F1AH	230080H	80H
	BP ACC, 0, LA	0F1DH	78003FH	80H
	DEC ACC	0F20H	7300H	7FH
	ROR	0F22H	C0H	BFH
LA:	INC ACC			

# BPC d9 b3 r8

Branch near relative address if direct ビット is positive , and clear

命令コード	0 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 ( 48H ~ 4FH , 58H ~ 5FH )
バイト数	3
サイクル数	2
機能	( PC ) ( PC ) + 3 , if ( d9 , b3 ) = 1 then ( PC ) ( PC ) + r8 ( d9 , b3 ) = 0
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ ( PC ) を 3 回インクリメントします。後に、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットがセットされている場合、そのビットをリセットした後、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。d8 ~ d0 で指定される RAM、または SFR の b2 ~ b0 で指定されるビットがリセットされている場合、次の命令を実行します。

## 例 1

BPC 命令実行時に B のビット 0 が 1 なので、リセットシラベル LA に分岐します。

		PC	命令コード	B
	MOV #003H, B	0F1AH	230203H	03H
	BPC B, 0, LA	0F1DH	58023FH	02H
LA:	INC B	0F5FH	6302H	03H
	NOP	0F61H	00H	03H

## 例 2

BPC 命令実行時に ACC のビット 0 が 0 なので、次の命令の実行に移ります。

		PC	命令コード	ACC
	MOV #080H, ACC	0F1AH	230080H	80H
	BPC ACC, 0, LA	0F1DH	58003FH	80H
	DEC ACC	0F20H	7300H	7FH
	ROR	0F22H	C0H	BFH
LA:	INC ACC			

### 注意

この命令をポート P0, P1, P2, P3, P4, P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# BN d9 b3 r8

Branch near relative address if direct ビット is negative

命令コード	1 0 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 ( 88H ~ 8FH , 98H ~ 9FH )
バイト数	3
サイクル数	2
機能	( PC ) ( PC ) + 3 , if ( d9 , b3 ) = 0 then ( PC ) ( PC ) + r8
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ ( PC ) を 3 回インクリメントします。後に、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットがリセットされている場合、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。d8 ~ d0 で指定される RAM、または SFR の b2 ~ b0 で指定されるビットがセットされている場合、次の命令を実行します。

## 例 1

BN 命令実行時に B のビット 0 が '0' なので、ラベル LA に分岐します。

		PC	命令コード	B
	MOV #0FEH, B	0F1AH	2302FEH	FEH
	BN B, 0, LA	0F1DH	98023FH	FEH
LA:	INC B	0F5FH	6302H	FFH
	NOP	0F61H	00H	FFH

## 例 2

BN 命令実行時に ACC のビット 0 が '1' なので、次の命令の実行に移ります。

		PC	命令コード	ACC
	MOV #001H, ACC	0F1AH	230001H	01H
	BN ACC, 0, LA	0F1DH	98003FH	01H
	DEC ACC	0F20H	7300H	00H
	ROR	0F22H	C0H	00H
LA:	INC ACC			

# DBNZ d9 r8

Decrement direct byte and branch near relative address if direct バイト is not zero

命令コード	0 1 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 ( 52H ~ 53H )
バイト数	3
サイクル数	2
機能	( PC ) ( PC ) + 3 ( d9 ) = ( d9 ) - 1 , if ( d9 ) 0 then ( PC ) ( PC ) + r8
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ ( PC ) を 3 回インクリメントし、さらに、d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) をデクリメントします。後に、デクリメントされた RAM、または SFR が 0 でない場合、r7 ~ r0 のデータを PC に加算し、その結果を PC に転送します。デクリメントされた RAM、または SFR が 0 の場合、次の命令を実行します。

## 例 1

DBNZ 命令実行時に B がデクリメントされて、B = 0 となるので、ラベル LA に分岐します。

		PC	命令コード	B
	MOV #002H, B	0F1AH	230202H	02H
	DBNZ B, LA	0F1DH	53023FH	01H
LA:	INC B	0F5FH	6302H	02H
	NOP	0F61H	00H	02H

## 例 2

DBNZ 命令実行時に ACC がデクリメントされて、ACC = 0 となるので、次の命令の実行に移ります。

		PC	命令コード	ACC
	MOV #001H, ACC	0F1AH	230001H	01H
	DBNZ ACC, LA	0F1DH	53003FH	00H
	DEC ACC	0F20H	7300H	FFH
	ROR	0F22H	C0H	FFH
LA:	INC ACC			

### 注意

この命令をポート P0, P1, P2, P3, P4, P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# DBNZ @Rj, r8

Decrement indirect byte and branch near relative address if indirect バイト is not zero

命令コード	0 1 0 1 0 1j1j0 r7r6r5r4r3r2r1r0 (54H~57H)
バイト数	2
サイクル数	2
機能	$(PC) \leftarrow (PC) + 2, ((Rj) \neq 0) \rightarrow ((Rj) \leftarrow (Rj) - 1),$ if $((Rj) = 0)$ then $(PC) \leftarrow (PC) + r8 \quad j = 0, 1, 2, 3$
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ(PC)を2回インクリメントし、さらに、j1~j0で指定される間接アドレスレジスタで指定されるRAM、または特殊機能レジスタ(SFR)をデクリメントします。後に、デクリメントされたRAM、またはSFRが0でない場合に、r7~r0のデータをPCに加算し、その結果をPCに転送します。デクリメントされたRAM、またはSFRが0の場合は、次の命令を実行します。

## 例 1

DBNZ 命令実行時に B がデクリメントされて、B = 0 となるので、ラベル LA に分岐します。

		PC	命令コード	B	RAM
					03H
	MOV	#002H, B	0F18H	230202H	02H
	MOV	#002H, 003H	0F1BH	220302H	02H
	DBNZ	@R3, LA	0F1EH	573FH	01H
LA:	INC	B	0F5FH	6302H	02H

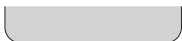
## 例 2

DBNZ 命令実行時に ACC がデクリメントされて、ACC = 0 となるので、次の命令の実行に移ります。

		PC	命令コード	ACC	RAM
					03H
	MOV	#001H, ACC	0F18H	230001H	01H
	MOV	#000H, 003H	0F1BH	220300H	01H
	DBNZ	@R3, LA	0F1EH	573FH	00H
	DEC	ACC	0F20H	7300H	FFH
LA:	INC	ACC			

### 注意

この命令をポート P0, P1, P2, P3, P4, P5 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。



# BE #i8,r8

Compare immediate data to accumulator and branch near relative address if equal

命令コード	0 0 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (31H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if (ACC) = # i8 then (PC) (PC) + r8
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、イミディエイトデータ (i7~i0) とアキュムレータ (ACC) の内容を比較し、比較したデータが同じ場合、r7~r0 のデータを PC に加算し、その結果を PC に転送します。比較したデータが異なる場合、次の命令を実行します。

また、ACC がイミディエイトデータよりも小さい場合、キャリーフラグ (CY) はセットされ、同じか、大きい場合、CY はリセットされます。

ACC < # i8 CY = 1

ACC # i8 CY = 0

## 例 1

BE 命令実行時に ACC = 02H なので、CY をリセットし、ラベル LA に分岐します。

			PC	命令コード	ACC	CY
	MOV	#002H, ACC	0F1AH	230002H	02H	-
	BE	#002H, LA	0F1DH	31023FH	02H	0
LA:	INC	ACC	0F5FH	6300H	03H	0

## 例 2

BE 命令実行時に ACC < 04H なので、CY をセットし、次の命令の実行に移ります。

			PC	命令コード	ACC	CY
	MOV	#003H, ACC	0F1AH	230003H	03H	-
	BE	#004H, LA	0F1DH	31043FH	03H	1
	DEC	ACC	0F20H	7300H	02H	1
LA:	INC	ACC				

# BE d9 r8

Compare direct byte to accumulator and branch near relative address if equal

命令コード	0 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (32H~33H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if (ACC) = (d9) then (PC) (PC) + r8
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、d8~d0 で指定される RAM、または特殊機能レジスタ (SFR) の内容とアキュムレータ (ACC) の内容を比較し、比較したデータが同じ場合、r7~r0 のデータを PC に加算し、その結果を PC に転送します。比較したデータが異なる場合、次の命令を実行します。

また、ACC が d8~d0 で指定された RAM、または特殊機能レジスタ (SFR) の内容よりも小さい場合、キャリーフラグ (CY) はセットされ、同じか、大きい場合、CY はリセットされます。

ACC < d9 (RAM or SFR) CY = 1  
ACC ≥ d9 (RAM or SFR) CY = 0

## 例 1

BE 命令実行時に ACC = B なので、CY をリセットしラベル LA に分岐します。

			PC	命令コード	ACC	B	CY
	MOV	#002H, ACC	0F17H	230002H	02H	-	-
	MOV	#002H, B	0F1AH	230202H	02H	02H	-
	BE	B, LA	0F1DH	33023FH	02H	02H	0
LA:	INC	ACC	0F5FH	6300H	03H	02H	0

## 例 2

BE 命令実行時に ACC < B なので、CY をセットし次の命令の実行に移ります。

			PC	命令コード	ACC	B	CY
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#0F2H, B	0F1AH	2302F2H	03H	F2H	-
	BE	B, LA	0F1DH	33023FH	03H	F2H	1
	DEC	ACC	0F20H	7300H	02H	F2H	1
LA:	INC	ACC					

# BE @Rj, #i8, r8

Compare immediate data to indirect byte and branch near relative address if equal

命令コード	0 0 1 1 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (34H~37H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if ((Rj)) = #i8 then (PC) (PC) + r8 j = 0, 1, 2, 3
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ(PC)を3回インクリメントします。後に、j1~j0で指定される間接アドレスレジスタによって指定されるRAM、または特殊機能レジスタ(SFR)の内容とイミディエイトデータ(i7~i0)の内容を比較し、比較したデータが同じ場合、r7~r0のデータをPCに加算し、その結果をPCに転送します。比較したデータが異なる場合、次の命令を実行します。

また、j1~j0で指定される間接アドレスレジスタによって指定されるRAM、または特殊機能レジスタ(SFR)の内容がイミディエイトデータ(i7~i0)よりも小さい場合、キャリーフラグ(CY)はセットされ、同じか、大きい場合、CYはリセットされます。

@Rj < #i8 CY = 1

@Rj #i8 CY = 0

## 例 1

BE 命令実行時に B = 05H なので、CY をリセットしラベル LA に分岐します。

			PC	命令コード	B	RAM	CY
						03H	
	MOV	#005H, B	0F17H	230205H	05H	-	-
	MOV	#002H, 003H	0F1AH	220302H	05H	02H	-
	BE	@R3, #5H, LA	0F1DH	37053FH	05H	02H	0
LA:	INC	B	0F5FH	6302H	06H	02H	0

## 例 2

BE 命令実行時に ACC < 09H なので、CY をセットし次の命令の実行に移ります。

			PC	命令コード	ACC	RAM	CY
						02H	
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#000H, 002H	0F1AH	220200H	03H	00H	-
	BE	@R2, #9H, LA	0F1DH	36093FH	03H	00H	1
	DEC	ACC	0F20H	7300H	02H	00H	1
LA:	INC	ACC					

# BNE #i8,r8

Compare immediate data to accumulator and branch near relative address if not equal

命令コード	0 1 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (41H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if (ACC) # i8 then (PC) (PC) + r8
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

まず、プログラムカウンタ (PC) を 3 回インクリメントします。後に、イミディエイトデータ (i7~i0) とアキュムレータ (ACC) の内容を比較し、比較したデータが異なる場合、r7~r0 のデータを PC に加算し、その結果を PC に転送します。比較したデータが同じ場合、次の命令を実行します。また、ACC がイミディエイトデータよりも小さい場合、キャリーフラグ (CY) はセットされ、同じか、大きい場合、CY はリセットされます。

ACC < # i8 CY = 1

ACC # i8 CY = 0

## 例 1

BNE 命令実行時に ACC > 00H なので、CY をリセットしラベル LA に分岐します。

		PC	命令コード	ACC	CY
	MOV #002H,ACC	0F1AH	230002H	02H	-
	BNE #000H,LA	0F1DH	41003FH	02H	0
LA:	INC ACC	0F5FH	6300H	03H	0

## 例 2

BNE 命令実行時に ACC = 03H なので、CY をリセットし次の命令の実行に移ります。

		PC	命令コード	ACC	CY
	MOV #003H,ACC	0F1AH	230003H	03H	-
	BNE #003H,LA	0F1DH	41033FH	03H	0
	DEC ACC	0F20H	7300H	02H	0
LA:	INC ACC				

# BNE d9, r8

Compare direct byte to accumulator and branch near relative address if not equal

命令コード	0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (42H~43H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if (ACC) (d9) then (PC) (PC) + r8
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、d8~d0 で指定される RAM、または特殊機能レジスタ (SFR) の内容とアキュムレータ (ACC) の内容を比較し、比較したデータが異なる場合、r7~r0 のデータを PC に加算し、その結果を PC に転送します。比較したデータが同じ場合、次の命令を実行します。

また、ACC が d8~d0 で指定された RAM、または特殊機能レジスタ (SFR) の内容よりも小さい場合、キャリーフラグ (CY) はセットされ、同じか、大きい場合、CY はリセットされます。

ACC < d9 (RAM or SFR) CY = 1

ACC ≥ d9 (RAM or SFR) CY = 0

## 例 1

BNE 命令実行時に ACC < B なので、CY をセットしラベル LA に分岐します。

			PC	命令コード	ACC	B	CY
	MOV	#002H, ACC	0F17H	230002H	02H	-	-
	MON	#003H, B	0F1AH	230203H	02H	03H	-
	BNE	B, LA	0F1DH	43023FH	02H	03H	1
LA:	INC	ACC	0F5FH	6300H	03H	03H	1

## 例 2

BNE 命令実行時に ACC = B なので、CY をリセットし次の命令の実行に移ります。

			PC	命令コード	ACC	B	CY
	MOV	#0F2H, ACC	0F17H	2300F2H	F2H	-	-
	MOV	#0F2H, B	0F1AH	2302F2H	F2H	F2H	-
	BNE	B, LA	0F1DH	43023FH	F2H	F2H	0
	DEC	ACC	0F20H	7300H	F1H	F2H	0
LA:	INC	ACC					

# BNE @Rj, #i8, r8

Compare immediate data to indirect byte and branch near relative address if not equal

命令コード	0 1 0 0 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (44H~47H)
バイト数	3
サイクル数	2
機能	(PC) (PC) + 3, if ((Rj)) # i8 then (PC) (PC) + r8 j = 0, 1, 2, 3
影響を受けるフラグ	CY
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容とイミディエイトデータ (i7~i0) の内容を比較し、比較したデータが異なる場合、r7~r0 のデータを PC に加算し、その結果を PC に転送します。比較したデータが同じ場合、次の命令を実行します。

また、j1~j0 で指定される間接アドレスレジスタによって指定される RAM、または特殊機能レジスタ (SFR) の内容がイミディエイトデータ (i7~i0) よりも小さい場合、キャリーフラグ (CY) はセットされ、同じか、大きい場合、CY はリセットされます。

@ Rj < # i8 CY = 1  
@ Rj # i8 CY = 0

## 例 1

BNE 命令実行時に B < 08H なので、CY をセットしラベル LA に分岐します。

			PC	命令コード	ACC	B	CY
	MOV	#002H, ACC	0F17H	230002H	02H	-	-
	MON	#003H, B	0F1AH	230203H	02H	03H	-
	BNE	B, LA	0F1DH	43023FH	02H	03H	1
LA:	INC	ACC	0F5FH	6300H	03H	03H	1

## 例 2

BNE 命令実行時に ACC = 03H なので、CY をリセットし次の命令の実行に移ります。

			PC	命令コード	ACC	RAM	CY
						02H	
	MOV	#003H, ACC	0F17H	230003H	03H	-	-
	MOV	#000H, 002H	0F1AH	220200H	03H	00H	-
	BNE	@R2, #3H, LA	0F1DH	46033FH	03H	00H	0
	DEC	ACC	0F20H	7300H	02H	00H	0
LA:	INC	ACC					

# サブルーチン命令

## CALL a12

Near absolute subroutine CALL

命令コード	0 0 0a11 1a10a9a8 a7a6a5a4a3a2a1a0 (08H~0FH, 18H~1FH)
バイト数	2
サイクル数	2
機能	(PC) (PC)+2(SP) (SP)+1((SP)) (PC7~0)(SP) (SP)+1((SP)) (PC15~8)(PC11~0) a12
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

### 説明

プログラムカウンタ(PC)を2回インクリメントします。後に、スタックポインタ(SP)をインクリメントし、PCの下位バイトをSPで指定されるRAMに格納します。さらにスタックポインタ(SP)をインクリメントし、PCの上位バイトをSPで指定されるRAMに格納します。最後に、a11~a0のデータをPCのビット11~00に転送します。

### 例 1

ラベル LA の値は、0F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH,SP	OFFAH	23061FH	1FH	-	-
	CALL LA	OFFDH	1F0EH	21H	FFH	0FH
LA:	INC ACC	0F0EH	6300H	21H	FFH	0FH
	RET	0F10H	A0H	1FH	FFH	0FH
	NOP	0FFFH	00H	1FH	FFH	0FH

### 例 2

ラベル LA の値は、1F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH,SP	OFFBH	23061FH	1FH	-	-
	CALL LA	OFFEH	1F0EH	21H	00H	10H
LA:	INC ACC	1F0EH	6300H	21H	00H	10H
	RET	1F10H	A0H	1FH	10H	
	INC ACC	1000H	6300H	1FH	00H	10H

# CALLF a16

Far absolute subroutine CALL

命令コード	0 0 1 0 0 0 0 0    a15a14a13a12a11a10a9a8    a7a6a5a4a3a2a1a0    ( 20H )
バイト数	3
サイクル数	2
機能	( PC ) ( PC ) + 3 ( SP ) ( SP ) + 1 ( ( SP ) ) ( PC7 ~ 0 ) ( SP ) ( SP ) + 1 ( ( SP ) ) ( PC15 ~ 8 ) ( PC )    a16
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

プログラムカウンタ ( PC ) を 3 回インクリメントします。後に、スタックポインタ ( SP ) をインクリメントし、PC の下位バイトを SP で指定される RAM に格納します。さらにスタックポインタ ( SP ) をインクリメントし、PC の上位バイトを SP で指定される RAM に格納します。最後に、a15 ~ a0 のデータを PC のビット 15 ~ 00 に転送します。

## 例 1

ラベル LA の値は、0F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV        #01FH, SP	0FF9H	23061FH	1FH	-	-
	CALLF      LA	0FFCH	200F0EH	21H	FFH	0FH
LA:	INC        ACC	0F0EH	6300H	21H	FFH	0FH
	RET	0F10H	A0H	1FH	FFH	0FH
	NOP	0FFFH	00H	1FH	FFH	0FH

## 例 2

ラベル LA の値は、0F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV        #01FH, SP	0FFAH	23061FH	1FH	-	-
	CALLF      LA	0FFDH	200F0EH	21H	00H	10H
LA:	INC        ACC	0F0EH	6300H	21H	00H	10H
	RET	0F10H	A0H	1FH	00H	10H
	INC        ACC	1000H	6300H	1FH	00H	10H

# CALLR r16

Far relative subroutine CALL

命令コード	0 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 r15r14r13r12r11r10r9r8 (10H)
バイト数	3
サイクル数	4
機能	(PC) (PC) + 3(SP) (SP) + 1((SP)) (PC7~0)(SP) (SP) + 1((SP)) (PC15~8)(PC) (PC) - 1 + r16
影響を受けるフラグ	
割り込みの受付	4 サイクル目で可

## 説明

プログラムカウンタ (PC) を 3 回インクリメントします。後に、スタックポインタ (SP) をインクリメントし、PC の下位バイトを SP で指定される RAM に格納します。さらにスタックポインタ (SP) をインクリメントし、PC の上位バイトを SP で指定される RAM に格納します。最後に、PC の内容をデクリメントして、PC の内容と r15 ~ r0 のデータを加算し、その結果を PC に転送します。

## 例 1

ラベル LA の値は、1100H です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH,SP	0FF9H	23061FH	1FH	-	-
	CALLR LA	0FFCH	100201H	21H	FFH	0FH
LA:	INC ACC	1100H	6300H	21H	FFH	0FH
	RET	1102H	A0H	1FH	FFH	0FH
	NOP	0FFFH	00H	1FH	FFH	0FH

## 例 2

ラベル LA の値は、1100H です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH,SP	0FFCH	23061FH	1FH	-	-
	CALLR LA	0FFDH	100101H	21H	00H	10H
LA:	INC ACC	1100H	6300H	21H	00H	10H
	RET	1102H	A0H	1FH	00H	10H
	INC ACC	1000H	6300H	1FH	00H	10H

# RET

Return for subroutine

命令コード	1 0 1 0 0 0 0 0 (A0H)
バイト数	1
サイクル数	2
機能	(PC15~8) ((SP)) (SP) (SP) - 1 (PC7~0) ((SP)) (SP) (SP) - 1
影響を受けるフラグ	
割り込みの受付	2 サイクル目で可

## 説明

スタックポインタ (SP) で指定される RAM の内容を、プログラムカウンタ (PC) の上位バイトに転送します。後に、SP をデクリメントし、SP で指定される RAM の内容を PC の下位バイトに転送し、さらに SP をデクリメントします。

## 例 1

ラベル LA の値は、0F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH, SP	0FF9H	23061FH	1FH	-	-
	CALLF LA	0FFCH	200F0EH	21H	FFH	0FH
LA:	INC ACC	0F0EH	6300H	21H	FFH	0FH
	RET	0F10H	A0H	1FH	FFH	0FH
	NOP	0FFFH	00H	1FH	FFH	0FH

## 例 2

ラベル LA の値は、0F0EH です。

		PC	命令コード	SP	RAM	RAM
					20H	21H
	MOV #01FH, SP	0FFAH	23061FH	1FH	-	-
	CALLF LA	0FFDH	200F0EH	21H	00H	10H
LA:	INC ACC	0F0EH	6300H	21H	00H	10H
RET		0F10H	A0H	1FH	00H	10H
	INC ACC	1000H	6300H	1FH	00H	10H

# RETI

Return for interrupt

命令コード	1 0 1 1 0 0 0 0 (B0H)
バイト数	1
サイクル数	2
機能	(PC15~8) ((SP)) (SP) (SP) - 1 (PC7~0) ((SP)) (SP) (SP) - 1
影響を受けるフラグ	
割り込みの受付	不可

## 説明

スタックポインタ (SP) で指定される RAM の内容を、プログラムカウンタ (PC) の上位バイトに転送します。後に、SP をデクリメントし、SP で指定される RAM の内容を PC の下位バイトに転送し、さらに、SP をデクリメントし、割り込み受付時に禁止された割り込み受付機能の動作を再開します。

## 例 1

	PC	命令コード	
NOP	OFFAH	00H	
NOP	OFFBH	00H	
MOV #001H, ACC	OFFCH	230001H	外部割り込み 0 発生
INC ACC	0003H	6300H	
RET1	0005H	B0H	
NOP	0FFFH	00H	

## 例 2

	PC	命令コード	
NOP	0FFCH	00H	
MOV #00EH, B	0FFDH	23020EH	外部割り込み 1 発生
INC ACC	0013H	6300H	
RET1	0015H	B0H	
INC ACC	1000H	6300H	

# ビット操作命令

## CLR1 d9 ,b3

Clear direct ビット

命令コード	1 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 ( C8H ~ CFH , D8H ~ DFH )
バイト数	2
サイクル数	1
機能	( d9 , b3 ) 0
影響を受けるフラグ	
割り込みの受付	可

### 説明

d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットをリセットします。

### 例 1

		ACC		
MOV	#001H,ACC	01H	0000	0001B
CLR1	ACC,0	00H	0000	0000B

### 例 2

		RAM		
		7FH		
MOV	#001H,07FH	01H	0000	0001B
CLR1	07FH,0	00H	0000	0000B

### 注意

この命令をポート P1 , P3 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# SET1 d9 b3

Set direct ビット

命令コード	1 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 ( E8H ~ EFH , F8H ~ FFH )
バイト数	2
サイクル数	1
機能	( d9 , b3 ) 1
影響を受けるフラグ	
割り込みの受付	可

## 説明

d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットをセットします。

## 例 1

		ACC		
MOV	#000H, ACC	00H	0000	0000B
SET1	ACC, 7	80H	1000	0000B

## 例 2

		RAM		
		7FH		
MOV	#001H, 07FH	01H	0000	0001B
SET1	07FH, 6	41H	0100	0001B

### 注意

この命令をポート P1、P3 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# NOT1 d9 b3

Not direct ビット

命令コード	1 0 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 ( A8H ~ AFH , B8H ~ BFH )
バイト数	2
サイクル数	1
機能	( d9 , b3 ) ( d9 , b3 )
影響を受けるフラグ	
割り込みの受付	可

## 説明

d8 ~ d0 で指定される RAM、または特殊機能レジスタ ( SFR ) の b2 ~ b0 で指定されるビットを反転します。

## 例 1

		ACC		
MOV	#000H,ACC	00H	0000	0000B
NOT1	ACC,7	80H	1000	0000B
NOT1	ACC,7	00H	0000	0000B

## 例 2

		RAM		
		7FH		
MOV	#001H,07FH	01H	0000	0001B
NOT1	07FH,6	41H	0100	0001B
NOT1	07FH,6	01H	0000	0001B

### 注意

この命令をポート P1 , P3 に適用した場合、各ポートのポートラッチが選択されます。ポートに加えられた外部信号は選択されません。また、ポート P7 に適用しても状態の変化はありません。

# その他の命令

## NOP

No operation

命令コード	0 0 0 0 0 0 0 0 ( 00H )
バイト数	1
サイクル数	1
機能	
影響を受けるフラグ	
割り込みの受付	可

### 説明

1 サイクルクロックを消費します。

# マクロ命令

## CHANGE ラベル名(またはアドレス)

プログラムモードの切り替え

### 説明

ROM 内システム BIOS とフラッシュメモリ内ユーザープログラムを切り替えます。

(1) ROM 内プログラム動作中での実行

- ・システム BIOS          アプリケーションへの移行
- ・ラベルまたはアドレスで指定されたフラッシュメモリのアドレスにプログラムカウンタがセットされます。

### 注意

システム BIOS からは、フラッシュメモリのバンク 0 のアドレス 0000H が呼び出されます。このアドレスは固定です。

(2) アプリケーション動作中での実行

- ・アプリケーション          システム BIOS への移行 (LDCEXT = 0 の場合)
- ・ラベルまたはアドレスで指定された ROM 内プログラムのアドレスにプログラムカウンタがセットされます。

### 注意

LDCEXT = 1 でかつアプリケーション実行中に CHANGE 命令を実行した場合でも、システム BIOS へのジャンプは行われません。この場合は、フラッシュメモリの CHANGE 命令で指定したアドレスへジャンプします。

(3) プログラムモードの移行は専用マクロ命令の実行後に移行します。

(4) 割り込みは受け付けません。

### 注意

ビジュアルメモリ開発セットのアセンブラに含まれている "GHEAD.ASM" ファイルを組み込んで、システム BIOS を呼び出してください。ROM 内アドレスを意識することなく OS プログラムを呼び出せます。

### 参照

OS プログラムの呼び出し方については『ビジュアルメモリ ハードウェアマニュアル』の「システム BIOS 編」を参照してください。



# 第 22 章 LC68K 命令セット一 覧

ニーモニック	命令コード	バイト	サイクル	動作	PSW		
					CY	AC	OV
ADD #i8	1 0 0 0 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A)+#i8	○	○	○
ADD d9	1 0 0 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A)+(d9)	○	○	○
ADD @Rj	1 0 0 0 0 1 j1 j0	1	1	(A) (A)+((Rj)) j:0,1,2,3	○	○	○
ADDC #i8	1 0 0 1 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A)+CY+#i8	○	○	○
ADDC d9	1 0 0 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A)+CY+(d9)	○	○	○
ADDC @Rj	1 0 0 1 0 1 j1 j0	1	1	(A) (A)+CY+((Rj)) j:0,1,2,3	○	○	○
AND #i8	1 1 1 0 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A) #i8			
AND d9	1 1 1 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A) (d9)			
AND @Rj	1 1 1 0 0 1 j1 j0	1	1	(A) (A) ((Rj)) j:0,1,2,3			
BE #i8, r8	0 0 1 1 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(A)=#i8, then(PC) (PC)+r8 if(A) < #i8, then CY 1elseCY 0	○		
BE d9, r8	0 0 1 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(A)=(d9), then(PC) (PC)+r8 if(A) < (d9), then CY 1elseCY 0	○		
BE @Rj, #i8, r8	0 0 1 1 0 1 j1 j0 i7 i6 i5 i4 i3 i2 i1 i0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if((Rj))=#i8, then(PC) (PC)+r8 if((Rj)) < #i8, then CY 1elseCY 0 j:0,1,2,3	○		
BE d9, b3, r8	1 0 0 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(d9.d3)=0 then(PC) (PC)+r8			
BNE #18, r8	0 1 0 0 0 0 0 1 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(A)=#i8, then(PC) (PC)+r8 if(A) < #i8, then CY 1elseCY 0	○		
BNE d9, r8	0 0 1 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(A) (d9), then(PC) (PC)+r8 if(A) < (d9), then CY 1elseCY 0	○		
BNE @Rj, #i8, r8	0 0 1 1 0 1 j1 j0 i7 i6 i5 i4 i3 i2 i1 i0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if((Rj))=#i8, then(PC) (PC)+r8 if((Rj)) < #i8, then CY 1elseCY 0 j:0,1,2,3	○		

ニーモニック	命令コード	バイト	サイクル	動作	PSW		
					CY	AC	OV
BNZ r8	1 0 0 1 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0	2	2	(PC) (PC)+2 if(A) 0, then(PC) (PC)+r8			
BP d9, b3, r8	0 1 1 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(d9,d3)=1, then(PC) (PC)+r8	○	○	○
BPC d9, b3, r8	0 1 0 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 if(d9,d3)=1, then(PC) (PC)+r8 (d9,d3) 0	○	○	○
BR r8	0 0 0 0 0 0 0 1 r7 r6 r5 r4 r3 r2 r1 r0	2	2	(PC) (PC)+2 (PC) (PC)+r8	○	○	○
BRF r16	0 0 0 1 0 0 0 1 r7 r6 r5 r4 r3 r2 r1 r0 r15 r14 r13 r12 r11 r10 r9 r8	3	4	(PC) (PC)+2 (PC) (PC)-1+r16	○	○	○
BZ r8	1 0 0 0 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0	2	2	(PC) (PC)+2 if(A)=0, then(PC) (PC)+r8	○	○	○
CALL a12	0 0 0 a11 1 a10 a9 a a7 a6 a5 a4 a3 a2 a1 a0	2	2	(PC) (PC)+2 (SP) (SP)+1 ((SP)) (PC7-0) (SP) (SP)+1 ((SP)) (PC15-8) (PC11-0) a12			
CALLF a16	0 0 1 0 0 0 0 0 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0	3	2	(PC) (PC)+3 (SP) (SP)+1 ((SP)) (PC7-0) (SP) (SP)+1 ((SP)) (PC15-8) (PC) a16			
CALLF r16	0 0 0 1 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0 r15 r14 r13 r12 r11 r10 r9 r8	3	4	(PC) (PC)+3 (SP) (SP)+1 ((SP)) (PC7-0) (SP) (SP)+1 ((SP)) (PC15-8) (PC) (PC)-1+r16			
CLR1 d9, b3	1 1 0 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(d9,d3) 0			
DBNZ d9, r8	0 1 0 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	3	2	(PC) (PC)+3 (d9)=(d9)-1 if(d9) 0, then(PC) (PC)+r18			
DBNZ @Rj, r8	0 1 0 1 0 1 j1 j0 r7 r6 r5 r4 r3 r2 r1 r0	2	2	(PC) (PC)+2 ((Rj))=((Rj))-1 j:0,1,2,3 if(d9) 0, then(PC) (PC)+r8			
DEC d9	0 1 1 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0 r7 r6 r5 r4 r3 r2 r1 r0	2	1	(d9) (d9)-1			
DEC @Rj	0 1 1 1 0 1 j1 j0	1	1	((Rj))=((Rj))-1 j:0,1,2,3			
DIV	0 1 0 0 0 0 0 0	1	7	(A)(C),mod(B) (A)(C)+(B)	○		○

ニーモニック	命令コード	バイト	サイクル	動作	PSW		
					CY	AC	OV
INC d9	0 1 1 0 0 0 1 d8 r7 r6 r5 r4 r3 r2 r1 r0	2	1	(d9) (d9)+1			
INC @Rj	0 1 1 0 0 1 j1 j0	1	1	((Rj)) (Rj))+1 j:0,1,2,3			
JMP a12	0 0 1 d8 1 b2 b1 b0 a7 a6 a5 a4 a3 a2 a1 a0	2	2	(PC) (PC)+2 (PC11-00) a12			
JMPF a16	0 0 1 0 0 0 0 1 a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0	3	2	(PC) a16			
LD d9	0 0 0 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (d9)			
LD @Rj	0 0 0 0 0 1 j1 j0	1	1	(A) (BNK)((TRR))+(A)) [ROM]			
LDC	1 1 0 0 0 0 0 1	1	2				
MOV #i8, d9	0 0 1 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0 i7 i6 i5 i4 i3 i2 i1 i0	3	2	(d9) #i8			
MOV #i8, @Rj	0 0 1 0 0 1 j1 j0 i7 i6 i5 i4 i3 i2 i1 i0	3	4	((Rj)) #i8 j:0,1,2,3			
MUL	0 0 1 1 0 0 0 0	2	1	(B)(A)(C) (A)(C) × (B)	○		○
NOP	0 0 0 0 0 0 0 0	3	2				
NOT1 d9, b3	1 0 1 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0	2	2	(d9,d3) (d9,d3)			
OR #i8	1 1 0 1 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A) #i8			
OR d9	1 1 0 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A) (d9)			
OR @Rj	1 1 0 1 0 1 j1 j0	1	1	(A) (A) ((Rj)) j:0,1,2,3			
POP d9	0 1 1 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	2	(d9) ((SP)) (SP) (SP)-1			
PUSH d9	0 1 1 0 0 0 0 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	2	(SP) (SP)+1 ((SP)) (d9)			
RET	1 0 1 0 0 0 0 0	1	2	(PC15-8) ((SP)) (SP) (SP)-1 (PC7-0) ((SP)) (SP) (SP)-1			
RETI	1 0 1 1 0 0 0 0	1	2	(PC15-8) ((SP)) (SP) (SP)-1 (PC7-0) ((SP)) (SP) (SP)-1			
ROL	1 1 1 0 0 0 0 0	1	1	A7 A6 A5 A4 A0 A1 A2 A3			
ROLC	1 1 1 1 0 0 0 0	1	1	CY A7 A6 A5 A4 A0 A1 A2 A3	○		
ROR	1 1 0 0 0 0 0 0	1	1	A7 A6 A5 A4 A0 A1 A2 A3			

ニーモニック	命令コード	バイト	サイクル	動作	PSW		
					CY	AC	OV
RORC	1 1 0 1 0 0 0 0	1	1	CY A7 A6 A5 A4 A0 A1 A2 A3	○		
SET d9, b3	1 1 1 d8 1 b2 b1 b0 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(d9,b3) 1			
ST d9	0 0 0 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(d9) (A)			
ST @Rj	0 0 0 1 0 1 j1 j0	1	1	((Rj)) (A) j:0 ,1 ,2 ,3			
SUB #i8	1 0 1 0 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A)-#i8	○	○	○
SUB d9	1 0 1 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A)-(d9)	○	○	○
SUB @Rj	1 0 1 0 0 1 j1 j0	1	1	(A) (A)-((Rj)) j:0 ,1 ,2 ,3	○	○	○
SUBC #i8	1 0 1 1 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A)-CY-#i8	○	○	○
SUBC d9	1 0 1 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(A) (A)-CY-(d9)	○	○	○
SUBC @Rj	1 1 0 0 0 0 j1 j0	1	1	(A) (A)-CY-((Rj)) j:0 ,1 ,2 ,3	○	○	○
XCH d9	1 1 0 0 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	(d9) (A)			
XCH @Rj	1 1 0 0 0 1 j1 j0	1	1	((Rj)) (A) j:0 ,1 ,2 ,3			
XOR #i8	1 1 0 1 0 0 0 1 i7 i6 i5 i4 i3 i2 i1 i0	2	1	(A) (A) #i8			
XOR d9	1 1 1 1 0 0 1 d8 d7 d6 d5 d4 d3 d2 d1 d0	2	1	A) (A) (d9)			
XOR @Rj	1 1 1 1 0 1 j1 j0	1	1	(A) (A) ((Rj)) j:0 ,1 ,2 ,3			

### 注意

ニーモニック覧に がついている命令は、バイトまたはビット単位のアドレッシングにおいてポートラッチが選択されます。また、その他の命令では、ポートに加えられる外部信号が選択されます。