Visual Memory Programmer's Guide Revision 1.01

Introduction

Thank you for collaborating in the development of applications for Sega's hardware.

This manual describes the assembler and other development tools, and the hardware instructions for the custom chips embedded in Visual Memory.

Visual Memory includes a custom version of Sanyo's LC86870. This chip is an embeddable microprocessor for a variety of applications, and is referred to generically as LC86K-series.

Use this manual for coding, in conjunction with the details of Visual Memory hardware in the Visual Memory Hardware Manual.

Sections of this manual

Installation
Describes the procedure for installing the Visual Memory development tools provided with the Dreamcast Software Developer's Kit (SDK) .

Assembler
Lists command line options and error messages for the M86K.EXE assembler.

Linkage Loader
Lists command line options and error messages for the L86K.EXE linkage loader.

Library Manager
Lists command line options and error messages for the LIB86K.EXE library manager.

EVA to HEX
The file output by the linker has the extension EVA. This is converted to a HEX format file that can be read into the Visual Memory simulator, using the E2H86K.EXE program.

This section lists the command options for this tool.

HEX to BIN
The output file from E2H86K.EXE has the extension .HEX or .H00. To create a file which can be read into the actual Visual Memory device, the HEX file must be converted to BIN format. H2BIN.EXE is used for this conversion.

This section lists the command options for this tool.

MAKE
The MAKE.EXE program maintenance utility allows just the updated source files to be automatically reassembled to create new object files, so that applications can be built in a convenient batch process.

This section describes how to write and use the files for controlling this process.

Coding

This provides a reference for the M86K.EXE assembler syntax, and assembler pseudoinstructions, together with an LC86870 instruction set reference.

Note:

This manual is based on the Visual Memory Programming Manual (Revision 1.00), which in turn was based on the LC86000 User's Manual (published by Sanyo Electric) with additions by Sega Enterprises for the specializations for Visual Memory. In the second edition there are amendments to the sections on installation, and additions regarding BIN to HEX.

CPU specifications and any of the contents of this manual are subject to future revision without notice.

Sega Enterprises can accept no responsibility whatsoever for consequences of actions taken using Visual Memory and this manual.

Trademarks

SEGA, Dreamcast, and Visual Memory are trademarks of Sega Enterprises.

Microsoft, Windows, and the Windows CE logo are registered or unregistered trademarks of Microsoft Corporation in the United States and other countries.

Other product names referred to in this manual are registered or unregistered trademarks of their respective owners. They are not indicated in the text by "TM" or "(r)" indications.

Revision History

August 31, 1998 First Edition

Second Edition November 30, 1998

Copyright © 1998 Sanyo Electric

Copyright © 1998 Sega Enterprises

Editing and Production: ASCII Corporation AAP Publishing Division

Relationship To Other Manuals

Visual Memory Hardware Manual

This is a technical reference covering the Visual Memory hardware and system BIOS specifications. The summary at the beginning provides a quick reference to the Visual Memory specification.

The summary will be useful for game designers, and the body of the manual for programmers.

Visual Memory Programmer's Guide

This manual provides comprehensive coverage of assembler, linker, library manager, and the E2H86K.EXE utility for creating executable files for the Sanyo LC86700 on which Visual Memory is based. It also covers the LC86700 instruction set and assembler syntax..

Visual Memory Tutorial

This describes the procedure for developing a Visual Memory application, and the method of porting applications to Visual Memory. It includes fully-commented sample programs.

Visual Memory Simulator Guide

This describes the software Visual Memory Simulator.

Sega Library Reference Volume 2

Refer to this manual when porting data and programs from Dreamcast to Visual Memory, or when displaying on the Visual Memory LCD screen connected to a Dreamcast controller.

For reading and writing Dreamcast game data in Visual Memory, refer to the functions in the Shinobi library.

For display on the LCD screen, refer to the LCD functions in the Shinobi library.

Restrictions

All development tools are MS-DOS programs.

Except for the Visual Memory simulator and Memory Card Utility, all Visual Memory tools are MS-DOS programs.

Technical Support

If, in the course of your development work, you have any technical questions, encounter hardware difficulties, need information that is not included in this manual, or experience a hardware malfunction, contact technical support at:

Sega Enterprises
Technical Support Center
1-2-12, Haneda, Ota-ku, Tokyo 144-8531
Japan
Tel: 03-5736-7355
Fax: 03-5736-5357
E-mail: katana@sft.sega.co.jp

contents

Introduction 2

Relationship To Other Manuals 4

Restrictions 5

Technical Support 6

PART 1 Installation **21**

Chapter 1
Setup **23**

1.1 Executing the Setup Program 23

1.2 Post-Installation Overview 30

Chapter 2
Setting Environment Variables **31**

2.1 Environment Variables for the Development Tools 31

2.1.1 Environment Variable Settings 32

PART 2Assembler 33

Chapter 3
Specifying Files for Assembly 35

3.1 Specifying File Names 35

3.2 Specifying Parameters on the Command Line 36

3.3 Specifying Parameters at the Prompts.....37

Chapter 4
Option Switches 39

-I Do not distinguish case in identifiers.....39
-D Do not output debugging information 39
-J Do not optimize jump instructions 39
-N Suppress copyright notice 39
-R Specify reserved word file 40
-P Specify working buffer size 40
-? Show options 40

Chapter 5
Environment Variables and Reserved Word File 41

5.1 Environment Variables.....41

5.2 Reserved Word File 42

Contents

Chapter 6
Errors 43

6.1 Warnings 43

6.2 Non-Fatal Errors 45

6.3 Fatal Errors 51

Chapter 7
Listing Format 55

PART 3PART 3 Linkage Loader 59

Chapter 8
Specifying Files for Linking 61

8.1 Specifying File Names 61

8.2 Specifying Parameters on the Command Line 62

8.3 Specifying Parameters at the Prompts.....63

8.4 Files Referenced During Linking 64

Chapter 9
Option Switches 65

-B=bank number
Creating a flash memory HEX file for the LC86800 series.....65
-C=address
Specifying a CSEG loading address.....65
-D=address
Specifying a DSEG loading address.....65
-E Allowing DSEG address overlaps 66
-I Do not distinguish case 66
-P Create a loading map 66
-L Create a local symbol listing 66
-W Issue warning messages for operand values.....67
-A, -F, -O, -R
Optimizing loading of CSEG FREE blocks.....67
-S Specify symbol sorting 68

Chapter 10
Object Alignment 69

10.1 -A option 69

10.2 -A -F options.....70

10.3 -A -O options 70

10.4 -A -R options 71

Chapter 11
Errors 73

11.1 Fatal Errors 73

11.2 Non-Fatal Errors 74

PART 4 Library Manager 75

Chapter 12
Starting the Program 77

12.1 Specifying File Names 77

12.2 Specifying Parameters on the Command Line 78
12.2.1 Option 79
12.2.2 Examples of Command Line Execution 79

12.3 Operation with the Prompts 80
12.3.1 Prompt Line Extension 80
12.3.2 Default Responses 80

Chapter 13
Error Messages 81

Chapter 14
Cross-Reference 83

Contents

PART 5 EVA to HEX 85

Chapter 15
Starting the Program 87

15.1 Specifying File Names 87

15.2 Specifying Parameters 88

15.3 Option Specification 89

/I Suppress information display during conversion 89

Chapter 16
Error Messages 91

16.1 Fatal Errors 91

PART 6 HEX to BIN 93

Chapter 17
Starting the Program 95

17.1 Specifying File Names 95

17.2 Specifying Parameters 95

Chapter 18
Error Messages 97

18.1 Fatal Errors 97

Contents

PART 7 MAKE 99

Chapter 19
Overview of MAKE 101

19.1 Running MAKE 101
19.1.1 Build Priority Sequence 102
19.1.2 Command Line Options 102
/E Give priority to external macros 102
/I Ignore result codes, and continue processing 102
/N Show sequence of build without executing commands 102
/R Do not read rule file 102
/S Suppress command display 103
/? Show help 103

19.2 Makefile Syntax 103
19.2.1 Generation Rules 103
19.2.2 Macros 105
19.2.3 Directives 107

19.3 Implicit Rules 108
19.3.1 Makerule file 108

PART 8 Coding 111

Chapter 20
Assembler Syntax 113

20.1 Statements 113
20.2 Label and Symbol Names 113
20.3 Comments 114
20.4 Operators 114
20.5 Numeric Constants 114
20.6 Character Constants 116
20.7 Character String Constants 116
20.8 Special Symbols 117

Chapter 21
Assembler Pseudoinstructions 119

ORG 121
WORLD 122
CSEG 123
DSEG 124
END 125
PUBLIC 126
EXTERN 128
OTHER_SIDE_SYMBOL 129
EQU 130
SET 131
DB 132
DW 133
DC 134
DS 135

Contents

MACRO 136
REPT 139
IRP 140
IRPC 141
ENDM 142
EXITM 143
LOCAL 144
IFDEF 146
IFNDEF 147
IFB 148
IFNB 149
IFE 150
IFNE 151
IFIDN 152
IFDIF 153
ELSE 154
ENDIF 155
.PRINTX 156
.LIST 157
.XLIST 158
.MACRO 159
.XMACRO 160
.IF 161
.XIF 162
INCLUDE 163
TITLE 164
PAGE 165
CHIP 166
COMMENT 167
WIDTH 168
BANK 169
CHANGE 170
RADIX 171
JMPO 172
BRO 173
CALLO 174
BZO 175
BNZO 176
BPO 177
BPCO 178
BNO 179
DBNZO 180
BEO 181
BNEO 182

Chapter 22
LC86K Instruction Summary 183

22.1 Instruction Summary.....183

22.1.1 Arithmetic Instructions.....183
22.1.2 Logical Instructions.....184
22.1.3 Data Transfer Instructions.....184
22.1.4 Jump Instruction.....184
22.1.5 Conditional Branch Instructions.....184
22.1.6 Subroutine Instruction.....185
22.1.7 Bit Manipulation Instructions.....185
22.1.8 Other Instructions.....185
22.1.9 Macro Instruction.....185
22.1.10 Addressing.....185
22.1.11 Program Memory Addressing.....185
22.1.12 RAM and Special Function Register (SFR) Addressing.....187

Chapter 23
Instruction Set Reference 191

Arithmetic Instructions.....192

ADD _i8.....192
ADD d9.....193
ADD @Rj.....194
ADDC _i8.....195
ADDC d9.....196
ADDC @Rj.....197
SUB _i8.....198
SUB d9.....199
SUB @Rj.....200
SUBC _i8.....201
SUBC d9.....202
SUBC @Rj.....203
INC d9.....204
INC @Rj.....205
DEC d9.....206
DEC @Rj.....207
MUL.....208
DIV.....209

Logical Instructions.....210

AND _i8.....210

AND d9 211
AND @Rj 212
OR _i8 213
OR d9 214
OR @Rj 215
XOR _i8 216
XOR d9 217
XOR @Rj 218
ROL 219
ROLC 220
ROR 221
RORC 222

Data Transfer Instructions 223
LD d9 223
LD @Rj 224
ST d9 225
ST @Rj 226
MOV _i8, d9 227
MOV _i8, @Rj 228
LDC 229
PUSH d9 230
POP d9 231
XCH d9 232
XCH @Rj 233

Jump Instructions 234
JMP a12 234
JMPF a16 235
BR r8 236
BRF r16 237

Conditional Branch Instructions 238
BZ r8 238
BNZ r8 239
BP d9, b3, r8 240
BPC d9, b3, r8 241
BN d9, b3, r8 242
DBNZ d9, r8 243
DBNZ @Rj, r8 244
BE _i8, r8 245
BE d9, r8 246
BE @Rj, _i8, r8 247
BNE _i8, r8 248

BNE d9, r8 249
BNE @Rj, _i8, r8 250

Subroutine Instructions 251
CALL a12 251
CALLF a16 252
CALLR r16 253
RET 254
RETI 255

Bit Manipulation Instructions 256
CLR1 d9, b3 256
SET1 d9, b3 257
NOT1 d9, b3 258

Miscellaneous Instruction 259
NOP 259

Macro Instruction 260
CHANGE <label (or address)> 260

Chapter 24
LC86K Instruction Set Summary 261

Dreamcast™

PART 1

Installation

This section describes the procedure for installing the assembler, linker, Visual Memory simulator, the Memory Card Utility for transferring data, and other tools.

Chapter 1 Setup

This installs the Visual Memory tools supplied with the Dreamcast SDK.

Caution	This description assumes Version 1.3J of the Dreamcast SDK.

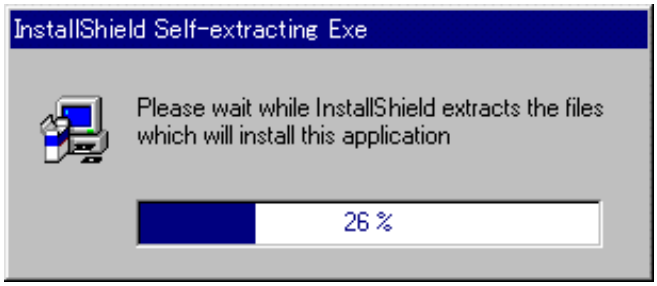
1.1 Executing the Setup Program

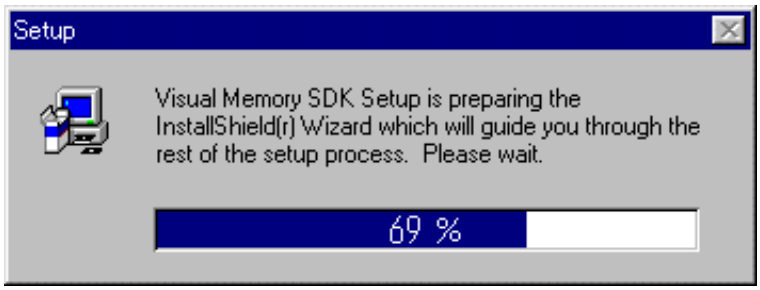
1. Insert the Dreamcast SDK CD-ROM into the drive, and using Explorer or another tool, open the VM_SDK directory.

Caution	There may be last-minute changes which do not appear in the manual. Be sure to check the README.TXT file.

2. For automatic setup, including environment variables, double-click on VMSETUP2.EXE.
3. When the next dialog box appears, click the Yes button.

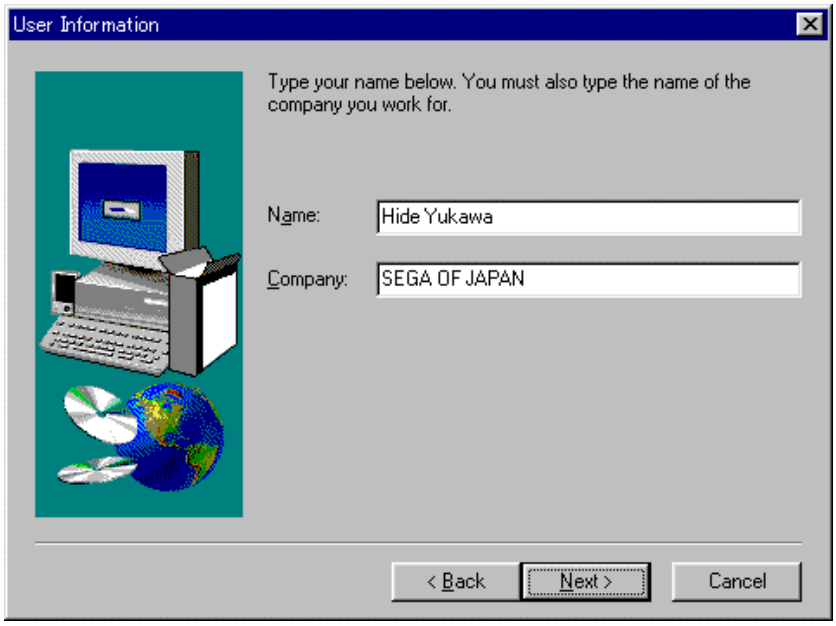
4. When the next dialog box appears, wait until the progress indicator reaches 100_. Two dialog boxes are shown.



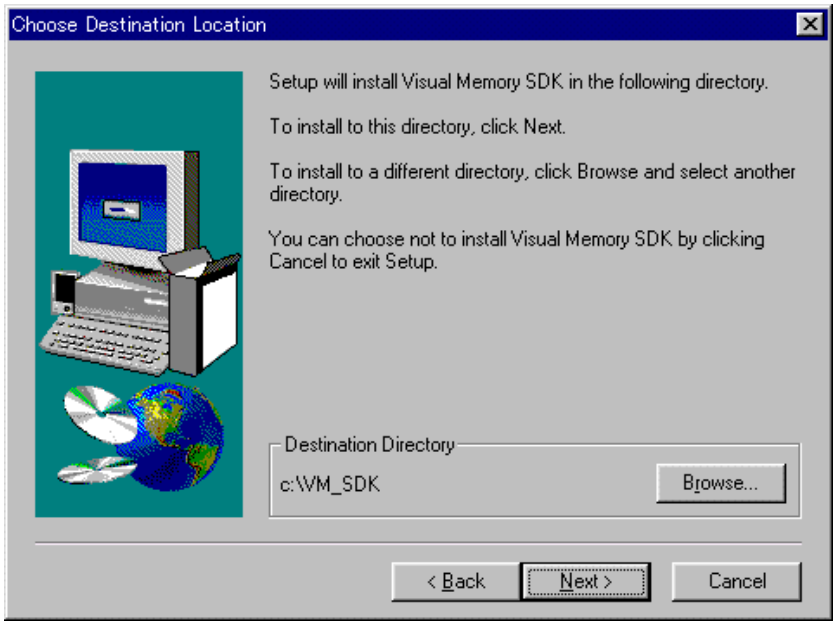


5. The installer has now started. Click the Next button in the dialog box.

6. When the next dialog box appears, enter your name and your company name. When these are complete, click Next.

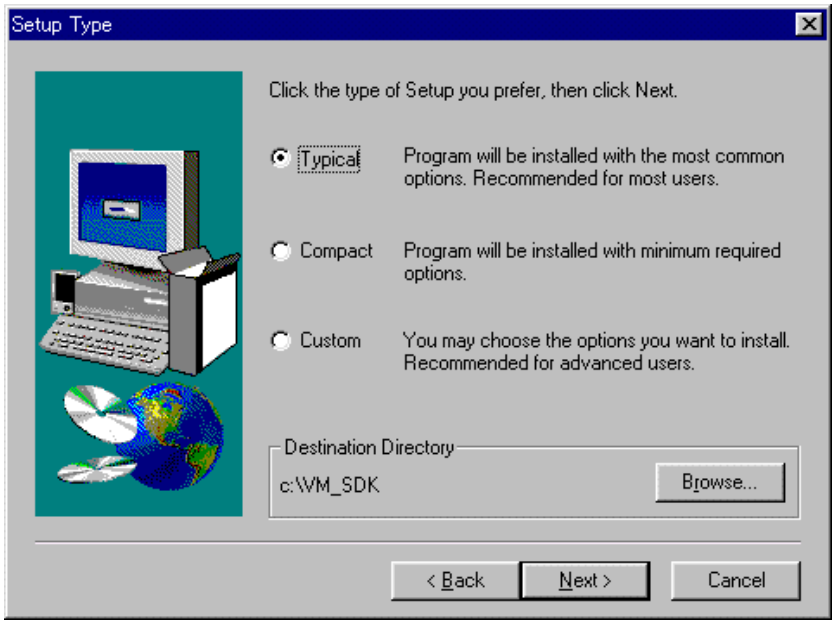


7. When the next dialog box appears, enter the drive and path specification of where you want to install the Visual Memory SDK.
After you have entered this, click Next.



To install with a different path specification, click Browse, and enter the drive and path specification.

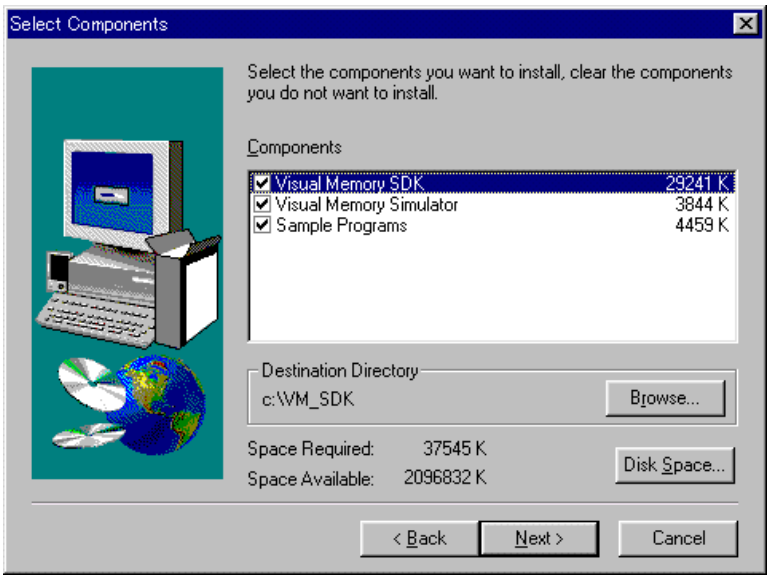
8. When the next dialog box appears, select the internal components of the Visual Memory SDK to be installed.



Select **Typical** to install all SDK components.

Select **Compact** to install the whole SDK except for sample programs.

Select **Custom** to select individual components in the next dialog box.

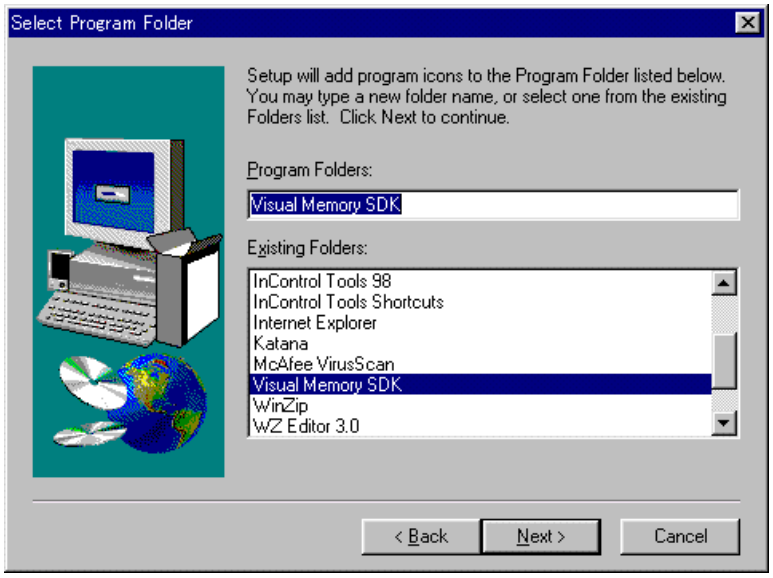


Select **Visual Memory SDK** to install the assembler, linker, and other tools, and the Memory Card Utility for application transfer.

Select **Visual Memory Simulator** to install the Visual Memory simulator.

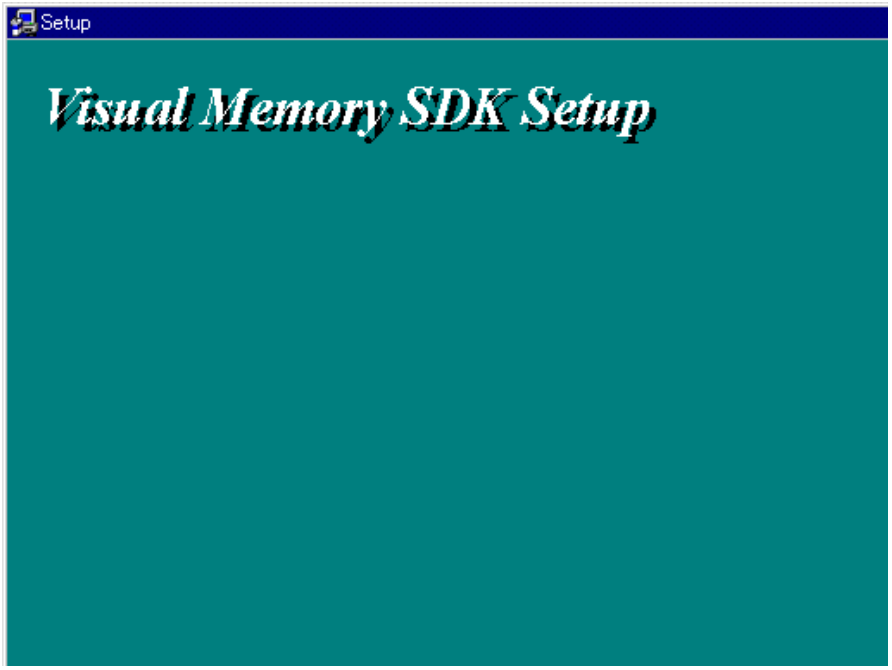
Select **Sample Programs** to install the sample programs.

9. Add the program to the start menu. Enter the name, then click Next.

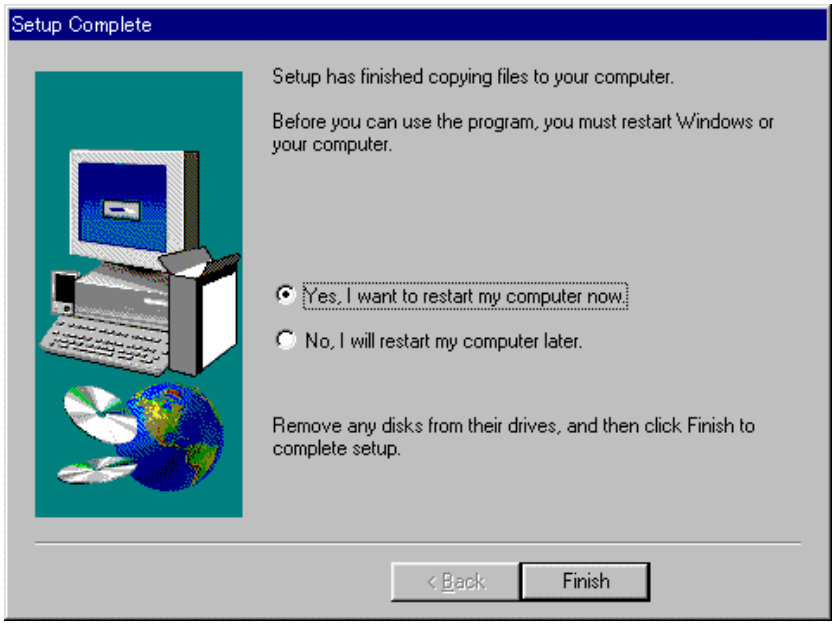


Caution	Only the Visual Memory simulator can be added to the Start menu. Other tools are **MS-DOS** programs, and cannot be added to the Start menu.

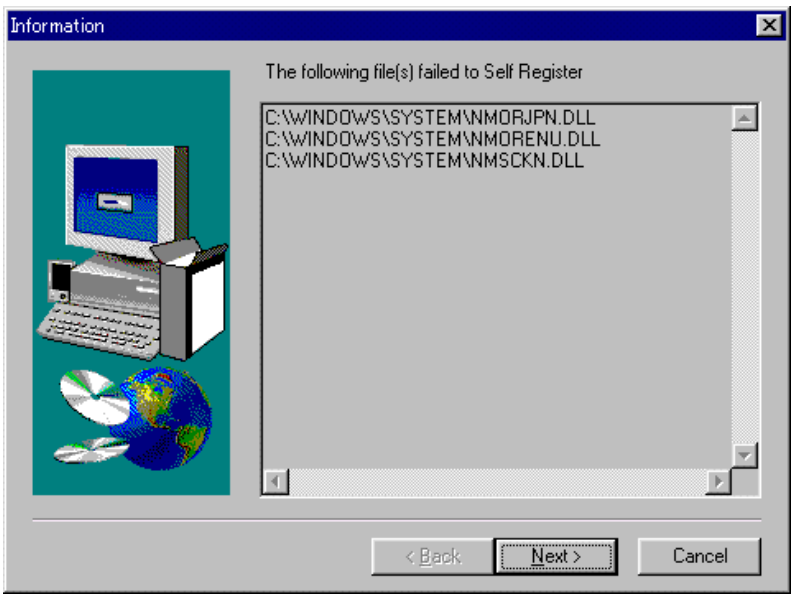
10. Copying of the files now starts. Wait until this finishes.



11. A restart confirmation dialog box appears.
    Select **Yes, I want to restart my computer now** and click Finish to restart the computer immediately.



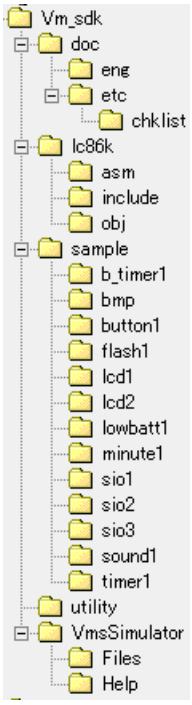
12. If the following dialog box appears, click Next.



Note	The setup program adds the following to the **AUTOEXEC.BAT** file.
	`SET PATH=%PATH%;C:\VM_SDK\LC86K`
	`SET CHIPNAME=LC868700`
	`SET M86KRSVDFILE=c:\VM_SDK\lc86k\m86krsvd.rwd`
	`SET TOOL86=c:\VM_SDK\lc86k`

1.2 Post-Installation Overview

Installing the Visual Memory SDK creates the following directories.



Name	Meaning
Utility	Contains the utility for transfers PC__Dev. Box__Visual Memory, This program is a Dev. Box executable file. Run it using CodeScape or similar.
Doc	Contains documents relating to Visual Memory.
lc86k	Contains the assembler, linker, and other development tools. Also includes header files, libraries, and the GHEAD.ASM file.
Sample	Contains sample programs.
VmsSimulator	Contains the Visual Memory simulator.

Chapter 2 Setting Environment Variables

2.1 Environment Variables for the Development Tools

The following environment variables are used with the L86K series development tools.

PATH

Defines the search path, by adding to the existing PATH definition.

Name	Search file
M86K	Searches for the reserved word file M86KRSVD.RWD in the directories specified by the PATH variable.
L86K	Searches for the reserved word defined symbol file LC86L.LIB in the directories specified by the PATH variable.
CGR86K	Searches for the default character generator data file DEFAULT.CGR in the directories specified by the PATH variable.

CHIPNAME

This defines the chip name (or series name) on which the software will run.

For developing Visual Memory applications, specify LC868700.

Name	Meaning
M86K	Defines the chip name for the assembly. This is ignored if there is a CHIP pseudoinstruction (see Chapter 21, "Assembler Pseudoinstructions") in the source program; it is referenced at assemble time if there is no CHIP pseudoinstruction. If the ROM size of the chip (the last two digits) is set to "00", then no ROM size check is carried out, and the assembly proceeds as though with 64K installed.
SU86K	Defines the chip name for option data creation. The ROM size specification (the last two digits of the chip name) is ignored.
CGR86K	Defines the chip name for character generator data file creation. The ROM size specification (the last two digits of the chip name) is ignored.

If the last two digits of the chip name are "00", this does not indicate a specific chip. It is a generic label for a series, but is given the maximum ROM capacity in the

series. Such a chip name can be used in order to determine the size of an assembled program.

M86KRSVDFILE

Defines the name of the reserved word file.

Name	Search file
M86K	Defines the directory in which the reserved word file is stored, and the file name. If this environment variable is not defined, the default file name is M86KRSVD.RWD, and the directories in the PATH specification are searched in sequence for the file M86KRSVD.RWD.

M86KWORKFILE

Defines the name of the work file.

Name	Search file
M86K	Specifies the name of the file which will be used as a kind of expansion memory for work space when the working memory dynamically allocated during the assembly process of the M86K compiler will not fit into main memory, and EMS memory is not installed, or is not available. It is possible to specify a drive and path name.

TOOL86K

Defines the directory holding the assembler, linker, LC86K.LIB, and so on.

Caution	This environment variable must be set.

TMP

Defines the directory for temporary workfiles.

Name	Search file
M86K	When M86K needs to create a work file (see "M86KWORKFILE"), if the environment variable M86KWORKFILE is not defined, then a work file is created in the directory specified by this environment variable. If this environment variable is not defined, the work file is created in the current directory. In either case, the file name is fixed, and is always M86KWORK.TMP.

2.1.1 Environment Variable Settings

Use the SET command to set environment variables in MS-DOS. For details of the SET command, refer to the documentation and help files supplied with MS-DOS.

Example: To set the default chip name to LC868700

```
C> SET CHIPNAME=LC868700↵
```

Dreamcast™

PART 2

Assembler

This section lists the command line options and error messages for the M86K.EXE assembler.

For assembler syntax, assembler pseudoinstructions, and the LC86870 instruction set, see the section "Coding."

Chapter 3 Specifying Files for Assembly

There are two ways of specifying the parameters when starting M86K.

(1) Specifying all parameters on the M86K command line

(2) Specifying parameters by responding to the prompts displayed by M86K

To force an exit from M86K, press the following key combinations.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Ctrl + Stop

3.1 Specifying File Names

When specifying file names to M86K, either in the command line, or in response to the prompts, case is ignored. For example, the following file names all refer to the **same file**.

```
sample.asm
SAmplE.ASM
SAMPLE.asM
```

If a file name is specified without an extension, M86K supplies the following default extensions.

File format	Default extension
Source file	.ASM
Object file	.OBJ
Listing file	.LST
Cross-reference file	.CRF
Error file	.ERR

3.2 Specifying Parameters on the Command Line

```
M86K _options_ _source_,_object_,_list_,_cross_,_error_↵
```

options

Specify assembler options as listed in Chapter 4, "*Option* Switches." If options are specified they must come before other parameters.

source

Specify the source file to be assembled. If the file extension is omitted, the default extension .ASM is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible.

object

Specify the object file name for the assembler output. A drive name and full path specification is possible. If this file specification is omitted, the object file name is the source file name, with the extension changed to .OBJ. If the file already exists, it is overwritten.

list

Specify a file name for the assembly listing output. A drive name and full path specification is possible. If this file specification is omitted, no listing file is created. If the listing file already exists, it is overwritten.

cross

Specify a file name for the assembly symbol cross-reference output. A drive name and full path specification is possible. If this file specification is omitted, no cross-reference is output. If the cross-reference file already exists, it is overwritten.

error

Specify a file name for the output of assembly error messages. A drive name and full path specification is possible. If this file specification is omitted, no error file is created. If the error file already exists, it is overwritten.

Example

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

This begins assembly of the MAIN.ASM source file in the current directory. The object file MAIN.OBJ is created; no listing file is output, but a cross-reference is written to file TEST.CXX,

3.3 Specifying Parameters at the Prompts

Enter the following command to start the assembler without specifying any file names. Then follow the prompts from the assembler to enter the file names.

```
prompt M86K [options]↵
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]: ……
```

options

Specify assembler options as listed in Chapter 4, "Option Switches."

Source filename

Specify the source file to be assembled. If the file extension is omitted, the default extension .ASM is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible. This file name may not be omitted. Pressing the Enter key alone results in a reprompt for the source file name. To cancel the operation at this point, use the following keys.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	Ctrl + C or Ctrl + Stop

Object filename

Specify the object file name for the assembler output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), the object file name is the source file name, with the extension changed to .OBJ. If the file already exists, it is overwritten.

Source listing

Specify a file name for the assembly listing output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no listing file is created. If the listing file already exists, it is overwritten.

Cross reference

Specify a file name for the assembly symbol cross-reference output. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no cross-reference is output. If the cross-reference file already exists, it is overwritten.

Error messages

Specify a file name for the output of assembly error messages. A drive name and full path specification is possible. If this file specification is omitted (by pressing the Enter key alone), no error file is created. If the error file already exists, it is overwritten.

Chapter 4 Option Switches

This chapter describes the assembler options which can be used to control the operation of M86K itself. All options are indicated by a prefixed minus sign or slash. The options are not case dependent. Thus, for example, -i, -I, /i, and /I all mean the same.

-I
Do not distinguish case in identifiers

When this switch is specified, the assembler ignores case in treating user-defined identifiers (labels, macro names, and symbols). If this switch is not specified, then uppercase and lowercase letters are distinguished. This applies only to user-defined identifiers, and not to mnemonics, SFR names and so forth.

-D
Do not output debugging information

When this switch is specified, the assembler does not include symbol information and source line numbers in the object file. If this information is not present in the object file, source line mode debugging is not possible. If this switch is not specified, symbol information and source line numbers are both included, and source line mode debugging is possible.

-J
Do not optimize jump instructions

If this switch is specified when assembling a source including pseudoinstructions (JMPO, CALLO, BRO) which require optimization, the optimization is suppressed. The result is that each of these pseudoinstructions is interpreted as a three-byte instruction, regardless of the jump destination. If this switch is not specified when assembling a source including pseudoinstructions which require optimization, then the optimization is carried out. If the source does not include pseudoinstructions which require optimization, this switch has no effect.

-N
Suppress copyright notice

When this switch is specified, the copyright notice displayed when the assembler starts up is suppressed. This is convenient when starting the assembler from make or another utility, so that error messages are not obscured by other unnecessary messages.

-R

Specify reserved word file

The character string immediately following this switch until the next space indicates the name of the reserved word file to the assembler. The following example shows this:

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

In this example the name of the reserved word file name is M86KRSVD.RWD. This specification takes precedence over a setting of the M86KRSVDFILE environment variable.

-P

Specify working buffer size

The numerical value immediately following this switch is used as the size of the assembler's internal working buffer. The working buffer is a memory area used by the assembler for increasing the speed of macro registration and expansion, and is reserved in main memory when the assembler is started. The default size is 4096 bytes, and this is normally sufficient for most source programs. If the assembler runs out of working buffer, however, the following error message is issued.

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

The assembler issues this message, then abandons processing (the two digits at the end of the message are an internal value, and may change). If this message should appear, use this switch to specify a large buffer size, and repeat the assembly. For example, use the following specification:

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

In this case, the working buffer size is set to 8192 bytes. The value must be in decimal, and immediately follow the switch letter 'p'. If the switch letter only is specified, or the numeric value cannot be found, the buffer size reverts to the default 4096 bytes.

-?

Show options

If this switch is specified, the assembler displays the following list of options, then immediately terminates. Note that any other option specifications are ignored.

```
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
```

Chapter 5 Environment Variables and Reserved Word File

5.1 Environment Variables

In MS-DOS, to set environment variables use the SET command. For details of the SET command, refer to your MS-DOS documentation and help files.

Example: set the default chip name to LC868700.

```
C> SET CHIPNAME=LC868700↵
```

M86K refers to the following environment variables as required.

PATH

This is used as the search path for the reserved word file. For details of the reserved word file and the search algorithm used to find it, see Section 5.2, "Reserved Word File."

CHIPNAME

This defined the chip for which assembly will be carried out. It is ignored if the CHIP pseudoinstruction appears in the source program. However, if the chip name in the CHIP pseudoinstruction is different from the environment variable chip name, a warning message is issued. This value is thus used if the CHIP pseudoinstruction does not appear in the source program.

M86KRSVDFILE

This defines the directory in which the reserved word file is stored, and the file name. Note that there is no default extension for the file name defined by this environment variable. Specify the drive name and path as required, but always specify the file name and extension.

M86KWORKFILE

Specifies the name of the file which will be used as a kind of expansion memory for work space when the working memory dynamically allocated during the assembly process of the M86K assembler will not fit into main memory, and EMS memory is not installed, or is not available. It is possible to specify a drive and path name.

TMP

When M86K needs to create a work file (see "M86KWORKFILE"), if the environment variable M86KWORKFILE is not defined, then a work file is created in the directory specified by this environment variable. If this environment variable is not defined, the work file is created in the current directory. In either case, the file name is fixed, and is always M86KWORK.TMP.

5.2 Reserved Word File

The reserved word file is always read when M86K starts up, and includes various information specific to the chip for which the program will be assembled (RAM/ROM size, SFR mnemonics, etc.). M86K will not operate correctly without the reserved word file. When M86K starts, it searches for the reserved word file in the following sequence.

1) When a file name is explicitly specified by assembler option-R, that file is loaded. If the file is not present or reads are inhibited, an error occurs.
2) If the environment variable M86KRSVDFILE is defined, the file specified in that variable is loaded. If the file is not present or a read-only file, an error occurs.
3) If reads are enabled and the file M86KRSVD.RWD is present in the directory containing M86K.EXE, that file is loaded.
4) If reads are enabled and the file M86KRSVD.RWD is present in the current directory, that file is loaded.
5) Directories specified by the environment variable PATH are accessed in succession a the first readable nd M86KRSVD.RWD file encountered is read.

If the reserved word file is not found after doing all of the above checks in succession, an error occurs and M86K operation stops. Ordinarily, reserved word files are stored in the same directory as M86K.EXE. This file's contents are vital for normal M86K operation, and Sega will not accept any responsibility for problems with M86K operation if they are deleted or altered. For this reason, it is strongly recommended that this file be write-protected.

Chapter 6 Errors

Errors detected by M86K fall into three categories: fatal errors, (non-fatal) errors, and warnings. When a fatal error is detected, M86K immediately abandons execution. This level includes problems such as the working buffer being exhausted. When a non-fatal error is detected, the assembler continues to the end of the current pass (pass 1 or pass 2), then aborts. This level includes syntax errors in the source program. Warnings are issued for problems that do not warrant "error" classification, such as operands with values outside the permitted range; in this case M86K continues execution.

When a fatal error is detected, all output files which M86K is creating are not produced. When a non-fatal error occurs in pass 1, the files are not produced, but if a non-fatal error occurs in pass 2 the listing file only is output (when specified). The following is the format for showing errors.

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

Example

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

The symbol xyz is undefined.

6.1 Warnings

The following are the warning messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

???: bit number exceeds limits

In a bit manipulation instruction, the number of bits exceeds the permitted range.

absolute expression expected

The expression must be able to be evaluated at assemble time.

address beyond zero

The value specified for the operand of an ORG instruction is negative.

address exceeds limits

The value specified for the operand of an ORG instruction exceeds the specified ROM capacity.

address exceeds ROM size

An address in assembled instruction exceeds the ROM capacity.

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

The operand of a CHIP instruction is different from the current environment variable setting.

END in included file

An END pseudoinstruction was encountered in a source file specified by an INCLUDE pseudoinstruction.

ENDF without FUNCTION

ENDF was encountered while not in a function definition.

ENDM without MACRO

ENDM was encountered while not in a macro definition.

EXITM outside MACRO

EXITM was encountered while not in a macro definition.

function code buffer overflow

A function definition is too large, and will not fit in the buffer.

illegal combination of attributes: ???

The two sides of an arithmetic operator have incompatible attributes (bank or segment values).

illegal style expression

The operand expression of a SET or EQU is illegal.

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

A JMP or CALL instruction was encountered at an address with the bottom 12 bits equal to 0FFEH or 0FFFH. Since the segment alignment mode is "FREE" there may be no problem with the result of linking, but when the segment is aligned from the beginning of a memory boundary, the linker will give errors.

Jump address is out of range (FREE)

The destination address of a jump is outside a memory boundary. Since the segment alignment mode is "FREE" there may be no problem with the result of linking, but when the segment is aligned from the beginning of a memory boundary, the linker will give errors.

LOCAL outside MACRO

LOCAL was encountered while not in a macro definition.

macro name in expression

A symbol defined as a macro was encountered in an expression.

macro name required

The macro name is missing from a macro definition.

no character in string

Character was not found in a string constant.

page width must be 72 ~ 132: ???

The operand of a WIDTH instruction must be in the range 72 to 132.

public ??? not defined

A symbol declared as public is not defined.

SET conflicts with PUBLIC

An attempt has been made to reset a symbol declared as public with a SET.

symbol name required

There is no symbol as the operand of a PUBLIC, EXTERN, or OTHER_SIDE_SYMBOL declaration.

undefined symbol in expression

An undefined symbol occurs in an expression (only detected on pass 2).

value is out of range

A value is outside the permitted range (the range depends on the operand).

zero divide: ??? modulo 0

The right-hand operand of a MOD operator is zero.

zero divide: ??? / 0

The right-hand operand of a division operator is zero.

6.2 Non-Fatal Errors

The following are the non-fatal error messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

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

The operand of a RADIX pseudoinstruction must be 2, 8, 10, or 16.

???: constant required

A numeric constant is missing.

???: duplicated label

A duplicate label occurred.

???: duplicated symbol

A duplicate symbol occurred.

???: illegal character in numeric constant

A numeric constant includes an illegal character.

???: no such chip in the table

A symbol specified in a CHIP instruction cannot be found in the reserved word file.

???: open error

An error occurred when opening a file.

???: undefined symbol

An undefined symbol has been referenced.

???: radix violation

A character in a numeric constant is improper for the base being used.

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

The address allocation in a data segment is outside the permitted range.

' not seen

The closing single quote of a character constant is missing.

':' not seen

In the explicit segment format of an EXTERN operand, the colon separating the segment and symbol is missing.

0x???: RAM address exceeds limits

The address allocation of a data segment is outside the permitted range.

address duplicated

There is a RAM address conflict specified by a DS pseudoinstruction.

address exceeds absolute limits

The address of an assembled instruction exceeds 65535.

bank number should be 0~15

The bank number must be in the range 0 to 15.

Branch address beyond zero

The destination address of a branch is below zero (the start address of the current code segment).

Branch address exceeds limits

The destination address of a branch exceeds the ROM capacity.

CSEG conflicts with WORLD EXTERNAL_DATA

It is not possible to have WORLD EXTERNAL_DATA and a pseudoinstruction specifying a segment in the same source file.

CSEG isn't allowed in macro

It is not possible to have a pseudoinstruction specifying a segment in a macro definition.

DS must be in DSEG

A DS pseudoinstruction can only appear in a data segment.

DSEG conflicts with WORLD EXTERNAL_DATA

It is not possible to have WORLD EXTERNAL_DATA and a pseudoinstruction specifying a segment in the same source file.

DSEG isn't allowed in macro

It is not possible to have a pseudoinstruction specifying a segment in a macro definition.

ELSE without IFxxx

The conditional assembly IFxxx pseudoinstruction corresponding to an ELSE is missing.

ENDF not seen

The ENDF pseudoinstruction ending a function definition is missing.

ENDIF without IFxxx

The conditional assembly IFxxx pseudoinstruction corresponding to an ENDIF is missing.

ENDM not seen

The ENDM pseudoinstruction ending a macro definition is missing.

external symbol can't be public

An external symbol has been declared as public_

Hardware configuration violation

An instruction (such as CHANGE) corresponds to a function not implemented on the specified chip.

identifier expected

Something other than an identifier was encountered in a macro definition formal parameter list or EXTERN operand.

illegal character in ??? constant

A numerical constant includes a character illegal for the specified base.

illegal character in binary constant

A numerical constant includes a character illegal for the specified base.

illegal symbol type

A symbol type is improper for a PUBLIC declaration.

illegal word in external list

A syntax error occurred in an EXTERN operand.

instructions can't be in DSEG

Instructions other than DS cannot be included in a data segment.

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

A JMP or CALL instruction was encountered at an address with the bottom 12 bits equal to 0FFEH or 0FFFH. Since the segment alignment mode is "INBLOCK" the linker will give errors.

Jump address beyond zero

The destination address of a jump is below zero (the start address of the current code segment).

Jump address exceeds limits

The destination address of a jump exceeds the ROM capacity.

Jump address is out of range (INBLOCK)

The destination address of a jump is outside a memory boundary. Since the segment alignment mode is "INBLOCK", the linker will give errors.

local symbol can't be public

A local symbol has been given a public declaration.

lost SET symbol

a symbol defined in a SET pseudoinstruction could not be found in pass 2. This may be an assembler internal error.

macro can't be public

A macro has been given a public declaration.

maximum nesting of macro is 10

Macros cannot be nested more than 10 deep.

Multiple WORLD specified

More than one WORLD pseudoinstruction appears in the same source file.

name required for macro

The name is missing from a macro definition.

no room for source line attribute object

There is insufficient memory to store source line information (for debugging).

no value for EXT

Even though a CHANGE instruction is used, a register EXT for an SFR is missing.

not the symbol defined by SET

An attempt was made to reset a value using SET, even though the symbol was not defined with SET.

operand exceeds limits

The repeat count in an REPT macro pseudoinstruction is not in the range 1 to 65535.

ORG isn't allowed in macro

The ORG pseudoinstruction may not appear in a macro.

other-side symbol isn't allowed

A symbol declared with OTHER_SIDE_SYMBOL cannot be specified here.

other-side symbol isn't allowed here

A symbol declared with OTHER_SIDE_SYMBOL cannot be specified here.

other-side symbol or absolute constant is required

A symbol declared with OTHER_SIDE_SYMBOL or constant is required.

positive value required

A negative value cannot be used.

public ??? not defined

A symbol declared as PUBLIC is not defined. This error is at the warning level when a PUBLIC symbol is neither given a value nor referenced, and at the (non-fatal) error level when the symbol is not given a value but is referenced.

string is too long

The length of a character string constant exceeds the limit (255 characters).

symbol name required

There is no symbol on the left-hand side of a SET or EQU.

symbol not defined

In pass 2, a symbol specified as a PUBLIC, EXTERN, or OTHER_SIDE_SYMBOL operand is missing. This may be an assembler internal error.

syntax error

A syntax error was detected.

syntax error near ???

A syntax error was detected near ???.

too complexed expression for an operand

An expression for an operand is too complex to be parsed.

too many CHIP pseudo operation

There is more than one CHIP pseudoinstruction in a single source file.

too nested if-statements

The nesting level of conditional assembly pseudoinstructions exceeds 10.

unbalanced conditional assembling controllers

The source file ended while skipping in a conditional assembly.

unbalanced IF statement

The source file ended while skipping in a conditional assembly.

unexpected end of file in string

The source file ended while reading a character string constant.

unexpected end of line in string

A line end was encountered while reading a character string constant.

unexpected EOF in conditional assembling

The source file ended while skipping in a conditional assembly.

unexpected terminator ??? in conditional assembling

The parser produced a fault while skipping in a conditional assembly. This may be an assembler internal error.

unmatched ELSE in skipping

The source file ended while skipping in a conditional assembly.

unmatched ENDIF

The source file ended while skipping in a conditional assembly.

WORLD conflicts xSEG

It is not possible to have WORLD EXTERNAL_DATA and a pseudoinstruction specifying a segment in the same source file.

6.3 Fatal Errors

The following are the fatal error messages which can be detected by M86K. In the list below, question marks indicate portions which may vary.

???(???): 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

There is a syntax error in the reserved word file.

???: illegal file name

The specified file name contains an illegal character.

???: no such chip in the table

The chip name specified by the environment variable CHIPNAME was not found in the reserved word file.

???: no such user

The user name specified by ~user is missing.

???: open error

An attempt to open the specified file failed.

???: unknown flag

An unknown assembler option was specified.

???: unreadable

The specified file cannot be read.

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

The EMS driver version is too old to support the application. Version 3.2 or later is required.

EMS allocation (??? pages) was failed

An attempt to allocate EMS memory failed.

EMS deallocation was failed

An error was detected when deallocating EMS memory.

flushing error in workfile

An error was detected when flushing a workfile (no disk space left, or similar errors).

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

An error was detected during the initialization of EMS memory, such as while checking the EMS driver version.

making temp. name for ??? failed

An error was detected while giving the output file a temporary name.

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

Neither the CHIP pseudoinstruction nor the environment variable CHIPNAME specify a chip, so assemble cannot continue.

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

Although there is a region which must be allocated to main memory, there is no more main memory left.

no more memory (???)

There is no dynamically allocatable memory (main memory, EMS memory, or work file).

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

The working space used for parsing expressions is used up.

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

The working space used for macro definitions and calling argument list processing is used up.

no reserved word file available.

An attempt to read in the reserved word file failed.

no room for file: ???

A file cannot be written because the disk is full.

Pxxxx must be less than 65536

The parameter buffer size must be specified to be 65535 or less.

read error in workfile (???)

An error occurred when reading a workfile.

removing ??? failed

After an error was detected, an attempt to delete a partially created file failed.

renaming ??? ==_ ??? failed

An error occurred at the point of attempting to rename a file created with a temporary name with its real name. This can occur if a file with the new name already exists, and is read-only.

too many file names

Five or more file names are specified on the command.

too many nested include files

The nesting level of include files exceeds 10.

unlinking work file is failed

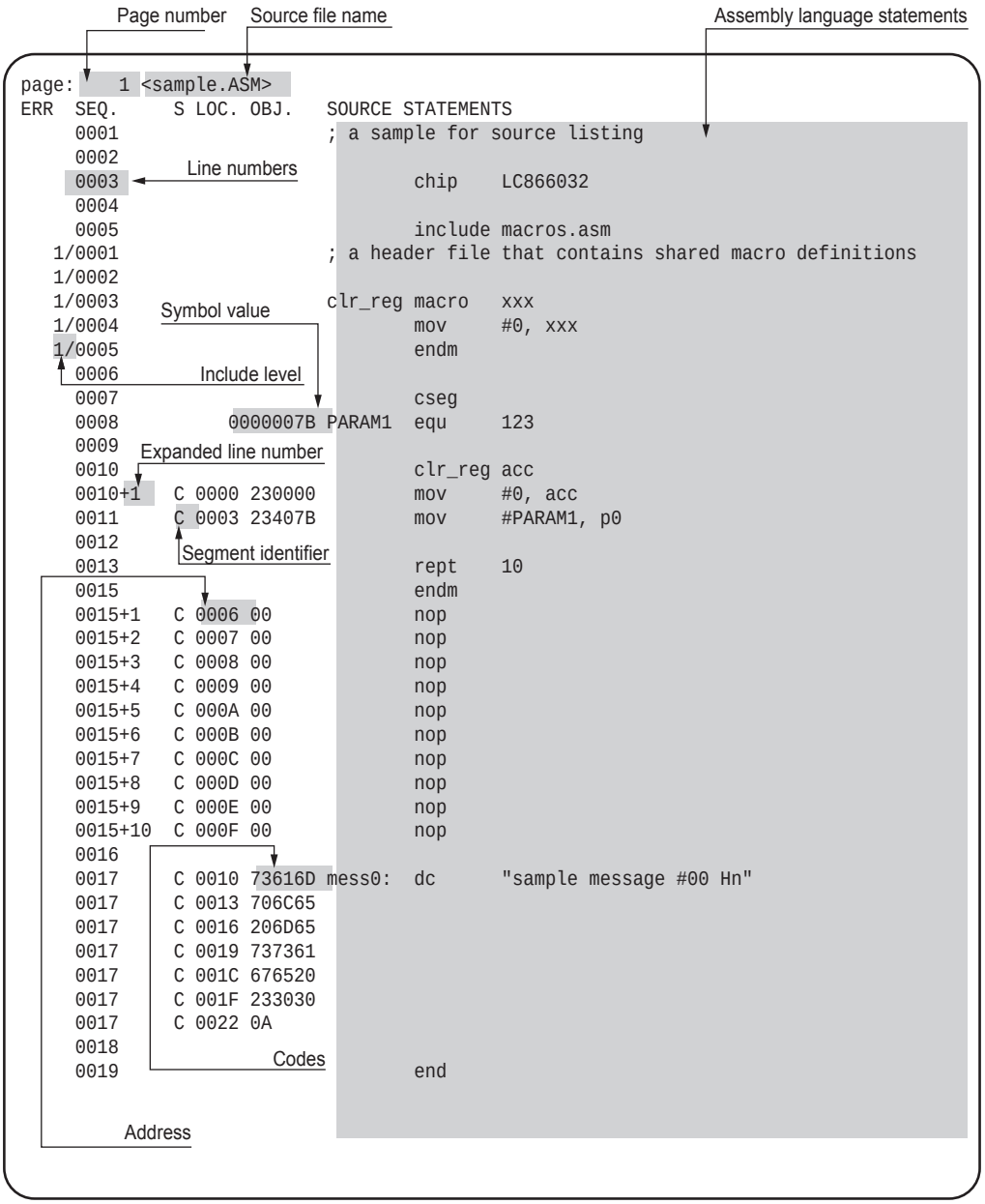
An error was detected attempting to delete a workfile.

workfile ???: already exist
workfile ???: open error

An error was detected creating a new workfile.

Chapter 7 Listing Format

The format of the listing file created by M86K is shown below. This basically reproduces the content of the source file, with line numbers and machine code on the left. The arrangement of rows and columns is also designed to fit the page size. That is, a page is 60 lines (including header), and each line consists of 132 character positions (source lines exceeding this limit are folded), and the left part has fixed character columns. Tab characters in the source are converted to spaces, to preserve layout.



Header

This appears at the top of each page, and includes a blank margin, the page number, source file name, and headings for the columns of the listing.

Page number

The pages are numbered from 1.

Source file name

This is the name of the source file specified for assembly. If the specification includes a drive name or path, this is also shown.

Assembly language statements

This shows the text of the source file, macro expansions (when their output is not suppressed), and include files.

Line numbers

These are line numbers (in decimal) in the source file. When the code section occupies more than one line in the listing the source file line numbers are repeated.

Include level

This shows the nesting level of include files. Lines from the source file itself have no include depth shown. Then the contents of an included file are shown as depth 1, and nested includes as 2, and so forth. The slash separates the include level from the line number.

Symbol value

When a symbol is given a value, if that value is explicit at assemble time it is shown here, as an 8-digit hexadecimal value. If the value is not known, it is not shown.

Expanded line number

This indicates a line not present in the source file, which has been produced by a macro expansion. The extra lines are numbered from 1 for each macro expansion.

Segment identifier

This character indicates whether the corresponding line is assembled as CSEG or DSEG code. Uppercase 'C' indicates CSEG INBLOCK, lowercase 'c' indicates CSEG FREE, and D indicates DSEG.

Address

When a line generates CSEG or DSEG code, this shows the address of the first byte as a four-digit hexadecimal value. This address is an offset from the beginning of the segment.

Codes

When code to be written to ROM is generated by assembling a source line, it is shown here, as two hexadecimal digits for each byte. A maximum of three bytes are shown on each line, in order of increasing address. When more than three bytes are assembled, they are shown on subsequent lines with the same source line number.

Dreamcast™

PART 3

Linkage Loader

This section lists the command line options and error messages for the L86K.EXE linker.

Chapter 8 Specifying Files for Linking

There are two ways of specifying the parameters when starting L86K.

(1) Specifying all parameters on the L86K command line

(2) Specifying parameters by responding to the prompts displayed by L86K

To force an exit from L86K, press the following key combinations.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Stop

8.1 Specifying File Names

When specifying file names to L86K, either in the command line, or in response to the prompts, case is ignored. For example, the following three file names all refer to the same file.

```
sample.obj
SAmplE.OBJ
SAMPLE.OBJ
```

If a file name is specified without an extension, L86K supplies the following default extensions.

File format	Default extension
Object file	.OBJ
Executable file	.EVA
Library file	.LIB
Option file	.OPT
Font file	.CGR
Flash memory data file	.Hnn

The number *nn* is that specified by the -B option.

8.2 Specifying Parameters on the Command Line

```
L86K_options_objectfiles_,_evafile__,_libraryfile___;_↵
```

options

Specify linkage loader options as listed in Chapter 9, "*Option* Switches." If options are specified they must come before other parameters.

objectfiles

Specify the object files to be linked, the linking start address, and library file names. At least one file name is required. When specifying multiple files, separate them by space characters. If the file names do not fit into one line, add a plus sign (+) to the end of the line to indicate a continuation. If the object file extension is omitted, the default extension .OBJ is added. If a file extension is specified, it takes precedence. In either case, a drive name and full path specification is possible. If .LIB is specified as the file name extension, the library is linked as well.

evafile

Specify a file name for execution on (downloading to) EVA86K. If no name is specified, the first file name specified in *objectfiles* is used, with the file name extension changed to .EVA.

Caution	EVA86K is a device for emulating the LC86K series on the computer. For Visual Memory, the EVA86K data is converted to HEX format, and run on the Visual Memory simulator.

libraryfile

Specify a library name. If no library is required, the specification can be omitted.

Caution	If a DUMMY.OBJ file is attached to the Visual Memory SDK, link this file in as well.

Example

```
C> L86K MAIN SUB0 SUB1,TEST,TEST.LIB ↵
```

This links the object modules MAIN.OBJ, SUB0.OBJ, SUB1.OBJ, MAIN.OPT, and MAIN.CGR. If there are any symbols remaining undefined after linking MAIN.OBJ”“SUB0.OBJ, and SUB1.OBJ, the linker searches TEST.LIB for these symbols, and links the appropriate modules.

8.3 Specifying Parameters at the Prompts

Enter the following command to start the linkage loader without specifying any file names.

```
L86K_options_↵
```

L86K produces the following prompts, one line at a time. Enter the parameters as required.

```
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 waits for a response before displaying the next prompt. Section 3.1, "Specifying File Name" describes the conventions for responding to the prompts.

The responses to the L86K command correspond to the parameters on the L86K command line, except for Option filename and Font filename. These are described below.

Reference	For details of the L86K command line, see Section 8.2, "Specifying Parameters on the Command Line."

Prompt	Command line argument
Object modules	*objectfiles*
EVA filename	*evafile*
Libraries	*libraryfi* les

When using the L86K prompts, enter the above four items, then the following two items.

Option filename

Enter the name of the option file for the chip for which the EVA file will be created.

Font filename

Enter the font file name.

If the last character typed on the response line is a plus sign (+), the prompt returns another line, so the input can be continued. In this case the plus sign must come at the end of a complete file or library name, path name, or drive name.

Caution	This file specification is not necessary for Visual Memory. This file specification is not necessary for Visual Memory.

Default Responses

To select the default response to the current prompt, press the Enter key, without entering a file name. The next prompt appears.

To select the default response to the current prompt and all of the remaining prompts, enter a semicolon (;) and press the Enter key. This disables input to the rest of the prompts for this linking session, and saves time when using the default settings. Since, however, there is no default for the Object modules prompt, it is not possible to enter a semicolon for this prompt.

The following table shows the defaults for the L86K prompts.

Prompt	Default
EVA filename	The name of the first file in the response to "Object modules," with the file extension changed from .OBJ to .EVA.
Libraries	No libraries are searched.
Option filename	The file name specified in response to the "EVA filename" prompt, with the file extension changed from .EVA to .OPT.
Font filename	The file name specified in response to the "Option filename" prompt, with the file extension changed from .OPT to .CGR.

8.4 Files Referenced During Linking

L86K always references the following files during linking.

Caution	For Visual Memory, the system BIOS used for reading and writing flash memory is in ROM, and therefore the following do not necessarily apply.

```
LC86K.LIB
```

System information is stored in LC86K.LIB. At linking time, L86K gets system information for the target CPU from LC86K.LIB, and stores it in the EVA file.

It also references the following option file during linking.

```
LCnn00.OPT  ...  nn is a two-digit value corresponding to the device
type
```

LC86K.LIB and LIBnn00.OPT must be present in the same directory as L86K.EXE or in a directory to be found in the PATH environment variable.

Chapter 9 Option Switches

This chapter describes how to use the linkage loader options to control the operation of L86K. The options are all introduced with the linkage loader option character, which is slash or minus.

-B=bank number

Creating a flash memory HEX file for the LC86800 series

The -B option specifies the bank number for LC86800 series flash memory (WORLD EXTERNAL_DATA). The bank number is written as a hexadecimal value from 1 to FF. Here the bank number is the data file extension.

Caution	Only bank 0 of flash memory is available to applications in Visual Memory. Do not specify this option.

Example

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

In this case the data file created is SAMPLE.H01.

-C=address

Specifying a CSEG loading address

The -C option applies to the immediately following object module, and specifies a code segment loading address. The address is written in hexadecimal.

If this option is omitted, L86K may load the object module code segment at any address.

-D=address

Specifying a DSEG loading address

The -D option applies to the immediately following object module, and specifies a data segment loading address. The address is written in hexadecimal.

If this option is omitted, L86K may load the object module data segment at any address.

-E
Allowing DSEG address overlaps

If the -E option, more than one symbol can be defined in data segments at the same address, without causing an error.

-I
Do not distinguish case

By default, L86K treats lowercase and uppercase letters as distinct. If the -I option is specified, case is ignored.

-P
Create a loading map

Specifying the -P option creates a file containing a listing of the linker mappings (the linking state of each segment, and public symbol locations). This map file has the file name specified on the command line or in the prompts for the EVA file, with the extension changed to .MAP. If, however, a fatal error prevents linking from continuing, the map file is not created.

Note	A map file will be necessary when using the reverse assembly function of the Visual Memory simulator to display address labels.

-L
Create a local symbol listing

The -L option is only valid in combination with the -P option, and adds a listing of local symbols for each module in the map file.

Note	A map file will be necessary when using the reverse assembly function of the Visual Memory simulator to display address labels.

```
   LC86K series Linkage map list                                  PAGE    1
                                  Linkage date: Wed Dec 09 13:28:02 1992


sample1.EVA loading map
                                                                  CSEG-I means segment attribute is INBLOCK; CSEG-F means FREE
                                                                  Start address of segment block
Module name                                                       Segment block size (bytes)
  sample1   CSEG-I          0000            0004  0004
  sample2   CSEG-F       1  0004            0005  0009
  sample2   CSEG-F       2  0009            0003  000C
                                                                  End address of
                                                                  segment block + 1
                                                                  Sequence numbers of segment blocks in order within a module,
                                                                  when there is more than one
public symbol list
  symbol        segment  address
  ACC           DSEG     0:0100    [  EQU  ]
  ADCR          DSEG     0:0160    [  EQU  ]
  ADRR          DSEG     0:0161    [  EQU        ]
          .                .
          .                .
          .                .
          .                .
                                                                  Outputs bank number and address for DSEG symbols
local symbol list
  symbol        segment   address
** sample1  **
  l1            CSEG        0000

                                                                  Output for each module when there are local symbols
** sample2  **
  l2            CSEG        0004
```

-W

Issue warning messages for operand values

When the -W option is specified, operands of JMP instructions and so forth are checked to be in range (for a JMP, the low 12 bits of the operand are allowed, and the value must thus be in the range 0 to 4095). If not, a warning message is issued (only for the number of valid bits stored in the instruction code; overflow bits are discarded).

-A, -F, -O, -R

Optimizing loading of CSEG FREE blocks

Normally L86K links the code segments in the order in the object modules and following the order of specification on the command line, and at this point L86K aligns the executable file segment data (code segments) on 4096-byte boundaries. For this reason, gap may appear between code segments.

To minimize this wasted space, there are four types of optimization of segment block positioning to use memory more efficiently.

The following are the loading methods.

-A option

With the -A option all code segment blocks to be linked are loaded in decreasing order of size. If a segment block that crosses a 4096-byte boundary has the INBLOCK attribute, it is aligned on a boundary; if it has the FREE attribute it is not repositioned.

-F option

The -F option is only valid in combination with -A. After code segment blocks with the INBLOCK attribute are linked in the order specified on the command line, code segment blocks with the FREE attribute are located in gaps occurring as described above (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

-O option

The -O option is only valid in combination with -A. After first linking code segment blocks with the INBLOCK attribute in decreasing order of size, code segment blocks with the FREE attribute are located in empty regions (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

-R option

The -R option is only valid in combination with -A. After first linking code segment blocks with the INBLOCK attribute in decreasing order of size, code segment blocks with the FREE attribute are located in empty regions. At this stage, if two consecutive 4096-byte regions include empty space, the later code segment block with the INBLOCK attribute is repositioned at the end of the region, and a code segment with the FREE attribute is located in the empty region (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

-S

Specify symbol sorting

If the number of public symbol definitions in the linked object files (including SFR definitions in the file LC86K.LIB) exceeds 8192, or the number of local symbol definitions in the linked object files exceeds 8192, then because of the drop in processing speed for symbol sorting, the following messages appear, showing the progress of the sorting.

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

However, if the -S option is specified, in this case symbol sorting is not carried out and linking is abandoned, with the following error message.

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

In this case the number of symbols shown is the excess over 8192.

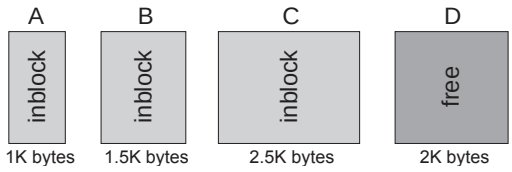
Chapter 10 Object Alignment

As described under "Optimizing loading of CSEG FREE blocks" in Chapter 9, "Option Switches" (the -A, -F, -O, and -R options), when optimization is specified for linking, objects are aligned differently from the normal case. The are four types of optimization, and the corresponding object alignment for each of the types is described below.

10.1 -A option

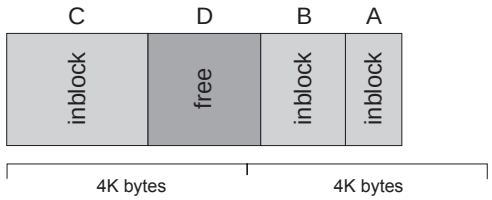
There are two alignment specifications for code segments: CSEG INBLOCK (aligned within a 4096-byte block) and CSEG FREE (align regardless of 4096-byte boundaries). When the -A option is specified segments are placed in the best positions in order of decreasing size, taking the INBLOCK and FREE segments together (the INBLOCK segment are located within 4096-byte blocks, but the FREE segments are aligned freely).

Consider, for example, the following objects:

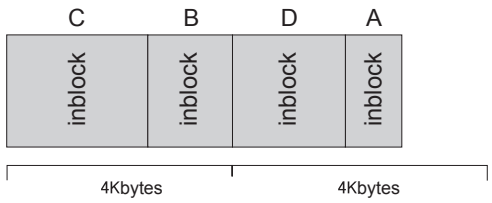


If the -A option is specified for linking, then the segments with INBLOCK/FREE specifications are located in decreasing order of size as follows.

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



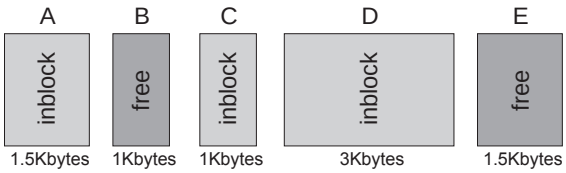
In this example, the segment straddling a 4096-byte boundary is object D, which has the FREE attribute, thus not requiring alignment. If object D has the INBLOCK attribute, the following is the result.



10.2 -A -F options

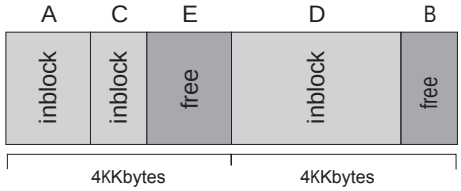
When -A-F is specified, the segments with the INBLOCK attribute are located in their order on the command line, then segments with the FREE attribute are located in decreasing order of size, in the best available positions.

Consider the following set of objects.



When these are linked with -A-F specified, A, C, and D are located in the command line order (with D aligned to a 4096-byte boundary), then E and B in sequence in the best available positions.

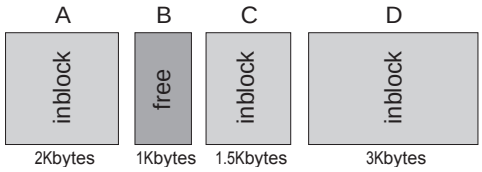
```
L86K -A-F A B C D E;
```



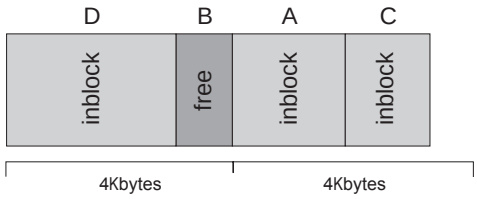
10.3 -A -O options

When -A-O is specified, the segments with the INBLOCK attribute are located in decreasing order of size, then segments with the FREE attribute are located in decreasing order of size, in the best available positions.

Consider the following set of objects.



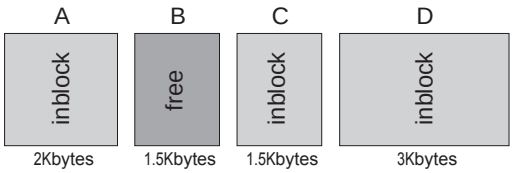
When these are linked with -A-O specified, D, A, and C are located in the best available positions in order of decreasing size, then B is located in the best available position.



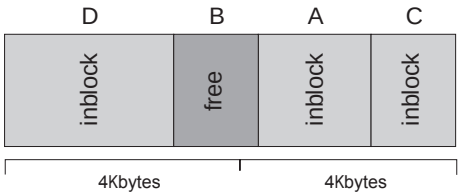
10.4 -A -R options

When -A-R is specified, the segments with the INBLOCK attribute are located in decreasing order of size, then if two consecutive 4096-byte regions include empty space, the segments in the second of the two regions are realigned to bring the empty space together, and segments with the FREE attribute are inserted in this space. alignment the later code segment block with the INBLOCK attribute is repositioned at the end of the region, and a code segment with the FREE attribute is located in the empty region (if there are no regions to locate code segment blocks with the FREE attribute, they are located from the last address, in decreasing order of size).

Consider the following set of objects.



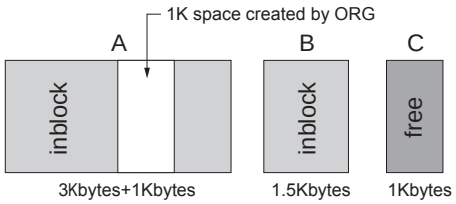
When these are linked with -A-R specified, after locating D, A, and C, segments A and C are realigned with the end boundary of their 4096-byte block, to open up a space, in which B can be located.



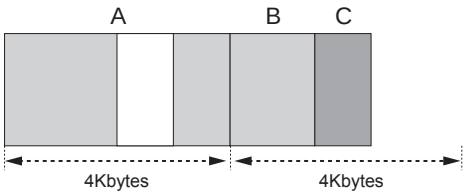
In all optimization operations, if an object module includes a space created by use of an ORG pseudoinstruction, this space is made available for optimized segment alignment.

The following is an example of this.

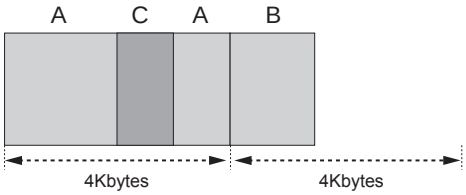
Consider the following set of objects.



If these are linked without optimization the following is the result.



When these are linked with optimization, the space created within segment A by the ORG pseudoinstruction is filled as follows.



If a loading address (-C option) is specified together with the optimization specification, then if the first segment of the file has a FREE attribute, the loading address specification is followed for this segment only (subsequent FREE blocks are optimized).

Chapter 11 Errors

11.1 Fatal Errors

When a fatal error is detected during linking, L86K displays a message, and aborts processing. The following are the L86K error messages.

Chip name unmatched

An attempt was made to link object modules for different chips.

Data file specified

A data file was specified for EVA file creation.

Data segment size exceeds

An attempt was made to link a DSEG object exceeding the RAM size.

External undefine symbol

An external symbol is undefined.

Illegal bank number specifed

The specified flash memory bank number is incorrect.

Illegal file format

The specified file is not for the LC86K series.

Illegal option specified

An illegal option was specified.

Internal module not specified

In linking an internal program file was not specified.

Loading address multiple assignment

More than one object has been allocated to the same address.

No such file or directory

The specified file does not exist.

Program file specified

A program file was specified when creating a flash memory data file (bank 1).

Public symbol multiple define

A public symbol is multiply defined._

Segment size exceeds

An attempt was made to link when exceeding the segment size.

WORLD attribute unmatched

A program file with the WORLD INTERNAL or WORLD EXTERNAL attribute and a data file with the WORLD EXTERNAL_DATA attribute are combined.

11.2 Non-Fatal Errors

If a non-fatal error is detected during linking, a message appears, but linking continues. L86K produces the following warning messages.

Cannot access file: LC86K.LIB

The library file LC86K.LIB containing the reserved words does not exist. LC86K.LIB contains the reserved words for each chip, and must be present in the current directory or a directory specified in the PATH environment variable.

Module not in library

A reserved word for the target chip is not present in the file LC86K.LIB.

Operand data overflow

The value specified in the operand field exceeds the designated range. (Range varies according to statement.)

Operand data type mismatch

An illegal segment symbol was specified in the operand field.

Dreamcast™

PART 4

Library Manager

This section lists the command line options and error messages for the LIB86K.EXE library manager.

Chapter 12 Starting the Program

There are two ways of starting LIB86K and specifying the parameters.

(1) Specifying all parameters on the LIB86K command line

(2) Specifying parameters by responding to the prompts displayed by LIB86K

To force an exit from M86K, press the following key combinations.

Computer type	Keys
PC/AT compatible	Ctrl + C or Ctrl + Pause/Break
NEC PC-9800 series	CTRL + C or Stop

12.1 Specifying File Names

When specifying file names to LIB86K, either in the command line, or in response to the prompts, case is ignored. For example, the following file names all refer to the same file.

```
sample.obj
SAmplE.OBJ
SAMPLE.OBJ
```

If a file name is specified without an extension, LIB86K supplies the following default extensions.

File format	Default extension
Library file	.LIB
Object file	.OBJ
Listing file	None

12.2 Specifying Parameters on the Command Line

```
LIB86K_option_ oldlibrary_commands__,_listfile__,_newlibrary___;_↵
```

option

The only *option* which can be specified is */?*.

oldlibrary

Specify the library file to be processed. This parameter cannot be omitted. If the library file extension is .LIB, it can be omitted. However, if the user's library file extension is other than .LIB, it cannot be omitted. There is no default for the library file, so if this parameter is missing an error message results. If the specified library file does not exist, the following prompt appears.

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

To create a new library, enter "Y". If any character other than "Y" is entered, the library manager terminates.

Entering an existing library file with just a semicolon carries out a consistency check on the library. This checks whether all of the modules within the library can be used. If an error is found, an error message is produced.

commands

The commands parameter includes symbols such as +, -, -+, *, and -*, which are used to control the program operation. One object file name or module name can be specified for each command, to carry out a number of operations. If the commands are omitted, no changes are made to the library file.

Command	Meaning
+	Add a module. The module in the object file specified following the command is added at the end of *oldlibrary*. This command cannot be used to merge libraries.
-	Delete a module. The module specified following the command is deleted from *oldlibrary*.
-+	Replace a module. The module in the object file specified following the command is added at the end of *oldlibrary*, and the existing module of the same name is deleted.
*	Copy a module. A search is made in *oldlibrary* for the module specified following the command, and this is written to an object file of the same name. The copied module is left in *oldlibrary*.
-*	Move a module. A search is made in *oldlibrary* for the module specified following the command, and this is written to an object file of the same name, but the module is deleted from *oldlibrary*.

listfile

In *listfile*, specify a file for output of a list of the public symbols, external reference symbols, and module names in the library. If omitted the listing is sent to standard output.

newlibrary

The *newlibrary* parameter specifies an output library name. If this is omitted, the existing version of the *oldlibrary* file has the extension changed to .BAK, and the *new library* is written to the file *oldlibrary*.

12.2.1 Option

Specify the */?* option to display a help message.

12.2.2 Examples of Command Line Execution

Example 1

```
LIB86K HOME-+ROM;↵
```

This example deletes the module ROM from the library HOME, and adds the object file ROM.OBJ at the end of the library.

Example 2

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

In the first version, the module ROM is deleted from the library HOME, and then the object file ROM.OBJ is added at the end of the library. In the second version, the ROM.OBJ object file is added to the library HOME first, and then the ROM module is deleted. Therefore, in the first version, the ROM module remains in the library, but in the second version it does not. This is because the command symbols are executed in the order they are specified.

Example 3

```
LIB86K HOME,LCROSS.PUB ↵
```

After carrying out a consistency check on HOME.LIB, a cross-reference listing is written to the file LCROSS.PUB.

Example 4

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

The module STUFF is extracted from the library FIRST.LIB and written to the file STUFF.OBJ, and then the module STUFF is deleted from the library. The module

MORE is written from the library to the file MORE.OBJ, but remains in the library. The rewritten library is named SECOND.LIB, and corresponds to FIRST.LIB, with the STUFF module deleted.

12.3 Operation with the Prompts

Enter the following command to start LIB86K without specifying any parameters. Then follow the prompts from the assembler to enter the parameters.

```
LIB86K_option_↵
```

LIB86K displays the following prompts, one at a time.

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

After displaying each prompt, LIB86K waits for user input. After user input, it displays the next prompt, and waits again.

The responses to the prompts correspond to the parameters entered on the command line, as shown in the following table.

Prompt	Corresponding command line parameter
Library name	Corresponds to the *oldlibrary* parameter. If the library name is followed by a semicolon, LIB86K carries out a consistency check.
Operations	Corresponds to the *commands* parameter.
List file	Corresponds to the *listfile* parameter.
Output library	Corresponds to the *newlibrary* parameter.

12.3.1 Prompt Line Extension

In response to the Operations prompt, entering an ampersand (&) at the end of the line produces another Operations prompt, so that the specification can be continued.

12.3.2 Default Responses

Except for the Library name prompt, there are default values, which are used when the response to the prompt is a semicolon or the Enter key. The following table shows the prompt default values.

Prompt	Default value
Operations	Make no changes to the library file.
List file	Select standard output for the listing. No list file is created.
Output library	The output library name is the same as the original library name.

Chapter 13 Error Messages

This chapter lists error message and their meanings.

cannot access file

LIB86K cannot open the specified file.

cannot create new library

The disk or root directory is full, or the library file already exists and is read-only.

cannot rename old library

The .BAK version is read-only, and LIB86K cannot rename the old library with the .BAK extension.

comma or newline missing

A comma or newline is missing on the command line.

error reading from library

LIB86K cannot read data from the specified library file.

error writing to new library

The disk or root directory is full

insufficient memory

There is insufficient memory for LIB86K to run.

interrupted by user

LIB86K execution was interrupted by the user.

invalid library header

The input library file is in an invalid format.

module not in library_ ignored

The specified module to be replaced was not found in the library.

output-library specification ignored

In the case of a new library name, an output library was also specified.

syntax error : illegal file specification

A command operator such as a minus sign (-) is not followed by a module name.

Chapter 14 Cross-Reference

This chapter describes the cross-reference listing format.

```
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
```

Dreamcast™

PART 5

EVA to HEX

This section describes the program E2H86K.EXE, which converts the EVA format file output by the linker to a HEX format file to be read into Visual Memory or the Visual Memory simulator, including command line options and error messages.

For a file to be transferred to Visual Memory, first create the HEX file, then use H2BIN.EXE to convert to a BIN format file.

Chapter 15 Starting the Program

Using E2H86K to convert an EVA format file into a HEX format file produces files with the extensions .HEX and .H00.

The file with the extension .HEX is a 64K-byte flash memory file in HEX format. The region not used for the program is filled with zeros. This file is not normally required.

The file with the extension .H00 is a HEX format file including only the program itself. This file can be read into the Visual Memory simulator, or converted to a BIN format file with H2BIN.EXE, for reading into a Visual Memory device.

15.1 Specifying File Names

When specifying file names to E2H86K, case is ignored. For example, the following three file names all refer to the **same** file.

```
sample.eva
SAmplE.EVA
SAMPLE.EVA
```

If a file name is specified without an extension, E2H86K supplies the following default extensions.

File format	Default extension
EVA file	.EVA
HEX file	.HEX

15.2 Specifying Parameters

```
E2H86K_option_ EVA_filename _HEX_filename_↵
```

option

Specify the option as described in Section 15.3, "*Option* Specification." This must immediately follow the command name.

EVA_filename

Specify the debugged file (with the extension .EVA). This is referred to as the EVA file.

HEX_filename

Specify the name of an Intellec HEX format file. If HEX_filename is omitted, the same file name as the *EVA_filename* is used. When a flash memory data file is converted the extension is .H00.

Caution	There is no prompt option.

Example 1

```
C>E2H86K PROG012 ↵
```

EVA file _____HEX file, _flash memory HEX file

PROG012.EVA___PROG012.HEX, PROG012.H00

Example 2

```
C>E2H86K ↵
```

This displays the following help message.

```
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 Option Specification

The option must be introduced with a slash (/).

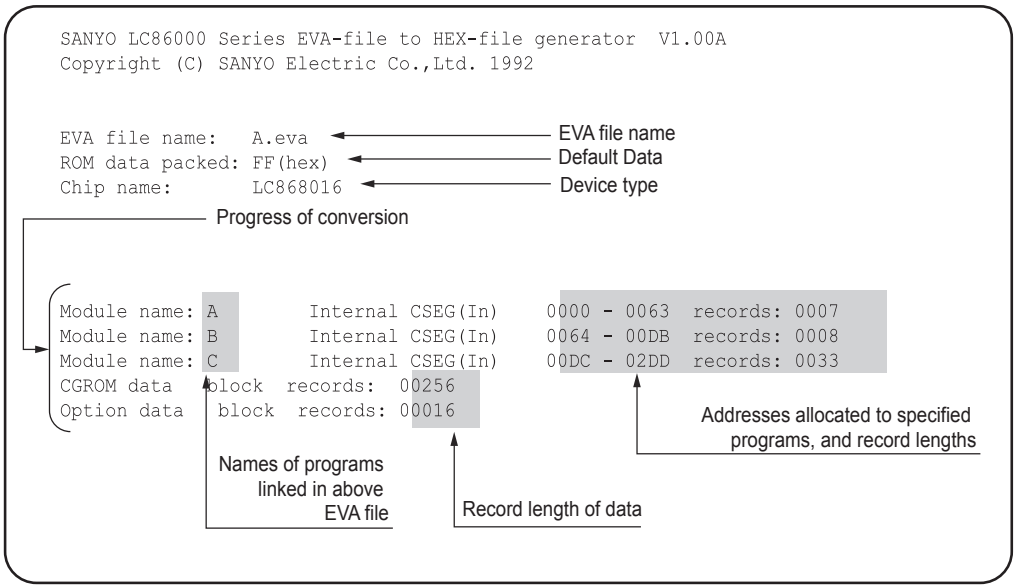
Caution	It is not possible to use a minus sign (-).

/I

Suppress information display during conversion

The /I option suppresses the display of information during conversion of the EVA file to the HEX file. If this option is not specified, the progress of the conversion is indicated.

Example display



Chapter 16 Error Messages

16.1 Fatal Errors

If E2H86K detects a fatal error during operation, it displays one of the messages listed below, and terminates.

```
Error message: Fatal error : ... message ...
```

‘filename' File not close.

The file 'filename' cannot be closed.

‘filename' File not create.

The file 'filename' cannot be created.

‘filename' File not open.

The file 'filename' cannot be found.

‘filename' not EVA file format.

The file 'filename' is not in EVA format.

‘filename' user disk full.

The disk became full while writing the file 'filename.'

Chipname undefined.

The chip name in the EVA file is not known.

ROM size over. (ROM size: XXXX)

The program size exceeds the ROM size.

Tablename allocation error.

There is insufficient memory to reserve space for tablename.

Dreamcast™

PART 6

HEX to BIN

This section describes the program H2BIN.EXE, which converts a HEX format file to be read into Visual Memory or the Visual Memory simulator, including command line options and error messages.

It is first necessary, using E2H86K.EXE, to convert an EVA format file to HEX format. For the Visual Memory simulator, on the other hand, a HEX format file can be used.

Chapter 17 Starting the Program

Using H2BIN, a BIN format file for reading into the Visual Memory device can be created. The BIN format file can be transferred to Visual Memory using a development computer, Dev. Box, and the Memory Card Utility special-purpose transfer utility.

Alternatively, using the Shinobi library backup functions, the BIN format file can be transferred to Visual Memory.

Reference	For the method of transfer using a development computer, Dev. Box, and the special-purpose transfer utility, see "Visual Memory Tutorial." For the method of transfer using the Shinobi library, refer to the buSaveExecFile() function under "Backup Functions" in the SEGA library manual, Vol. 2.

17.1 Specifying File Names

When specifying file names to H2BIN, case is ignored. For example, the following three file names all refer to the **same** file.

```
sample.H00
SAmplE.H00
SAMPLE.H00
```

17.2 Specifying Parameters

H2BIN ***_HEX_filename_ _BIN_filename_***↵

Note	H2BIN has no command line options.

HEX_filename

Specify the file created with E2H86K with the extension .H00.

Starting H2BIN without specifying a file name produces a brief help message.

Note	This file must be in the Intellec HEX format.

BIN_filename

Specify the name for the BIN format file. If *BIN_filename* is omitted, the same file name as the *HEX_filename* is used, with the extension .BIN.

Caution	**H2BIN** does not run in the full screen mode of **MS-DOS**. If started in the full screen mode, it forcibly switches to the window mode. This is important when using **MAKE** or a batch operation.

Example 1

```
C>H2BIN PROG012.H00 ↵
```

HEX file _____BINfile _

PROG012.H00___PROG012.BIN

Example 2

```
C>H2BIN ↵
```

The following message (brief help) appears.

```
INTELLEC HEX to binary converter Version 0.10 SEGA SYSTEM R&D
Usage:    H2BIN <HEX file name > [<binary file name>]
          HEX file name is the source file
          Binary file name is the output file (can be omitted)
Function: converts INTELLEC HEX format file to binary.
          If the output file name is omitted, it is the source file
          name with the extension changed to .BIN.
Example:  C:\VM_SDK\LC86K\H2BIN.EXE lcd_puu3.hex lcd_puu3.bin
```

Chapter 18 Error Messages

18.1 Fatal Errors

If H2BIN detects a fatal error during operation, it displays one of the messages listed below, and terminates.

file 'filename' not found.

The specified HEX format file cannot be found. Check the path and file name. Do not omit the extension .H00.

file 'filename' cannot be created.

A write to the specified BIN format file failed. Check the available disk space, and whether other applications have the file open.

Conversion failed.

The HEX format file CRC is bad. The HEX file may be corrupted. Run E2H86K.EXE to recreate the HEX format file.

Not an INTELLEC HEX format file.

The HEX format file is of a different type. Run E2H86K.EXE to recreate the HEX format file.

error: extended address code detected - not currently supported.

The HEX format file is of a different type. Run E2H86K.EXE to recreate the HEX format file.

error: unknown record type detected.

The HEX file may be corrupted. Run E2H86K.EXE to recreate the HEX format file.

Dreamcast™

PART 7

MAKE

This section describes the MAKE.EXE program maintenance utility, which checks which source files have been updated, and minimizes the work in rebuilding an application, including command line options and error messages.

Chapter 19 Overview of MAKE

MAKE automates program development. It automatically updates an executable file (EVA) when a source file (ASM), object file (OBJ), option file (OPT), CGROM file (CGR), and so on is updated.

To run MAKE requires a file, the makefile, including the necessary information. This is a text file, of the instruction used to build the program. These instructions include generation rules, macros, directives, and implicit rules. A generation rule consists of a target, and files it depends on, together with commands for building the target. MAKE compares the time stamp of the target with the time stamp of the files on which it depends, and if any of these have been updated more recently than the target, uses the specified command to rebuild the target.

19.1 Running MAKE

To start MAKE, enter the following command.

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

options

Enter any MAKE options. For details, see Section 19.1.2, "Command Line *Options*."

makefile

Specify the name of the *makefile*. Note that a space is required between '/f' and '*makefile*'. The makefile name may be omitted if it is 'MAKEFILE.'

errorfile

Specify a file for output of *errors*. Note that a space is required between '/x' and '*errorfile*'. The error output is normally to the display, but with this option can be to a file.

targets

Specify the target or *targets* to be built. If this specification is omitted, and there is no .TARGET directive, the first target in the makefile is built.

19.1.1 Build Priority Sequence

MAKE looks for the rules for a build in the following priority sequence.

1) If the /f option is specified, MAKE looks for the specified makefile in the current directory or specified directory. If the file is not found, MAKE terminates.
2) If the /f option is not specified, MAKE assumes the file name is 'MAKEFILE,' and looks in the current directory.
3) Unless the /r option is specified, MAKE looks for a makerule file in the current directory. If not found in the current directory, MAKE looks in the directory containing MAKE itself. If the makerule file cannot be found, MAKE terminates.

19.1.2 Command Line Options

The following options control the operation of MAKE. The option letters are not case-dependent, but are always preceded by a slash (/).

/E

Give priority to external macros

When referencing a macro, give priority to an external macro. The default is to give priority to an internal macro.

/I

Ignore result codes, and continue processing

This ignores the result code from a command specified in the makefile. MAKE continues processing to the end of the makefile, even if errors occur. To ignore result codes in a particular part of the makefile only, use the hyphen (-) command modifier or .IGNORE directive.

/N

Show sequence of build without executing commands

This just displays the commands which would be executed in the makefile build, without actually carrying them out. This is useful for makefile debugging, and for checking which target files should be updated.

/R

Do not read rule file

When this option is specified, MAKE will not read a makerule file. The default is to allow the makerule file to be read.

/S
Suppress command display

Do not echo commands in the makefile. To suppress command display for parts of the makefile only, use the at sign (@) command modifier or .SILENT directive.

/?
Show help

Display the MAKE command line syntax.

19.2 Makefile Syntax

The makefile is a text file: create it using a text editor. Normally the makefile is called "MAKEFILE," but if there are a number of different makefiles, they can have distinct names. The makefile contains generation rules, macros, implicit rules, and directives.

19.2.1 Generation Rules

The generation rules are the core of the makefile. They are written as follows.

```
Target  :  dependent_files
sample.eva : sample.obj sample.opt sample.cgr __dependency rule
     l86k/p sample;                           __generating command
     copy          sample.eva c:\myprog\      __generating command
```

Dependency rules

Each generation rule starts with a dependency rule. This is in two parts, separated by a colon. The item before the colon is the target, which is the file which MAKE is going to update. The items after the colon are the files on which the target is dependent (also referred to as the source files). In the above example, sample.eva is the target, depending on the files sample.obj, sample.opt, and sample.cgr. There must be no spaces or tabs at the beginning of the dependency rule. There can be more than one target or dependent file, separated by spaces.

The dependency rule means that the target must be updated if it either does not exist, or is older than any of the files on which it depends. As an exception, if there are no files to the right of the colon, the target is always updated. If there are multiple dependency rules in the makefile, by default the first is the final target, so it is best to make the dependency rule for the EVA file the first in the makefile (it is also possible to specify the target explicitly when running MAKE). For the final target, the default target file extension can be set using the .TARGET directive. If a dependency rule is long, it can be broken into a number of lines, by ending all but the last line with a backslash character.

Example of continued lines

```
sample.eva :  sample.obj \
              sample.opt \
              sample.cgr
       186k/p sample;
sample.obj :  sample.asm
       m86k sample;
```

Commands

The commands immediately follow a dependency rule. Each of the command lines must begin with a space or tab character. MAKE uses the existence or not of space or tab characters to distinguish dependency rules from commands. The commands, one on each line, are the DOS commands required to update the target. These commands can include DOS internal commands (such as dir). If a command is long, it can be broken into a number of lines, by ending all but the last line with a backslash character. The commands are passed to DOS by MAKE, so are subject to the line length restriction for DOS (maximum 127 characters).

Caution	During execution of MAKE, about 100 KB is required for MAKE itself and work areas. It is therefore possible to run out of memory when by starting make.

Command modifiers

Command modifiers provide more detailed control of command execution. The command modifiers precede the command, and more than one can be attached to a single command.

_command

Do not echo this command to the display when executing it. This does not affect output to the display by the command itself. See the related functions, the /S option switch, and .SILENT directive.

Note	The /S option suppresses command echoing for the whole makefile. The .SILENT directive switches the mode on and off through the text of the makefile.

-command

Ignore the command result code. MAKE normally terminates if the result code from a command is other than zero, but if the - modifier is used, MAKE continues regardless of the result code. See the related functions, the /I option switch, and .IGNORE directive.

Note	The /I option causes result codes to be ignored for the whole makefile. The .IGNORE directive switches the mode on and off through the text of the makefile.

Examples of command modifiers

```
sample.eva : sample.obj subr.obj sample.opt sample.cgr
 @echo Now creating sample.eva       __Echo progress
 l86k/p sample+subr;
sample.obj : sample.asm
 -m86k sample;                    __Ignore assembly errors
subr.obj : subr.asm
 -m86k subr;                      __Ignore assembly errors
```

Phantom targets

By deliberately specifying as a target a file which does not exist, it is possible to force MAKE to execute particular commands. This can be referred to as "phantom target." Obviously, when using a particular name for a phantom target, it is necessary to check that this file does not actually exist in the current directory.

Example using a phantom target

```
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\
```

In the above makefile, if the "all" target is specified to MAKE, or no target specified, the phantom target all causes bother the sampe1.eva and sample2.eva targets to be built, and also, before this, the phantom target "copy" causes sample1.exa and sample2.eva to be copied to the directory c:\old_prog.

19.2.2 Macros

A macro allows one character string appearing in the makefile to be replaced by a different character string. Its function is very similar to a "#define" preprocessor statement in C. There are two types of macro: user-defined macros, and built-in macros.

User macro definition

To define a new macro, use the following syntax.

```
macroname=string
```

Here, macroname can be any combination of alphanumeric characters and underscores, up to a maximum of 255 characters. The characters in the macro name are not case dependent, so for example MacroName and MACRONAME are regarded as the same macro. Another macro can be referenced within macroname, as long as it has already been defined in the makefile.

The right hand string specifies a character string of any length. It must be contained within a single logical line, but can be continued over line breaks by using a backslash character immediately before the line break. It is also possible to specify an empty string of zero length. In this case, when the macro is referenced, since it is replaced by an empty string, this can be used to delete a character string. If the same macro is defined more than once, the latest definition is the one which is used.

Internal macros and external macros

There are two types of user definition macro: "internal macros", which are defined and referenced in the makefile, and "external macros," which are supported by the MS-DOS shell function using environment variables. The format of the two is the same. By default the internal macros take precedence over external macros, but if the /E option is specified, external macros take precedence.

Referencing user macros

To reference a macro, enter a dollar sign followed by the macro name in parentheses. If the macro name is a single character, the parentheses can be omitted.

```
$(macroname) or $c
```

If an undefined macro is referenced, it is replaced by an empty string.

Referencing built-in macros

MAKE provides the following built-in macros for file names.

$@	Full name of current target file (including path, base, and extension)
$*	Name of current target file, excluding extension
$?	List of dependent files newer than target

Example of macro setting and reference

```
ASM = m86k                          # LC86000 series assembler
LINK = l86k/p                     # LC86000 series linker

all : sample.eva                    __uses phantom target
sample.obj :  $*.asm                __references base name of target
              $ (ASM) $*;           __references assembler command
sample.eva :  $*.obj $*.opt $*.cgr     __references target base name
              $ (LINK) $*;          __references linker command
```

19.2.3 Directives

The following directives can appear in the makefile. Each directive is written on a line with no space (or tab) characters at the start; it must not be within the body of a generation rule.

.IGNORE: {+|-}

This switches on and off the mode for ignoring the result codes from programs. When followed by a plus sign, this directive switches to the mode in which the result codes are ignored; when followed by a minus sign, it switches to the mode in which the result codes affect MAKE execution. By default, if a result code is other than zero, MAKE terminates. To ignore the result code from a single command only, use the minus sign modifier. To ignore result codes for the whole makefile, use the /I option.

.SILENT: {+|-}

This switches on and off the mode for echoing programs run from the makefile. When followed by a plus sign, this directive switches to the mode in which commands are not echoed; when followed by a minus sign, it switches to the mode in which commands are echoed. By default, commands are echoed. Suppress the echo from a single command only, use the at sign modifier. To ignore suppress echoing for the whole makefile, use the /S option.

.DEFAULT:

When a generation rule in the makefile consists of a dependency rule with no following commands, MAKE uses the implicit rules to generate commands. If there are no creation rules, a default set of commands can be supplied with the .DEFAULT directive on a line followed by the commands.

.DEFAULT

commands

.TARGET: suffixes

The final target to be built can be specified to MAKE on the command line, and if this is omitted, the target of the first generation rule in the makefile is built, but the .TARGET directive specifies an extension for the default final target, so that the extension of a file to be built can be specified for the final target. A suffix is a period followed by up to three characters. More than one suffix can be specified by separating them with spaces.

19.3 Implicit Rules

The implicit rules define how the general way to make a file of one extension from a file of another extension. MAKE follows these implicit rules to derive the commands needed to update a target, from the target dependency rules. Using implicit rules generally makes writing the makefile simpler. The implicit rules can be included in the makefile, or written in the makerule file MAKERULE.DEF.

For each of the source files in a generation rule, MAKE checks whether there is generation rule with that file as target, and if not it uses an implicit rule. The conditions for using an implicit rule are thus as follows:

1) There must be no dependency rule for the file in question.
2) There must be a generation rule for making the file.
3) The file or files required for making the file must exist.

If these conditions are met, MAKE adds generation rules as follows.

1) If a generation rule does not exist, it is added.
2) basename.sss is added as the source file (sss: source file extension)
3) The commands from the (implicit) generation rule are used as the commands for creating the target.

19.3.1 Makerule file

The makerule file MAKERULE.DEF contains the rules that MAKE uses to create implicit rules, in the following format.

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

The first line specifies two file extensions. The first, "sss," is the extension of the source file, and the second, "ttt," is the extension of the target file. The extensions are not case-dependent. The first period, before "sss," must come at the very beginning of a line. The following lines are the commands, written as in the makefile, for creating the target. For example, to create an object file "basename.obj" from an assembly language source "basename.asm," the rule is ".asm.obj."

Example makerule file

```
# ************************************************************
# ***                                                      ***
# ***  Implicit rules for EVA86000 utility make.           ***
# ***  Definition for M86K                                 ***
# ***                                                      ***
# ************************************************************
       ASM = m86k

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

.TARGET: .EVA .HEX __default final target
.DEFAULT:
       @echo --------------------------------
       @echo ??? Undefined build commands ???
       @echo --------------------------------
# end of makerule.def
```

Example makefile using a makerule file

```
ASM = m86k                          # LC86000 series  assembler
LINK = l86k/p                     # LC86000 series  linker

all : sample.eva

sample.obj : $*.asm
                                    _ assembler command omitted
sample.eva : $*.obj $*.opt $*.cgr
       $ (LINK) $*;
```

Dreamcast™

PART 8

Coding

This chapter describes assembler syntax, assembler pseudoinstructions, and the LC86K instruction set.

Chapter 20 Assembler Syntax

Each line of the source file is a character string of up to 511 characters (including the terminating CR and LF). Except for symbols defined in the source program (labels, macros, and so on), uppercase and lowercase letters are not distinguished. For example, “Nop” and “nop” are both recognized as the mnemonic for the NOP instruction. By specifying the -I assembler option, it is possible to remove the case distinctions for labels and other symbols as well.

20.1 Statements

Statements consist of the instruction mnemonics and operands which define the object code to be created by the assembly process, and comments. One line of source code corresponds to one instruction mnemonic. A statement is not allowed to be continued over more than one line. Each statement comprises the following four fields.

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

Field	Purpose
label	Identifies the location of this statement, so that it can be referenced from another statement. It must always be followed by the colon.
operation	Specifies the function of the statement.
operand	Specifies the operand (or operands) on which the function operates.
comment	This is for explanatory purposes, and does not directly affect the result of assembly.

Caution	The square brackets *[ ]* identify Items which can be omitted.

20.2 Label and Symbol Names

Label and symbol names consist of character strings of any (nonzero) length, but only the first 32 characters are used to distinguish names. The following characters can be used:

```
A to Z, a to z, 0 to 9, $, ?, _(underscore), @, . (period)
```

Label and symbol names must begin with a letter, underscore, period or '@.' If the -i assembler option is specified, uppercase and lowercase letter are regarded as the same. Note that a label must be followed by a colon.

20.3 Comments

Comments are delimited by a semicolon, and extend to the end of the line.

20.4 Operators

The following table lists the operators which can be used in M86K assembly language, and their order of precedence. For operators such as NOT whose names consist of letters, case is not distinguished, and thus “NOT” and “not” are both the same operator.

Operator	Meaning	Precedence order
NOT	One's complement logical not	1
HIGH	High order byte	
LOW	Low order byte	
*	Multiplication	2
/	Division	
MOD	Modulo (remainder)	
+	Addition	3
-	Subtraction	
SHR	Shift right	4
SHL	Shift left	
LAND	Logical AND	5
LOR	Logical (inclusive) OR	
LXOR	Logical exclusive OR	
EQ	Equals	6
NE	Not equals	
LT	Less than	
LE	Less than or equals	
GT	Greater than	
GE	Greater than or equals	

20.5 Numeric Constants

M86K allows numeric constants to be written in binary, octal, decimal, or hexadecimal. Constants can be written with an explicit indication of the radix, or base, as for example in “123H,” or with the default radix defined by the RADIX pseudoinstruction. Thus a constant “123,” for example, with no explicit radix is interpreted according to the specification of the RADIX pseudoinstruction. By default, if there is no RADIX pseudoinstruction, such numbers are taken as decimal.

However they are written, constants are handled internally by the assembler as 32-bit values. When the final result of a numeric expression is evaluated and stored as immediate data as the operand of an instruction, only the number of bits required for the operand are stored, and any more significant bits are discarded.

Notation of constants with an explicit radix

Radix	Format	Examples		
2	'%' followed by one or more digits 0, 1	`%01111011`	`%1111111`	`%0000010000000000`
	One or more digits 0, 1 followed by 'B' *	`01111011B`	`1111111B`	`0000010000000000B`
	One or more digits 0, 1 followed by '.B' *	`01111011.B`	`1111111.B`	`0000010000000000.B`
8	One or more digits 0 to 7 followed by '.O'	`273.O`	`377.O`	`2000.O`
10	One or more digits 0 to 9 followed by '.D'	`123.D`	`255.D`	`1024.D`
16	'$' followed by one or more digits 0 to 9, a to f, or A to F	`$7B`	`$FF`	`$0400`
	One or more digits 0 to 9, a to f, or A to F, followed by 'H' (must start with 0 to 9)	`7BH`	`0FFH`	`0400H`
	One or more digits 0 to 9, a to f, or A to F, followed by '.H' (must start with 0 to 9)	`7B.H`	`0FF.H`	`0400.H`

Note	The radix letters B, O, D, and H can be uppercase or lowercase. This is not affected by the assembler -i option.

Caution	These formats are affected by the RADIX setting. See the table below.

Interpretation of numeric constants without explicit radix notation

Format	Example	Values for each RADIX setting (in decimal)			
		2	8	10	16
One or more 0, 1	0101	5_{10}	65_{10}	101_{10}	257_{10}
One or more 0 to 7	123	Error	83_{10}	123_{10}	291_{10}
One or more 0 to 9	789	Error	Error	789_{10}	1929_{10}
One or more 0, 1 followed by 'B'	101B	5_{10}	5_{10}	5_{10}	4123_{10}
One or more digits 0 to 9, a to f, or A to F, starting with 0 to 9	0FF	Error	Error	Error	255_{10}

20.6 Character Constants

A character enclosed in single quotes (') is treated as a character constant. A character constant is a type of numeric constant, with the value of the ASCII codes of the specified characters. In addition to all printable ASCII characters, the following codes can be used to enter other characters. If more than one character is enclosed in the quotes, this is not a character constant but a character string constant (see Section 20.7, "Character String Constants").

Codes for use in character constants and character string constants

Notation	Hexadecimal value	Character name
\n	0A	Linefeed
\r	0D	Carriage return
\t	09	Horizontal tab
\b	08	Backspace
\f	0C	Form feed
\"	22	Double quote
\'	27	Single quote
\\	5C	Backslash
\ooo		Octal value ooo
\xhh		Hexadecimal value hh

Example 1: `ADD _'A'`
Example 2: `DB 'A', '\012', 'C'`
Example 3: `DB 'ABC'`

Caution	In example 3, 'ABC' is a character string constant, and is therefore an error as the operand for DB.

20.7 Character String Constants

One or more characters enclosed in double quotation marks ("), or two or more characters enclosed in single quotation marks (') are treated as a character string constant. A character string constant can be used as the operand of a DC or .PRINTX pseudoinstruction. Within a character string, any printable ASCII characters can be used, and also the codes listed in Section 20.6, "Character Constants."

Example

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

20.8 Special Symbols

In an operand, an asterisk represents the address of the current location.

Example 1

The following represents the address 6 bytes before the current address.

```
BR      *-6
```

Example 2

The following represents the address 12 bytes after the current address.

```
BR      *+12
```

Chapter 21 Assembler Pseudoinstructions

A pseudoinstruction differs from an ordinary instruction (such as ADD or MOV in the LC86K instruction set); it gives directives or definitions to the assemble, and a pseudoinstruction of itself does not generate a machine instruction. (This does not apply to JMPO and other optimization pseudoinstructions, or to the CHANGE pseudoinstruction.) Pseudoinstructions are often used in combination with ordinary instructions.

Group	Pseudoinstruction	Function
Linking control	ORG	Specify origin
	WORLD	Select the ROM to hold code
	CSEG	Declare the beginning of a code segment
	DSEG	Declare the beginning of a data segment
	END	End program
	PUBLIC	Declare public symbol
	EXTERN	Declare external symbol
	OTHER_SIDE_SYMBOL	Declare CHANGE instruction jump label
Symbol definitions	EQU	Assign a fixed value
	SET	Assign temporary value
Data definitions	DB	Define byte data
	DW	Define word data
	DC	Define character string data
	DS	Define byte area
Macro control	MACRO	Define macro
	REPT	Repeat macro
	IRP	Iteration macro
	IRPC	Character string macro
	ENDM	End macro definition
	EXITM	End macro expansion
	LOCAL	Define local label
Conditional assembly	IFDEF	Assemble if defined
	IFNDEF	Assemble if undefined
	IFB	Assemble if operand empty
	IFNB	Assemble if operand nonempty
	IFE	Assemble if zero
	IFNE	Assemble if nonzero
	IFIDN	Assemble if identical
	IFDIF	Assemble if different
	ELSE	Else case of conditional assembly
	ENDIF	End conditional assembly
	.PRINTX	Display message during assembly
	.LIST	Resume listing
	.XLIST	Suppress listing

Group	Pseudoinstruction	Function
	.MACRO	List macro expansions
	.XMACRO	End macro expansion listing
	.IF	List skipped statements in conditional assembly
	.XIF	End listing of skipped statements
Miscellaneous	INCLUDE	Include file
	TITLE	Set listing title
	PAGE	New page
	CHIP	Specify chip for assembly
	COMMENT	Add comment to object file
	WIDTH	Specify columns in listing file
	BANK	Specify RAM bank
	CHANGE	Jump between flash memory and ROM
	RADIX	Specify default radix
Optimization	JMPO	Optimized JMP instruction
	BRO	Optimized BR instruction
	CALLO	Optimized CALL instruction
	BZO	BZ instruction guaranteeing no address error
	BNZO	BNZ instruction guaranteeing no address error
	BPO	BP instruction guaranteeing no address error
	BPCO	BPC instruction guaranteeing no address error
	BNO	BN instruction guaranteeing no address error
	DBNZO	DBNZ instruction guaranteeing no address error
	BEO	BE instruction guaranteeing no address error
	BNEO	BNE instruction guaranteeing no address error

ORG

Specify origin

Syntax

ORG ***expression***

Description

The ORG pseudoinstruction specifies the start address in program memory (flash memory) as *expression*. Expression must be a numeric constant, or an expression which can be evaluated at assembly time.

Example

```
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

Select the ROM to hold code

Syntax

WORLD ***selection***

Description

This specifies the ROM which will hold the assembled code. This pseudoinstruction is only valid when the target chip is the LC86800 series. There are three values which can be specified for *selection*.

INTERNAL	Store in the on-chip ROM.
EXTERNAL	Store in flash memory bank 0.
EXTERNAL_DATA	Store in flash memory bank 1.

Caution	For Visual Memory, always specify EXTERNAL. Other specifications may lead to data corruption or misoperation.

If there is more than one WORLD pseudoinstruction in a single file, an error results. For chips other than the chips other than the LC86800 series, if a value other than INTERNAL is selected for the WORLD pseudoinstruction, an error results.

CSEG

Declare the beginning of a code segment

Syntax

CSEG ***mode***

Description

This indicates to the assembler the beginning of a segment holding program code. When *mode* is not specified or is INBLOCK, the segment is aligned within 4K boundaries. If the *mode* is FREE, this indicates that the segment can be located regardless of 4K boundaries.

Example

```
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
     0011     C 0009 0000
     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
```

Local address is reset to zero at the beginning of each segment.

Independent segments

DSEG

Declare the beginning of a data segment

Syntax

DSEG

Description

This indicates to the assembler the beginning of a segment holding data.

Caution	Data segments are copied into RAM. It is not possible to open a data segment in flash memory.

Example

```
page:    1 <dseg.ASM>
ERR  SEQ.     S LOC. OBJ.   SOURCE STATEMENTS
     0001                   ; a sample program for CSEG
     0002                           chip    lc864024
     0003                           extern  wait1s
     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   wait1s
     0016     C 0003 210000'        jmpf    label1
     0017
     0018                           dseg
     0019     D 0000        min1:   ds      1
     0020     D 0001        min0:   ds      1
     0021                           end
```

Code segments

Data segment

END

End program

Syntax

END

Description

This indicates the end of the source program. When the assembler encounters this instruction, it ends the pass currently being executed, so any text beyond this point, even if valid statements, is ignored.

Example

```
; 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
```

All text after END is ignored.

```
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

Declare public symbol

Syntax

PUBLIC ***symbol {, symbol}***

Description

The PUBLIC pseudoinstruction declares that a *symbol* defined in the program can be referenced from other source files.

Example

```
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
```

To reference a symbol defined in another file, it must be declared EXTERN.

To allow a symbol in this file to be referenced from another file, it must be declared PUBLIC.

```
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
```

Caution	To reference a symbol defined in another source file, it must be declared EXTERN. To allow a symbol in this file to be referenced from another file, it must be declared PUBLIC.

```
page:   1 <public.ASM>
ERR SEQ.    S LOC. OBJ.      SOURCE STATEMENTS
    0001                     ; 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
```

Note	The combination of PUBLIC and EXTERN declarations allows a symbol to be referenced even when it is defined in another file.

EXTERN

Declare external symbol

Syntax

EXTERN ***[segment:] symbol {, [segment:] symbol}***

Description

The EXTERN pseudoinstruction is used when a *symbol* or symbols are defined in other source program files. The optional *segment* parameter is either CSEG or DSEG, indicating the segment type. If this is not specified, a code segment, CSEG, is the default.

Reference	For examples see the previous item "PUBLIC - Declare public symbol."

OTHER_SIDE_SYMBOL

Declare CHANGE instruction jump label

Syntax

OTHER_SIDE_SYMBOL ***label {, label}***

Description

This declares an address label which can be specified as the operand of a CHANGE instruction, which in the LC86800 series is used for switching between ROM and flash memory. The label declared is a type of external symbol, but one difference is that in a source file of code to be stored in ROM, a label is declared in flash memory (or in ROM in a source file of code to be stored in flash memory). This pseudoinstruction is only valid for the LC86800 series, and in other cases an error results.

Reference	For examples, see under "CHANGE - Jump between flash memory and ROM in this chapter.

EQU

Assign a fixed value

Syntax

Symbolname **EQU** ***expression***

Description

The EQU pseudoinstruction assigns the value *expression* to *symbolname*. A symbol defined with the EQU pseudoinstruction cannot be redefined. Used appropriately, the EQU pseudoinstruction can aid the visual identification of constant data, and improve maintenance efficiency.

Example

When the defined value can be computed it appears here (hexadecimal).

No colon between the defined symbol and the "EQU"

Any expression can be written.

```
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    sub1
     0014     C 0008 230202         mov     #mode_b, b
     0015     C 000B 0818'          CALL    sub1
     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
```

SET

Assign temporary value

Syntax

Symbolname **SET** ***expression***

Description

The SET pseudoinstruction assigns the value *expression* to *symbolname*. A symbol defined with the SET pseudoinstruction can be redefined by a subsequent SET. However, a symbol set with this pseudoinstruction cannot be the subject of a PUBLIC declaration, nor can it be redefined with EQU.

Example

```
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
```

When the defined value can be computed it appears here (hexadecimal).

No colon between the defined symbol and the "SET"

Any expression can be written, including the symbol currently being set.

DB

Define byte data

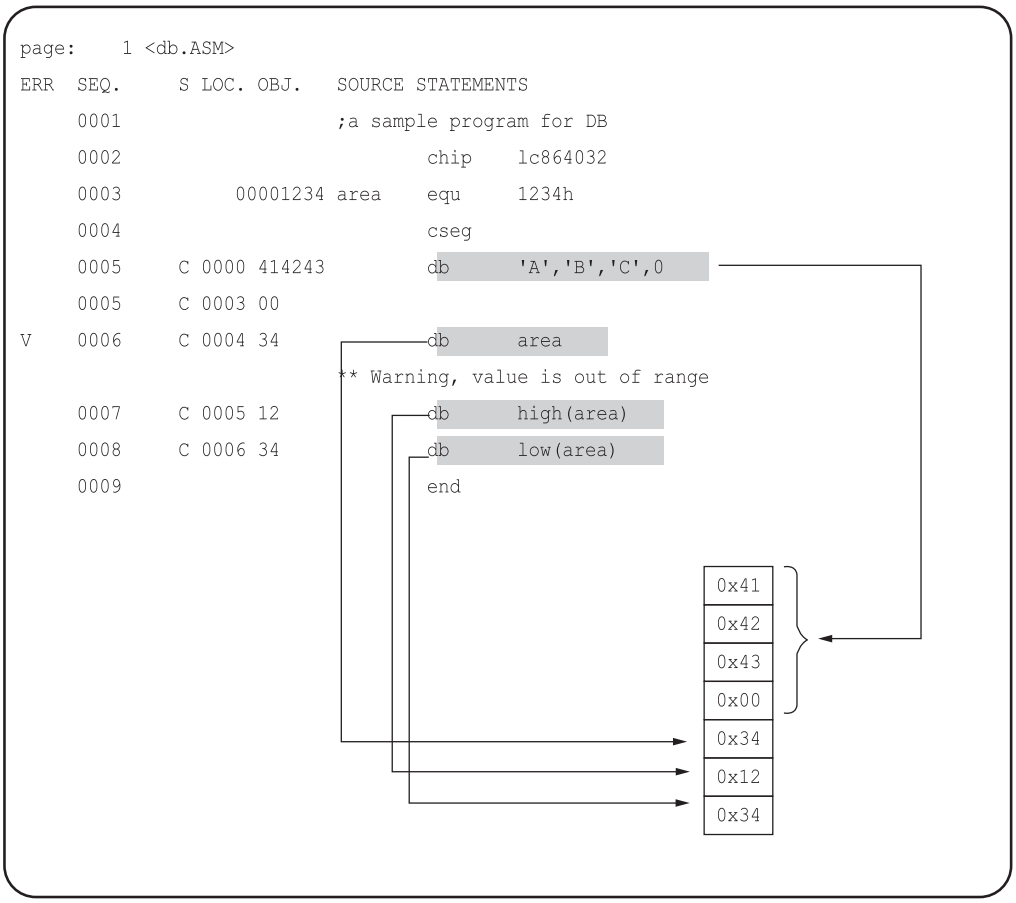
Syntax

Labelname DB expression {, expression}

Description

The DB pseudoinstruction stores the 8-bit data value or values represented by *expression* in program memory (ROM). Any number of operands may be specified, separated by commas. When two or more operands are specified, they are evaluated in order left to right, and stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

Example



In the above example, because the “db area” statement references the symbol “area,” which is a 16-bit value, at assembly time a warning level message, “Value is out of range,” is generated. The bottom eight bits of the value are used in the object code.

DW

Define word data

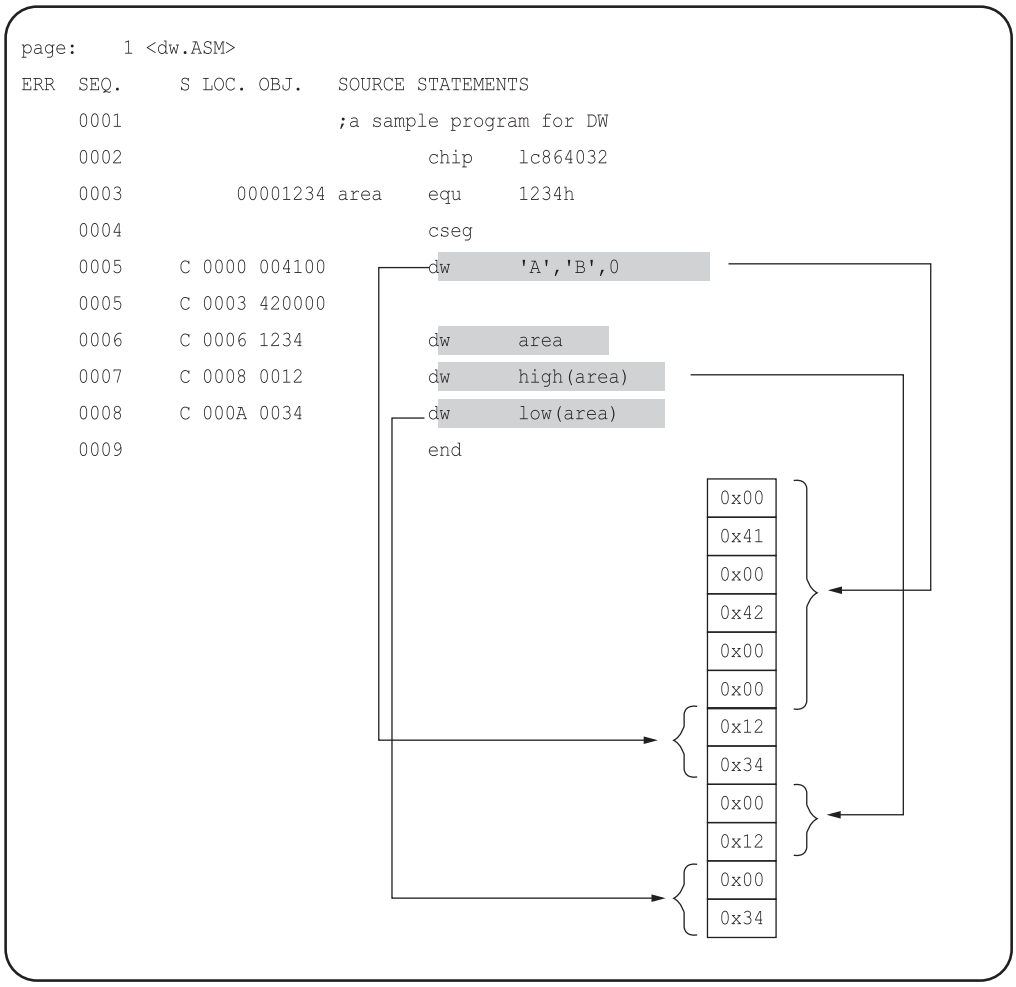
Syntax

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

Description

The DW pseudoinstruction stores the 16-bit data value or values represented by expression in program memory (ROM). The more significant byte is stored first, and the less significant byte at the address one higher. Any number of operands may be specified, separated by commas. When two or more operands are specified, they are stored in successive addresses. If there are two commas with nothing between them, this is interpreted as a zero value.

Example



If the DW pseudoinstruction is used to define 8-bit values, the upper 8 bits of the 16-bit result are always 0.

DC

Define character string data

Syntax

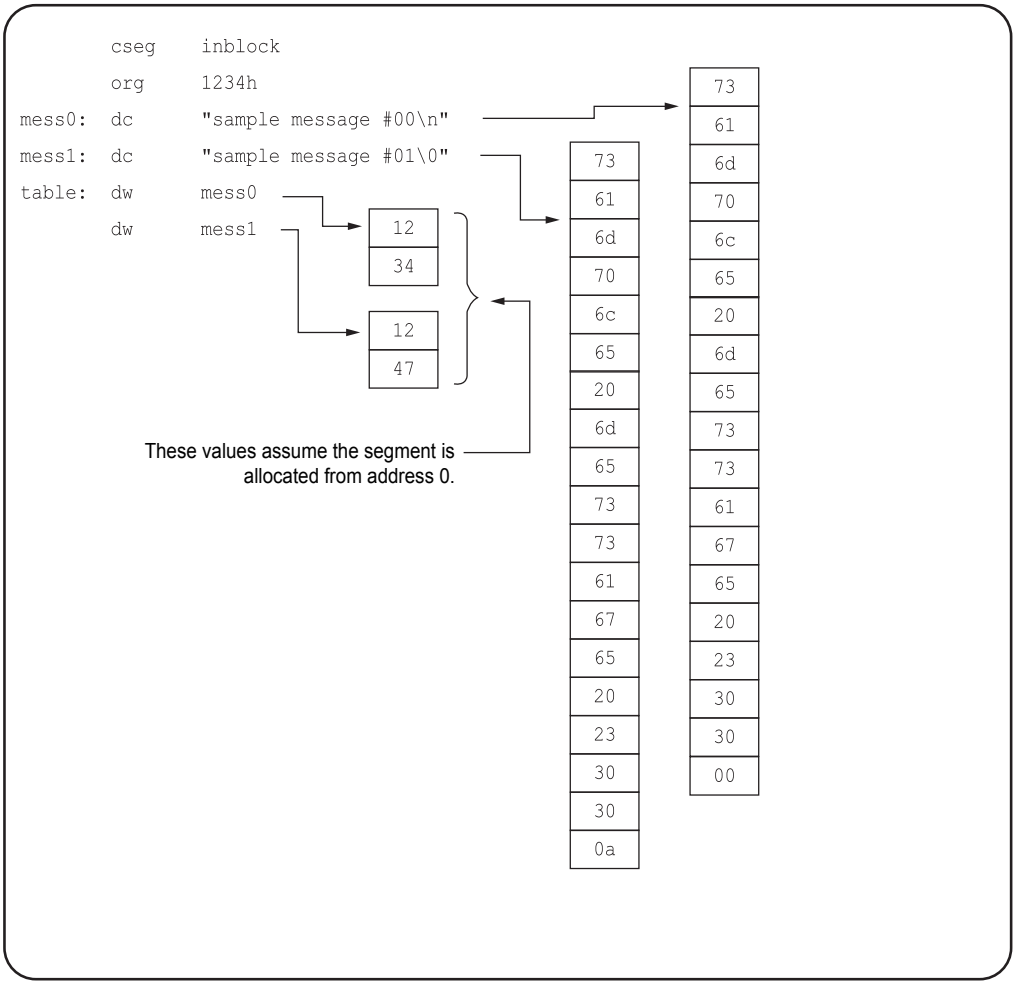
labelname **DC** ***"string"***

Description

This stores the ASCII codes of *string* (a character string constant) in sequence in program memory (ROM).

Reference	For details of character string constants, see Section 20.7, "Character String Constants."

Example



DS
Define byte area

Syntax

labelname **DS** ***absolute_expression***

Description

The DS pseudoinstruction allocates any area of data memory (RAM) of the number of bytes specified by *absolute_expression*. The *absolute_expression* must have a value completely determined at assembly time. This pseudoinstruction can only be used after a DSEG pseudoinstruction.

Caution	A DS pseudoinstruction can only be used to allocate RAM (a data segment). It cannot be used for flash memory. Use DB or DW statements instead.

Example

```
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:     ├──────────┤
                         │ ┄┄┄┄┄┄┄┄ │
                         └──────────┘
```

The example above defines a 1-byte area named area0 that is immediately followed by a 2-byte area named area1.

MACRO

Define macro

Syntax

name **MACRO** ***parameter {, parameter}***

Description

The MACRO pseudoinstruction defines a macro. The statements from the MACRO pseudoinstruction to the following ENDM pseudoinstruction form the body of the macro. The parameter *name* is the *name* by which the macro can be called, which is replaced by the body of the macro, and is therefore mandatory. The formal parameter list specified by *parameter* is optional, depending on the macro being defined.

Caution	When calling one macro from within another, or when using a pseudoinstruction such as IFB which requires angle brackets (< >), a sufficient depth of angle brackets to correspond to the nesting level is required. See the section "EXITM – End macro expansion" in this chapter.

Example

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

Push acc, c, and b onto the stack.

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

Pop b, c, and acc off the stack.

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

Generates code to left shift by the number of positions shown by the parameter. However, if the parameter is zero, generates no code.

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

Format of source program

```
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

Repeat macro

Syntax

REPT ***count***

Description

The REPT pseudoinstruction repeats the statements up to the ENDM instruction, generating the number of copies specified by *count*. This value can be any integer from 1 to 65535.

Example

In the following example, the area not occupied by the program is filled with NOP codes (for a 256-byte boundary).

```
page:    1 <rept.ASM>
ERR  SEQ.     S LOC. OBJ.   SOURCE STATEMENTS
     0001                   ; a sample program for REPT
     0002                           chip    lc864024
     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
```

Body of macro definition is not shown.

Expanded statements

IRP

Iteration macro

Syntax

IRP ***parameter, argument {, argument}…***

Description

The IRP pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each *argument* specified. In each copy, *parameter* is replaced by the corresponding *argument*.

Example

```
_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

Character string macro

Syntax

IRPC ***parameter, string***

Description

The IRPC pseudoinstruction repeats the statements up to the ENDM instruction, generating one copy for each character in *string*. As distinct from a character string constant, *string* is not enclosed in quotation marks. Codes beginning with a backslash cannot be used. In each copy, *parameter* is replaced by the corresponding character in *string*.

Example

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

Formal parameter

Argument string

The formal parameter is replaced by successive characters from the argument string.
The ampersand delimits the formal parameter when it appears within an identifier.

```
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

End macro definition

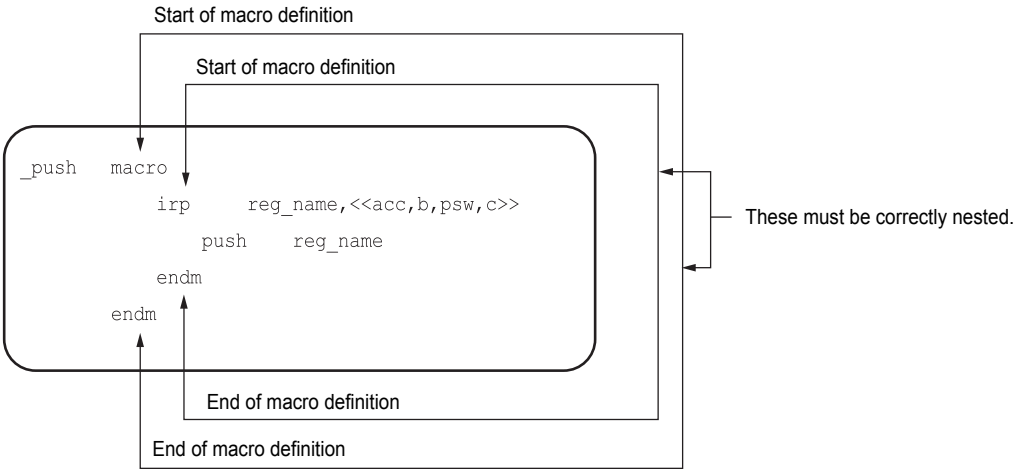
Syntax

ENDM

Description

This marks the end of a macro definition.

Example



EXITM

End macro expansion

Syntax

EXITM

Description

The EXITM pseudoinstruction ends expansion of a macro. In combination with conditional assembly pseudoinstructions, this can be used to create different forms of expansion of a macro depending on the arguments supplied.

Example

The angle brackets must be double, because one layer is removed in each macro expansion.

```
page:    1 <exitm.ASM>
ERR  SEQ.     S LOC. OBJ.   SOURCE STATEMENTS
     0001                   ; a sample program for EXITM
     0002                           chip    LC866032
     0003                   rpush   macro   a1,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
```

When the first argument is applied, this section is assembled.

Since there is no second argument, this section is assembled, and the expansion terminates at EXITM.

LOCAL

Define local label

Syntax

LOCAL ***name {, name}***

Description

The LOCAL pseudoinstruction declares a label which can be used internally to the body of the macro. During macro expansion, the *name* declared in the LOCAL pseudoinstruction is replaced by the assembler with a unique identifier to avoid *name* conflicts.

Example

```
; 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
```

In the above example, the BRO pseudoinstruction is used to define the B_NE macro which generates different instructions depending on the destination of a jump; this is then used in the example. The following is the result of assembly.

```
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
```

The identifier declared as LOCAL is replaced with different names.

The format of the name generated is _L######L_ (where ###### is a serial number starting with 000000).

IFDEF

Assemble if defined

Syntax

IFDEF ***symbol***

Description

If *symbol* is defined, the statements after the IFDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

Example

```
page:    1 <ifdef.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
```

Symbol efg is undefined,
so this section is not assembled.

Symbol abc is defined,
so this section is assembled.

IFNDEF

Assemble if undefined

Syntax

IFNDEF ***symbol***

Description

If *symbol* is undefined, the statements after the IFNDEF pseudoinstruction until the next ELSE or ENDIF are assembled.

Example

```
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
```

Symbol efg is undefined,
so this section is not assembled.

Symbol abc is defined,
so this section is assembled.

IFB

Assemble if operand empty

Syntax

IFB ***<argument>***

Description

If *argument* is empty, the statements after the IFB pseudoinstruction until the next ELSE or ENDIF are assembled. “Empty” means that there are no characters at all (even spaces or tabs) between the angle brackets in which *argument* must be enclosed.

Example

```
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
```

The angle brackets must be double, because one layer is removed in each macro expansion.

This is assembled because the argument to IFB is empty.

This is assembled because the argument to IFB is nonempty.

IFNB

Assemble if operand nonempty

Syntax

IFNB ***<argument>***

Description

If *argument* is nonempty, the statements after the IFNB pseudoinstruction until the next ELSE or ENDIF are assembled. “Empty” means that there are no characters at all (even spaces or tabs) between the angle brackets in which *argument* must be enclosed.

Example

```
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>>
     0005                                   inc     a
     0006                           else
     0007                                   inc     b
     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
```

The angle brackets must be double, because one layer is removed in each macro expansion.

This is assembled because the argument to IFB is empty.

This is assembled because the argument to IFB is nonempty.

IFE

Assemble if zero

Syntax

IFE ***expression***

Description

If the value of *expression* is zero, the statements after the IFE pseudoinstruction until the next ELSE or ENDIF are assembled.

Example

```
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
```

The expression value is zero, so this section is assembled.

The expression value is zero, so this section is assembled.

IFNE

Assemble if nonzero

Syntax

IFNE ***expression***

Description

If the value of *expression* is nonzero, the statements after the IFNE pseudoinstruction until the next ELSE or ENDIF are assembled.

Example

```
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
```

The expression value is zero, so this section is assembled.

The expression value is zero, so this section is assembled.

IFIDN

Assemble if identical

Syntax

IFIDN ***<string1>, <string2>***

Description

If the two strings *string1* and *string2* are identical, the statements after the IFIDN pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

Example

```
page:    1 <ifidn.ASM>
ERR  SEQ.     S LOC. OBJ.   SOURCE STATEMENTS
     0001                   ; a sample program for IFIDN
     0002                           chip    lc866032
     0003                           cseg
     0004                   tifidn  macro   arg1,arg2
     0005                           ifidn   <<arg1>>,<<arg2>>
     0006                                   inc     a
     0007                           else
     0008                                   dec     a
     0009                           endif
     0010                           endm
     0011
     0012                           tifidn  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                           tifidn  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
```

The angle brackets must be double, because one layer is removed in each macro expansion.

The strings are different, so this section is assembled.

The strings are the same, so this section is assembled.

IFDIF

Assemble if different

Syntax

IFDIF ***<string1>, <string2>***

Description

If the two strings *string1* and *string2* are different, the statements after the IFDIF pseudoinstruction until the next ELSE or ENDIF are assembled. The strings must be enclosed in angle brackets, and within them, comparison is carried out with spaces and tabs considered significant.

Example

```
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
```

The angle brackets must be double, because one layer is removed in each macro expansion.

The strings are different, so this section is assembled.

The strings are the same, so this section is assembled.

ELSE

Else case of conditional assembly

Syntax

ELSE

Description

The statements after the ELSE pseudoinstruction until the next ENDIF are assembled when the test condition of the preceding IF pseudoinstruction fails to hold.

Reference	See under “IFDEF - Assemble if defined” in this section.

ENDIF

End conditional assembly

Syntax

ENDIF

Description

Marks the end of a conditional assembly.

Reference	See under "IFDEF - Assemble if defined" in this section.

.PRINTX

Display message during assembly

Syntax

.PRINTX ***"string"***

Description

The .PRINTX pseudoinstruction displays the character *string* constant string during assembly.

Reference	For details of character string constants, see Section 20.7, "Character String Constants."

Example

Source program

```
; 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
```

Screen display

```
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
```

Because of the IFDEF pseudoinstruction, this section is not assembled, so the corresponding output does not appear.

.LIST

Resume listing

Syntax

.LIST

Description

The .LIST pseudoinstruction resumes listing output, when it has been suppressed with the .XLIST pseudoinstruction.

Example

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

From the .XLIST line onwards, output to the listing file is suppressed. However, line numbers are still counted, so there is no loss of synch.

```
.xlist
abc     equ     10h
        dseg
count:  ds      4
.list
        cseg    inblock
        ld      b
        sub     #abc
        st      count
        end

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
```

From the .LIST line onwards, output to the listing file is resumed.

.XLIST

Suppress listing

Syntax

.XLIST

Description

The .XLIST pseudoinstruction suppresses output to the listing file.

Reference	See under “.LIST - Resume listing,” in this section.

.MACRO

List macro expansions

Syntax

.MACRO

Description

The .MACRO pseudoinstruction causes the expanded body of macro calls to be output to the listing file.

Example

```
; 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
```

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing. This means that the generated statements and code both disappear.

The .MACRO pseudoinstruction resumes causes the listing of expanded macro calls.

.XMACRO

End macro expansion listing

Syntax

.XMACRO

Description

The .XMACRO pseudoinstruction ends the output of expanded macro calls to the listing.

Reference	For an example, see under the previous item, “.MACRO - List macro expansions.”

.IF

List skipped statements in conditional assembly

Syntax

.IF

Description

The .IF pseudoinstruction causes source program statements skipped in a conditional assembly to be output to the listing file.

Example

```
; 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
```

The .XIF pseudoinstruction stops source program statements skipped in a conditional assembly from being output to the listing file. The part of a conditional assembly section which is assembled appears in the listing regardless of this pseudoinstruction.

The .IF pseudoinstruction causes even the source program statements skipped in a conditional assembly to be output to the listing file.

```
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                         else
     0014+4   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

End listing of skipped statements

Syntax

.XIF

Description

The .XIF pseudoinstruction stops source program statements skipped in a conditional assembly from being output to the listing file.

Reference	For an example, see under the previous item, “.IF - List skipped statements in conditional assembly.”

INCLUDE

Include file

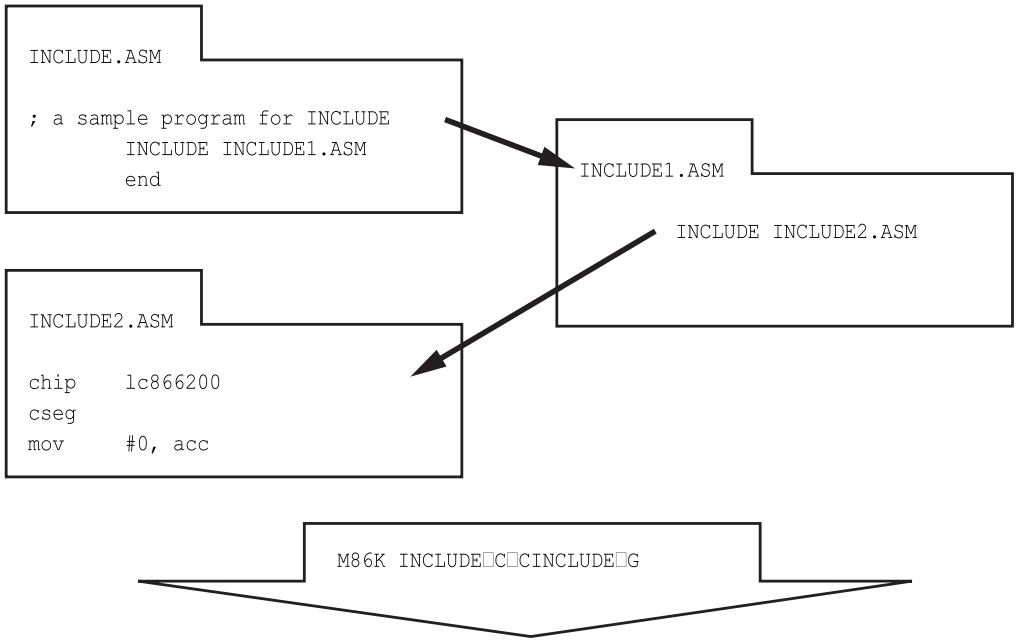
Syntax

INCLUDE ***filename***

Description

The INCLUDE pseudoinstruction causes the source file specified by *filename* to be read into the current point in the source program and assembled. The specification of *filename* must include the extension. The INCLUDE pseudoinstruction can be nested to a maximum depth of nine. Note that if an END pseudoinstruction occurs in the included file, this terminates the assembly.

Example




```
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    lc866200
   2/0002                           cseg
   2/0003     C 0000 230000         mov     #0, acc
     0003
```

Indicates the nesting depth of includes.

TITLE

Set listing title

Syntax

TITLE ***string***

Description

The TITLE pseudoinstruction specifies *string* as the title for the listing file. Unlike a character *string* constant, string is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol.

Example

This string appears on all pages of the listing.

```
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

New page

Syntax

PAGE

Description

The PAGE pseudoinstruction forces a new page in the listing file. The page break appears immediately after this pseudoinstruction.

Example

Source file

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

Listing file

```
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

Specify chip for assembly

Syntax

CHIP ***chipname***

Description

The CHIP pseudoinstruction informs the assembler of the chip for which assembly is to be carried out. According to the value of *chipname*, the assembly changes the reserved words, and carries out a memory size check. This pseudoinstruction must appear at the beginning of the source file, before any other instructions or pseudoinstructions. If this pseudoinstruction is not found, the environment variable CHIPNAME is referenced. If the chip name specified by this pseudoinstruction is different from the chip specified by the CHIPNAME environment variable, a warning level error is issued.

Note	For developing Visual Memory applications, the chip name must be set to LC868700.

COMMENT

Add comment to object file

Syntax

COMMENT ***comment_string***

Description

The COMMENT pseudoinstruction adds a comment directly into the assembled object code. Unlike a character string constant, *comment_string* is not enclosed in quotation marks. It is also not possible to include codes with the backslash (\) symbol. The comment is stored from byte 680 of the object file. A maximum of 255 characters can be used for the comment.

Example

Source file

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

Dump of object file (part only)

Character count (1 byte)

```
00000260  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
00000270  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

Specify columns in listing file

Syntax

WIDTH ***number***

Description

The WIDTH pseudoinstruction specifies the number of character columns in the listing file, that is, the number of characters in each line. The parameter *number* may be any value from 72 to 132 inclusive, but the recommended minimum is the number of columns of the source file plus 28. Although this pseudoinstruction can appear any number of times in a single source file, normally it is specified once only at the beginning of the file. If this pseudoinstruction is not found, the default listing file has 132 columns.

Example

WIDTH is evaluated on pass 1 and pass 2, but the listing output occurs in pass 2 only, so the last value found in pass 1 is used here, causing the lines to be folded at 78 characters.

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
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 indicat
     0003                   e WIDTH's effect
     0004                           WIDTH   72
     0005     C 0000 00              nop     ; this is also a long line
     0005                    to indicate WIDTH's effect
     0006                           WIDTH   78
     0007                           end
```

Carriage return and linefeed is inserted at character position 72, folding the lines here.

BANK

Specify RAM bank

Syntax

BANK ***expression***

Description

The BANK pseudoinstruction supplies the bank number for symbols defined by DS pseudoinstructions for RAM after a DSEG pseudoinstruction.

Example

```
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
```

These symbols are assigned to bank 1.

These symbols are assigned to bank 0.

CHANGE

Jump between flash memory and ROM

Syntax

CHANGE ***symbol***

Description

For the LC86800 series, this is a special jump instruction for switching between code in flash memory and code in ROM (system BIOS). The operand *symbol* must have been declared with the pseudoinstruction OTHER_SIDE_SYMBOL. Note that this pseudoinstruction is special to the LC86800 series, and in other cases an error results.

Note	For Visual Memory, use this instruction to call an operating system function.

Reference	For details of operating system function calls, refer to the “System BIOS” section of the Visual Memory Hardware Manual.

Example

```
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

Specify default radix

Syntax

RADIX ***expression***

Description

The RADIX pseudoinstruction specifies the radix, or base, of a numeric constant with no explicit radix indication. The value of *expression* must be a constant value from the set 2, 8, 10, and 16. This specification takes effect from this statement until a subsequent RADIX pseudoinstruction. If this pseudoinstruction is not present, the default radix is 10.

Example

```
Xxx SET     10  _   interpreted by default as 10 decimal.
    RADIX   16
Xxx SET     10  _   interpreted as 16 decimal, because of the radix value 16.
    RADIX   2
Xxx SET     10  _   interpreted as 2 decimal, because of the radix value 2.
```

JMPO

Optimized JMP instruction

Syntax

JMPO ***expression***

Description

The JMPO pseudoinstruction compares *expression* with the current location, and if this is a jump within the same block (only the bottom 12 bits of the addresses are different) generates a JMP instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a JMPF instruction.

Example

JMP instruction when within the same memory block

```
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
```

JMPF instruction when in a different memory block

BRO

Optimized BR instruction

Syntax

BRO ***expression***

Description

BRO pseudoinstruction compares *expression* with the current location, and if the branch address is within the range -128 to +127 generates a BR instruction; when outside the range -128 to +127 generates a BRF instruction.

Example

Generates a BR instruction because destination is within the range -128 to +127.

```
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
```

Generates a BRF instruction because destination is outside the range -128 to +127.

CALLO

Optimized CALL instruction

Syntax

CALLO ***expression***

Description

The CALLO pseudoinstruction compares *expression* with the current location, and if this is a call within the same block (only the bottom 12 bits of the addresses are different) generates a CALL instruction. Otherwise, that is, if the address is in a different block, or if the address cannot be determined because for example it is an external symbol, then this generates a CALLF instruction.

Example

CALL instruction when within the same memory block

```
page:    1 <CALLo.ASM>
ERR  SEQ.     S LOC. OBJ.   SOURCE STATEMENTS
     0001                   ; a sample program for CALLO
     0002                           chip    1c866032
     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
```

CALLF instruction when in a different memory block

BZO

BZ instruction guaranteeing no address error

Syntax

BZO *expression*

Description

The BZO macro generates code equivalent to the BZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BZO macro uses a BNZ instruction, which is the logical inverse of the BZ instruction, and a BRO instruction. Enter the branch destination for *expression*.

Code generation macro

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

BNZO

BNZ instruction guaranteeing no address error

Syntax

BNZO ***expression***

Description

The BNZO macro generates code equivalent to the BNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNZO macro uses a BZ instruction, which is the logical inverse of the BNZ instruction, and a BRO instruction. Enter the branch destination for *expression*.

Code generation macro

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

BPO

BP instruction guaranteeing no address error

Syntax

BPO *expression*

Description

The BPO macro generates code equivalent to the BP instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPO macro uses a BP instruction, and BR and BRO instructions. Enter the branch destination for *expression*.

Code generation macro

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

BPCO

BPC instruction guaranteeing no address error

Syntax

BPCO ***expression***

Description

The BPCO macro generates code equivalent to the BPC instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BPCO macro uses a BPC instruction, and BR and BRO instructions. Enter the branch destination for *expression*.

Code generation macro

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

BNO

BN instruction guaranteeing no address error

Syntax

BNO *expression*

Description

The BNO macro generates code equivalent to the BN instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNO macro uses a BN instruction, and BR and BRO instructions. Enter the branch destination for *expression*.

Code generation macro

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

DBNZO

DBNZ instruction guaranteeing no address error

Syntax

DBNZO ***expression***

Description

The DBNZO macro generates code equivalent to the DBNZ instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The DBNZO macro uses a DBNZ instruction, and BR and BRO instructions. The function of *expression* is the same as in the DBNZ instruction.

Code generation macro

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

BEO

BE instruction guaranteeing no address error

Syntax

BEO *expression*

Description

The BEO macro generates code equivalent to the BE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BEO macro uses a BNE instruction and BRO instruction. The function of *expression* is the same as in the BE instruction.

Code generation macro

```
; *** 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

BNE instruction guaranteeing no address error

Syntax

BNEO ***expression***

Description

The BNEO macro generates code equivalent to the BNE instruction, with no restrictions on the distance between the branch destination in the same segment of the same source and the location of this instruction. The BNEO macro uses a BE instruction and BRO instruction. The function of *expression* is the same as in the BNE instruction.

Code generation macro

```
; *** 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
```

Chapter 22 LC86K Instruction Summary

This chapter describes general features of flag handling and addressing, before the complete listing of the instruction set.

22.1 Instruction Summary

22.1.1 Arithmetic Instructions

The arithmetic instructions operate principally on the accumulator, and carry out the four basic operations, incrementing, and decrementing. The carry, auxiliary carry, and overflow flags are set according to the results of arithmetic operations, as follows.

CY (carry flag)

Arithmetic operation	Operation result	CY
Add instructions	When there is a carry from bit 7 (MSB)	1
	When there is no carry from bit 7 (MSB)	0
Subtraction and comparison instructions	When a borrow from bit 7 (MSB) is required	1
	When no borrow from bit 7 (MSB) is required	0
Multiplication and division instructions	-	0

AC (auxiliary carry flag)

Arithmetic operation	Operation result	AC
Add instructions	When there is a carry from bit 3	1
	When there is no carry from bit 3	0
Subtraction instructions	When a borrow from bit 3 is required	1
	When no borrow from bit 3 is required	0

OV (overflow flag)

Arithmetic operation	Operation result	OV
Add and subtract instructions	When there is a carry from bit 7 but not from bit 6	1
	When there is a carry from bit 6 but not from bit 7	1
	When an overflow error occurs in a signed variable addition instruction	1
	All other cases	0
Multiplication instructions	When the product is 256 or more	1
	When the product is 255 or less	0
Division instructions	When the divisor is zero	1
	When the divisor is nonzero	0

22.1.2 Logical Instructions

Logical instructions carry out logical rotates. The RORC and ROLC instructions also affect the carry flag.

22.1.3 Data Transfer Instructions

The data transfer instructions read, write, back up, and exchange data to and from RAM and special function registers (SFR).

22.1.4 Jump Instruction

A jump instruction is an unconditional transfer to a new instruction.

22.1.5 Conditional Branch Instructions

Conditional branch instructions determine the value of a specified condition as true or false, and transfer to the specified destination if true. If false, there is no transfer, and control passes to the next instruction.

The BE and BNE instructions branch on the basis of a comparison of two 8-bit data values, and the carry flag is set or cleared according to the result, as follows.

	Operands			Carry flag (CY)
	#i8, r8	d9, r8	@Rj, #i8, r8	
Magnitude relation	#i8 > (ACC)	(d9) > (ACC)	#i8 > ((Rj))	1
	#i8 = (ACC)	(d9) = (ACC)	#i8 = ((Rj))	0
	(d9) < (ACC)	(d9) < (ACC)	#i8 < ((Rj))	0

22.1.6 Subroutine Instruction

The subroutine instruction performs an unconditional branch to another instruction. An address is stored on the stack In order that, after the branch, a return instruction (RET or RETI) can return to the instruction following the CALL instruction. The stack is in RAM, and is pointed to by the stack pointer (SP). Enough RAM must be reserved for the stack to allow for the nesting level of subroutine calls.

Note	The Visual Memory stack is held in bank 0 of RAM. When an application is started, the system BIOS sets it to 7FH. When a value is pushed onto the stack, the stack pointer is incremented before storing the data, so the actual values are stored from address 80H. The stack consumes addresses upwards from 7FH to 0FFH. The internal clock function also needs 20 bytes on the stack.

22.1.7 Bit Manipulation Instructions

The bit manipulation instructions operate on individual bits of specified RAM or special function registers (SFR).

22.1.8 Other Instructions

The NOP instruction has no effect other than to consume one clock cycle.

22.1.9 Macro Instruction

This is a dedicated standard macro instruction. It switches between the execution of the system BIOS in ROM and the application program in flash memory.

22.1.10 Addressing

There are a number of different methods of addressing for flash memory, RAM, and special function registers (SFRs).

22.1.11 Program Memory Addressing

The program ROM address of the destination of a jump, branch, or subroutine instruction is shown by the instruction code. In this case the address is shown by one of the following addressing methods.

r8 (8-bit relative addressing)

The transfer is to an address in the range -128 to +127 from the start address of the currently executed instruction. This is shown by a signed 8-bit value.

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

r16 (16-bit relative addressing)

This allows a transfer anywhere within the 64K-byte flash memory address space. It is shown by an unsigned 16-bit value.

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

a12 (12-bit absolute addressing)

The top four bits PC15 to PC12 (the current page) of the address of the instruction after the current instruction (PC15 to PC00) are kept the same, and the remaining 12 bits (PC11 to PC00) are replaced by the address data (000H to FFFH). This allows a jump anywhere within the current page (PC15 to PC12).

Caution	Care is required, because if a JMP instruction or CALL instruction occurs at the final address within a page, the current page changes.

a16 (16-bit absolute addressing)

This allows a transfer anywhere within the 64K-byte flash memory address space.

The 16-bit value is used unchanged as the address.

[0000H to FFFFH: 0 to 65535]

Table jumps

If the destination address of a jump is on the stack, a RET instruction forces the address into the program counter (PC), thus achieving a jump.

In Example 1, the first line sets the stack pointer (SP) to 09H. Executing a RET instruction now causes a jump to the address whose upper byte is the value of byte 08H in RAM, and whose lower byte is the value of byte 07H in RAM; the jump address is set accordingly in lines 2 and 3.

Since the jump destination is PC = 0C13H, in line 2 the lower byte is set to 13H, and in line 3 the upper byte is set to 0CH. In the fourth line, when the RET instruction is executed, the stack pointer is set to 07H, and a jump to 0C13H occurs. However, in Example 1, since the stack pointer value must be known explicitly, normally a PUSH instruction is used as in Example 2.

Example 1

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

Example 2

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

Example 3 carries out a 128-way branch to 00H to 7FH on the basis of the values of RAM address 70H.

In lines 1 and 2, the lower byte of the branch destination address is set, and in line 4 the upper byte of the address. The RET instruction in line 6 branches to the jump table in lines 7 and 8, thence to the branch destination.

This is referred to as a “table jump” and can be used to branch to a number of different addresses according to conditions.

Example 3

```
A0: LD  070H
  ROL
  ADD #LOW(A1)
  PUSH  ACC
  MOV #HIGH(A1),ACC
  PUSH  ACC
  RET
;
  ORG 0C00H
A1: JMP B00          jump table
    _
    JMP B7F
;
B00:    XXXXXX
```

22.1.12 RAM and Special Function Register (SFR) Addressing

d9 (direct addressing)

Addresses RAM or SFR directly with nine bits d8 to d0.

Addresses 000H to 0FFH . . . specify RAM.

Addresses 100H to 1FFH . . . specify an SFR.

b3 (bit addressing)

In the bit manipulation instructions (SET1, CLR1, NOT1), and the BP, BPC, and BN instructions, 3-bit bit-address data is used in combination with d9 (direct addressing), to specify individual bits within RAM or an SFR.

	MSB							LSB
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	7	6	5	4	3	2	1	0
Bit-address value	(111)	(110)	(101)	(100)	(011)	(010)	(001)	(000)

@Rj (indirect addressing)

For indirect addressing the destination RAM or SFR address is stored in a particular location in RAM, and the access made through specification of this address in RAM.

Reference	For more details of indirect addressing refer to the Visual Memory Hardware Manual.

The particular addresses in RAM are referred to as indirect address registers, and are indicated as @R0, @R1, @R2, and @R3. The indirect address registers are accessed using a 2-bit indirect addressing value (j1, j0), allowing a specification from @R0 to @R3.

A bank of four indirect address registers is assigned to the first 16 bytes (addresses 00H to 0FH) of each RAM bank. The RAM bank is selected with RAMBK0 (bit 1 of PSW). The indirect address register bank is selected with IRBK1 and 0 (bits 4 and 3 of PSW).

When an indirect addressing instruction is executed, for the indirect address register and the RAM address specified by the indirect address register, the RAM address used is in the RAM bank specified by IRBK1 and 0 and RAMBK0. On a reset, IRBK0, and 1 are both zero, and RAMBK0 is also set to zero, so the absolute addresses of @R0, @R1, @R2, and @R3 are respectively 00H, 01H, 02H, and 03H in RAM bank 0.

Indirect address registers . . . @R3 @R2 @R1 @R0

Indirect addressing values (j1, j0) . . . (11) (10) (01) (00)

Indirect addressing register map

Indirect address register	Function	Bank 0 (IRBK1 = 0) (IRBK0 = 0)	Bank 1 (IRBK1 = 0) (IRBK0 = 1)	Bank 2 (IRBK1 = 1) (IRBK0 = 0)	Bank 3 (IRBK1 = 1) (IRBK0 = 1)
@R0	RAM access	RAM 00H	RAM 04H	RAM 08H	RAM 0CH
@R1	RAM access	RAM 01H	RAM 05H	RAM 09H	RAM 0DH
@R2	SFR access	RAM 02H	RAM 06H	RAM 0AH	RAM 0EH
@R3	SFR access	RAM 03H	RAM 07H	RAM 0BH	RAM 0FH

Examples of indirect addressing

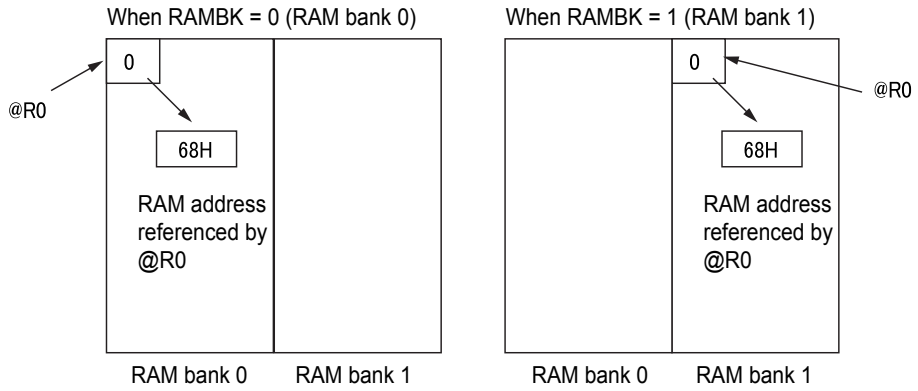
The following are examples of calculation using indirect address registers.

In Example 1, in the second line immediate data 68H is stored in RAM (address 00H). Using RAM (address 00H) as an indirect address register, RAM (address 68H) is accessed. For example, in line 3, the indirect address register (@R0) is specified to store immediate data 10H in RAM (address 68H).

In line 5, by specifying the indirect address register (@R0), the contents of RAM (address 68H) is added to the accumulator.

Example 1

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



The next example uses indirect addressing to access an SFR.

In Example 2, the first two lines clear bits 4 and 3 of the PSW, selecting RAM addresses 00H to 03H for the indirect address registers. In the fourth line immediate data 02H is stored in RAM address 02H. Then using RAM (address 02H) as an indirect address register accesses RAM (address 02H). For example, in line 5, immediate data 12H is stored by specifying the indirect address register (@R2) in an SFR (address 02H: B register). In line 6, the indirectly addressed B register is incremented.

Example 2

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

The next example uses bank-switching with the PSW, to indirectly address an SFR.

In Example 3, the first two lines set the PSW bank to 2, so that RAM addresses 08H to 0BH are used as indirect address registers. In line 4, immediate data 02H is stored in RAM address 0BH. Using RAM (address 0BH) as an indirect address register accesses RAM (address 02H). For example, in line 5 immediate data 12H is stored in the SFR (address 02H: B register) by specifying the indirect address register (@R2). In line 6 the indirectly addressed B register is incremented.

Example 3

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

Chapter 23 Instruction Set Reference

The comprehensive LC86K instruction set includes some 70 instructions. Identified by some 45 operation codes these can be grouped into the following eight categories.

Arithmetic instructions	ADD, ADDC, SUB, SUBC, INC, DEC, MUL, DIV
Logical instructions	AND, OR, XOR, ROL,ROLC,ROR, RORC
Data transfer instructions	LD, ST, MOV, LDC, PUSH, POP, XCH
Jump instructions	JMP, JMPF, BR, BRF
Conditional branch instructions	BZ, BNZ, BP, BPC, BN, DBNZ, BE, BNE
Subroutine instructions	CALL, CALLF, CALLR, RET, RETI
Bit manipulation instructions	CLR1, SET1, NOT1
Miscellaneous instruction	NOP
Macro instruction	CHANGE

Arithmetic Instructions

ADD _i8

ADD immediate data to accumulator

Instruction code	`1 0 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 (81H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) + #i8
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

Example

```
                                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

Instruction code	`1 0 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 (82H to 83H)`
Byte count	2
Cycles	1
Function	(ACC) _ (ACC) + (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

Example 1

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

Example 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

Instruction code	1 0 0 0 0 1j1j0 (84H to 87H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) + ((Rj)) j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

Example 1

		ACC	RAM 00H	RAM 68H	CY	AC	OV
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

Example 2

		ACC	RAM 02H	TRL	CY	AC	OV
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

Instruction code	`1 0 0 1 0 0 0 1 i7i6i5i4i3i2i1i0 (91H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) + (CY) + #i8
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and carry flag to the immediate data (i7 to i0), and stores the result in the accumulator.

Example

```
                                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

Instruction code	`1 0 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (92H to 93H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) + (CY) + (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and carry flag to the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

Example 1

		ACC	RAM 23H	CY	AC	OV
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

Example 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

Instruction code	`1 0 0 1 0 1j1j0 (94H to 97H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) + (CY) + ((Rj)) j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Adds the contents of the accumulator and carry flag to the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

Example 1

		ACC	RAM 00H	RAM 68H	CY	AC	OV
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

Example 2

		ACC	RAM 02H	TRL	CY	AC	OV
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

Instruction code	`1 0 1 0 0 0 0 1 i7i6i5i4i3i2i1i0 (A1H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) - #i8
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts immediate data (i7 to i0) from the contents of the accumulator, and stores the result in the accumulator.

Example

```
                                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

Instruction code	**1 0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (A2H to A3H)**
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) - (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts the contents of the RAM address or SFR specified by d8 to d0 from the contents of the accumulator, and stores the result in the accumulator.

Example 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

Example 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

Instruction code	`1 0 1 0 0 1j1j0 (A4H to A7H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) - ((Rj)) j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 from the contents of the accumulator, and stores the result in the accumulator.

Example 1

		ACC	RAM 00H	RAM 68H	CY	AC	OV
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

Example 2

		ACC	RAM 02H	TRL	CY	AC	OV
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

Instruction code	`1 0 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 (B1H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) - (CY) - #i8
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts immediate data (i7 to i0) and carry flag from the contents of the accumulator, and stores the result in the accumulator.

Example

```
                                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

Instruction code	1 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (B2H to B3H)
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) - (CY) - (d9)
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts the contents of the RAM address or SFR specified by d8 to d0 and carry flag from the contents of the accumulator, and stores the result in the accumulator.

Example 1

		ACC	RAM 23H	CY	AC	OV
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

Example 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

Instruction code	1 0 1 1 0 1j1j0 (B4H to B7H)
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) - (CY) - ((Rj)) j = 0, 1, 2, 3
Flags affected	CY, AC, OV
Interrupts enabled	Yes

Description

Subtracts the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 and carry flag from the contents of the accumulator, and stores the result in the accumulator.

Example 1

		ACC	RAM 00H	RAM 68H	CY	AC	OV
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

Example 2

		ACC	RAM 02H	TRL	CY	AC	OV
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

Instruction code	`0 1 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (62H to 63H)`
Byte count	2
Cycles	1
Function	(d9) ← (d9) + 1
Flags affected	
Interrupts enabled	Yes

Description

Increments the contents of the RAM address or SFR specified by d8 to d0.

Example 1

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

Example 2

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

Caution	
	• The flags, CY, AC, and OV are not changed. • When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

INC @Rj

Increment indirect byte

Instruction code	0 1 1 0 0 1j1j0 (64H to 67H)
Byte count	1
Cycles	1
Function	((Rj)) ← ((Rj)) + 1 j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Increments the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

Example 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
```

Example 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
```

Caution

- The flags, CY, AC, and OV are not changed.
- When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

DEC d9

Decrement direct byte

Instruction code	`0 1 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (72H to 73H)`
Byte count	2
Cycles	1
Function	(d9) ← (d9) - 1
Flags affected	
Interrupts enabled	Yes

Description

Decrements the contents of the RAM address or SFR specified by d8 to d0.

Example 1

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

Example 2

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

Caution	
	• The flags, CY, AC, and OV are not changed. • When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

DEC @Rj

Decrement indirect byte

Instruction code	`0 1 1 1 0 1j1j0 (74H to 77H)`
Byte count	1
Cycles	1
Function	((Rj)) ← ((Rj)) - 1 j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Decrements the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

Example 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
```

Example 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
```

Caution	
	• The flags, CY, AC, and OV are not changed. • When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

MUL

Multiply accumulator and C register by B register

Instruction code	0 0 1 1 0 0 0 0 (30H)
Byte count	1
Cycles	7
Function	(B)(ACC)(C) ← (ACC)(C) × (B)
Flags affected	CY, OV
Interrupts enabled	Yes on the 7th cycle

Description

Multiplies the unsigned 16-bit value represented by the accumulator and C register by the unsigned 8-bit value of the B register. Of the 24-bit calculation result, the bottom 8 bits are stored in C, the middle 8 bits in the accumulator, and the top 8 bits in B.

As a result of the calculation, if the contents of B are zero, the overflow flag is cleared, and if the contents of B are nonzero the overflow flag is set. The carry flag is always cleared.

Example 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

Example 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

Instruction code	0 1 0 0 0 0 0 0 (40H)
Byte count	1
Cycles	7
Function	(ACC)(C), mod(B) ← (ACC)(C) ÷ (B)
Flags affected	CY, OV
Interrupts enabled	Yes on the 7th cycle

Description

Divides the 16-bit value represented by the contents of the accumulator (upper byte) and C register (lower byte) by the contents of the B register (unsigned 8-bit value). The quotient is stored in the accumulator (upper byte) and C (lower byte), and the remainder is stored in B.

Caution	If this instruction is executed with the contents of the B register zero, the accumulator is set to FFH, and the overflow flag is set. If the B register is nonzero, then the overflow flag is cleared, and the carry flag is also always cleared.

Example 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	

Example 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	**error**

Logical Instructions

AND _i8

AND immediate data to accumulator

Instruction code	`1 1 1 0 0 0 0 1 i7i6i5i4i3i2i1i0 (E1H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ^ #i8
Flags affected	
Interrupts enabled	Yes

Description

Logical ANDs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

Example 1

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

Example 2

```
                                ACC
MOV       #0FFH,ACC             FFH
AND       #0FEH                 FEH
AND       #0FDH                 FCH
AND       #0FBH                 F8H
AND       #0F7H                 F0H
AND       #0EFH                 E0H
AND       #0DFH                 C9H
AND       #0BFH                 80H
AND       #07FH                 00H
```

AND d9

AND direct byte to accumulator

Instruction code	`1 1 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 (E2H to E3H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ^ (d9)
Flags affected	
Interrupts enabled	Yes

Description

Logical ANDs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

Example 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
```

Example 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                     F0H  F7H
```

AND @Rj

AND indirect byte to accumulator

Instruction code	`1 1 1 0 0 1j1j0 (E4H to E7H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) ^ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Logical ANDs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

Example 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
```

Example 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

Instruction code	`1 1 0 1 0 0 0 1 i7i6i5i4i3i2i1i0 (D1H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) v #i8
Flags affected	
Interrupts enabled	Yes

Description

Logical ORs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

Example 1

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

Example 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

Instruction code	`1 1 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (D2H to D3H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) v (d9)
Flags affected	
Interrupts enabled	Yes

Description

Logical ORs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

Example 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
```

Example 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

Instruction code	`1 1 0 1 0 1j1j0 (D4H to D7H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) v ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Logical ORs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

Example 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
```

Example 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

Instruction code	`1 1 0 1 0 1j1j0 (D4H to D7H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) v ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Logical XORs the contents of the accumulator and immediate data (i7 to i0), and stores the result in the accumulator.

Example 1

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

Example 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

Instruction code	`1 1 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 (F2H to F3H)`
Byte count	2
Cycles	1
Function	(ACC) ← (ACC) ∀ (d9)
Flags affected	
Interrupts enabled	Yes

Description

Logical XORs the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0, and stores the result in the accumulator.

Example 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
```

Example 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

Instruction code	`1 1 1 1 0 1j1j0 (F4H to F7H)`
Byte count	1
Cycles	1
Function	(ACC) ← (ACC) ∀ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Logical XORs the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0, and stores the result in the accumulator.

Example 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
```

Example 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

Instruction code	1 1 1 0 0 0 0 0 (E0H)
Byte count	1
Cycles	1
Function	←A7←A6←A5←A4←A3←A2←A1←A0← (A7)
Flags affected	
Interrupts enabled	Yes

Description

Rotates the 8-bit value of the accumulator left by one bit position. This transfers bit 7 of the accumulator to bit 0.

Example 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

Instruction code	1 1 1 1 0 0 0 0 (F0H)
Byte count	1
Cycles	1
Function	←A7←A6←A5←A4←A3←A2←A1←A0←CY← (A7)
Flags affected	CY
Interrupts enabled	Yes

Description

Rotates the 8-bit value of the accumulator left by one bit position through the carry flag. This transfers bit 7 of the accumulator to the carry flag, and the contents of the carry flag to bit 0.

Example 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                            C0H  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

Instruction code	1 1 0 0 0 0 0 0 (C0H)
Byte count	1
Cycles	1
Function	(A0) →A7→A6→A5→A4→A3→A2→A1→A0→
Flags affected	
Interrupts enabled	Yes

Description

Rotates the 8-bit value of the accumulator right by one bit position. This transfers bit 0 of the accumulator to bit 7.

Example 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

Instruction code	1 1 0 1 0 0 0 0 (D0H)
Byte count	1
Cycles	1
Function	(A0) →CY→A7→A6→A5→A4→A3→A2→A1→A0→
Flags affected	CY
Interrupts enabled	Yes

Description

Rotates the 8-bit value of the accumulator right by one bit position through the carry flag. This transfers bit 0 of the accumulator to the carry flag, and the contents of the carry flag to bit 7.

Example 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
```

Data Transfer Instructions

LD d9

Load direct byte to accumulator

Instruction code	`0 0 0 0 0 0 1d8 d8d7d6d5d4d3d2d1d0 (02H to 03H)`
Byte count	2
Cycles	1
Function	(ACC) ← (d9)
Flags affected	
Interrupts enabled	Yes

Description

Transfers the contents of the RAM address or SFR specified by d8 to d0 to the accumulator.

Example 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
```

Example 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

Instruction code	`0 0 0 0 0 1j1j0 (04H to 07H)`
Byte count	1
Cycles	1
Function	(ACC) ← ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Transfers the contents of the RAM address or SFR specified by the indirect address register specified by jl to j0 to the accumulator.

Example 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
```

Example 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 from accumulator

Instruction code	`0 0 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 (12H to 13H)`
Byte count	2
Cycles	1
Function	(d9) ← (ACC)
Flags affected	
Interrupts enabled	Yes

Description

Transfers the contents of the accumulator to the RAM address or special function register (SFR) specified by d8 to d0.

Example 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
```

Example 2

```
                                ACC  B    SP
MOV       #012H,ACC             12H  -    -
MOV       #0F0H,B               12H  F0H  -
MOV       #00FH,SP              12H  F0H  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 from accumulator

Instruction code	`0 0 0 1 0 1j1j0 (14H to 17H)`
Byte count	1
Cycles	1
Function	((Rj)) ← (ACC) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Transfers the contents of the accumulator to the RAM address or special function register (SFR) specified by the indirect address register specified by jl to j0.

Example 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
```

Example 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

Instruction code	0 0 1 0 0 0 1d8 d7d6d5d4d3d2d1d0 i7i6i5i4i3i2i1i0 (22H to 23H)
Byte count	3
Cycles	2
Function	(d9) ← #i8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Transfers immediate data (i7 to i0) to the RAM address or special function register (SFR) specified by d8 to d0.

Example 1

		RAM 00H	RAM 01H	RAM 02H	RAM 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

Example 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

Instruction code	`0 0 1 0 0 1j1j0 i7i6i5i4i3i2i1i0 (24H to 27H)`
Byte count	2
Cycles	1
Function	((Rj)) ← #i8 j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Transfers immediate data (i7 to i0) to the RAM address or special function register (SFR) specified by the indirect address register specified by j1 to j0.

Example 1

		RAM 00H	RAM 01H	RAM 7EH	RAM 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

Example 2

		RAM 02H	RAM 03H	ACC 100H	B 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

Instruction code	`1 1 0 0 0 0 0 1 (C1H)`
Byte count	1
Cycles	2
Function	(ACC) ← (BNK)((TRR) + (ACC)) [ROM]
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Loads into the accumulator the contents of the program memory (ROM) address specified by adding the contents of the accumulator to the contents of the table reference register (TRR). The ROM address accessed is different for a program running in ROM and a program running in flash memory. For a program running in ROM, ROM is accessed, and for a program running in flash memory, bank 0 of flash memory is accessed.

The LDC instruction cannot access bank 1 of flash memory. To access bank 1 of flash memory, use the system BIOS function provided by Visual Memory.

Reference	For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

Example

```
                          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

Instruction code	`0 1 1 0 0 0 0d8 d7d6d5d4d3d2d1d0 (60H to 61H)`
Byte count	2
Cycles	2
Function	(SP) ← (SP) + 1, ((SP)) ← (d9)
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the stack pointer (SP), then transfers the contents of the RAM address or SFR specified by d8 to d0 to the address indicated by the stack pointer.

Note	Even when the application is accessing RAM bank 1, the stack is in RAM bank 0.

Example

		ACC	B	RAM 00H	SP	RAM 20H	RAM 21H	RAM 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

Instruction code	`0 1 1 1 0 0 0d8 d7d6d5d4d3d2d1d0 (70H to 71H)`
Byte count	2
Cycles	2
Function	(d9) ← ((SP)), (SP) ← (SP) - 1
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Transfers the contents of the RAM address indicated by the stack pointer to the RAM address or special function register (SFR) specified by d8 to d0, then decrements the stack pointer.

Note	Even when the application is accessing RAM bank 1, the stack is in RAM bank 0.

Example

		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

Instruction code	`1 1 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 (C2H to C3H)`
Byte count	2
Cycles	1
Function	(ACC) ←→ (d9)
Flags affected	
Interrupts enabled	Yes

Description

Exchanges the contents of the accumulator and the contents of the RAM address or SFR specified by d8 to d0.

Example 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
```

Example 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

Instruction code	`1 1 0 0 0 1j1j0 (C4H to C7H)`
Byte count	1
Cycles	1
Function	(ACC) ←→ ((Rj)) j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes

Description

Exchanges the contents of the accumulator and the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0.

Example 1

```
                                ACC  RAM  RAM
                                     01H  68H
MOV       #0FFH,ACC             FFH  -    -
MOV       #068H,001H            FFH  68H  -
MOV       #0F0H,@R1             FFH  68H  F0H
XCH       @R1                   F0H  68H  FFH
XCH       @R1                   FFH  68H  F0H
XCH       @R1                   F0H  68H  FFH
XCH       @R1                   FFH  68H  F0H
```

Example 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
```

Jump Instructions

JMP a12

Jump near absolute address

Instruction code	0 0 1a11 1a10a9a8 a7a6a5a4a3a2a1a0 (28H to 2FH, 38H to 3FH)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, (PC11 to 00) ← a12
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then transfers the value of a11 to a0 to PC bits 11 to 00.

Example 1

The value of label LA is 0F0EH.

```
                                PC        Instruction code
     NOP                        0FFBH     00H
     NOP                        0FFCH     00H
     JMP       LA               0FFDH     3F0EH
LA:  INC       ACC              0F0EH     6300H
     ROR                        0F10H     C0H
```

Example 2

The value of label LA is 1F0EH.

```
                                PC        Instruction code
     NOP                        0FFCH     00H
     NOP                        0FFDH     00H
     JMP       LA               0FFEH     3F0EH
LA:  INC       ACC              1F0EH     6300H
     ROR                        1F10H     C0H
```

JMPF a16

Jump far absolute address

Instruction code (21H)	0 0 1 0 0 0 0 1 a15a14a13a12a11a10a9a8 a7a6a5a4a3a2a1a0
Byte count	3
Cycles	2
Function	(PC) ← a16
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Transfers the value of a15 to a0 to the program counter (PC).

Example 1

The value of label LA is 0F0EH.

```
                                PC        Instruction code
     NOP                        0FFAH     00H
     NOP                        0FFBH     00H
     JMPF      LA               0FFCH     210F0EH
LA:  INC       ACC              0F0EH     6300H
     ROR                        0F10H     C0H
```

Example 2

The value of label LA is 0F0EH.

```
                                PC        Instruction code
     NOP                        0FFCH     00H
     NOP                        0FFDH     00H
     JMPF      LA               0FFEH     210F0EH
LA:  INC       ACC              0F0EH     6300H
     ROR                        0F10H     C0H
```

BR r8

Branch near relative address

Instruction code	0 0 0 0 0 0 0 1 r7r6r5r4r3r2r1r0 (01H)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then adds the value of r7 to r0 to the PC, leaving the result in PC.

Example 1

The value of label LA is 0F5FH.

```
                                PC        Instruction code
     NOP                        0F1CH     00H
     NOP                        0F1DH     00H
     BR        LA               0F1EH     013FH
LA:  INC       ACC              0F5FH     6300H
     ROR                        0F61H     C0H
```

Example 2

The value of label LA is 1F0EH.

```
                                PC        Instruction code
     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

Instruction code (11H)	0 0 0 1 0 0 0 1 r7r6r5r4r3r2r1r0 r15r14r13r12r11r10r9r8
Byte count	3
Cycles	4
Function	(PC) ← (PC) + 3, (PC) ← (PC) – 1 + r16
Flags affected	
Interrupts enabled	Yes on the 4th cycle

Description

Increments the program counter (PC) three times, then decrements PC and further adds the value of r15 to r0 to PC, leaving the result in PC.

Example 1

The value of label LA is 105FH.

```
                                PC        Instruction code
     NOP                        0F1CH     00H
     NOP                        0F1DH     00H
     BRF        LA              0F1EH     113F01H
LA:  INC        ACC             105FH     6300H
     ROR                        1061H     C0H
```

Example 2

The value of label LA is 1F0EH.

```
                                PC        Instruction code
     NOP                        1FFCH     00H
     NOP                        1FFDH     00H
LA:  INC        ACC             1F0EH     6300H
     ROR                        1F10H     C0H
     NOP                        1F11H     00H
     NOP                        1F12H     00H
     BRF        LA              1F13H     11F8FFH
```

Conditional Branch Instructions

BZ r8

Branch near relative address if accumulator is zero

Instruction code	`1 0 0 0 0 0 0 0 r7r6r5r4r3r2r1r0 (80H)`
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, if (ACC) = 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then if the accumulator is zero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the accumulator is nonzero, continues to the next instruction.

Example 1

When the BZ instruction is executed, the accumulator is zero, so control branches to label LA.

```
                                PC        Instruction code   ACC
     MOV       #000H,ACC        0F1BH     230000H            00H
     BZ        LA               0F1EH     803FH              00H
LA:  INC       ACC              0F5FH     6300H              01H
     ROR                        0F61H     C0H                80H
```

Example 2

When the BZ instruction is executed, the accumulator is nonzero, so control passes to the next instruction.

```
                                PC        Instruction code   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

Instruction code	`1 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 (90H)`
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, if (ACC) ≠ 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then if the accumulator is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the accumulator is zero, continues to the next instruction.

Example 1

When the BNZ instruction is executed, the accumulator is nonzero, so control branches to label LA.

```
                                PC        Instruction code   ACC
     MOV       #001H,ACC        0F1BH     230001H            01H
BNZ  LA        0F1EH            903FH     01H
LA:  INC       ACC              0F5FH     6300H              02H
     ROR                        0F61H     C0H                01H
```

Example 2

When the BNZ instruction is executed, the accumulator is zero, so control passes to the next instruction.

```
                                PC        Instruction code   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 bit is one ("positive")

Instruction code	0 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (68H to 6FH, 78H to 7FH)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 1 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), continues to the next instruction.

Example 1

When the BP instruction is executed, bit 0 of B is 1, so control branches to the label LA.

```
                                PC        Instruction code   B
     MOV       #001H,B          0F1AH     230201H            01H
     BP        B,0,LA           0F1DH     78023FH            01H
LA:  INC       B                0F5FH     6302H              02H
     NOP                        0F61H     00H                02H
```

Example 2

When the BP instruction is executed, bit 0 of the accumulator is 0, so control passes to the next instruction.

```
                                PC        Instruction code   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 bit is one ("positive"), and clear

Instruction code	`0 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (48H to 4FH,` `58H to 5FH)`
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 1 then (PC) ← (PC) + r8, (d9, b3) = 0
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), first clears the bit, then adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), continues to the next instruction.

Example 1

When the BPC instruction is executed, bit 0 of B is 1, so it is cleared, then control branches to the label LA.

```
                                PC        Instruction code   B
     MOV       #003H,B          0F1AH     230203H            03H
     BPC       B,0,LA           0F1DH     58023FH            02H
LA:  INC       B                0F5FH     6302H              03H
     NOP                        0F61H     00H                03H
```

Example 2

When the BPC instruction is executed, bit 0 of the accumulator is 0, so control passes to the next instruction.

```
                                PC        Instruction code   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
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

BN d9, b3, r8

Branch near relative address if direct bit is zero ("negative")

Instruction code	1 0 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (88H to 8FH, 98H to 9FH)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (d9, b3) = 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then if the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is cleared (0), adds the value of r7 to r0 to PC, leaving the result in PC.

If the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0 is set (1), continues to the next instruction.

Example 1

When the BN instruction is executed, bit 0 of B is zero, so control branches to the label LA.

```
                                PC        Instruction code   B
     MOV       #0FEH,B          0F1AH     2302FEH            FEH
     BN        B,0,LA           0F1DH     98023FH            FEH
LA:  INC       B                0F5FH     6302H              FFH
     NOP                        0F61H     00H                FFH
```

Example 2

When the BN instruction is executed, bit 0 of the accumulator is 1, so control passes to the next instruction.

```
                                PC        Instruction code   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 byte is nonzero

Instruction code	`0 1 0 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (52H to 53H)`
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, (d9) = (d9)-1, if (d9) ≠ 0 then (PC) ← (PC) + r8
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then decrements the address in RAM or special function register (SFR) indicated by d8 to d0. Next, if the value of the RAM address or SFR after decrementing is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the value of the RAM address or SFR after decrementing is zero, continues to the next instruction.

Example 1

When the DBNZ instruction is executed, B is decremented, and since B is then nonzero, control branches to the label LA.

```
                                PC        Instruction code   B
     MOV       #002H,B          0F1AH     230202H            02H
     DBNZ      B,LA             0F1DH     53023FH            01H
LA:  INC       B                0F5FH     6302H              02H
     NOP                        0F61H     00H                02H
```

Example 2

When the DBNZ instruction is executed, the accumulator is decremented, and since the accumulator is then zero, control passes to the next instruction.

```
                                PC        Instruction code   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
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

DBNZ @Rj, r8

Decrement indirect byte and branch near relative address if indirect byte is not zero

Instruction code	`0 1 0 1 0 1j1j0 r7r6r5r4r3r2r1r0 (54H to 57H)`
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, ((Rj)) = ((Rj)) - 1, if ((Rj)) ≠ 0 then (PC) ← (PC) + r8 j = 0, 1, 2, 3
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then decrements the address in RAM or special function register (SFR) indicated by the indirect address register specified by j1 to j0. Next, if the value of the RAM address or SFR after decrementing is nonzero, adds the value of r7 to r0 to PC, leaving the result in PC.

If the value of the RAM address or SFR after decrementing is zero, continues to the next instruction.

Example 1

When the DBNZ instruction is executed, B is decremented, and since B is then nonzero, control branches to the label LA.

```
                                PC        Instruction    B    RAM
                                          code
                                                              03H
     MOV       #002H,B          0F18H     230202H        02H  -
     MOV       #002H,003H       0F1BH     220302H        02H  02H
     DBNZ      @R3,LA           0F1EH     573FH          01H  02H
LA:  INC       B                0F5FH     6302H          02H  02H
```

Example 2

When the DBNZ instruction is executed, the accumulator is decremented, and since the accumulator is then zero, control passes to the next instruction.

```
                                PC        Instruction    ACC  RAM
                                          code
                                                              03H
     MOV       #001H,ACC        0F18H     230001H        01H  -
     MOV       #000H,003H       0F1BH     220300H        01H  00H
     DBNZ      @R3,LA           0F1EH     573FH          00H  00H
     DEC       ACC              0F20H     7300H          FFH  00H
LA:  INC       ACC
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

BE _i8, r8

Compare immediate data to accumulator and branch near relative address if equal

Instruction code (31H)	0 0 1 1 0 0 0 1 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (ACC) = #i8 then (PC) ← (PC) + r8
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares immediate data (i7 to i0) with the contents of the accumulator, and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the accumulator is less than the immediate data value, the carry flag is set; if equal or more, the carry flag is cleared.

ACC < #i8 → CY = 1

ACC ≥ #i8 → CY = 0

Example 1

When the BE instruction is executed, ACC=02H, so CY is cleared, and control branches to the label LA.

```
                                PC        Instruction    ACC  CY
                                          code
     MOV       #002H,ACC        0F1AH     230002H        02H  -
     BE        #002H,LA         0F1DH     31023FH        02H  0
LA:  INC       ACC              0F5FH     6300H          03H  0
```

Example 2

When the BE instruction is executed, ACC< 04H, so CY is set, and control passes to the next instruction.

```
                                PC        Instruction    ACC  CY
                                          code
     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

Instruction code	0 0 1 1 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (32H to 33H)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (ACC) = (d9) then (PC) ← (PC) + r8
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by d8 to d0 with the contents of the accumulator, and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the accumulator is less than the contents of the RAM address or SFR specified by d8 to d0, the carry flag is set; if equal or more, the carry flag is cleared.

ACC < d9 (RAM or SFR) → CY = 1

ACC ≥ d9 (RAM or SFR) → CY = 0

Example 1

When the BE instruction is executed, ACC=B, so CY is cleared, and control branches to the label LA.

```
                                PC   Instruction    ACC  B    CY
                                     code
     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
```

Example 2

When the BE instruction is executed, ACC_B, so CY is set, and control passes to the next instruction.

```
                                PC   Instruction    ACC  B    CY
                                     code
     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

Instruction code	`0 0 1 1 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (34H to 37H)`
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if ((Rj)) = #i8 then (PC) ← (PC) + r8 j = 0, 1, 2, 3
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 with immediate data (i7 to i0), and if the values are equal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the RAM address or SFR specified by the indirect address register specified by j1 to j0 is less than the immediate data (i7 to i0), the carry flag is set; if equal or more, the carry flag is cleared.

@Rj < #i8 → CY = 1
@Rj ≥ #i8 → CY = 0

Example 1

When the BE instruction is executed, B=05H, so CY is cleared, and control branches to the label LA.

```
                                PC   Instruction    B    RAM  CY
                                     code
                                                         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
```

Example 2

When the BE instruction is executed, ACC_09H, so CY is set, and control passes to the next instruction.

```
                                PC   Instruction    ACC  RAM  CY
                                     code
                                                         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

Instruction code	0 1 0 0 0 0 0 1 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (41H)
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (ACC) ≠ #i8 then (PC) ← (PC) + r8
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares immediate data (i7 to i0) with the contents of the accumulator, and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are equal, continues to the next instruction. If the value in the accumulator is less than the immediate data value, the carry flag is set; if equal or more, the carry flag is cleared.

ACC < #i8 → CY = 1
ACC ≥ #i8 → CY = 0

Example 1

When the BNE instruction is executed, ACC_00H, so CY is cleared, and control branches to the label LA.

```
                                PC        Instruction    ACC  CY
                                          code
     MOV       #002H,ACC        0F1AH     230002H        02H  -
     BNE       #000H,LA         0F1DH     41003FH        02H  0
LA:  INC       ACC              0F5FH     6300H          03H  0
```

Example 2

When the BNE instruction is executed, ACC=03H, so CY is cleared, and control passes to the next instruction.

```
                                PC        Instruction    ACC  CY
                                          code
     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

Instruction code	`0 1 0 0 0 0 1d8 d7d6d5d4d3d2d1d0 r7r6r5r4r3r2r1r0 (42H to 43H)`
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if (ACC) ≠ (d9) then (PC) ← (PC) + r8
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by d8 to d0 with the contents of the accumulator, and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are equal, continues to the next instruction.

If the value in the accumulator is less than the contents of the RAM address or SFR specified by d8 to d0, the carry flag is set; if equal or more, the carry flag is cleared.

ACC < d9(RAM or SFR) → CY = 1
ACC ≥ d9(RAM or SFR) → CY = 0

Example 1

When the BNE instruction is executed, ACC_B, so CY is set, and control branches to the label LA.

```
                                PC   Instruction    ACC  B    CY
                                     code
     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
```

Example 2

When the BNE instruction is executed, ACC=B, so CY is cleared, and control passes to the next instruction.

```
                                PC   Instruction    ACC  B    CY
                                     code
     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

Instruction code	`0 1 0 0 0 1j1j0 i7i6i5i4i3i2i1i0 r7r6r5r4r3r2r1r0 (44H to 47H)`
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, if ((Rj)) ≠ #i8 then (PC) ← (PC) + r8 j = 0, 1, 2, 3
Flags affected	CY
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then compares the contents of the RAM address or SFR specified by the indirect address register specified by j1 to j0 with immediate data (i7 to i0), and if the values are unequal adds the value of r7 to r0 to PC, leaving the result in PC.

If the values are different, continues to the next instruction.

If the value in the RAM address or SFR specified by the indirect address register specified by j1 to j0 is less than the immediate data (i7 to i0), the carry flag is set; if equal or more, the carry flag is cleared.

@Rj < #i8 → CY = 1
@Rj ≥ #i8 → CY = 0

Example 1

When the BNE instruction is executed, B_08H, so CY is set, and control branches to the label LA.

```
                                PC    Instruction    ACC  B    CY
                                      code
     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
```

Example 2

When the BNE instruction is executed, ACC=03H, so CY is cleared, and control passes to the next instruction.

```
                                PC    Instruction    ACC  RAM  CY
                                      code
                                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
```

Subroutine Instructions

CALL a12

Near absolute subroutine call

Instruction code	0 0 0a11 1a10a9a8 a7a6a5a4a3a2a1a0 (08H to 0FH, 18H to 1FH)
Byte count	2
Cycles	2
Function	(PC) ← (PC) + 2, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0), (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC11 to 0) ← a12
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) twice, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, transfers the value of a11 to a0 to bits 11 to 00 of PC.

Example 1

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM 20H	RAM 21H
	MOV	#01FH,SP	0FFAH	23061FH	1FH	-	-
	CALL	LA	0FFDH	1F0EH	21H	FFH	0FH
LA:	INC	ACC	0F0EH	6300H	21H	FFH	0FH
	RET		0F10H	A0H	1FH	FFH	0FH
	NOP		0FFFH	00H	1FH	FFH	0FH

Example 2

The value of label LA is 1F0EH.

			PC	Instruction code	SP	RAM 20H	RAM 21H
	MOV	#01FH,SP	0FFBH	23061FH	1FH	-	-
	CALL	LA	0FFEH	1F0EH	21H	00H	10H
LA:	INC	ACC	1F0EH	6300H	21H	00H	10H
	RET	1F10H	A0H	1FH	00H	10H	

```
INC       ACC            1000H     6300H     1FH  00H  10H
```

CALLF a16

Far absolute subroutine call

Instruction code (20H)	0 0 1 0 0 0 0 0 a15a14a13a12a11a10a9a8 a7a6a5a4a3a2a1a0
Byte count	3
Cycles	2
Function	(PC) ← (PC) + 3, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0), (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC) ← a16
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Increments the program counter (PC) three times, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, transfers the value of a15 to a0 to bits 15 to 00 of PC.

Example 1

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM 20H	RAM 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

Example 2

The value of label LA is 0F0EH.

			PC	Instruction code	SP	RAM 20H	RAM 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

Instruction code (10H)	0 0 0 1 0 0 0 0 r7r6r5r4r3r2r1r0 r15r14r13r12r11r10r9r8
Byte count	3
Cycles	4
Function	(PC) ← (PC) + 3, (SP) ← (SP) + 1, ((SP)) ← (PC7 to 0), (SP) ← (SP) + 1, ((SP)) ← (PC15 to 8), (PC) ← (PC) - 1 + r16
Flags affected	
Interrupts enabled	Yes on the 4th cycle

Description

Increments the program counter (PC) three times, then increments the stack pointer (SP), stores the lower byte of PC at the RAM address indicated by SP; increments the stack pointer (SP) again, and stores the upper byte of PC at the RAM address indicated by SP; finally, decrements PC_, then adds the value of r15 to r0 to PC, leaving the result in PC.

Example 1

The value of label LA is 1100H.

			PC	Instruction code	SP	RAM 20H	RAM 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

Example 2

The value of label LA is 1100H.

			PC	Instruction code	SP	RAM 20H	RAM 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 from subroutine

Instruction code	1 0 1 0 0 0 0 0 (A0H)
Byte count	1
Cycles	2
Function	(PC15 to 8) ← ((SP)), (SP) ← (SP) - 1, (PC7 to 0) ← ((SP)), (SP) ← (SP) - 1
Flags affected	
Interrupts enabled	Yes on the 2nd cycle

Description

Transfers the contents of RAM indicated by the stack pointer (SP) to the upper byte of the program counter (PC), then decrements SP, then transfers the contents of RAM indicated by the stack pointer (SP) to the lower byte of the program counter (PC), then decrements SP again.

Example 1

The value of label LA is 0F0EH.

			PC	Instruction code		SP	RAM 20H	RAM 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	

Example 2

The value of label LA is 0F0EH.

			PC	Instruction code		SP	RAM 20H	RAM 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

Instruction code	1 0 1 1 0 0 0 0 (B0H)
Byte count	1
Cycles	2
Function	(PC15 to 8) ← ((SP)), (SP) ← (SP) - 1, (PC7 to 0) ← ((SP)), (SP) ← (SP) - 1
Flags affected	
Interrupts enabled	No

Description

Transfers the contents of RAM indicated by the stack pointer (SP) to the upper byte of the program counter (PC), then decrements SP, then transfers the contents of RAM indicated by the stack pointer (SP) to the lower byte of the program counter (PC), then decrements SP again, and resumes the interrupt handling function which was inhibited while handling an interrupt.

Example 1

```
                    PC        Instruction code
NOP                 0FFAH     00H
NOP                 0FFBH     00H
MOV  #001H,ACC      0FFCH     230001H   ← external interrupt 0 occurs
INC  ACC            0003H     6300H
RET1                0005H     B0H
NOP                 0FFFH     00H
```

Example 2

```
                    PC        Instruction code
NOP                 0FFCH     00H
MOV  #00EH,B        0FFDH     23020EH   ← external interrupt 1 occurs
INC  ACC            0013H     6300H
RET1                0015H     B0H
INC  ACC            1000H     6300H
```

Bit Manipulation Instructions

CLR1 d9, b3

Clear direct bit

Instruction code	`1 1 0d8 1b2b1b0 d7d6d5d4d3d2d1d0 (C8H to CFH, D8H to DFH)`
Byte count	2
Cycles	1
Function	(d9, b3) ← 0
Flags affected	
Interrupts enabled	Yes

Description

Clears the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

Example 1

```
                    ACC
MOV  #001H,ACC      01H            0000 0001B
CLR1 ACC,0          00H            0000 0000B
```

Example 2

```
                    RAM
                    7FH
MOV  #001H,07FH     01H            0000 0001B
CLR1 07FH,0         00H            0000 0000B
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

SET1 d9, b3

Set direct bit

Instruction code	`1 1 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 (E8H to EFH, F8H to FFH)`
Byte count	2
Cycles	1
Function	(d9, b3) ← 1
Flags affected	
Interrupts enabled	Yes

Description

Set the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

Example 1

```
                    ACC
MOV  #000H,ACC      00H            0000 0000B
SET1 ACC,7          80H            1000 0000B
```

Example 2

```
                    RAM
                    7FH
MOV  #001H,07FH     01H            0000 0001B
SET1 07FH,6         41H            0100 0001B
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

NOT1 d9, b3

Not direct bit

Instruction code	`1 0 1d8 1b2b1b0 d7d6d5d4d3d2d1d0 (A8H to AFH, B8H to BFH)`
Byte count	2
Cycles	1
Function	(d9, b3) ← (d9, b3)
Flags affected	
Interrupts enabled	Yes

Description

Inverts the bit selected by the bit address b2 to b0 at the address in RAM or special function register (SFR) indicated by d8 to d0.

Example 1

```
                    ACC
MOV  #000H,ACC      00H            0000 0000B
NOT1 ACC,7          80H            1000 0000B
NOT1 ACC,7          00H            0000 0000B
```

Example 2

```
                    RAM
                    7FH
MOV  #001H,07FH     01H            0000 0001B
NOT1 07FH,6         41H            0100 0001B
NOT1 07FH,6         01H            0000 0001B
```

Caution	When this instruction is applied to ports P1 and P3, the latch of each port is selected. The external signal applied to the port is not selected. Even when applied to port P7, there is no change in status.

Miscellaneous Instruction

NOP

No operation

Instruction code	0 0 0 0 0 0 0 0 (00H)
Byte count	1
Cycles	1
Function	
Flags affected	
Interrupts enabled	Yes

Description

Consumes one clock cycle.

Macro Instruction

CHANGE <label (or address)>

Change program mode

Description

Switches between system BIOS in ROM and user program in flash memory.

1) Executed while running program in ROM:
   - Switch from system BIOS to application
   - Set the program counter to the address in flash memory specified by the label or address.

Caution	From the system BIOS, flash memory bank 0 address 0000H is accessed. This address is fixed.

2) Executed while running an application:
   - Switch from application to system BIOS (when LDCEXT=0)
   - Set the program counter to the program address in ROM specified by the label or address.

Caution	When the CHANGE instruction is executed with LDCEXT=1 and while an application is running, a jump to the system BIOS is not carried out. In this case, there is a jump to the address in flash memory specified by the CHANGE instruction.

3) The program mode switch occurs after the dedicated macro instruction has executed.
4) Interrupts are disabled.

Caution	Install the GHEAD.ASM file included with the Visual Memory development set assembler, to call the system BIOS. This calls OS functions without needing to be aware of ROM addresses.

Reference	For details of operating system function calls, refer to the "System BIOS" section of the Visual Memory Hardware Manual.

Chapter 24 LC86K Instruction Set Summary

Mnemonic	Instruction code	Bytes	Cycles	Function	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 d	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
					○		

Mnemonic	Instruction code	Bytes	Cycles	Function	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)	○		○

Mnemonic	Instruction code	Bytes	Cycles	Function	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←			

Mnemonic	Instruction code	Bytes	Cycles	Function	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			

Caution	An asterisk in the "Mnemonic" column for an instruction indicates that in byte or bit addressing the port latch is selected. For these instructions, an external signal supplied to the port is selected.

index

Symbol

.IF 161
.LIST 157
.MACRO 159
.PRINTX 156
.XIF 162
.XLIST 158
.XMACRO 160
@Rj 188
12-bit absolute addressing 186
16-bit absolute addressing 186
16-bit relative addressing 186
8-bit relative addressing 186

_

a12 186
a16 186
AC 183
ADD _i8 192
ADD @Rj 194
Add comment to object file 167
ADD d9 193
ADD direct byte and carry flag to accumulator 196
ADD direct byte to accumulator 193
ADD immediate data and carry flag to accumulator 195
ADD immediate data to accumulator 192
ADD indirect byte and carry flag to accumulator 197
ADD indirect byte to accumulator 194
ADDC _i8 195
ADDC @Rj 197
ADDC d9 196
Addressing 185
AND _i8 210
AND @Rj 212
AND d9 211
AND direct byte to accumulator 211
AND immediate data to accumulator 210
AND indirect byte to accumulator 212
Arithmetic Instructions 183, 192
Assemble if defined 146
Assemble if different 153
Assemble if identical 152
Assemble if nonzero 151
Assemble if operand empty 148
Assemble if operand nonempty 149
Assemble if undefined 147
Assemble if zero 150
Assembler pseudoinstructions 119
Assembler syntax 113
Assign a fixed value 130
Assign temporary value 131
AUTOEXEC.BAT 29
Auxiliary carry flag 183

_

b3 187
BANK 169
BE _i8\, r8 245
BE @Rj\, _i8\, r8 247
BE d9\, r8 246
BE instruction guaranteeing no address error 181
BEO 181
Bit addressing 187
Bit Manipulation Instructions 185, 256
BN d9\, b3\, r8 242
BN instruction guaranteeing no address error 179
BNE _i8\, r8 248
BNE @Rj\, _i8\, r8 250
BNE d9\, r8 249
BNE instruction guaranteeing no address error 182
BNEO 182
BNO 179
BNZ instruction guaranteeing no address error 176
BNZ r8 239
BNZO 176
BP d9\, b3\, r8 240
BP instruction guaranteeing no address error 177
BPC d9\, b3\, r8 241

BPC instruction guaranteeing no address error....178
BPCO....178
BPO....177
BR r8....236
Branch far relative address....237
Branch near relative address....236
Branch near relative address if accumulator is not zero....239
Branch near relative address if accumulator is zero....238
Branch near relative address if direct bit is one ("positive")....240
Branch near relative address if direct bit is one ("positive")\, and clear....241
Branch near relative address if direct bit is zero ("negative")....242
BRF r16....237
BRO....173
Build Priority Sequence....102
BZ r8....238
BZO....175

C

CALL a12....251
CALLF a16....252
CALLO....174
CALLR r16....253
Carry flag....183
CHANGE....170, 260
Change program mode....260
Character Constants....116
character string constants....116
Character string macro....141
CHIP....31, 166
CHIPNAME....31, 41
Clear direct bit....256
CLR1 d9\, b3....256
Commands (MAKE)....104
COMMENT....167
Comments....114
Compare direct byte to accumulator and branch near relative address if equal....246
Compare direct byte to accumulator and branch near relative address if not equal....249
Compare immediate data to accumulator and branch near relative address if equal....245
Compare immediate data to accumulator and branch near relative address if not equal....248
Compare immediate data to indirect byte and branch near relative address if equal....247
Compare immediate data to indirect byte and branch near relative address if not equal....250
Conditional Branch Instructions....184, 238
Cross-reference....36
Cross-Reference....83
CSEG....123
CY....183

D

d9....187
Data Transfer Instructions....184, 223
DB....132
DBNZ @Rj\, r8....244
DBNZ d9\, r8....243
DBNZ instruction guaranteeing no address error....180
DBNZO....180
DC....134
DEC @Rj....207
DEC d9....206
Declare CHANGE instruction jump label 129
Declare external symbol....128
Declare public symbol....126
Declare the beginning of a code segment....123
Declare the beginning of a data segment....124
Decrement direct byte....206
Decrement direct byte and branch near relative address if direct byte is nonzero. 243
Decrement indirect byte....207
Define byte area....135
Define byte data....132
Define character string data....134
Define macro....136
Define word data....133
Dependency rules (MAKE)....103
Direct addressing....187

Directives (MAKE) 107
Display message during assembly 156
DIV 209
Divide accumulator and C register by B register 209
DS 135
DSEG 124
DUMMY.OBJ 62
DW 133

E

ELSE 154
Else case Else case of conditional assembly 154
END 125
End conditional assembly 155
End macro definition 142
End macro expansion 143
End macro expansion listing 160
End program 125
ENDIF 155
ENDM 142
Environment variab 31
EQU 130
EVA 62
EVA format file 87
EVA format file into a HEX format 87
Examples of indirect addressing 189
Exchange direct byte with accumulator ...232
Exchange indirect byte with accumulator 233
EXITM 143
EXTERN 128
External macros (MAKE) 106

F

Far absolute subroutine call 252
Far relative subroutine call 253

G

Generation rules (MAKE) 103
GHEAD.ASM 260

H

H00 87
H2BIN.EXE 87
HEX 87
HEX format file 87

I

IFB 148
IFDEF 146
IFDIF 153
IFE 150
IFIDN 152
IFNB 149
IFNDEF 147
IFNE 151
Implicit rules 108
INC @Rj 205
INC d9 204
INCLUDE 163
Increment direct byte 204
Increment indirect byte 205
Indirect addressing 189
Internal macros 106
IRP 140
IRPC 141
Iteration macro 140

J

JMP a12 234
JMPF a16 235
JMPO 172
Jump between flash memory and ROM .. 170
Jump far absolute address 235
Jump Instruction 184
Jump Instructions 234
Jump near absolute address 234

L

L86K command line option 63
L86K error messages 73
Labels 113
LC86K.LIB 64
LD @Rj 224
LD d9 223
LDC 229
LIB 61
LIB86K 77
library 62
Library manager 78
Linker 62
List macro expansions 159
Listing file format 55
Listing output 157

Listing title 164
Load code byte relative to TRR to accumulator 229
Load direct byte to accumulator 223
Load indirect byte to accumulator 224
LOCAL 144
Logical Instructions 184, 210

M86K command line option 36
M86K errors detected by 43
M86KRSVDFILE 32, 41
M86KWORK.TMP 41
M86KWORKFILE 32, 41
MACRO 136
Macro Instruction 185
Macros (MAKE) 105
MAKE 101
MAKE command line options 101
MAKEFILE 101, 102, 103
Makerule file 108
MAKERULE.DEF 108
MOV _i8\, @Rj 228
MOV _i8\, d9 227
Move immediate data to direct byte 227
Move immediate data to indirect byte 228
MUL 208

Near absolute subroutine CALL 251
New page 165
No operation 259
NOP 259
Not direct bit 258
NOT1 d9\, b3 258
Numeric Constants 114

OBJ 62
object 62
Object alignment 69
Operand 114
Operators 114
Optimized BR instruction 173
Optimized CALL instruction 174
Optimized JMP instruction 172
OR _i8 213
OR @Rj 215
OR d9 214
OR direct byte to accumulator 214
OR immediate data to accumulator 213
OR indirect byte to accumulator 215
ORG 121
OTHER_SIDE_SYMBOL 129
OV 184
Overflow flag 184

PAGE 165
Phantom targets 105
POP d9 231
Pop direct byte from stack 231
Post-Installation Overview 30
PUBLIC 126
PUSH d9 230
Push direct byte to stack 230

r16 186
r8 186
RADIX 171
RAM and special function register (SFR) addressing 187
Repeat macro 139
REPT 139
Reserved word file 42
Reserved word file name 32
RET 254
RETI 255
Return for interrupt 255
Return from subroutine 254
ROL 219
ROLC 220
ROR 221
RORC 222
Rotate accumulator left 219
Rotate accumulator left through the carry flag 220
Rotate accumulator right 221
Rotate accumulator right through the carry flag 222

Select the ROM to hold code 122
SET 131
SET command 32

Set direct bit 257
SET1 d9\, b3 257
Shinobi library 95
Skipped statements, end listing 162
Skipped statements, listing 161
source file 113
Special symbols 117
Specify chip for assembly 166
Specify default radix 171
Specify origin 121
Specify RAM bank 169
ST @Rj 226
ST d9 225
Statements 113
Store direct byte from accumulator 225
Store indirect byte from accumulator 226
SUB _i8 198
SUB @Rj 200
SUB d9 199
SUBC _i8 201
SUBC @Rj 203
SUBC d9 202
Subroutine Instruction 185
Subroutine Instructions 251
Subtract direct byte from accumulator 199
Subtract immediate data and carry flag from accumulator 201
Subtract immediate data from accumulator 198
Subtract indirect byte and carry flag from accumulator 203
Subtract indirect byte from accumulator 200
Suppress listing 158
Symbol 113

_

Table jumps 186
TITLE 164
TMP 32, 41
TOOL86K 32

_

User macro (MAKE) 106

_

Visual Memory, transferred to 95
VM_SDK 23

_

WIDTH 168
WORLD 122

_

XCH @Rj 233
XCH d9 232
XOR _i8 216
XOR @Rj 218
XOR d9 217
XOR direct byte to accumulator 217
XOR immediate data to accumulator 216
XOR indirect byte to accumulator 218

_

Z instruction guaranteeing no address error 175