

F²MC-8L/8FX FAMILY
SOFTUNE™ ASSEMBLER
MANUAL
for V3

F²MC-8L/8FX FAMILY
SOFTUNETM ASSEMBLER
MANUAL
for V3

FUJITSU LIMITED

PREFACE

■ Purpose of this manual and target readers

This manual describes the functions and operations of the Fujitsu SOFTUNE Assembler.

This manual is intended for engineers who develop application programs using the F²MC-8L/8FX family microcontroller.

Note: F²MC is the abbreviation of FUJITSU Flexible Microcontroller.

■ Trademarks

SOFTUNE is a trademark of FUJITSU LIMITED.

Microsoft, Windows and Windows Media are either registered trademarks of Microsoft Corporation in the United States and/or other countries.

Other company names and products names are trademarks or registered trademarks of their respective companies.

■ Configuration of This Manual

This manual consists of two parts and an appendix.

PART I OPERATION

PART I explains how to use the SOFTUNE assembler.

PART II SYNTAX

PART II describes the syntax and format for writing assembly source programs.

APPENDIX

The appendix explain error messages and note restrictions that must be observed.

- The contents of this document are subject to change without notice.
Customers are advised to consult with sales representatives before ordering.
- The information, such as descriptions of function and application circuit examples, in this document are presented solely for the purpose of reference to show examples of operations and uses of FUJITSU semiconductor device; FUJITSU does not warrant proper operation of the device with respect to use based on such information. When you develop equipment incorporating the device based on such information, you must assume any responsibility arising out of such use of the information. FUJITSU assumes no liability for any damages whatsoever arising out of the use of the information.
- Any information in this document, including descriptions of function and schematic diagrams, shall not be construed as license of the use or exercise of any intellectual property right, such as patent right or copyright, or any other right of FUJITSU or any third party or does FUJITSU warrant non-infringement of any third-party's intellectual property right or other right by using such information. FUJITSU assumes no liability for any infringement of the intellectual property rights or other rights of third parties which would result from the use of information contained herein.
- The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for use requiring extremely high reliability (i.e., submersible repeater and artificial satellite).
Please note that FUJITSU will not be liable against you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products.
- Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions.
- Exportation/release of any products described in this document may require necessary procedures in accordance with the regulations of the Foreign Exchange and Foreign Trade Control Law of Japan and/or US export control laws.
- The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

CONTENTS

PART I	OPERATION	1
CHAPTER 1	OVERVIEW	3
1.1	SOFTUNE Assembler	4
1.2	Assembler Syntax	5
CHAPTER 2	ENVIRONMENT VARIABLES AND DIRECTORY STRUCTURE OF THE DEVELOPMENT ENVIRONMENT	7
2.1	FETOOL	8
2.2	FELANG	9
2.3	TMP	10
2.4	INC896	11
2.5	OPT896	12
2.6	Directory Structure of the Development Environment	13
CHAPTER 3	STARTUP METHOD	15
3.1	fasm896s Commands	16
3.2	Specifying a File	17
3.3	Handling of File Names	18
3.3.1	Format for Specifying a File Name	19
3.3.2	Specifying a File Name with File Name Components Omitted	20
3.4	Option File	21
3.5	Comments Described in an Option File	22
3.6	Default Option File	23
3.7	Termination Code	24
CHAPTER 4	STARTUP OPTIONS	25
4.1	Rules for Startup Options	26
4.2	Startup Option List	27
4.3	Details of the Startup Options	29
4.4	Options Related to Objects and Debugging	30
4.4.1	-o, -Xo	31
4.4.2	-g, -Xg	32
4.5	Options Related to Listing	33
4.5.1	-l, -lf, -Xl	34
4.5.2	-pl, -pw	35
4.5.3	-linf, -lsrc, -lsec, -lcros	36
4.5.4	-linc, -lexp	38
4.5.5	-tab	39
4.6	Options Related to the Preprocessor	40
4.6.1	-p	41
4.6.2	-P, -Pf	42

4.6.3	-D, -U	43
4.6.4	-I	44
4.6.5	-H	45
4.6.6	-C	46
4.7	Target-Dependent Options	47
4.7.1	-sa, -Xsa	48
4.7.2	-div_check, -Xdiv_check	49
4.8	Other Options	50
4.8.1	-Xdof	51
4.8.2	-f	52
4.8.3	-w	53
4.8.4	-name	54
4.8.5	-V, -XV	55
4.8.6	-cmsg, -Xcmsg	56
4.8.7	-cwno, -Xcwno	57
4.8.8	-help	58
4.8.9	-cpu	59
4.8.10	-kanji	60
4.8.11	-LBA, -XLBA	61
4.8.12	-OVFW, -XOVFW	62
4.8.13	-cif	64

CHAPTER 5 OPTIMIZATION CODE CHECK FUNCTIONS 65

5.1	Optimization Code Check Functions of fasm896s	66
-----	---	----

CHAPTER 6 ASSEMBLY LIST 69

6.1	Composition	70
6.2	Page Format	72
6.3	Information List	74
6.4	Source List	76
6.4.1	Preprocessor Operations	77
6.4.2	Error Display	78
6.4.3	Include File	79
6.4.4	.END, .PROGRAM, .SECTION	80
6.4.5	.ALIGN, .ORG, .SKIP	82
6.4.6	.EXPORT, .GLOBAL, .IMPORT	83
6.4.7	.EQU	84
6.4.8	.DATA, .BIT, .BYTE, .HALF, .EXTEND, .LONG, .WORD, .DATAB	85
6.4.9	.FDATA, .FLOAT, .DOUBLE, .FDATAB	87
6.4.10	.RES, .FRES	88
6.4.11	.SDATA, .ASCII, .SDATAB	89
6.4.12	.DEBUG	90
6.4.13	.LIBRARY	91
6.4.14	.FORM, .TITLE, .HEADING, .LIST, .PAGE, .SPACE	92
6.5	Section List	95
6.6	Cross-reference List	96

PART II SYNTAX	97
CHAPTER 7 BASIC LANGUAGE RULES	99
7.1 Statement Format	100
7.2 Character Set	102
7.3 Names	103
7.4 Forward Reference Symbols and Backward Reference Symbols	105
7.5 Integer Constants	106
7.6 Location Counter Symbols	107
7.7 Character Constants	108
7.8 Strings	110
7.9 Floating-Point Constants	111
7.10 Data Formats of Floating-Point Constants	113
7.11 Expressions	115
7.11.1 Terms	117
7.11.2 Range of Operand Values	121
7.11.3 Operators	122
7.11.4 Values Calculated from Names	124
7.11.5 Precedence of Operators	127
7.12 Comments	128
CHAPTER 8 SECTIONS	129
8.1 Section Description Format	130
8.2 Section Types	132
8.3 Section Types and Attributes	135
8.4 Section Allocation Patterns	136
8.5 Section Linkage Methods	138
8.6 Multiple Descriptions of a Section	140
8.7 Setting ROM Storage Sections	141
CHAPTER 9 MACHINE INSTRUCTIONS	143
9.1 Machine Instruction Format	144
9.2 Operand Field Format	145
CHAPTER 10 ASSEMBLER PSEUDO-INSTRUCTIONS	147
10.1 Scope of Integer Constants Handled by Pseudo-Instructions	148
10.2 Program Structure Definition Instructions	149
10.2.1 .PROGRAM Instruction	150
10.2.2 .END Instruction	151
10.2.3 .SECTION Instruction	152
10.3 Address Control Instructions	154
10.3.1 .ALIGN Instruction	155
10.3.2 .ORG Instruction	156
10.3.3 .SKIP Instruction	157
10.4 Program Linkage Instructions	158
10.4.1 .EXPORT Instruction	159
10.4.2 .GLOBAL Instruction	160

10.4.3	.IMPORT Instruction	161
10.5	Symbol Definition Instructions	162
10.5.1	.EQU Instruction	163
10.6	Area Definition Instructions	164
10.6.1	.DATA, .BIT, .BYTE, .HALF, .LONG, and .WORD Instructions	165
10.6.2	.DATAB Instruction	167
10.6.3	.FDATA, .FLOAT, and .DOUBLE Instructions	168
10.6.4	.FDATAB Instruction	170
10.6.5	.RES Instruction	171
10.6.6	.FRES Instruction	172
10.6.7	.SDATA and .ASCII Instructions	173
10.6.8	.SDATAB Instruction	174
10.6.9	.STRUCT and .ENDS Instructions	175
10.7	Debugging Information Output Control Instruction	177
10.8	Library File Specification Instruction	178
10.9	List Output Control Instructions	179
10.9.1	.FORM Instruction	180
10.9.2	.TITLE Instruction	182
10.9.3	.HEADING Instruction	183
10.9.4	.LIST Instruction	184
10.9.5	.PAGE Instruction	187
10.9.6	.SPACE Instruction	188

CHAPTER 11 PREPROCESSOR PROCESSING 189

11.1	Preprocessor	191
11.2	Basic Preprocessor Rules	193
11.2.1	Preprocessor Instruction Format	195
11.2.2	Comments	196
11.2.3	Continuation of a Line	197
11.2.4	Integer Constants	198
11.2.5	Character Constants	199
11.2.6	Macro Names	202
11.2.7	Formal Arguments	203
11.2.8	Local Symbols	204
11.3	Preprocessor Expressions	205
11.4	Macro Definitions	208
11.4.1	#macro Instruction	209
11.4.2	#local Instruction	210
11.4.3	#exitm Instruction	211
11.4.4	#endm Instruction	212
11.5	Macro Call Instructions	213
11.6	Repeat Expansion	214
11.7	Conditional Assembly Instructions	216
11.7.1	#if Instruction	217
11.7.2	#ifdef Instruction	218
11.7.3	#ifndef Instruction	219
11.7.4	#else Instruction	220

11.7.5	#elif Instruction	221
11.7.6	#endif Instruction	223
11.8	Macro Name Replacement	224
11.8.1	#define Instruction	225
11.8.2	Replacing Formal Macro Arguments by Character Strings (# Operator)	227
11.8.3	Concatenating the Characters to be Replaced by Macro Replacement (## Operator)	228
11.8.4	#set Instruction	229
11.8.5	#undef Instruction	230
11.8.6	#purge Instruction	231
11.9	#include Instruction	232
11.10	#line Instruction	234
11.11	#error Instruction	235
11.12	#pragma Instruction	236
11.13	No-operation Instruction	237
11.14	Defined Macro Names	238
11.15	Differences from the C Preprocessor	240
CHAPTER 12	ASSEMBLER PSEUDO- MACHINE INSTRUCTIONS	241
12.1	Assembler Pseudo Machine Instructions for the F ² MC-8L/8FX Family	242
12.1.1	Branch Pseudo Machine Instructions	243
12.1.2	Operation Pseudo Machine Instructions	244
12.1.3	Miscellaneous Pseudo Machine Instructions	245
12.1.4	Optimum Allocation Branch Pseudo Machine Instructions	247
CHAPTER 13	STRUCTURED INSTRUCTIONS	251
13.1	Overview of Structured Instructions	252
13.2	Structured Program Instructions	253
13.2.1	Conditional Expressions in a Structured Program Instruction	254
13.2.2	Generation Rules for Structured Program Instructions	257
13.2.3	Format for Structured Program Instructions	258
13.2.4	2-process Selection Syntax	259
13.2.5	Multiple-Process Selection Syntax	260
13.2.6	Computed Repetition Syntax	262
13.2.7	At-end Condition Repetition Syntax	264
13.2.8	Execution Condition Repetition Syntax	265
13.2.9	Control Transfer Instructions	266
13.3	Expressions (Assignment Expressions, Operation and Assignment Expressions, Increment/Decrement Expressions)	268
13.3.1	Format for Expressions	269
13.3.2	Assignment Expressions	270
13.3.3	Operation and Assignment Expressions	272
13.3.4	Increment/Decrement Expressions	275
APPENDIX	277
APPENDIX A	Error Messages	278
APPENDIX B	Restrictions	302
APPENDIX C	The Acquisition Method of an Extended Direct Access Area Bank Number	303

APPENDIX D Difference in Specification of SOFTUNE Assembler (FASM896S) and Old Assembler (ASM96)

.....	304
D.1 Assembler Language Basic Rules	305
D.2 Expression Processing	309
D.3 Pseudo Instruction	312
D.4 Macro Processing	315
D.5 Structured Control Instruction	316
D.6 Machine Instruction	317
D.7 Difference in Command Line	318
D.8 Environmental Variables	319
D.9 Options	320
D.10 Alleviation of Restrictions	322

INDEX.....	323
-------------------	------------

PART I OPERATION

PART I explains how to use the **SOFTUNE** assembler.

CHAPTER 1 OVERVIEW

CHAPTER 2 ENVIRONMENT VARIABLES AND DIRECTORY STRUCTURE OF THE
DEVELOPMENT ENVIRONMENT

CHAPTER 3 STARTUP METHOD

CHAPTER 4 STARTUP OPTIONS

CHAPTER 5 OPTIMIZATION CODE CHECK FUNCTIONS

CHAPTER 6 ASSEMBLY LIST

CHAPTER 1

OVERVIEW

This chapter provides an overview of the SOFTUNE assembler.

1.1 SOFTUNE Assembler

1.2 Assembler Syntax

1.1 SOFTUNE Assembler

The SOFTUNE assembler (hereafter the assembler) assembles source programs written in assembly language for the F²MC-8L/8FX families. The assembler processing consists of two phases: the preprocessor phase and the assembly phases, also outputs relocatable objects and assembly lists.

■ Overview (SOFTUNE Assembler)

The SOFTUNE assembler assembles source programs written in assembly language for the F²MC-8L/8FX families. The assembler also outputs relocatable objects and assembly lists.

Assembler processing consists of two phases: the preprocessor phase and the assembly phases.

Figure 1.1-1 shows the assembler configuration.

■ Preprocessor Phase

In this phase, the assembler performs preprocessing. Preprocessing refers to text-related processing such as macro definition or expansion.

Since the assembler supports the C language preprocessor specification as one of its function specifications, the header file can be shared with C language when preprocessor processing is performed.

Only C language preprocessor instructions and comments description delimited by /* and */ can be shared with C language.

The assembler also supports assembler-specific functions, including macro definition and expansion.

The results of preprocessing can be stored a file.

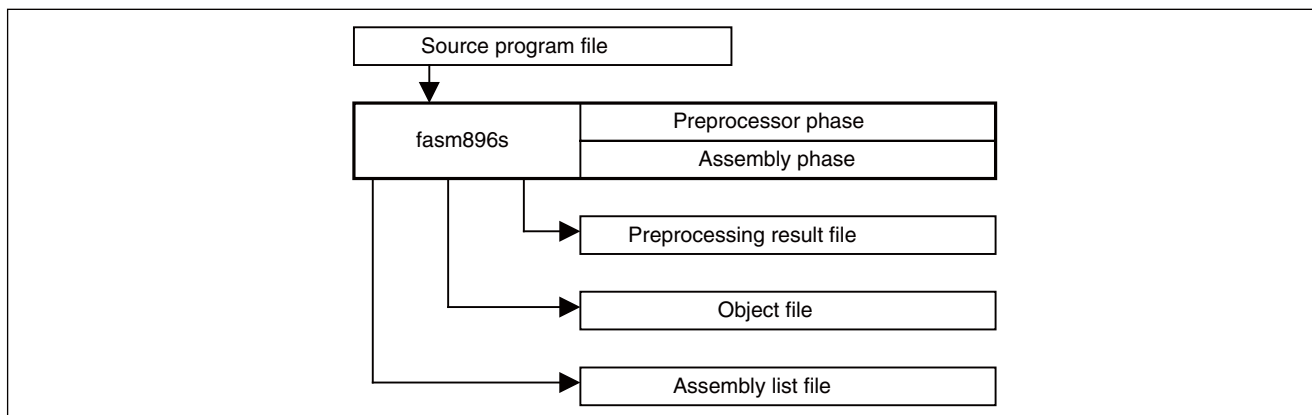
■ Assembly Phase

In this phase, the assembler translates machine instructions and pseudo-instructions to generate object code.

The following functions are supported in the assembly phase.

- Comment description shared with C
- Debugging information output
- Optimization check function for machine instructions

Figure 1.1-1 Assembler Configuration



1.2 Assembler Syntax

The assembler supports extended functions that facilitate the user programming, as well as a language specification that complies with IEEE-649 specifications.

■ Overview (Assembler Syntax)

The assembler supports the following four functions, in addition to a language specification that complies with IEEE-649 specifications:

● Comment description shared with C language

Even though the assembler is being used, comments can be inserted in the same way as with C.

[Example]

```
/* -----
   Main processing
   ----- */
.SECTION CODE, CODE, ALIGN=2
CALL  _init      /* Initialization processing */
```

● Assembler pseudo-instructions

Assembler functions has been expanded with the addition of list control instructions and area definition instructions, as well as assembler pseudo-instructions complying with IEEE-649 specifications.

● Preprocessor processing

The assembler supports the C language preprocessor specification.

Therefore the header file can be shared with C language when preprocessor processing is performed.

Only C language preprocessor instructions and comments description delimited by `/*` to `*/` can also be used in C language.

In addition, the assembler also supports assembler-specific functions, including macro definition and expansion.

[Example]

```
#ifdef    SPC_MODE
#include  "spc.h"
        :
#endif
#define   SIZE_MAX    256    /* Maximum size */
```

● Debugging information output

Debugging information can be included in an object.

This function is required for debugging a program.

CHAPTER 2

ENVIRONMENT VARIABLES AND DIRECTORY STRUCTURE OF THE DEVELOPMENT ENVIRONMENT

This chapter describes environment variables used with the assembler and the directory structure of the development environment.

2.1 FETOOL

2.2 FELANG

2.3 TMP

2.4 INC896

2.5 OPT896

2.6 Directory Structure of the Development Environment

2.1 FETOOL

FETOOL specifies the directory in which the development environment is to be installed.

If this environment variable is not set, the system assumes the parent directory of the directory that contains the assembler that has been started.

■ FETOOL

[Format]

SET FETOOL=directory

[Description]

FETOOL specifies the directory in which the development environment is to be installed.

The files required for the development environment, such as message files, include files, and library files, are accessed in this directory.

For details, of the directory structure of the development environment, see Section "2.6 Directory Structure of the Development Environment".

If FETOOL is not set, the system assumes the parent directory of the directory (location-of-directory-containing-assembler\..) that contains the assembler that has been started.

[Example]

SET FETOOL=D:\SOFTUNE

2.2 FELANG

**FELANG specifies the format in which messages are output.
This environment variable can be omitted.**

■ FELANG

[Format]

`SET FELANG={ ASCII|EUC|SJIS }`

[Description]

FELANG specifies the format in which messages are output.

● When ASCII is specified

Messages are encoded in ASCII.

Messages are output in English.

Specify this format for a system that does not support Japanese-language environment.

● When EUC is specified

Messages are encoded in EUC.

Messages are output in Japanese.

Specify this format for an EUC terminal.

● When SJIS is specified

Messages are encoded in SJIS.

Messages are output in Japanese.

Specify this format for an SJIS terminal.

When SJIS is specified in a Windows environment, Japanese messages are usually displayed.

Reference:

This specification of FELANG also applies to help messages and error messages.

This environment variable can be omitted.

The default output format is ASCII.

[Example]

`SET FELANG=ASCII`

2.3 TMP

**TMP specifies a work directory used by the assembler.
This environment variable can be omitted.**

■ TMP

[Format]

SET TMP=directory

[Description]

TMP specifies a work directory used by the assembler.

If a directory that cannot be accessed is specified, the assembler terminates abnormally.

This environment variable can be omitted.

The default work directory is the current directory.

[Example]

SET TMP=D:\TMP

2.4 INC896

INC896 specify an include path.

Specify a path to search for an include file specified using the #include instruction.

This environment variable can be omitted.

■ INC896

[Format]

SET INC896=path

[Description]

INC896 specify an include path.

Specify a path to search for an include file specified using the #include instruction.

The system first searches the path that has been specified using the include path specification for startup option (-I). If no include file is found, the system then searches the path set for INC896.

This environment variable can be omitted.

[Example]

SET INC896=E:\INCLUDE

2.5 OPT896

**OPT896 specify the directory that contains the default option file.
This environment variable can be omitted.**

■ OPT896

[Format]

SET OPT896=directory

[Description]

OPT896 specify the directory that contains the default option file.

For details, the default option file, see Section "3.6 Default Option File".

These environment variables can be omitted.

If these variables are not omitted, a default option file under the development environment directory is accessed.

The default option files under the development environment directory are as follows:

%FETOOL%\LIB\896\FASM896.OPT

[Example]

SET OPT896=D:\USR

2.6 Directory Structure of the Development Environment

This section describes the directory structure of the development environment.

■ Directory Structure of the Development Environment

The development environment consists of the following structures, each directory includes the following files.

- %FETOOL%\BIN

This is a load module directory.

This directory contains the C compiler, assembler, linker, and workbench.

- %FETOOL%\LIB

This is a library directory.

This directory contains appended files such as libraries.

- %FETOOL%\LIB\896

1. These directories contain the F²MC-8L/8FX libraries.
2. These directories contain message files, library files, and include files.

- %FETOOL%\LIB\896\INCLUDE

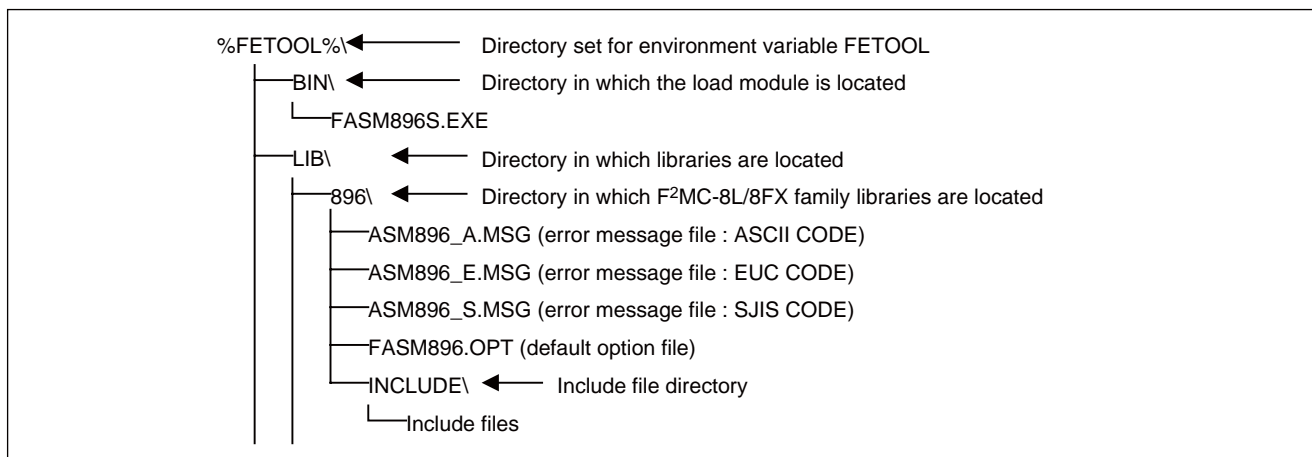
These directories contain the F²MC-8L/8FX include files.

These directories contain the standard C compiler include file.

These directories are searched last in a search using the #include instruction.

The directory structure for the development environment is as follows:

Figure 2.6-1 Directory Structure for Development Environment



CHAPTER 3

STARTUP METHOD

**This chapter describes how to start the assembler.
The assembler startup commands are as follows:**

- **fas²m896s:** F²MC-8L/8FX family assembler

3.1 fasm896s Commands

3.2 Specifying a File

3.3 Handling of File Names

3.4 Option File

3.5 Comments Described in an Option File

3.6 Default Option File

3.7 Termination Code

3.1 fasm896s Commands

The fasm896s commands are described in the following format:

- fasm896s [startup-option] ... [file-name]
-

■ fasm896s Command Lines

[Format]

fasm896s [startup-option] ... [file-name]

[Description]

A startup option and a file name can be specified on the fasm896s command line.

A startup option and a file name can be specified at any position on the command line.

More than one startup option can be specified.

A startup option and a file name are delimited with a space.

The fasm896s commands identify a startup option and a file name in the following steps:

- 1 Any character string prefixed with the option symbol is assumed to be a startup option. The option symbol is a hyphen (-).
- 2 If a startup option is accompanied by an argument, a character that follows the startup option is assumed to be the argument.
- 3 Any character string that is not a startup option is assumed to be a file name.

For details of the startup options, see "CHAPTER 4 STARTUP OPTIONS".

If "-f option-file-name" is specified as a startup option, the system reads the file specified by -f and executes the fasm896s command described in the file.

This function allows the fasm896s commands to be stored in a file.

For details, see Section "3.4 Option File".

If a startup option and a file name are omitted and nothing is specified after the fasm896s command, a startup option list (help message) is output.

The fasm896s commands support the default option file function.

The fasm896s option described in the default option file is executed first.

For details, see Section "3.6 Default Option File".

[Example]

```
fasm896s -f def.opt -l -pw 80 prog.asm
```

3.2 Specifying a File

Specify an assembly source file.

Only a single assembly source file can be specified.

If the file extension is omitted, the system appends ".asm" to the file name.

■ Specifying a File

[Example]

File specification	File to be assembled
fasm896s test	test.asm
fasm896s test.	test.
fasm896s D:\WORK\test	D:\WORK\test.asm
fasm896s . . \FMC8L\abc.src	. . \FMC8L\abc.src

Note:

For an explanation of how to write a file, see the relevant OS manual.

3.3 Handling of File Names

This section describes how to handle file names in the assembler.

This section covers the following two items:

- **Format for specifying a file name**
 - **Specifying a file name with components omitted**
-

■ Format for Specifying a File Name

The assembler assumes that a file name consists of these three parts: <path-name >, <primary-file-name >, and <file-extension >.

<file-extension > refers to the characters that follow the period (.).

<path-name > and <file-extension > can be omitted.

■ Specifying a File Name with Components Omitted

This section explains how to handle a file name in the assembler when file name components are omitted.

3.3.1 Format for Specifying a File Name

This section describes the format for specifying a file name.

The assembler assumes that a file name consists of these three parts: <path-name >, <primary-file-name >, and <file-extension >.

<file-extension > indicates characters that follow the period (.).

<path-name > and <file-extension > can be omitted.

■ Format for Specifying a File Name

[Format]

```
"<path-name >" <primary-file-name > "<file-extension >"
```

[Description]

For an explanation of file names, see the relevant OS manual.

The assembler assumes that a file name consists of these three parts: <path-name >, <primary-file-name >, and <file-extension >.

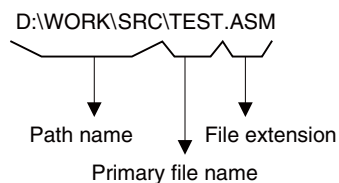
<file-extension > indicates the characters that follow the period (.).

<path-name > and <file-extension > can be omitted.

For an explanation of handling file names when file name components are omitted, see Section "3.3.2 Specifying a File Name with File Name Components Omitted".

When the assembler for Windows is used, <drive-name > is included in <path-name >.

[Example]



3.3.2 Specifying a File Name with File Name Components Omitted

This section describes the handling of file names when file name components are omitted.

■ Specifying a File Name with Components Omitted

This section describes the handling of file names when file name components are omitted.

When only a path name is used to specify an object file or list file, the primary file name of the source file is used.

- When only a path name is specified

<specified-path-name > <primary-file-name-of-source-file > <default-file-extension >

Table 3.3-1 shows how the system handles a file name when file name components are omitted.

Table 3.3-1 Handling of Omitted File Name

Omitted component		Value used
Path name		Current
File-Extension	Source for file-extension	.asm
	Object for file-extension	.obj
	List for file-extension	.lst
	Option for file-extension	.opt
	Preprocessing result for file-extension	.as

[Example]

```
fasm896s TEST -o D:\WORK\SRC -lf abc
```

Source file name: TEST.asm

Object file name: D:\WORK\SRC\TEST.obj

List file name: abc.lst

3.4 Option File

The option file function is used to describe the fasm896s option in a file so that they can be specified as a batch. With this function, frequently specified startup options can be stored in a file.

To specify an option file, use the **-f** startup option.

■ Option File

[Format]

```
-f option-file-name
```

[Description]

The option file function is used to describe the fasm896s option in a file so that they can be specified as a batch. With this function, frequently specified startup options can be stored in a file.

To specify an option file, use the **-f** startup option.

If the file extension is omitted from the option file name, the system appends ".opt" to the file name.

fasm896s option can be described in an option file.

Comments description can be placed in an option file.

For details, see Section "3.5 Comments Described in an Option File".

An option file can be nested up to eight levels deep.

[Example]

option file : def.opt

```
-I D:\usr\include
-D SMAP
-I
```

```
fasm896s -V -f def.opt test
```

The above example is interpreted as follows:

```
fasm896s -V -I D:\usr\include -D SMAP -I test
```

3.5 Comments Described in an Option File

A comment can start at any column.

A comment starts with a number sign (#) and continues to the end of a line.

■ Comments Described in an Option File

[Format]

```
# Comment
```

Comments can be also used in the following formats:

```
/* Comment */  
// Comment  
; Comment
```

[Description]

A comment can start at any column.

A comment starts with a number sign (#) and continues to the end of a line.

[Example]

```
#  
# F2MC8L customization option  
#  
# Include path  
#  
-I D:\usr\test\include # Test include  
#  
# Define  
#  
-D SMAP  
-D VER=2
```

3.6 Default Option File

A default option file is supported as one of the option file functions. If an option file is to be used but the -f startup option is not specified, a predetermined option file is read and executed.

This file is called the default option file.

■ Default Option File

The default option file is always read when the assembler is started. The startup options suitable to the user environment can be specified beforehand.

To suppress the default option file function, specify the -Xdof startup option on the command line.

When this option is specified, the default option file is not read.

The default option file name is predetermined as follows:

Command name	Default option file name
fasm896s	FASM896.OPT

The default option file is referenced as described below.

● When environment variable OPT896 is set

The system references the file under the directory set for environment variable OPT896.

%OPT896%\FASM896.OPT

● When environment variable OPT896 is not set

The system references the default option file under the development environment directory.

%FETOOL%\LIB\896\FASM896.OPT

The default option file is not always required.

3.7 Termination Code

A termination code is output when the assembler terminates processing and returns control to the OS.

■ Termination Code

A termination code is output when the assembler terminates processing and returns control to the OS.

The value of this code informs the user of the approximate processing status of the assembler.

Table 3.7-1 lists the termination codes.

Table 3.7-1 Termination Codes

Processing status	Termination code
Normal termination	0
Warning	0 or 1
Error	2
Abnormal termination	3

Notes:

- The termination code output when a warning occurs varies with the specification of the -cwno or -Xcwno option. For details, see Section "4.8.7 -cwno, -Xcwno".
 - If a warning and an error occur simultaneously, a termination code is returned for the error.
 - If an error occurs, no object file is created.
-

CHAPTER 4

STARTUP OPTIONS

**This chapter explains the assembler startup options.
The startup options can control assembly processing.
The startup options are identified by an option
identification symbol.
The option identification symbol is a hyphen (-).**

- 4.1 Rules for Startup Options
- 4.2 Startup Option List
- 4.3 Details of the Startup Options
- 4.4 Options Related to Objects and Debugging
- 4.5 Options Related to Listing
- 4.6 Options Related to the Preprocessor
- 4.7 Target-Dependent Options
- 4.8 Other Options

4.1 Rules for Startup Options

This section describes the rules for startup options.

■ Rules for Startup Options

The specifications for overall startup options are given below.

From this point, a startup option is simply referred to as an option.

● Specifying a single option more than once

If an option is specified more than once, the one specified last is used.

[Example]

```
fasm896s -o abc test.asm -o def
```

The system uses "-o def", thus creating an object file named def.obj.

● Options that can be specified more than once

- -D name[=def]: Specifies a macro name.
- -U name: Cancels a macro name.
- -I path: Specifies an include path.
- -f filename: Specifies an option file.

The above options can be specified more than once. Each specification is valid.

● Positioning of options

The position in which an option is specified has no special meaning. An option has the same meaning no matter where it is specified on the command line.

[Example]

```
1 fasm896s -C -name prog test.asm -l
```

```
2 fasm896s test.asm -l -name prog -C
```

The assembler performs the same processing in both 1) and 2).

● Mutually exclusive and dependent relationships

Some options have either mutually exclusive or dependent relationships. If such types of options are specified simultaneously, the one specified last is valid.

[Example]

```
fasm896s -lf t1 test.asm -Xl
```

The system accepts -Xl, but does not create a list file.

4.2 Startup Option List

List the startup options show in Table 4.2-1 .

■ Startup Options

Table 4.2-1 Startup Options (1 / 2)

Specification format	Function overview	Initial value
Options related to objects and debugging		
-o [filename]	Specifies an object file name.	Output
-Xo	Creates no object file.	
-g	Outputs debugging information.	Not output
-Xg	Cancels the output of debugging information.	
Options related to listing		
-l	Outputs a list file.	Not output
-lf filename	Outputs a list file (with a file name specified).	
-Xl	Cancels the output of a list file.	
-pl {0 20-255}	Specifies the number of lines on a list page.	60
-pw {80-1023}	Specifies the number of columns in a list line.	100
-linf {ON OFF}	Outputs an information list.	ON
-lsrc {ON OFF}	Outputs a source list.	ON
-lsec {ON OFF}	Outputs a section list.	ON
-lcros {ON OFF}	Outputs a cross-reference list.	OFF
-linc {ON OFF}	Outputs an include file list.	ON
-lexp {ON OFF OBJ}	Outputs a macro expansion section to a list.	OBJ
-tab {0-32}	Specifies the number of tab expansion characters	8
Options related to the preprocessor		
-p	Specifies not to start the preprocessor.	Start
-P	Starts only the preprocessor.	
-Pf filename	Starts only the preprocessor (with a file name specified).	
-D name[=def]	Specifies a macro name.	
-U name	Cancels a macro name.	

Table 4.2-1 Startup Options (2 / 2)

Specification format	Function overview	Initial value
-I path	Specifies an include path.	
-H	Outputs an include path.	Not output
-C	Specifies to leave comments in the preprocessor output.	Comments are not left.
Target-dependent options		
-sa	Outputs an accumulator protection code.	Not output
-Xsa	Suppresses output of an accumulator protection code.	
-div_check	A warning message is outputted to a DIVU instruction.	F ² MC 8L : not output
-Xdiv_check	A warning message is not outputted to a DIVU instruction.	F ² MC 8FX: output
Other options		
-Xdof	Cancels a default option file.	
-f filename	Specifies an option file.	
-w [0-3]	Specifies the output level of warning messages.	2
-name module-name	Specifies a module name.	
-V	Outputs a startup message.	Not displayed
-XV	Cancels output of a start message.	
-cmsg	Outputs a termination message.	Not output
-Xcmsg	Suppresses the output of a terminating message.	
-cwno	Specifies 1 as the termination code when a warning message is output.	0 is specified
-Xcwno	Specifies 0 as the termination code when a warning message is output.	
-help	Outputs a help message.	Not displayed
-cpu MB-number	Specifies the target chip.	
-kanji [EUC SJIS]	Specifies Japanese encoding.	SJIS
-LBA	Specifies to handle the next label of the .bit pseudo-instruction as byte addressing.	Bit Addressing
-XLBA	Specifies to handle the next label of the .bit pseudo-instruction as bit addressing.	
-OVFW	Specifies to generate a code as a WARNING level for overflow.	Errors
-XOVFW	Specifies not to generate a code as an error level for overflow.	
-cif CPU-information file name	Specifies a CPU information file name	

4.3 Details of the Startup Options

The startup options are classified as follows based on function:

- Options related to objects and debugging
- Options related to listing
- Options related to the preprocessor
- Target-dependent options
- Other options

This section describes each of the startup options in details.

■ Options Related to Objects and Debugging

Options used to specify an object file name or to control output of debugging information.

■ Options Related to Listing

Options used to specify a list file name or a list format.

■ Options Related to the Preprocessor

Options used to specify preprocessor operations.

■ Target-dependent Options

Options dependent on the target chip.

■ Other Options

Other options include those used to specify an option file, the output level of warning messages, or a module name.

4.4 Options Related to Objects and Debugging

The options related to objects and debugging are used to specify an object file name or to control output of debugging information.

■ Options Related to Objects and Debugging

The following four options related to objects and debugging are supported:

- -o Specifies an object file name.
- -Xo Creates no object file.
- -g Outputs debugging information.
- -Xg Cancels output of debugging information.

4.4.1 -o, -Xo

-o creates an object file having the specified object file name.

-Xo creates no object file.

If neither **-o** nor **-Xo** is specified, an object is output to the file having the primary file name of the source file suffixed with the file-extension **.obj**.

■ -o

[Format]

-o [object-file-name]

[Description]

-o creates an object file having the specified object file name.

If an object file name is omitted or if only a path name is specified, an object is output to the file having the primary file name of the source file suffixed with the file-extension **.obj**.

For details, see Section "3.3.2 Specifying a File Name with File Name Components Omitted".

[Example]

```
fasm896s ex1 -o ex1_a
```

■ -Xo

[Format]

-Xo

[Description]

-Xo creates no object file.

[Example]

```
fasm896s ex1 -Xo
```

4.4.2 -g, -Xg

-g outputs debugging information.

-Xg outputs no debugging information.

If the **-g** option is not specified, debugging information is not output.

■ -g

[Format]

-g

[Description]

-g outputs debugging information to an object file.

Outputting debugging information enables symbolic debugging using the simulator debugger or emulator debugger.

Specify this option when performing high-level language debugging.

[Example]

```
fasm896s c_test -g
```

■ -Xg

[Format]

-Xg

[Description]

-Xg outputs no debugging information to an object file.

[Example]

```
fasm896s c_test -Xg
```

4.5 Options Related to Listing

The options related to listing are used to specify an assembly list file name or a list format.

■ Options Related to Listing

The following 12 options related to listing are supported:

- -l Outputs an assembly list file.
- -lf..... Outputs an assembly list file (with a file name specified).
- -Xl..... Cancels output of an assembly list file.
- -pl..... Specifies the number of lines on an assembly list page.
- -pw Specifies the number of columns in an assembly list line.
- -linf Outputs an information list.
- -lsrc Outputs a source list.
- -lsec..... Outputs a section list.
- -lcros Outputs a cross-reference list.
- -linc Outputs an include file to a list.
- -lexp Outputs a macro expansion section to a list.
- -tab Specifies the number of tab expansion characters.

4.5.1 -l, -lf, -XI

-l creates an assembly list file.

-lf creates an assembly list file having the specified file name.

-XI creates no assembly list file.

If -l, -lf, nor -XI is specified, no assembly list is created.

■ -l

[Format]

-l

[Description]

-l creates an assembly list file.

An assembly list is output to the file having the primary file name of the source file suffixed with the file-extension .lst.

[Example]

```
fasm896s test -l
```

■ -lf

[Format]

-lf assembly-list-file-name

[Description]

-lf creates an assembly list file having the specified assembly list file name.

If only a path name is specified, an assembly list is output to the file having the primary file name of the source file suffixed with the file-extension .lst.

For details, see Section "3.3.2 Specifying a File Name with File Name Components Omitted".

[Example]

```
fasm896s test -lf\asm\src
```

■ -XI

[Format]

-XI

[Description]

-XI creates no assembly list file.

[Example]

```
fasm896s test -XI
```

4.5.2 -pl, -pw

-pl specifies the number of lines on an assembly list page.

-pw specifies the number of columns in an assembly list line.

■ -pl

[Format]

-pl {0|20-255}

[Description]

-pl specifies the number of lines on an assembly list page.

An assembly list is created so that a single page contains the specified number of lines.

A number between 20 and 255 can be specified as the number of lines.

If 0 is specified as the number of lines, no page break is created.

If this option is not specified, 60 is the default.

[Example]

fasm896s test -pl 0

■ -pw

[Format]

-pw {80-1023}

[Description]

-pw specifies the number of columns in an assembly list line.

A number between 80 and 1023 can be specified as the number of columns.

If this option is not specified, 100 is the default.

[Example]

fasm896s test -pw 80

4.5.3 -linf, -lsrc, -lsec, -lcros

An assembly list consists of the following four lists:

- Information list
- Source list
- Section list
- Cross-reference list

Specify whether or not each of these lists is output.

-linf specifies whether or not an information list is output.

-lsrc specifies whether or not a source list is output.

-lsec specifies whether or not a section list is output.

-lcros specifies whether or not a cross-reference list is output.

■ -linf

[Format]

-linf {ON|OFF}

ON: An information list is output. <default >

OFF: An information list is not output.

[Description]

-linf specifies whether or not an information list is output.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, ON is the default.

[Example]

```
fasm896s test -linf off
```


■ -lsrc**[Format]**

-lsrc {ON OFF}

ON: A source list is output. <default >

OFF: A source list is not output.

[Description]

-lsrc specifies whether or not a source list is output.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, ON is the default.

[Example]

fasm896s test -lsrc on

■ -lsec**[Format]**

-lsec {ON OFF}

ON: A section list is output. <default >

OFF: A section list is not output.

[Description]

-lsec specifies whether or not a section list is output.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, ON is the default.

[Example]

fasm896s test -lsec on

■ -lcros**[Format]**

-lcros {ON OFF} .

ON: A cross-reference list is output.

OFF: A cross-reference list is not output. <default >

[Description]

-lcros specifies whether or not a cross-reference list is output.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, OFF is the default.

[Example]

fasm896s test -lcros on

4.5.4 -linc, -lexp

-linc and -lexp control output of an include file and macro expansion section in the source list.

-linc controls output of an include file to the list.

-lexp controls output of a macro expansion section to the list.

■ -linc

[Format]

`-linc {ON|OFF}`

ON: An include file is output to the list. <default >

OFF: An include file is not output to the list.

[Description]

-linc controls output of an include file to the list.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, ON is the default.

[Example]

fasm896s test -linc off

■ -lexp

[Format]

`-lexp {ON|OFF|OBJ}`

ON: A macro expansion section is output to the list.

OFF: A macro expansion section is not output to the list.

OBJ: Only the object code is output to the list. <default > The text of a macro expansion section is not output to the list.

[Description]

-lexp controls output of a macro expansion section to the list.

Either uppercase character or lowercase character can be used to specify ON or OFF.

If this option is not specified, OBJ is the default.

[Example]

fasm896s test -lexp obj

4.5.5 -tab

-tab specifies the number of space characters used to expand tabs when a list is output.

■ -tab

[Format]

-tab {0-32}

[Description]

-tab specifies the number of space characters used to expand tabs when a list is output.

If this option is not specified, 8 is the default.

[Example]

fasm896s test -tab 4

4.6 Options Related to the Preprocessor

The options related to the preprocessor are used to specify preprocessor operations.

■ Options Related to the Preprocessor

The following eight options related to the preprocessor are supported:

- -p Do not start the preprocessor.
- -P Starts only the preprocessor.
- -Pf Starts only the preprocessor (with a file name specified).
- -D Specifies a macro name.
- -U Cancels a macro name.
- -I Specifies an include path.
- -H Outputs an include path.
- -C Leaves comments as is in the preprocessor output.

4.6.1 -p

-p specifies that the preprocessor is not started.

-p must be specified using a lowercase letter.

■ -p

[Format]

-p

[Description]

-p specifies that the preprocessor is not started.

This means that the preprocessor phase is skipped, and the assembly phase is performed directly.

Specifying this option reduce processing time because the preprocessor processing is not performed.

This option is valid when an assembly source that does not include any preprocessor instructions output by a high-level language compiler is assembled.

[Example]

fasm896s test -p

4.6.2 -P, -Pf

-P outputs the results of preprocessing performed in the preprocessor phase.

-Pf outputs the results of preprocessing performed in the preprocessor phase to the file having the specified file name.

■ -P

[Format]

-P

[Description]

-P outputs the results of preprocessing performed in the preprocessor phase to a file.

The results are output to the file having the primary file name of the source file suffixed with the extension .as.

When this option is specified, only preprocessor processing is performed; the processing in the assembly phase is not performed.

[Example]

fasm896s test -P

■ -Pf

[Format]

-Pf preprocessing-result-file name

[Description]

-Pf outputs the results of preprocessing performed in the preprocessor phase to the file having the specified file name.

If only a path name is specified, the results are output to the file having the primary file name of the source file suffixed with the extension .as.

For details, see Section "3.3.2 Specifying a File Name with File Name Components Omitted".

When this option is specified, only preprocessor processing is performed; the processing in the assembly phase is not performed.

[Example]

fasm896s test -Pf\fasm\src

4.6.3 -D, -U

-D defines a defined character string for a macro name.

-U cancels a macro name specified using -D.

■ -D

[Format]

`-D macro-name="definition-character-string"`

[Description]

-D defines a definition character string for a macro name.

If "-D macro-name" is described and "=defined-character-string is omitted" 1 is defined.

If "-D macro-name=" is described and no definition character string is described, a null character is defined.

This option can be specified more than once.

This option has the same function as the #define instruction.

[Example]

fasm896s test -D OS_type=3 -D WINDOWS

■ -U

[Format]

`-U macro-name`

[Description]

-U cancels a macro name specified -D.

If the same macro name is specified by using both the -D and -U options, the macro name is canceled regardless of the order in which the options are specified.

This option can be specified more than once.

This option has the same function as the #undef instruction.

[Example]

fasm896s test -D ABC=10 -U ABC

4.6.4 -I

-I specifies an include path.

Specify a path to search for an include file specified using the #include instruction.

■ -I

[Format]

-I include-path

[Description]

-I specifies an include path.

Specify a path to search for an include file specified the #include instruction.

This option can be specified more than once. The specified paths are searched in the order they are specified.

For details of the #include instruction, see Section "11.9 #include Instruction" in "PART II SYNTAX".

[Example]

```
fasm896s test -I \include -I \F2MC8L
```


4.6.5 -H

-H outputs the path name of an include file that is read using the **#include** instruction to the standard output.

Path names are output one to a line in the order in which they are read.

■ -H

[Format]

-H

[Description]

-H outputs the path name of an include file that is read using the **#include** instruction to the standard output.

Path names are output one to a line in the order in which they are read.

A path name is not output if an include file was searched when the path was not found.

[Example]

```
fasm896s test -I \include -I \F2MC8L -H
```

4.6.6 -C

-C leaves all comments and space during preprocessor processing.

If this option is not specified, a comment and a space are replaced with a single blank character.

■ -C

[Format]

-C

[Description]

-C leaves all comments and space during preprocessor processing.

If this option is not specified, a comment and a space are replaced with a blank character.

Specifying this option, reduces the processing load in the assembly phase.

[Example]

fasm896s test -C

4.7 Target-Dependent Options

The target-dependent options are options dependent on the target chip.

■ Target-dependent Options

The following four target-dependent options are supported:

Only for the F²MC-8L/8FX family

- -sa..... Specifies the output of an accumulator protection code.
- -Xsa..... Suppresses the output of an accumulator protection code.
- -div_check..... Specifies the warning message is outputted to a DIVU instruction.
- -Xdiv_check.. Suppresses the warning message is no outputted to a DIVU instruction.

4.7.1 -sa, -Xsa

-sa and -Xsa are options related to the output of an accumulator protection code.

-sa outputs an accumulator protection code.

-Xsa does not output an accumulator protection code.

■ -sa

[Format]

-sa

[Description]

-sa outputs an accumulator protection code.

The protection code is output so that the accumulator cannot be changed before and after a structured instruction statement.

[Example]

fasm896s test -sa

■ -Xsa

[Format]

-Xsa

[Description]

-Xsa suppresses the output of an accumulator protection code.

[Example]

fasm896s test -sa -Xsa

4.7.2 -div_check, -Xdiv_check

-div_check outputs a warning message to a DIVU instruction.

-Xdiv_check does not output a warning message to a DIVU instruction.

■ -div_check

[Format]

```
-div_check
```

[Description]

-div_check outputs a warning message to a DIVU instruction as follows.

```
W1806A      DIVU is detected
```

[Example]

```
fasm896s test -div_check
```

■ -Xdiv_check

[Format]

```
-Xdiv_check
```

[Description]

-Xdiv_check does not output a warning message to a DIVU instruction.

[Example]

```
fasm896s test -div_check -Xdiv_check
```

■ About Default Operation

Family Type	Default Operation
F ² MC-8L	It is the same as "-Xdiv_check" specification.
F ² MC-8FX	It is the same as "-div_check" specification.

■ About a DIVU Instruction Check Function

It is necessary to check whether an F²MC-8FX family is satisfactory about the part where the DIVU command is used, when diverting the assembler source for F²MC -8L family to F²MC -8FX family, since operation of a DIVU command differs from the conventional F²MC -8L family.

In order to investigate about use of the DIVU command in assembler source, it is detectable at the time of assembling by specifying -div_check option by referring to an editor etc., although it is possible.

4.8 Other Options

Other options include options used to specify an option file, output level of warning messages, or module name.

■ Other Options

The following 18 options are also supported:

- -Xdof..... Cancels the default option file.
- -f..... Specifies an option file.
- -w Specifies the output level of warning messages.
- -name Specifies a module name.
- -V Displays a startup message.
- -XV Cancels the display of a startup message.
- -cmsg..... Outputs a termination message.
- -Xcmsg..... Suppresses the output of a termination message.
- -cwno Specifies 1 as the termination code when a warning.
- -Xcwno Specifies 0 as the termination code when a warning.
- -help Displays a help message.
- -cpu Specifies the target chip.
- -kanji Specifies Japanese encoding.
- -LBA Handles the next level of the .bit pseudo-instruction as byte addressing.
- -XLBA Handles the next level of the .bit pseudo-instruction as bit addressing.
- -OVFW Generates a code as a WARNING label for overflow.
- -XOVFW Generates a code as an error label for overflow.
- -cif..... Specifies to reference a CPU information.

4.8.1 -Xdof

-Xdof cancels reading of the default option file.

If this option is not specified, the default option file is always read.

■ -Xdof

[Format]

-Xdof

[Description]

-Xdof cancels reading of the default option file.

If this option is not specified, the default option file is always read.

For details of the default option file, see "3.6 Default Option File".

[Example]

fasn896s test -Xdof

4.8.2 -f

-f reads the specified option file.

The fasm896s options can be placed in an option file.

■ -f

[Format]

-f option-file-name

[Description]

-f reads the specified option file.

If the file-extension is omitted from the option file name, .opt is automatically added.

The fasm896s options can be placed in an option file.

Option file can be specified more than once.

For details of option files, see Section "3.4 Option File".

[Example]

fasm896s test -f test.opt

4.8.3 -w

-w sets the output level of warning messages.

If 0 is specified as the warning level, no warning messages will be output.

■ -w

[Format]

```
-w "warning-level"
```

[Description]

-w sets the output level of warning messages.

If 0 is specified for warning-level, no warning messages will be output.

If warning-level is omitted, 2 is the default.

If this option is not specified, 2 is the default.

For details of warning levels and warning messages that can be output, see "APPENDIX A Error Messages".

The following table lists the warning levels and warning messages that can be output:

Table 4.8-1 Warning Levels and Warning Messages Cat can be Output

Warning level	Warning message
0	No warning messages are output.
1, 2	Warning messages other than error numbers W1551A and W1711A are output.
3	All warning messages are output.

[Example]

```
fasm896s test -w 0
```

■ Error Number W1551A

A warning is output if there is no .END instruction at the end of the source file.

■ Error Number W1711A

A warning is output if an address is returned to a .ORG instruction.

4.8.4 -name

-name specifies a module name.

A module name specified this option is assumed to be valid even though it is also specified the .PROGRAM instruction.

■ -name

[Format]

-name module-name

[Description]

-name specifies a module name.

A module name must comply with naming rules.

A module name specified this option is assumed to be valid even though it is also specified using the .PROGRAM instruction.

[Example]

```
fasm896s test -name prog
```

4.8.5 -V, -XV

-V displays a startup message when the assembler is executed.

-XV cancels the display of a startup message.

If neither -V nor -XV is specified, a startup message is not displayed.

■ -V

[Format]

-V

[Description]

-V displays a startup message when the assembler is executed.

A startup message contains the version information and copyright information of the executed assembler.

[Example]

fasm896s test -V

■ -XV

[Format]

-XV

[Description]

-XV cancels the display of a startup message.

[Example]

fasm896s test -V -XV

4.8.6 -cmsg, -Xcmsg

-cmsg displays a termination message when the assembler is executed.

-Xcmsg cancels the display of a termination message.

If neither -cmsg nor -Xcmsg is specified, a termination message is not displayed.

■ -cmsg

[Format]

-cmsg

[Description]

-cmsg displays a termination message when the assembler is executed.

[Example]

fasm896s test -cmsg

■ -Xcmsg

[Format]

-Xcmsg

[Description]

-Xcmsg cancels the display of a termination message.

[Example]

fasm896s test -cmsg -Xcmsg

4.8.7 -cwno, -Xcwno

-cwno specifies 1 as the assembler termination code when a warning message is output.

-Xcwno specifies 0 as the assembler termination code when a warning message is output.

If neither **-cwno** nor **-Xcwno** is specified, 0 is the assembler termination code when a warning message is output.

■ -cwno

[Format]

-cwno

[Description]

-cwno specifies 1 as the assembler termination code when a warning message is output.

[Example]

fasm896s test -cwno

■ -Xcwno

[Format]

-Xcwno

[Description]

-Xcwno specifies 0 as the assembler termination code when a warning message is output.

[Example]

fasm896s test -cwno -Xcwno

4.8.8 -help

**-help displays the startup option list.
This is referred to the help message.**

■ -help

[Format]

-help

[Description]

-help displays the startup option list.

This list is referred to as the help message.

If this option is specified, assembly processing is not performed.

[Example]

fasm896s test -help

4.8.9 -cpu

-cpu specifies the target chip.

The target chip specifies the name of the product to be used.

■ -cpu

[Format]

-cpu target

target: name of the product to be used

[Description]

-cpu specifies the target chip.

Target chip specifies the name of the product to be used.

[Example]

fasm896s test -cpu MB89051

fasm896s test -cpu MB89F051

4.8.10 -kanji

-kanji specifies Japanese encoding.

■ -kanji

[Format]

-kanji "EUC SJIS"

EUC: EUC encoding is specified.

SJIS: Shift JIS encoding is specified.

[Description]

-kanji specifies that Japanese encoding is used for comments, character constants, and character string constants in a source program.

If this option is not specified, SJIS is the default.

[Example]

fasm896s test -kanji EUC

4.8.11 -LBA, -XLBA

-LBA handles those labels as bit addressing when only the label of definition is coded after the pseudo-instruction for bit addressing of `.bit`, etc.

-XLBA handles those labels as bit addressing when only the label of definition is coded after the pseudo-instruction for bit addressing of `.bit`, etc.

If neither **-LBA** nor **-XLBA** is specified, operations are the same as when **-XLBA** has been specified.

■ -LBA

[Format]

-LBA

[Description]

Processes labels as byte addresses when only the line of label definition is coded immediately after the line in which the pseudo-instruction of either `.bit`, `.data.i`, or `.res.i` is coded. In that case, the label address is adjusted to the byte boundary.

[Example]

fasm896s test -LBA

The following shows the results of the execution.

000000:0 0		.bit 0
000000:1 0	LBL1:	bit 0
000001	LBL2:	
000001:0 1		.bit 1

■ -XLBA

[Format]

-XLBA

[Description]

Processes labels as bit addresses when only the line of label definition is coded immediately after the line in which the pseudo-instruction of either `.bit`, `.data.i`, or `.res.i` is coded. In that case, the label address is the bit address after the immediately preceding bit address.

[Example]

fasm896s test -XLBA

The following shows the results of the execution.

000000:0 0		.bit 0
000000:1 0	LBL1:	.bit 0
000000:	LBL2:	
000000:2 1		.bit 1

4.8.12 -OVFW, -XOVFW

The **-OVFW** option displays a warning message when the operation result of an operand that describes an operational equation exceeds that operand size.

The **-XOVFW** option displays an error message when the operation result of an operand that describes an operational equation exceeds that operands size.

If neither **-OVFW** nor **-XOVFW** is specified, an **ERROR** message is displayed when the results of the operation coded in the immediate value operand exceed that operand size.

■ **-OVFW**

[Format]

-OVFW

[Description]

The **-OVFW** option performs the following processing when the operation result of an operand that describes an operational equation exceeds that operand size.

The operand that describes an operational equation includes an immediate value and an address value.

- Displays a warning message. (W1541A: Value out of range.)
- Outputs an object file.
- Masks the operand operation result in accordance with the operand size, and sets only the lower bits to generates a code.
- Outputs an assembly list is output when an assembly list output specification option (-l) is specified.

The following shows an example of this option, and an example of an output list.

To output an assembly list, specify the assembly list output specification option (-l).

[Example]

fasm896s test -OVFW -l
• Output list

The operation result of ~(8008H) is FFFF7FF7H, because assembler operates the expression by 32 bits. ~(8008H)=FFFF7FF7H is masked by 16 bits. The value output to the object and the list is 7FF7H.

When -OVFW is specified, assembler output WARNING.

SN LOC OBJ

LLINE SOURCE

=====

<test.asm>

:

CO 0000 E47FF7

2 T MOVW A,#~(8008H)

*** test.asm(2) W1541A: Value out of range (in operand 2)

:

■ -XOVFW

[Format]

-XOVFW (Default)

[Description]

The -XOVFW option performs the following processing when the operation result of an operand that describes an operational equation exceeds that operand size.

The operand that describes an operational equation includes an immediate value and an address value.

- Displays an error message. (E4541A: Value out of range.)
- Does not output an object file.
- Masks the operand operation result in accordance with the operand size, and sets only the lower bits to output a code to the assembly list.
- Outputs an assembly list is output when an assembly list output specification option (-l) is specified.

The following shows an example of this option, and an example of an output list.

To output an assembly list, specify the assembly list output specification option (-l).

The assembly list is also output when an error occurs.

[Example]

fasm896s test -XOVFW -l

• Output list

SN LOC OBJ

LLINE SOURCE

<test.asm>

=====

CO 0000 E47FF7

2 T MOVW A,#~(8008H)

*** test.asm(2) E4541A: Value out of range (in operand 2)

:

When -XOVFW is specified, assembler output ERROR.

The operation result of ~(8008H) is FFFF7FF7H, because assembler operates the expression by 32 bits. ~(8008H)=FFFF7FF7H is masked by 16 bits. The value output to the object and the list is 7FF7H.

4.8.13 -cif

-cif specifies a CPU information file that SOFTUNE Tools reference.

■ -cif

[Format]

-cif CPU-information-filename

CPU-information-filename: CPU information file name to be referenced

[Description]

Specify a CPU information file that SOFTUNE Tools reference.

[Example]

fasm896s test -cpu MB89603 -cif "C:\SOFTUNE\lib\896\896.csv"

Note:

SOFTUNE Tools get CPU information by referencing the CPU information file. Reference to a CPU information file different between the related tools may cause an error to the program to be created. The CPU information file that comes standard with SOFTUNE Tools is located at:

Installation directory\lib\896\896.csv

When installing the compiler assembler packs in a different directory and using the compiler, assembler and linkage editor instead of SOFTUNE Workbench, specify -cif that each tool can reference the same CPU information file.

CHAPTER 5

OPTIMIZATION CODE CHECK FUNCTIONS

This chapter describes the optimization code check functions of assemblers.

The optimization code check functions locate those instruction codes in a program that can be rewritten to speed up program execution.

5.1 Optimization Code Check Functions of fasm896s

5.1 Optimization Code Check Functions of fasm896s

For fasm896s, those portions of a program that can be optimized, provided optimization does not disturb program operation, are located.

- Optimization of branch instructions

■ Optimization of Branch Instructions

The optimal instruction is created based on the distance to the branch destination label.

The following instructions are optimized (16-bit extended branch instructions).

- Instructions optimized: Bcc16

Optimization is always performed.

Optimization operates on 16-bit extended branch instructions.

Table 5.1-1 show the relationships between 16-bit extended branch instructions and the instructions that are created.

Table 5.1-1 Instructions Created from 16-bit Extended Branch Instructions (1 / 3)

Extended branch instruction	Action	Distance	Created instruction
BZ16 label	Branchs if (Z) =1.	-128 to +127	BZ label
		Other	BNZ \$+5 JMP label
BNZ16 label	Branchs if (Z) =0.	-128 to +127	BNZ label
		Other	BZ \$+5 JMP label
BEQ16 label	Branchs if (Z) =1.	-128 to +127	BEQ label
		Other	BNE \$+5 JMP label
BNE16 label	Branchs if (Z) =0.	-128 to +127	BNE label
		Other	BEQ \$+5 JMP label
BC16 label	Branchs if (C) =1.	-128 to +127	BC label
		Other	BNC \$+5 JMP label
BNC16 label	Branchs if (C) =0.	-128 to +127	BNC label
		Other	BC \$+5 JMP label

Table 5.1-1 Instructions Created from 16-bit Extended Branch Instructions (2 / 3)

Extended branch instruction	Action	Distance	Created instruction
BLO16 label	Branchs if (C) =1.	-128 to +127	BLO label
		Other	BHS \$+5 JMP label
BHS16 label	Branchs if (C) =0.	-128 to +127	BHS label
		Other	BLO \$+5 JMP label
BN16 label	Branchs if (N) =1.	-128 to +127	BN label
		Other	BP \$+5 JMP label
BP16 label	Branchs if (N) =0.	-128 to +127	BP label
		Other	BN \$+5 JMP label
BLT16 label	Branchs if (V) xor (N)=1.	-128 to +127	BLT label
		Other	BGE \$+5 JMP label
BGE16 label	Branchs if (V) xor (N)=0.	-128 to +127	BGE label
		Other	BLT \$+5 JMP label
BBC16 bit,label	Branchs if bit=0.	-128 to +127	BBC bit,label
		Other	BBS bit,\$+6 JMP label
BBS16 bit,label	Branchs if bit=1.	-128 to +127	BBS bit,label
		Other	BBC bit,\$+6 JMP label
BV16 label	Branchs if (V)=1.	-128 to +122	BN \$+7 BLT label JMP \$+5 BGE label
		Other	BN \$+7 BGE \$+10 JMP label BLT \$+5 JMP label

Table 5.1-1 Instructions Created from 16-bit Extended Branch Instructions (3 / 3)

Extended branch instruction	Action	Distance	Created instruction
BNV16 label	Branches if (V)=0.	-128 to +122	BN \$+7 BGE label JMP \$+5 BLT label
		Other	BN \$+7 BLT \$+10 JMP label BGE \$+5 JMP label
BLE16 label	Branches if (Z) or ((N) or (V))=1.	-128 to +125	BEQ label BLT label
		Other	BEQ \$+4 BGE \$+5 JMP label
BGT16 label	Branches if (Z) or ((N) or (V))=0.	-128 to +127	BEQ \$+4 BLT label
		Other	BEQ \$+7 BGE \$+5 JMP label
BLS16 label	Branches if (Z) or (C)=1.	-128 to +125	BEQ label BLO label
		Other	BEQ \$+4 BHS \$+5 JMP label
BHI16 label	Branches if (Z) or (C)=0.	-128 to +127	BEQ label BHS label
		Other	BEQ \$+7 BLO \$+5 JMP label

CHAPTER 6

ASSEMBLY LIST

This chapter describes the contents of the assembly list.

- 6.1 Composition
- 6.2 Page Format
- 6.3 Information List
- 6.4 Source List
- 6.5 Section List
- 6.6 Cross-reference List

6.1 Composition

The assembly list is created if the start-time option **-l** or **-lf** has been specified.

The assembly list consists of the following four lists:

- **Information list**
 - **Source list**
 - **Section list**
 - **Cross-reference list**
-

■ Composition

The assembly list is created if the start-time option **-l** or **-lf** has been specified.

The assembly list consists of the following four lists.

● Information list

The information list consists of specified start-time contents, the number of errors, the number of warnings, and the names of source files, include files name, and option files name.

● Source list

The source list consists of various items of information about assembling the source program. Information is displayed for each line.

Error information, location and object data are output.

● Section list

The section list consists of the names and attributes of defined in the source program.

● Cross-reference list

The cross-reference list consists of the definition of symbol names used in the source program and line numbers that are referenced by those symbols.

■ Relationship with Start-time Options

Each list can control output using start-time options.

For details of the start-time options, see Section "4.5 Options Related to Listing".

Table 6.1-1 shows the relationship of lists and start-time options.

Table 6.1-1 Relationship of Lists and Start-time Options

List	Start-time option	Initial value
Information list	-linf {ON OFF}	ON: Displayed
Source list	-lsrc {ON OFF}	ON: Displayed
Section list	-lsec {ON OFF}	ON: Displayed
Cross-reference list	-lcros {ON OFF}	OFF: Not displayed

6.2 Page Format

The format of pages composing the assembly list can be specified using start-time options or pseudo-instructions.

When using start-time options, specify the number of lines with `-pl`, and the number of column with `-pw`.

When using pseudo-instructions, specify the number of lines with `".FORM LIN"`, and the number of columns with `".FORM COL"`.

■ Information List

The format of pages composing the assembly list can be specified using start-time options or pseudo-instructions.

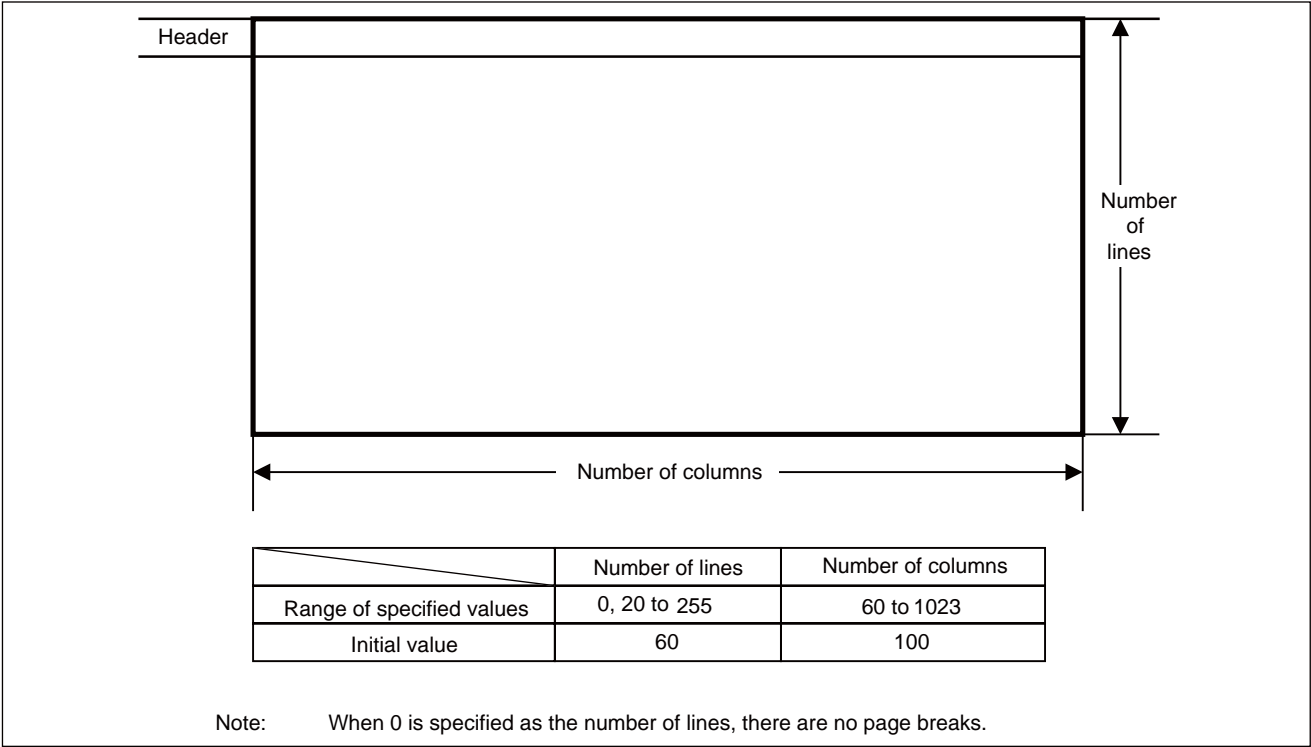
When using start-time options, specify the number of lines with `-pl`, and number of column with `-pw`.

When using pseudo-instructions, specify the number of lines with `".FORM LIN"`, and the number of columns with `".FORM COL"`.

The header of each page consists of four lines.

Figure 6.2-1 shows the page format.

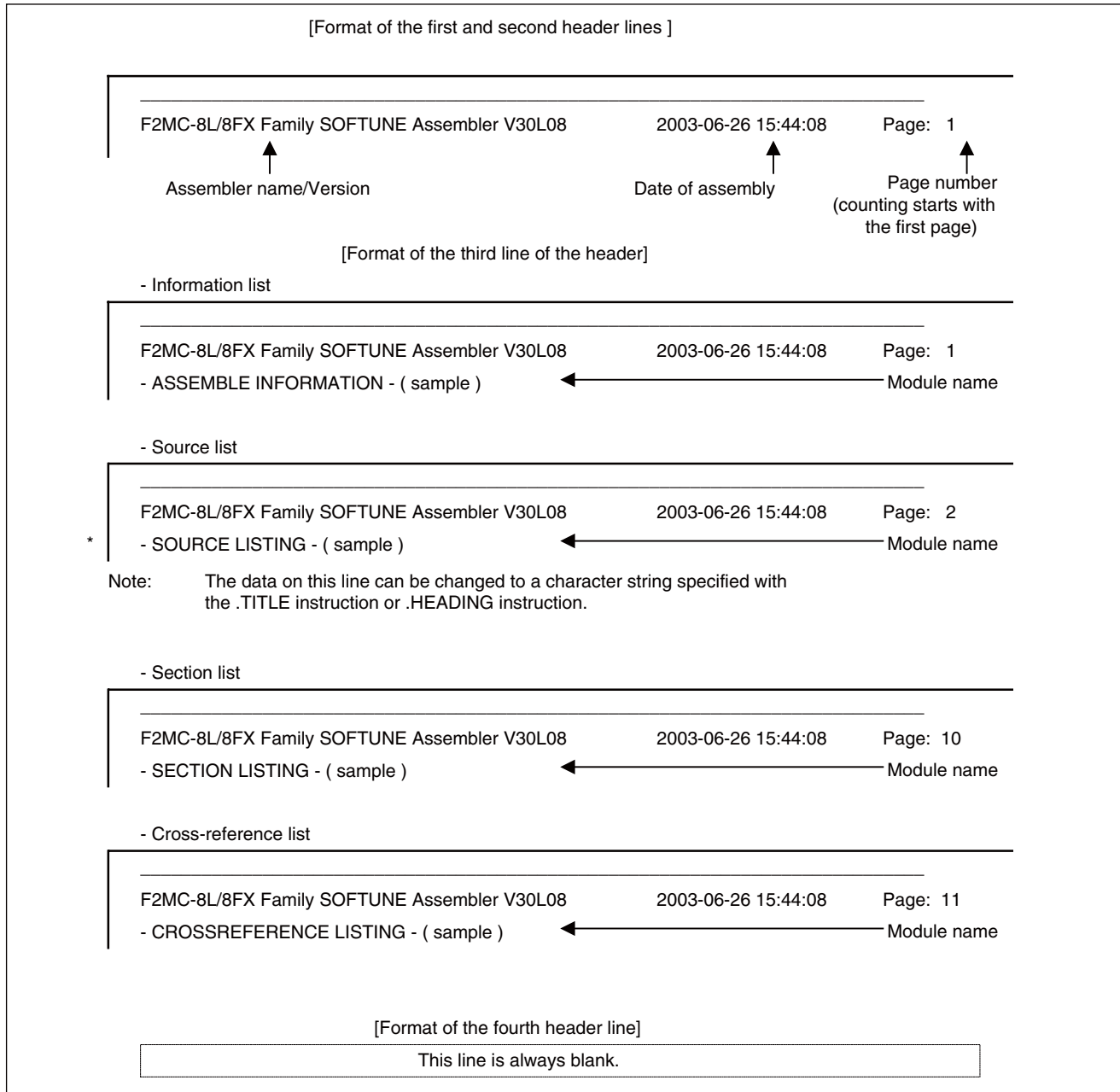
Figure 6.2-1 Page Format



■ Header Format

The header consists of four lines. The fourth line is a blank line.

Figure 6.2-2 Header Format



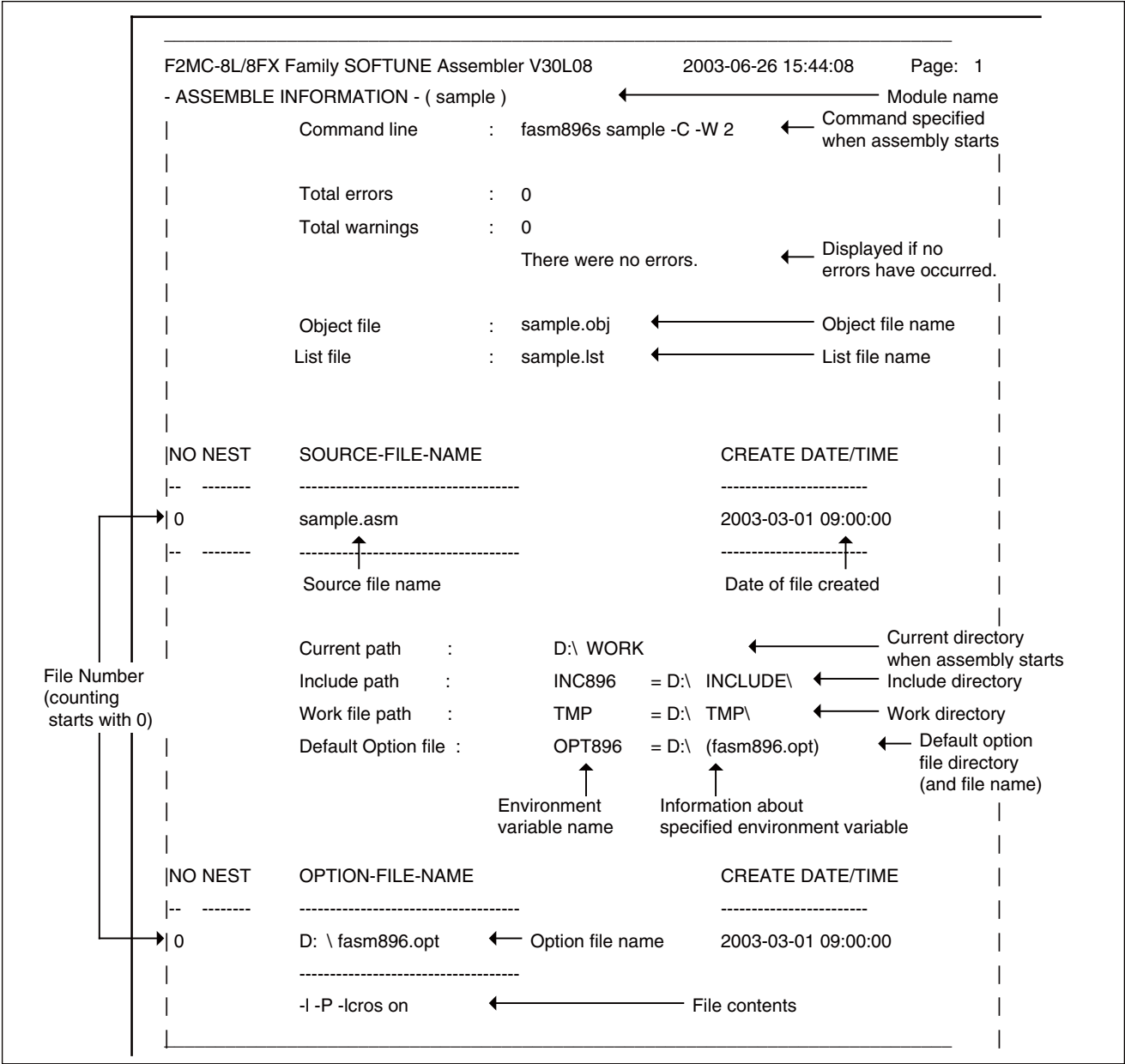
6.3 Information List

The format (sizes) of pages composing the assembly list can be specified using start-time options or pseudo-instructions.

■ Information List

[Format 1]

Figure 6.3-1 Information List [Format 1]



[Format 2]

Figure 6.3-2 Information List [Format 2]

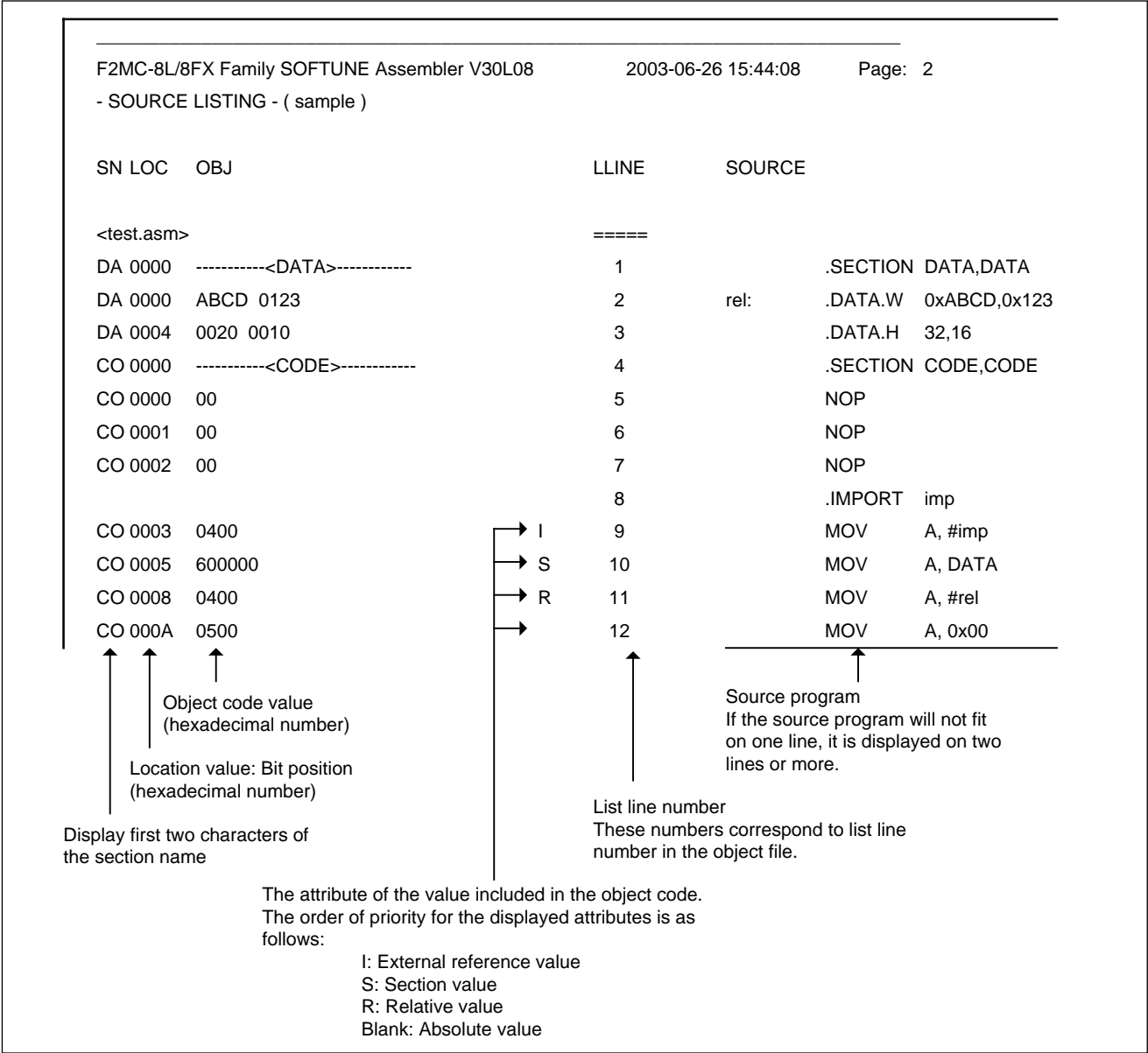
F2MC-8L/8FX Family SOFTUNE Assembler V30L08		2003-06-26 15:44:08	Page: 1
- ASSEMBLE INFORMATION - (sample)		Module name	
Command line : fasm896s sample -C -W 2			
** Total errors : 2 (First Line: 22)			
** Total warnings : 3 (First Line: 7)			
Object file : - None -		If any errors or warnings have occurred, the number of each and the list line number of the line of the first occurrence of the error or warning is found are displayed.	
List file : sample.lst			
NO NEST		SOURCE-FILE-NAME	CREATE DATE/TIME
-- -----		-----	-----
0		sample.asm	2003-03-01 09:00:00
-- -----		-----	-----
1 *		sample.h	2003-03-01 09:02:00
-- -----		-----	-----
:			
Information about include nest (The number of asterisks represents the nesting depth.)			
Current path : D:\ WORK\			
Include path : INC896 = -None- = D:\ TOOL\LIB\ 896			
		\INCLUDE\	
Work file path : TMP = D:\ TMP\			
Default Option file : OPT896= -None- = D:\ TOOL\LIB\ 896			
		\ (fasm896.opt)	
		If no option files error exist, no file contents are displayed.	

6.4 Source List

The source list output various items of information about assembling the source program. Information is displayed for each line of the program. Error information, location, object data and, other information are displayed.

■ Source List

Figure 6.4-1 Source List



6.4.1 Preprocessor Operations

If any operation is performed on a line by the preprocessor, a symbol is displayed for the line.

■ Preprocessor Operations

Figure 6.4-2 Preprocessor Operations

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			2	#ifdef DEF1
DA 0000	0001		3	X .DATA 0 /* then */
			4	#else
			5	.DATA 1 /* else */
			6	#endif

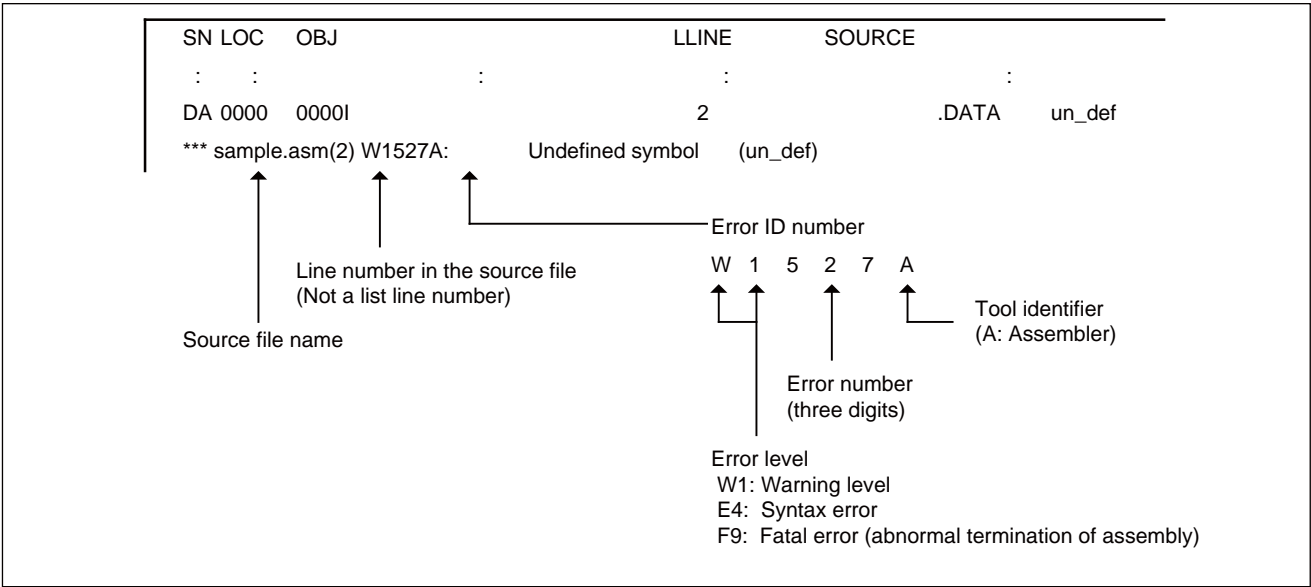
Information about operations performed by the preprocessor.
Blank: Normal.
Preprocessor
X: The line has not been assembled.
&: Macro expansion is performed for this line.

6.4.2 Error Display

The following list format is displayed for a line containing an error.

■ Error Display

Figure 6.4-3 Error List Format



6.4.3 Include File

The following list format is displayed for a file that has been read with the `#include` instruction.

■ Include File

Figure 6.4-4 Include File List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			2	#include "sample.h"
			=====	
	<sample.h>		3=1	NOP
CO 0000	00		4=1	NOP
CO 0001	00		5=1	NOP
CO 0002	00		=====	
	<sample.asm>			

↑ Include file name
 Name of the file located when control returns from the include file.

↑ Number of include nests (8 maximum)

In case of -line off /.LIST NOINC, these instructions are not displayed. However, a error message is displayed.

6.4.4 .END, .PROGRAM, .SECTION

This section describes the list format of the following program structure definition instructions:

- **.END:** Ends the source program
- **.PROGRAM:** Declares a module name
- **.SECTION:** Defines a section

■ **.END**

Figure 6.4-5 .END List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
		==	25	.END
Or				
		=> 00000300	25	.END 0x300
Or				
		=> 00000010	R 25	.END start

The start address specification is displayed by the hexadecimal number.

R : Relative value
Blank : Absolute value

Anything written after the .END instruction is not assembled.
Anything written after the .END instruction is also not displayed in the list.

■ **.PROGRAM**

Figure 6.4-6 .PROGRAM List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
		MODULE NAME = test_module1	9	.PROGRAM test_module1

A character string specified as the module name is displayed.
If the character string is long, it is displayed on more than one line.

■ .SECTION

Figure 6.4-7 .SECTION List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
SE 0000	-----	<SEC1>-----	4	.SECTION SEC1,CODE,ALIGN1
:	:	:	:	:
SE 0010	-----	<SEC2>-----	19	.SECTION SEC2,CODE,LOCATE=0x10

↑ ↑ ↑
 Location Section name
 If the section name is longer than the display space, it is displayed
 with truncating the exceeded characters.

↑
 First two characters of the section name

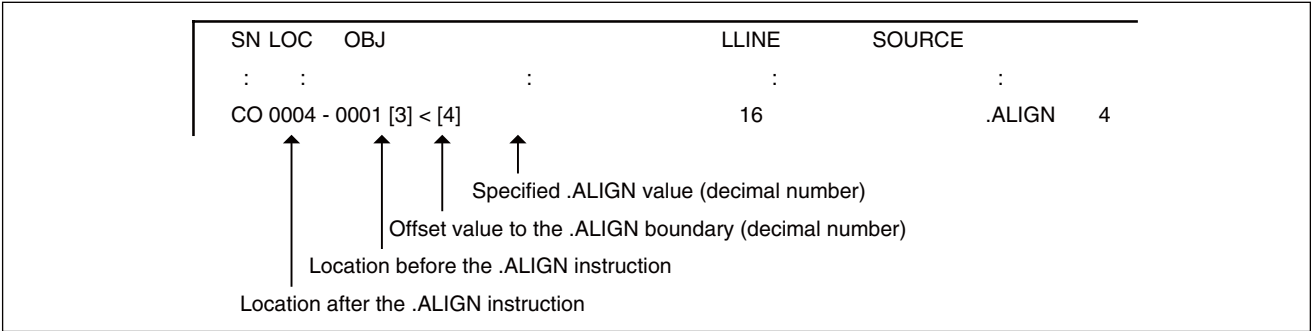
6.4.5 .ALIGN, .ORG, .SKIP

This section describes the list format for the following address control instructions:

- **.ALIGN:** Performs boundary alignment
- **.ORG:** Changes the value of a location counter
- **.SKIP:** Increments the value of a location counter

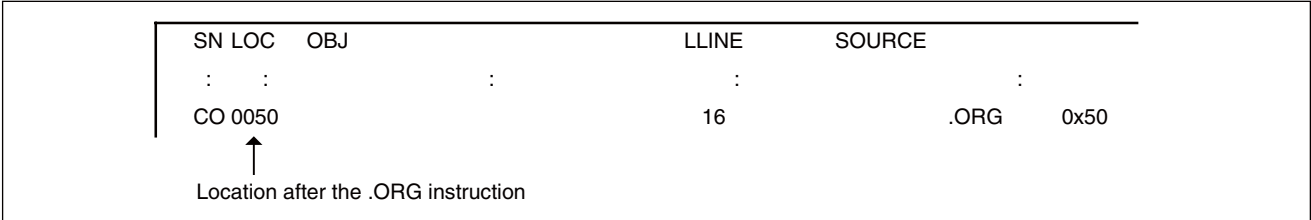
■ **.ALIGN**

Figure 6.4-8 .ALIGN List Format



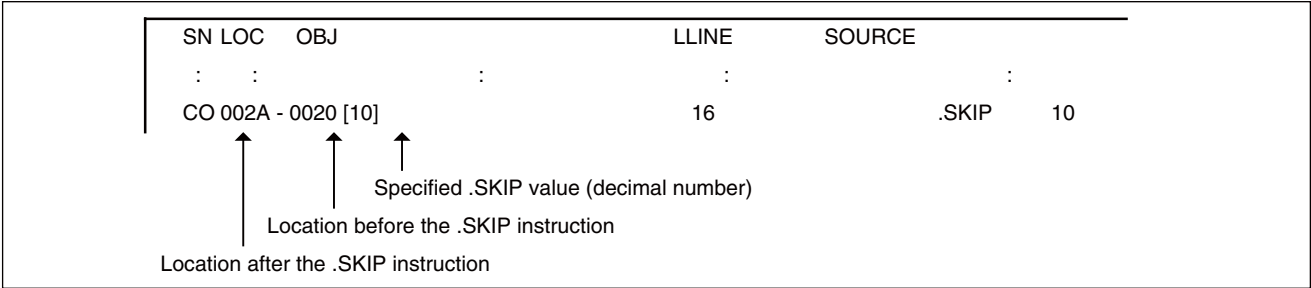
■ **.ORG**

Figure 6.4-9 .ORG List Format



■ **.SKIP**

Figure 6.4-10 .SKIP List Format



6.4.6 .EXPORT, .GLOBAL, .IMPORT

The following program link instructions are not changed in the list:

- **.EXPORT:** Declares an external definition symbol
- **.GLOBAL:** Declares an external definition symbol or external reference symbol
- **.IMPORT:** Declares an external reference symbol

■ .EXPORT

Figure 6.4-11 .EXPORT List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			16	.EXPORT exp1,exp2

■ .GLOBAL

Figure 6.4-12 .GLOBAL List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			16	.GLOBAL exp1,exp2

■ .IMPORT

Figure 6.4-13 .IMPORT List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			16	.IMPORT imp1,imp2

6.4.7 .EQU

This section describes the list format for the following symbol definition instructions:

- **.EQU:** **Assigns a symbol a value**

■ **.EQU**

Figure 6.4-14 .EQU List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
		= 00000100	16	sym01: .EQU 0x100
		↑		
		Set symbol value (hexadecimal number)		

6.4.8 .DATA, .BIT, .BYTE, .HALF, .EXTEND, .LONG, .WORD, .DATAB

This section describes the list format of the following area definition instructions (integer):

- **.DATA:** Defines constants (integer)
- **.BIT:** Defines constants (1-bit integer)
- **.BYTE:** Defines constants (8-bit integer)
- **.HALF:** Defines constants (16-bit integer)
- **.LONG:** Defines constants (32-bit integer)
- **.WORD:** Defines constants (16-bit integer, 32-bit integer)
- **.DATAB:** Defines constant blocks (integer)

■ .DATA

Figure 6.4-15 .DATA List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
SE 0000	10		10	.DATA.B 0x10
SE 0001	20 10		11	.DATA.B 0x20,16
SE 0003	00R 00S		12	.DATA.B REL01,SEC1
SE 0005	0010		13	.DATA.H 0x10
SE 0007	0020 0010		14	.DATA.H 0x20,16
SE 000B	0000R 0000S		15	.DATA.H REL01,SEC1
SE 000F	00000010		16	.DATA.L 0x10
SE 0013	00000020 00000010		17	.DATA.L 0x20,16
SE 001B	00000000R 00000000S		18	.DATA.L REL01,SEC1

↑

Location

↑

Data Value
If multiple data values are defined, as many values as possible are displayed side by side.
(hexadecimal number)

Symbol following a data value:

I: External reference value

S: Section value

R: Relative value

Blank: Absolute value

■ .BIT

The format in the list is the same as that of .DATA.I.

■ .BYTE

The format in the list is the same as that of .DATA.B.

■ **.HALF**

The format in the list is the same as that of .DATA.H.

■ **.LONG**

The format in the list is the same as that of .DATA.L.

■ **.WORD**

The format in the list is the same as that of .DATA.W.

For the F²MC-8L/8FX family, the format in the list is the same as that of .DATA.H.

■ **.DATAB**

Figure 6.4-16 .DATAB List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
SE	0023	[2] 10	10	.DATAB.B 2,0x10
SE	0025	[16] 0020	11	.DATAB.H 16,0x20
SE	0045	[2] 00000010	12	.DATAB.L 2,0x10
SE	004D	[2] 00000000	R 13	.DATAB.L 2,REL01
SE	0055	[2] 00	S 14	.DATAB.B 2,SEC1

Location

Repeat count
(decimal number)

Data value
(hexadecimal number)

Attribute included in a data value
I: External reference value
S: Section value
R: Relative value
Blank: Absolute value

6.4.9 .FDATA, .FLOAT, .DOUBLE, .FDATAB

This section describes the list format for the following area definition instructions (floating-point data):

- **.FDATA:** Defines constants (floating-point numbers)
- **.FLOAT:** Defines constants (32-bit floating-point numbers)
- **.DOUBLE:** Defines constants (64-bit floating-point numbers)
- **.FDATAB:** Defines constants blocks (floating-point numbers)

■ .FDATA

Figure 6.4-17 .FDATA List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
SE	0057	3F800000 40133333	16	.FDATA.S 0r1.0,0r2.3
SE	005F	3FF0000000000000	17	.FDATA.D 0r1.0
SE	0067	11112222	18	.FDATA.S 0x11112222
SE	006B	1111222233334444	19	.FDATA.D 0x1111222233334444

↑ Location
 ↑ Data Value
 If multiple data value are defined, as many values as possible are displayed side by side. (hexadecimal number)

■ .FLOAT

The format in the list is the same as that of .FDATA.S.

■ .DOUBLE

The format in the list is the same as that of .FDATA.D.

■ .FDATAB

Figure 6.4-18 .FDATAB List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
SE	0073	[2] 3F800000	16	.FDATAB.S 2,0r1.0
SE	007B	[2] 3FF0000000000000	17	.FDATAB.D 2,0r1.0

↑ Location
 ↑ Repeat count (decimal number)
 ↑ Data value (hexadecimal number)

6.4.10 .RES, .FRES

This section describes the list format of the following area definition instructions (no data values):

- **.RES:** Defines area instructions (no data values: integer)
- **.FRES:** Defines area instructions (no data values: floating-point number)

■ **.RES**

Figure 6.4-19 .RES List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
CO 0000	[2]B		16	.RES.B 2
CO 002	[5]H		17	.RES.H 5
CO 000C	[3]L		18	.RES.L 3+1-1

Location

Repeat count
(decimal number)

Data size
B : 1byte
H : 2byte length
L : 4byte length
W : 2byte length

■ **.FRES**

Figure 6.4-20 .FRES List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
CO 0018	[2]S		16	.FRES.S 2
CO 0020	[2]D		17	.FRES.D 2

Location

Repeat count
(decimal number)

Data size
S: Single-precision floating point number data (area size: 4byte length)
D: Double-precision floating point number data (area size: 8byte length)

6.4.11 .SDATA, .ASCII, .SDATAB

This section describes the list format for the following area definition instructions (character string):

- **.SDATA:** Defines a character string
- **.ASCII:** Defines a character string
- **.SDATAB:** Defines a character string block

■ .SDATA

Figure 6.4-21 .SDATA List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
DA	0000	61 62 63 64 65 66 67 68 69 6A	16	.SDATA "abcdefghijklmnopqrstuv\
		6B 6C 6D 6E 6F 70 71 72 73 74	17	wxyz0123456789" /*continuation-line */
		75 76 77 78 79 7A 30 31 32 33		
		34 35 36 37 38 39		
DA	0024	31 32 33 FF 31 32 33	18	.SDATA "123\xff123","12345\t\n"
DA	002B	31 32 33 34 35 09 0A		
DA	0032	31 32 33 09 31 32 33	19	.SDATA "123\t123","\t",\
DA	0039	22	20	"1234567890"
DA	003A	31 32 33 34 35 36 37 38 39 30		
DA	0044		21	.SDATA "" /*null-character-string */

↑ Location
 ↑ Data value
 Displayed byte by byte (hexadecimal number)

■ .ASCII

The format in the list is the same as that of .SDATA.

■ .SDATAB

Figure 6.4-22 .SDATAB List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
DA	0044	[2]	16	.SDATAB 2,"" /*null-character-string */
DA	0044	[5] 31 32 33 34 35 36 37 38	17	.SDATAB 5,"12345678901234567890"
		39 30 31 32 33 34 35 36 37 38		
		39 30		

↑ Location
 ↑ Data value
 Displayed byte by byte (hexadecimal number)

6.4.12 .DEBUG

This section describes the list format for the following debugging information display control instruction:

- **.DEBUG:** Specifies which portion of the debugging information to display

■ .DEBUG

Figure 6.4-23 .DEBUG List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
=====		DEBUG INFORMATION Already ON	28	.DEBUG ON
:	:	:	:	:
=====		DEBUG INFORMATION on -> OFF	31	.DEBUG OFF
:	:	:	:	:
=====		DEBUG INFORMATION off -> ON	40	.DEBUG ON

↑

If the debugging information display option -g is not specified at start time, "Ignore" is displayed.

↑

ON/OFF status
off -> ON: Debugging information display is on from this line.
on -> OFF: Debugging information display is off from this line.
Already ON: Debugging information display has already been started.
Already OFF: Debugging information display has already been stopped.

6.4.13 .LIBRARY

The following instruction specifying a library file is not changed in the list:

- **.LIBRARY:** Specifies a library file
-

■ .LIBRARY

Figure 6.4-24 .LIBRARY List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			16	.LIBRARY "sample.lib"

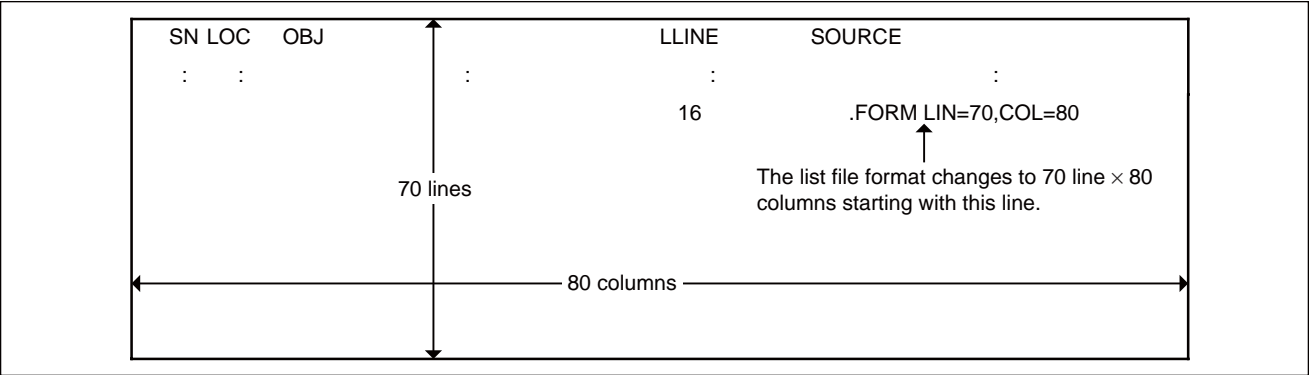
6.4.14 .FORM, .TITLE, .HEADING, .LIST, .PAGE, .SPACE

This section describes the list format for the following list display control instructions:

- **.FORM:** Specifies the number of lines and the number of columns on a page
- **.TITLE:** Specifies a title
- **.HEADING:** Changes a title
- **.LIST:** Specifies details of displaying the assembly source list
- **.PAGE:** Specifies a page break
- **.SPACE:** Outputs blank lines

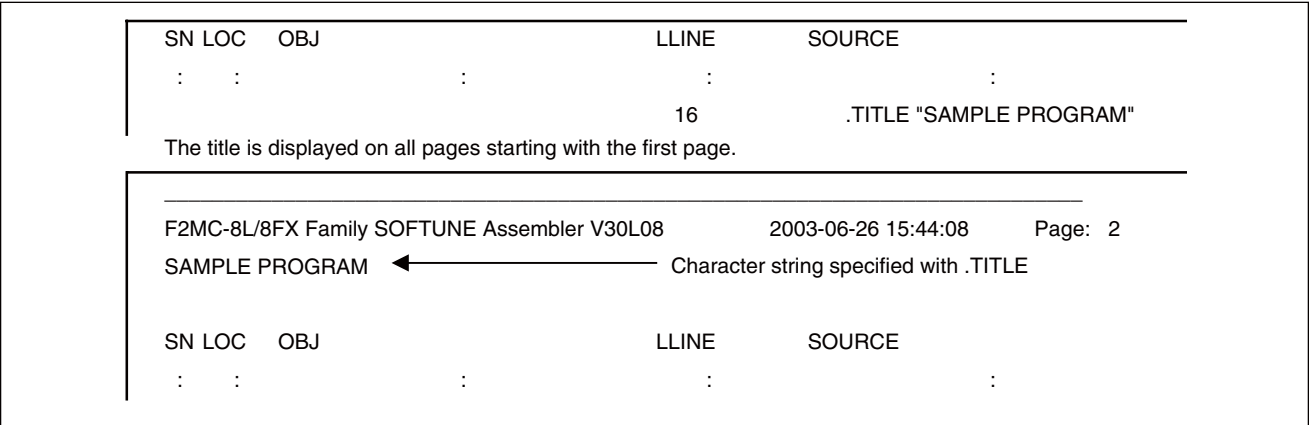
■ **.FORM**

Figure 6.4-25 .FORM List Format



■ **.TITLE**

Figure 6.4-26 .TITLE List Format



■ .HEADING

Figure 6.4-27 .HEADING List Format

F2MC-8L/8FX Family SOFTUNE Assembler V30L08				2003-06-26 15:44:08	Page: 2
SAMPLE PROGRAM					
SN	LOC	OBJ		LLINE	SOURCE
:	:	:	:	:	:
Page break occurs at the line where .HEADING is described. The title changes on the next page.					
F2MC-8L/8FX Family SOFTUNE Assembler V30L08				2003-06-26 15:44:08	Page: 3
PROG-1					
SN	LOC	OBJ		LLINE	SOURCE
.HEADING "PROG-1"					

■ .LIST

Figure 6.4-28 .LIST List Format

SN	LOC	OBJ		LLINE	SOURCE
:	:	:	:	:	:
				6	
				7	; .LIST OFF is on the next line (eighth line)
The list is not displayed from the line where .LIST OFF occurs to the line immediately before the line where .LIST ON occurs. (The line number is increasing.)				10	
				11	
				10	.LIST ON

■ .PAGE

Figure 6.4-29 .PAGE List Format

SN	LOC	OBJ		LLINE	SOURCE
:	:	:	:	:	:
				6	
				7	; .PAGE is on the next line (eighth line)
A page break occurs at the line where .PAGE is specified. (If .PAGE is on the first line of the page, there is no page break.)					

■ .SPACE

Figure 6.4-30 .SPACE List Format

SN	LOC	OBJ	LLINE	SOURCE
:	:	:	:	:
			5	; .SPACE 2 is on the next line (sixth line)
The specified number of blank lines are created.			7	

6.5 Section List

The section list consists of the names and attributes of and other data about sections defined in the source program.

■ Section list

Figure 6.5-1 Section List

F2MC-8L/8FX Family SOFTUNE Assembler V30L08		2003-06-26 15:44:08	Page: 10
- SECTION LISTING - (sample)		←	Module name
NO	SECTION-NAME	SIZE	ATTRIBUTES
0	SEC1	0008	CODE REL ALIGN=1
1	SEC02	0008	CODE REL ALIGN=1
2	ON	0004	CODE REL ALIGN=1
3	SEC05	0008	CODE REL ALIGN=1
4	SEC06	0008	DATA REL ALIGN=1
5	SEC07	0008	CONST REL ALIGN=1
6	SEC08	0008	COMMON REL ALIGN=1
7	SEC09	0008	STACK REL ALIGN=1
	SEC10	0008	DUMMY
8	SEC11	0008	CODE ABS LOCATE=0100
9	SEC12	0008	CODE REL ALIGN=1
10	SEC13	0008	CODE REL ALIGN=1
11	SEC14	0008	DATA ABS LOCATE=0000
12	SEC15	0008	CONST REL ALIGN=1
13	SEC16	0008	COMMON ABS LOCAE=0010
14	SEC17	0008	STACK REL ALIGN=1

↑

Section name
(Displayed in the order
in which they appear.)

↑

Section size
(hexadecimal number)

↑

Section type

Section number in the order that the sections appear
Starts with 0.
Dummy section are not numbered.
These numbers correspond to the section numbers in
the object file.

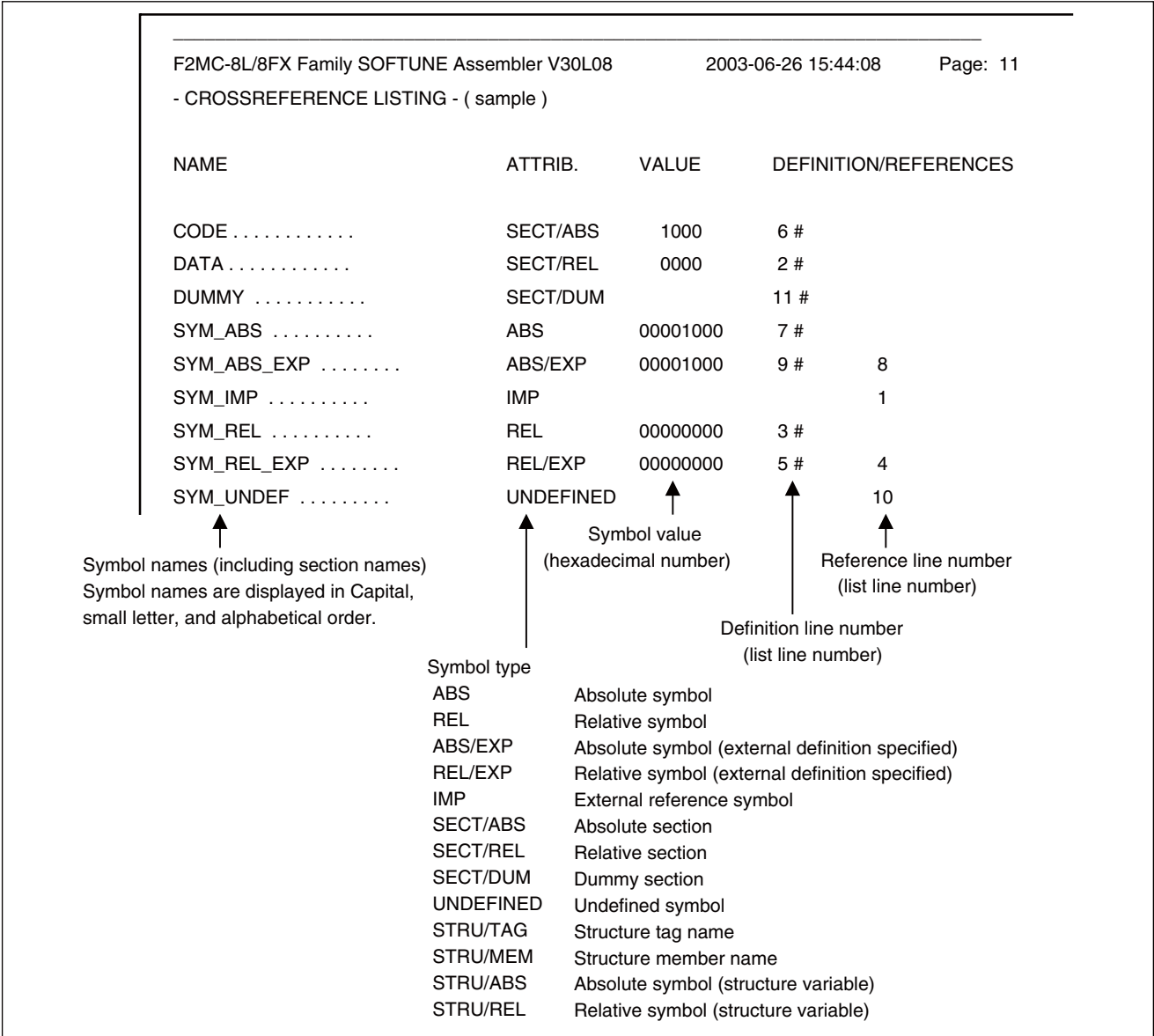
Section placement format
ALIGN values are displayed for
relative sections.
LOCATE values are displayed for
absolute sections.
Blanks are displayed for dummy
sections.

6.6 Cross-reference List

The cross-reference list consists of the definition of the symbol names used in the source program and the line numbers that are referenced by those symbols.

■ Cross-reference List

Figure 6.6-1 Cross-reference List



PART II SYNTAX

PART II describes the syntax and format for writing assembly source programs.

CHAPTER 7 BASIC LANGUAGE RULES

CHAPTER 8 SECTIONS

CHAPTER 9 MACHINE INSTRUCTIONS

CHAPTER 10 ASSEMBLER PSEUDO-INSTRUCTIONS

CHAPTER 11 PREPROCESSOR PROCESSING

CHAPTER 12 ASSEMBLER PSEUDO- MACHINE INSTRUCTIONS

CHAPTER 13 STRUCTURED INSTRUCTIONS

CHAPTER 7

BASIC LANGUAGE RULES

This chapter describes the rules that govern the construction of programs in assembly language.

- 7.1 Statement Format
- 7.2 Character Set
- 7.3 Names
- 7.4 Forward Reference Symbols and Backward Reference Symbols
- 7.5 Integer Constants
- 7.6 Location Counter Symbols
- 7.7 Character Constants
- 7.8 Strings
- 7.9 Floating-Point Constants
- 7.10 Data Formats of Floating-Point Constants
- 7.11 Expressions
- 7.12 Comments

7.1 Statement Format

An assembly statement line consists of the following five fields:

- Symbol field
- Operation field
- Operand field
- Comment field
- Continuation field

Each line can contain up to 4095 characters.

■ Statement Format

The format of an assembly statement is as follows:

Symbol field	Operation field	Operand field	Comment field	Continuation field
--------------	-----------------	---------------	---------------	--------------------

Every field is optional.

The symbol, operation, and operand fields must be separated by one or more spaces or tabs.

Each statement line can contain up to 4095 characters.

■ Symbol Field

This field is used to write a symbol.

A symbol starts in the first column. Alternatively, a symbol can start in any column after the first provided it ends with a colon (:).

[Example]

SYMBOL SYMBOL: SYMBOL:

■ Operation Field

This field is used to write the operation mnemonic for a machine or pseudo-instruction.

This field starts in the second or any subsequent column.

The symbol and operation fields must be separated by one or more spaces or tabs.

[Example]

.SECTION CODE NOP .DATA 100

■ Operand Field

This field is used to write one or more operands for a machine or pseudo-instruction.

Two or more operands must be separated by a comma (,).

The operation and operand fields must be separated by one or more spaces or tabs.

[Example]

```
.SECTION    DATA,DATA,ALIGN=4
.DATA      1,2,4
.SECTION    CODE,CODE,ALIGN=2
MOV        A, R1
```

■ Comment Field

This field is used to write a comment.

This field can start in any column.

A semicolon (;) or two slashes (//) indicates the start of a comment, which extends to the end of the line. Comments of this type are called line comments.

A comment can be enclosed by /* and */, as in C. Comments of this type are called range comments. A range comment can be inserted anywhere.

[Example]

```
/*-----
Comment
-----*/
        PUSHW    IX        ;Comment
        MOVW     A, /* Comment */ SP    // Comment
```

■ Continuation Field

A backslash (\), specified at the end of a line, allows a statement to continue on the next line.

If any character other than a new line character follows a backslash (\), continuation is not allowed.

A backslash (\) can also be specified within a comment, character constant, or string.

[Example]

```
.DATA      0x01,0x02,0x03,        \
           0x04,0x05,0x06,        ; Comment \
           0x07,0x08,0x09
.SDATA     "abcdefghijklmnpqrstuvwxy\
ABCDEFHIJKLMNOPQRSTUVWXYZ"      /* String continuation */
```

7.2 Character Set

The followings are the characters that can be used to write programs in assembly language:

- **Alphabetic characters:** Uppercase letters (A to Z) and lowercase letters (a to z)
 - **Numeric characters:** 0 to 9
 - **Symbols:** + - * / % < > | & ~ = () ! ^ \$ @ # _ ' " : . \ ; space tab
-

■ Character Set

Table 7.2-1 shows the character set that can be used for the assembler.

Table 7.2-1 Character Set

Type	Character
Uppercase letter	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Lowercase letter	abcdefghijklmnopqrstuvwxyz
Numeric character	0123456789
Symbols	+ - * / % < > & ~ = () ! ^ \$ @ # _ ' " : . \ ; Space Tab \$ @ # _ ' " : . \ ; Space Tab

Note:

Some terminals display a yen sign (¥) in place of a backslash (\).

Symbols that are not included in the character set, as well as Japanese-encoded characters (SJIS or EUC code), can be used in a comment or string.

7.3 Names

The assembler uses names to identify and refer program elements.

Names such as module names, symbols, section names, structure tag names, and macro names are used.

The following are reserved words:

- Register names: R0 to R7, A, T, IX, EP, SP, PC, PS
 - Operators: SIZEOF, BITADR, BITPOS
-

■ Naming Rules

- A name can contain up to 255 characters.
- The first character of a name must be an alphabetic character or underscore (_).
- The second and subsequent characters of a name are alphabetic characters, numeric characters, and underscores (_).
- Names are case-sensitive.

■ Name Classification

Names are classified by purpose, as described below:

● Module names

Module names are defined with the .PROGRAM instruction. They are used to identify objects.

● Symbols

Symbols are names to which values have been assigned.

They are usually defined in a symbol field.

They are used as branch destination labels or data addresses.

Values are assigned with instructions such as .EQU.

● Section names

Section names are defined with the .SECTION instruction.

They are used to identify sections.

They are assigned to symbols.

When a section name is used as a term in an expression, it indicates the start address of the section that is to be set after linking is completed.

● Structure tag names

Structure tag names are defined with the .STRUCT instruction.

They are used to identify structures.

● Macro names

Macro names are used with the preprocessor.

For information about the preprocessor, see "CHAPTER 11 PREPROCESSOR PROCESSING."

■ Reserved Words

Reserved words are names that have already been reserved for the assembler. The user cannot redefine them.

Reserved words are not case-sensitive.

The following are reserved words:

● Register names:

R0 to R7, A, T, IX, EP, SP, PC, PS

● Operators

SIZEOF, BITADR, BITPOS

● Other reserved names

The followings are reserved names for the linker.

In the assembler, these cannot be used even though no error detection is provided to detect the use of these reserved names.

- Section names and symbol names starting with `_ROM_` or `_RAM_`.

7.4 Forward Reference Symbols and Backward Reference Symbols

In an assembly program, symbols used as operands for machine or pseudo-instructions are handled as forward reference symbols or backward reference symbols. When a used symbol has not yet been defined, it is handled as a forward reference symbol.

When a used symbol has already been defined, it is handled as a backward reference symbol.

■ Forward Reference Symbols and Backward Reference Symbols

In an assembly program, symbols used as operands for machine or pseudo-instructions are handled as forward reference symbols or backward reference symbols.

When a used symbol has not yet been defined, it is handled as a forward reference symbol.

When a used symbol has already been defined, it is handled as a backward reference symbol.

[Example]

```
BEQ    aaa    /* aaa is a forward reference symbol for the BEQ instruction */
aaa:
BLT    aaa    /* aaa is a backward reference symbol for the BLT instruction */
```

The following restrictions apply to symbols handled as forward reference symbols:

- An expression containing a forward reference symbol, even when the symbol has an absolute value, is handled as if it were a relative expression. A forward reference symbol, therefore, cannot be used as an operand for which only an absolute value can be specified.

■ Size Operator

In general, the size operator is handled as a relative value because the size is calculated by the linker. However, the size of a dummy section is calculated by the assembler.

In this case, the size operator is handled as a forward reference symbol for processing reasons.

For information about the size operator, see Section "7.11.4 Values Calculated from Names".

7.5 Integer Constants

There are four integer constant types: binary constant, octal constant, decimal constant, and hexadecimal constant.

■ Integer Constants

There are four integer constant types: binary constant, octal constant, decimal constant, and hexadecimal constant.

Also, the long type specification (such as 123L) and unsigned type specification (for example, 123U) in C are supported.

■ Binary Constants

Binary constants are integer constants represented in binary.

Each binary constant must be given a prefix (B' or 0b) or suffix (B).

Prefixes and suffixes are not case-sensitive.

[Example]

B'0101 0b0101 0101B

■ Octal Constants

Octal constants are integer constants represented in octal.

Each octal constant must be given a prefix (Q' or 0) or suffix (Q).

Prefixes and suffixes are not case-sensitive.

[Example]

Q'377 0377 377Q

■ Decimal Constants

Decimal constants are integer constants represented in decimal.

Each decimal constant can be given a prefix (D') or suffix (D).

Prefixes and suffixes are optional only for decimal constants.

Prefixes and suffixes are not case-sensitive.

[Example]

D'1234567 1234567 1234567D

■ Hexadecimal Constants

Hexadecimal constants are integer constants represented in hexadecimal.

Each hexadecimal constant must be given a prefix (H' or 0x) or suffix (H).

Prefixes and suffixes are not case-sensitive.

[Example]

H'ff 0xFF 0FFH

7.6 Location Counter Symbols

Location counter symbols represent the current location counter.
The dollar sign (\$) is used as a location counter symbol.

■ Location Counter Symbols

The assembler uses the location counter for addressing during assembly.

The current location value can be referred by using location counter symbols.

The dollar sign (\$) is used as a location counter symbol.

The location counter value is an absolute value for an absolute section, and a relative value for a relative section.

[Example]

```
BNE      $+4
```

7.7 Character Constants

Character constants represent character values.
Each character constant must be enclosed in single quotation marks (').
Each character constant can contain up to four characters.

■ Character Constants

Each character constant must be enclosed in single quotation marks (').

Characters, escape sequences, octal notations, and hexadecimal notations can be used as character constants.

Each character constant can contain up to four characters.

Character constants are handled as a base-256 system.

● Characters

Any character (including the space character) except a backslash (\) and single quotation mark (') can be used as character constants.

[Example]

'P' '@A' '0A'''

● Escape sequences

A backslash (\) followed by a specific character can be used as a character constant.

Character constants of this type are called escape sequences.

Table 7.7-1 lists the escape sequences.

Table 7.7-1 Escape Sequence

Character	Character constant	Value
New line	\n	0x0A
Horizontal tab	\t	0x09
Backspace	\b	0x08
Carriage return	\r	0x0D
Form feed	\f	0x0C
Backslash	\\	0x5C
Single quotation mark	\'	0x27
Double quotation mark	\"	0x22
Alert	\a	0x07
Vertical tab	\v	0x0B
Question mark	\?	0x3F

Note:

Only lower case letters can be used in escape sequence.

[Example]

`\n`

`\"`

`\\"\\`

● Octal escape sequences

When an octal escape sequence is used, a character code bit pattern can be directly specified to represent one-byte data.

Each octal escape sequence starts with a backslash (`\`), and contains up to three octal digits.

[Example]

Character constant	Bit pattern
<code>\0</code>	<code>b'00000000</code>
<code>\377</code>	<code>b'11111111</code>
<code>\53</code>	<code>b'00101011</code>
<code>\0123</code>	<code>b'00001010</code> → Divided into <code>\012</code> and <code>'3'</code>

● Hexadecimal escape sequences

When a hexadecimal escape sequence is used, a character code bit pattern can be directly specified to represent one-byte data.

Each hexadecimal escape sequence starts with a backslash (`\`), and contains one or two hexadecimal digits.

[Example]

Character constant	Bit pattern
<code>\x0</code>	<code>b'00000000</code>
<code>\xff7</code>	<code>b'11111111</code>
<code>\x2B</code>	<code>b'00101011</code>
<code>\x0A5</code>	<code>b'00001010</code> → Divided into <code>\x0A</code> and <code>'5'</code>

7.8 Strings

Each string must be enclosed in double quotation marks (").

■ Strings

Each string must be enclosed in double quotation marks (").

Strings use the same formats as those for character strings. For more information, see Section "7.7 Character Constants".

To specify a double quotation mark (") in a string, use an escape sequence (\").

A single quotation mark (') can be specified without using an escape sequence.

[Example]

```
"ABCD\n"
```

```
"012345\n\0"
```

```
"\xff Fujitsu\t Co., Ltd.\n\0377\0"
```

Note:

When a Japanese string is written, it is output to the object in its Japanese encoding (SJIS or EUC). The assembler does not convert the code.

7.9 Floating-Point Constants

The following are the formats for floating-point constants:

- `[0r][+|-]{.d|d[.d]}` `[e[+|-]d]`: `d` is a decimal number.
- `[F'][+|-]{.d|d[.d]}` `[e[+|-]d]`: `d` is a decimal number.
- `0xh`: `h` is a hexadecimal number.
- `H'h`: `h` is a hexadecimal number.

■ Notation for Floating-point Constants

[Format 1]

<code>[0r] [+ -] { .d d [.d] } [e [[+ -] d]]</code> <code>[F'] [+ -] { .d d [.d] } [e [[+ -] d]]</code>	<code>d</code> is a decimal number.
--	-------------------------------------

[Description]

A value is used to specify a floating-point constant.

The letter "e" indicates the specification of the exponent part.

The value that follows "e" is the exponent part.

An uppercase "E" can be used instead of a lowercase "e".

A prefix (0r or F') is optional.

[Example]

0r954	0r-12e+0	415.
F' .5	F' 2.0e2	-2.5E-4

[Format 2]

<code>0xh</code> <code>H'h</code>	<code>h</code> is a hexadecimal number.
--------------------------------------	---

[Description]

A bit pattern is used to directly specify a floating-point constant.

A bit pattern for the data length is specified using a hexadecimal number.

[Example 1: To represent negative infinity with double precision:]

`0xFFFF000000000000`

[Example 2: To represent negative infinity with single precision:]

`0xFF800000`

■ Specification of Single or Double Precision

Whether a floating-point constant has single precision (32 bits) or double precision (64 bits) depends on the pseudo-instructions or the size specification.

● Single precision (32 bits) is specified when:

1. No size specification appears in a pseudo-instruction for a floating-point constant.
2. The size specification symbol S is used.
3. The .FLOAT instruction is used.

● Double precision (64 bits) is specified when:

1. The size specification symbol D is used.
2. The .DOUBLE instruction is used.

[Example]

.FDATA.S 1.2	/*Single precision is specified because the S size specification is used */
.FDATA.D 1.2	/*Double precision is specified because the D size specification is used */
.FDATA 1.2	/*Single precision is specified because no size specification is used */
.FLOAT 0.1	/*Single precision is specified because the .FLOAT instruction is used */
.DOUBLE 0.1	/*Double precision is specified because the .DOUBLE instruction is used */

7.10 Data Formats of Floating-Point Constants

The data formats of floating-point constants comply with the ANSI/IEEE Std754-1985 specifications.

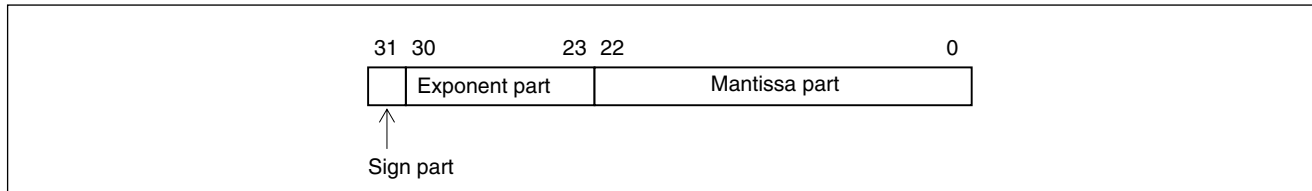
The following two data formats are provided for floating-point constants:

- Data format for single-precision floating-point constants
- Data format for double-precision floating-point constants

The range of the representable floating-point constants is shown below.

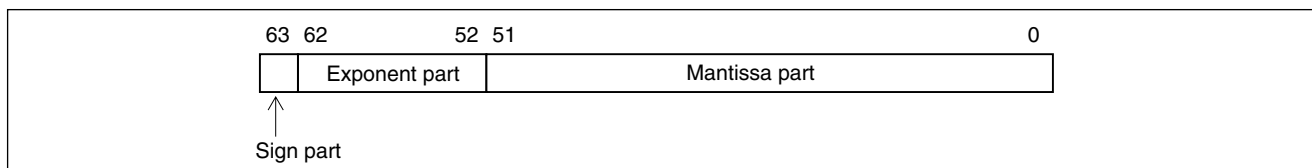
■ Data Format of Single-precision Floating-point Constants

The data format of single-precision floating-point constants includes one bit in the sign part, eight bits in the exponent part, and 23 bits in the mantissa part.



■ Data Format of Double-precision Floating-point Constants

The data format of double-precision floating-point constants includes one bit in the sign part, 11 bits in the exponent part, and 52 bits in the mantissa part.



■ Range of the Representable Floating-point Constants

Table 7.10-1 shows the range of the representable floating-point constants.

Table 7.10-1 Range of the Representable Floating-Point Constants

Precision	Range of representable floating-point constants
Single precision	-3.40282356779733661637e+38 to -1.17549431578982589985e-38 -0 0 1.17549431578982589985e-38 to 3.40282356779733661637e+38
Double precision	-1.79769313486231580793e+308 to -2.22507385850720125958e-308 -0 0 2.22507385850720125958e-308 to 1.79769313486231580793e+308

7.11 Expressions

Expressions can be used in places where addresses or data values can be specified. C-like operators are provided for expressions.

■ Expression Syntax

The following is the BNF expression syntax:

[Syntax]

<code><expression > ::= <expression > ' ' <expression ></code>	Logical OR expression
<code> ::= <expression > ' & ' <expression ></code>	Logical AND expression
<code> ::= <expression > ' ' <expression ></code>	Bitwise OR expression
<code> ::= <expression > '^' <expression ></code>	Bitwise XOR expression
<code> ::= <expression > ' & ' <expression ></code>	Bitwise AND expression
<code> ::= <expression > { '==' '!==' } <expression ></code>	Equality expression
<code> ::= <expression > { ' < ' <= ' > ' >= ' } <expression ></code>	Relational expression
<code> ::= <expression > { ' < < ' > > ' } <expression ></code>	Shift expression
<code> ::= <expression > { '+' '-' } <expression ></code>	Addition expression
<code> ::= <expression > { '*' '/' '%' } <expression ></code>	Multiplication expression
<code> ::= { '!' '~' '+' '-' 'SIZEOF' } <expression ></code>	Unary expression
<code> ::= '(' <expression > ')'</code>	Parenthesized expression
<code> ::= <term ></code>	
<code><term > ::= { numeric-constant character-constant symbol location-counter }</code>	

Note:

Every expression is an integer expression. Floating-Point constant expression are not supported.

■ Expression Types

Expressions are evaluated during assembly. Relative symbols, external reference symbols, and section symbols are kept as terms, then processed by the linker. Therefore, for any expressions containing relative symbols, external reference symbols, or section symbols, relocation information is generated, and then the expressions are processed by the linker.

Expressions are classified as shown below:

● Absolute expressions

- Expression that consists of only numeric constants, character constants, and/or symbols having absolute values
- Expression that does not contain any forward reference symbols
- Expression that does not contain any size operators

● Simple relative expressions

- Expression that contains a single relative value
- Expression that does not contain any forward reference symbols
- Expression that does not contain any size operators

● Compound relative expressions

- Expression that contains two or more relative values
- Expression that contains one or more external reference values
- Expression that contains one or more section values
- Expression that contains one or more forward reference symbols
- Expression that contains one or more size operators

A relative expression usually refers to a compound relative expression.

■ Precision in Operations on Expressions

Operations with a precision of 32 bits are performed on expressions. Note that the result of an operation with a precision of more than 32 bits is not assured. (If such an operation is performed, no error will result.)

7.11.1 Terms

Terms represent absolute values, relative values, external reference values, or section values. They can be used in expressions.

Each term can be given a bit address and addressing specifier.

■ Term Types

The following term types are provided:

- Numeric constant
- Character constant
- Symbol (absolute, relative, external reference, and section)
- Location counter (\$)

Each term has an absolute value, relative value, external reference value, or section value.

Each term can be given a bit address or addressing specifier.

■ Absolute Values

Any of the following terms has an absolute value:

- Numeric constant
- Character constant
- Symbol defined with the .EQU instruction
- Symbol that represents an address within an absolute section
- Location counter within an absolute section
- Section symbol for an absolute section
- Size operator for calculating the size of a dummy section

■ Relative Values

Since relative values are resolved by the linker, relocation information is generated.

Any of the following terms has a relative value:

- Symbol that represents an address within a relative section
- Location counter within a relative section
- Size operator for calculating the size of a non-dummy section

■ External Reference Values

Since external reference values are resolved by the linker, relocation information is generated.

The following term has an external reference value:

- External reference symbol

■ Section Values

Each section value has the start address of a section.

Since section values are resolved by the linker, relocation information is generated.

Only section symbols for relative sections have section values.

Section symbols for absolute sections have absolute values.

■ Bit Addresses

Bit addresses can be defined as symbols defined with instructions such as `.BIT` or `.RES.I`, or can be specified using one of the special formats shown below.

[Format 1]

:bit-address

A bit address indicates a bitwise location allocated in the memory area.

[Format 2]

address:bit-position

A value of 8 or greater can be specified for a bit position. This value is divided by 8, and the obtained value is added to the specified address. The remainder is handled as the bit position.

■ Addressing Specifiers

An addressing specifier can be an operand.

[Format]

addressing-specifier : address

addressing-specifier: {D|E|X}

D: Direct address specification

E: Extend address specification

X: Extended direct address specification

Addressing methods are classified as shown below:

- Direct address specification: Specify the lower 8 bits of an address
- Extend address specification: Specify a 16-bit address
- Extended direct address specification: Specify the lower 8 bits of the results of an operation of the formula: $(\text{address} \ \& \ 0x007F) + 0x0080$. This specification has meaning only in the F²MC-8FX family.

[Restriction 1]

In the event that a machine instruction operand can specify either of the direct address specification, the extend address specification, or the extended direct address specification, if the symbol specified by the operand has the following conditions, the specification of the address specifier is ignored. Specification is always the extend address.

- For forward reference symbols (excluding external reference symbols)
- For symbols that are made an external reference declaration outside a section

Symbols that are made an external reference declaration outside a section are symbols that are made an external reference declaration with `.import` or `.global` pseudo instruction from the beginning of the source program until the `.section` pseudo instruction appears first.

[Restriction 2]

The extended direct address specification is a function that has meaning only in the F²MC-8FX family. However, note that the Assembler will not detect errors even if specified by the F²MC-8L family.

[Supplement 1]

fasm896s automatically selects addressing mode, as follows.

Table 7.11-1 Automatic Selection of Addressing Mode

Format	Conditions	Addressing Mode
Immediate	0x0000 to 0x00FF	Direct addressing
	0x0100 to 0xFFFF	Extend addressing
Symbols	Definition/Declaration by DIR section	Direct addressing
	Definition/Declaration by other sections	Extend addressing

Therefore, when assembling an assembler source for F²MC-8FX, access to address at 0x0080 to 0x00FF cannot be made correctly, when the extended direct access area bank is other than 0. In such a case, use the address specifier "E" to specify the extend addressing. (See example 1 and 2.)

[Example 1]

```
SETDB    #1          /* Direct bank pointer = 1 */
MOV      A,0x80      /* Access to address at 0x0100 */
                        /* with the direct address specification. */
MOV      A, E:0x80   /* Access to address at 0x0080 */
                        /*with the extend address specification. */
```

[Example 2] When the symbol SYM is DIR section and at 0x0080 address

```
SETDB    #1          /* Direct bank pointer = 1 */
MOV      A,SYM       /* Access to address at 0x0100 */
                        /* with the direct address specification. */
MOV      A, E:SYM    /* Access to address at 0x0080 */
                        /*with the extend address specification. */
```

[Supplement 2]

The extended direct address is an address specification for when a direct bank pointer is specified. The relationships between the direct bank pointer and extended direct address and mapping area are described below. To access with an extended direct address using a symbol assigned to areas where the direct bank pointer is anything other than 0, follow the procedure in Example 3.

Table 7.11-2 Relationships among Direct Bank Pointer, Extended Direct Address and Mapping Area

Direct bank pointer	Extended direct address	Mapping area (16-bit address)
0	0x0080 to 0x00FF	0x0080 to 0x00FF
1		0x0100 to 0x017F
2		0x0180 to 0x01FF
3		0x0200 to 0x027F
4		0x0280 to 0x02FF
5		0x0300 to 0x037F
6		0x0380 to 0x03FF
7		0x0400 to 0x047F

[Example 3] When the symbol **SYM** is at 0x0100 address

```

SETDB    #1          /* Direct bank pointer = 1 */
MOV      A,D: SYM    /* Access to address at 0x0000 */
                        /* with the direct address specification. */
MOV      A, E: SYM   /* Access to address at 0x0100 */
                        /* with the extend address specification. */
MOV      A,X: SYM    /* Access to address at 0x0100 */
                        /* with the extended direct address specification (Coded: access at 0x80) */

```

7.11.2 Range of Operand Values

The range of the value of the operands that describes an operational equation that can be described is determined according to the type.

The assembler displays a warning or an error when the value of the operand operation result is out of range.

Whether the message is a warning or an error is determined by the specification of the -OVFW and -XOVFW options ("4.8.12 -OVFW, -XOVFW" Reference).

This section explains the range of the operand value.

■ Range of Operand Values

Table 7.11-3 lists examples of operand value range.

If the operational result is outside of the range in the table below (Table 7.11-3), the assembler will display a warning or an error.

For details on each operand, see the Hardware Manual.

Table 7.11-3 Example of Ranges of Operand Values

Operand Types	Range of Values
Direct address (dir)	0 to 255
Offset (off)	-128 to 127
Extended address (ext)	0 to 65535
Vector table number (#vct)	0 to 7
Immediate data (#d8)	-128 to 255
Immediate data (#d16)	-32768 to 65535
Bit direct address (dir: b)	di: 0 to 255/b: 0 to 7
Branch relative address (rel)	-128 to 127

7.11.3 Operators

Operators are used in expressions.

Boolean values are assumed to be true if they are nonzero; they are assumed to be false if they are zero.

The following are the operators:

- Logical operators: `|| & & !`
 - Bitwise operators: `| & ^ ~`
 - Relational operators: `== != < <= > >=`
 - Arithmetic operators: `+ - * / % >> << +(unary-expression) -(unary-expression)`
 - Operators for calculating values from names: `BITADR BITPOS SIZEOF`
-

■ Boolean Values

Boolean values are assumed to be true if they are nonzero; they are assumed to be false if they are zero.

■ Logical Operators

The followings are the logical operators:

- `expression||expression:` Boolean value OR operation
- `expression & &expression:` Boolean value AND operation
- `!expression:` Boolean value inversion

■ Bitwise Operators

The following are the bitwise operators:

- `expression|expression:` Bitwise OR operation
- `expression & expression:` Bitwise AND operation
- `expression^expression:` Bitwise XOR operation (exclusive OR operation)
- `~expression:` Bitwise inversion

■ Relational Operators

For evaluation of an expression based on a relational operator, 1 is assumed when the expression is true; 0 is assumed when the expression is false.

The following are the relational operators:

- `expression==expression:` Equal to
- `expression!=expression:` Not equal to
- `expression < expression:` Less than
- `expression <= expression:` Less than or equal to
- `expression > expression:` Greater than
- `expression >= expression:` Greater than or equal to

■ Arithmetic Operators

The following are the arithmetic operators:

- `expression+expression`: Addition
- `expression-expression`: Subtraction
- `expression*expression`: Multiplication
- `expression/expression`: Division
- `expression%expression`: Modulo operation
- `expression <<expression`: Left shift
- `expression >>expression`: Right shift
- `+expression`: Positive
- `-expression`: Negative

■ Operators for Calculating Values from Names

The following are the operators for calculating values from names.

For more information, see Section "7.11.4 Values Calculated from Names".

- `BITADR` symbol: Bit symbol address operator
- `BITPOS` symbol: Bit position number operator
- `SIZEOF` section: Size operator

7.11.4 Values Calculated from Names

The following special operators are provided to refer values required for addressing or programming:

- **BITADR:**Bit symbol address operator
 - **BITPOS:**Bit position number operator
 - **SIZEOF:**Size operator
-

■ Bit Symbol Address Operator (BITADR Operator)

[Format]

BITADR bit-symbol BITADR (bit-symbol)
--

A bit symbol can be enclosed in parentheses.

[Description]

The bit address symbol operator is used to calculate the address of a bit symbol.

■ Bit Position Number Operator (BITPOS Operator)

[Format]

BITPOS bit-symbol BITPOS (bit-symbol)
--

A bit symbol can be enclosed in parentheses.

[Description]

The bit position number operator is used to calculate the bit position number of a bit symbol.

■ Section Size Determination (SIZEOF Operator)

[Format]

```
SIZEOF section-symbol
SIZEOF (section-symbol)
```

A section symbol can be enclosed in parentheses.

[Description]

The size operator is used to calculate the size of a section.

Section symbols, therefore, are the only terms that can be used as operands for SIZEOF.

Because the section size is calculated by the linker, any expression that contains the size operator is handled as a relative expression.

For dummy sections, however, because the code is not output to an object file, the size is calculated by the assembler.

In this case, the assembler handles the size operator as if it were a forward reference symbol. This is because multiple subsections are allowed for composing a section so that the section size is unknown until the assembler completes internal processing PASS1. That is, each expression containing the size operator is handled as if it were a relative expression, regardless of whether it eventually takes an absolute value.

[Example]

```
.SECTION ABC,DATA,ALIGN=1
:
.SECTION PROGRAM,CODE,ALIGN=1
MOVW  A, #SIZEOF(ABC)
MOVW  A, #SIZEOF(DMY)
:
.SECTION DATA,DATA,ALIGN=1
.DATA SIZEOF(PROGRAM),SIZEOF(DMY)
:
SECTION DMY,DUMMY,ALIGN=1
:
```

■ Pseudo-instructions for which an Expression Containing the Size Operator cannot be Specified

An expression containing the size operator cannot be used for the pseudo-instructions listed below.

Any expression containing the size operator is valid (can be specified) for machine instructions.

- .END instruction (start address)
- .SECTION instruction (boundary value and start address)
- .ALIGN instruction (boundary value)
- .ORG instruction (expression)
- .SKIP instruction (expression)
- .EQU instruction (expression)
- .DATAB instruction (expression 1)
- .FDATAB instruction (expression 1)
- .RES instruction (expression)
- .FRES instruction (expression)
- .SDATAB instruction (expression)
- .FORM instruction (numbers of lines and digits)
- .SPACE instruction (number of blank lines)

■ Obtaining the Size of a Section in another Module

To obtain the size of a section in another module, create a blank section for the desired section, then use the size operator.

[Example] To obtain the size of section S in another module:

```
.SECTION    S,STACK,ALIGN=4      /* Create a blank section for section S */
.SECTION    ,CODE,ALIGN=2
MOVW       A, #SIZEOF(S)        /* Use the size operator to obtain the size of section S*/
```

7.11.5 Precedence of Operators

The precedence of operators is the same as that in C.

■ Precedence of Operators

Table 7.11-4 shows the precedence of operators.

Table 7.11-4 Precedence of Operators

Precedence	Operator	Associativity	Target expression
1	()	Left-to-right	Parenthesized expression
2	! ~ + - BITADR BITPOS SIZEOF	Right-to-left	Unary expression
3	* / %	Left-to-right	Multiplication expression
4	+ -	Left-to-right	Addition expression
5	<< >>	Left-to-right	Shift expression
6	< <= > >=	Left-to-right	Relational expression
7	= = !=	Left-to-right	Equality expression
8	&	Left-to-right	Bitwise AND expression
9	^	Left-to-right	Bitwise XOR expression
10		Left-to-right	Bitwise OR expression
11	&&	Left-to-right	Logical AND expression
12		Left-to-right	Logical OR expression

[Example]

```

        .IMPORT      imp
        .DATA        10 + 2 * 3
data_h:  .DATA        imp >> 8 & 0xff
data_l:  .DATA        imp & 0xff

```

7.12 Comments

There are two types of comment: line comment and range comment.

A line comment starts with a semicolon (;) or two slashes (//).

A comment can be enclosed in /* and */, as in C.

■ Comments

[Format]

```
/* Range comment */
// Line comment
; Line comment
```

[Description]

A comment can start in any column.

There are two types of comment: line comment and range comment.

A line comment starts with a semicolon (;) or two slashes (//).

A comment can be enclosed in /* and */, as in C.

Comments of this type are called range comments.

A range comment can be inserted anywhere.

[Example]

```
/*-----
```

```
Range comment
```

```
-----*/
```

```
; Line comment
```

```
// Line comment
```

```
        .SECTION D1,DATA,ALIGN=2    // Line comment
```

```
/* Range comment */ .DATA 1
```

```
        .DATA /* Range comment */ 0xff ; Line comment
```

CHAPTER 8

SECTIONS

The memory space can be divided into different areas. Each area that is used meaningfully is called a section. Based on how used, the memory space is divided into sections. Sections having the same name can be grouped.

Because linkage between sections having the same name is created during linking, using sections enables the memory space to be handled logically.

This chapter describes the coding of sections.

8.1 Section Description Format

8.2 Section Types

8.3 Section Types and Attributes

8.4 Section Allocation Patterns

8.5 Section Linkage Methods

8.6 Multiple Descriptions of a Section

8.7 Setting ROM Storage Sections

8.1 Section Description Format

The `.SECTION` instruction declares the start of a section description.
Multiple descriptions of a section are allowed.
The section type is determined by the type of the section.
The allocation pattern of a section is determined by its allocation pattern.

■ Section Description Format

[Format]

```
.SECTION section-name [,section-type] [, section-allocation-pattern]
:
text
:
```

- section-type: { CODE|DATA|CONST|COMMON|STACK|DUMMY|O|IOCOMMON|DIR|DIRCONST|DIRCOMMON }
- section-allocation-pattern: { ALIGN = boundary-value | LOCATE=start-address }
- boundary-value: Expression (absolute expression)
- start-address: Expression (absolute expression)

[Description]

The `.SECTION` instruction declares the start of a section description.
Instructions for generating object code or updating the location counter cannot precede the first occurrence of the `.SECTION` instruction.
Multiple descriptions of a section are allowed.
For more information, see Section "8.6 Multiple Descriptions of a Section".

■ Section Types

Specifying the type of a section determines the attribute of that section.

For more information, see Section "8.2 Section Types".

There are 11 section types, which are listed below.

By default, CODE is assumed.

- CODE Code section
- DATA Data section
- CONST Data section with initial values
- COMMON Common section
- STACK Stack section
- DUMMY Dummy section
- IO I/O section
- IOCOMMON Common I/O section
- DIR Direct section
- DIRCONST Direct section with initial values
- DIRCOMMON Common direct section

■ Section Allocation Patterns

One of the following section allocation patterns is specified:

● ALIGN: Relative section

An allocation address in memory is determined by the linker.

A section is aligned on a memory boundary specified by the boundary value.

● LOCATE: Absolute section

A section is allocated starting at the specified start address.

By default, ALIGN=1 is used.

For more information, see Section "8.4 Section Allocation Patterns".

[Example]

```
.SECTION          PROG,CODE,ALIGN=2
/* Section name:          PROG */
/* Section type:          Code section */
/* Section allocation pattern:  Relative section (boundary value 2) */
:
.SECTION          VAL,DATA,LOCATE=0x1000
/* Section name:          VAL */
/* Section type:          Data section */
/* Section allocation pattern:  Absolute section starting at address 0x1000 */
:
```

8.2 Section Types

The type of a section depends on the type of data that will be stored in.

■ Section Types

The type of a section depends on the type of data that will be stored in.

The following explains the section types and general data types:

● Code section (CODE specification)

Program code is stored.

Machine instructions are usually used.

[Example]

```

        .SECTION      PROG, CODE, ALIGN=1
start_program:
        MOVW          A, #0
        PUSHW         A
        CALL          _func

```

● Data section (DATA specification)

Data without initial values is stored.

The .RES and .FRES instructions are usually used.

[Example]

```

        .SECTION      VAL, DATA, ALIGN=1
v1:     .RES          1
v2:     .RES.H 2

```

Initial-value data that is allowed to be changed is also stored in a data section.

For more information, see Section "8.7 Setting ROM Storage Sections".

[Example]

```

        .SECTION      INIT, DATA, ALIGN=1
ptbl:   .DATA         10, 11, 12      /* Data values that are allowed to be changed */

```

● Data section with initial values (CONST specification)

Initial-value data that does not change is stored.

This section type is usually specified to code data values that will be stored in ROM.

[Example]

```

        .SECTION      PARAM, CONST, ALIGN=1
param1: .DATA         10
param2: .DATA         -1

```


● Common section (COMMON specification)

This section type is specified to allocate shared variables or shared areas.
Shared linkage between common sections is created by the linker.

[Example]

```

        .SECTION      COMval,COMMON,ALIGN=1
val:    .DATA          500
tbl:    .DATAB.B       10,0xff

```

● Stack section (STACK specification)

This section is used to allocate a stack area.
The .RES instruction is usually used to allocate an area.

[Example]

```

        .SECTION      STACKAREA,STACK,ALIGN=1
        .RES.B         0x1000      /* 4K bytes */

```

● Dummy section (DUMMY specification)

Dummy section descriptions are not output as object code.
Dummy section descriptions, therefore, are meaningful only when defined symbols within them are handled or when the dummy section size is calculated.

[Example: To use the IO area as a dummy section:]

Header file iodef.h

```

        .SECTION      IO_MAP,DUMMY,LOCATE=0x100
iodata1: .RES.H 1
iodata2: .RES.H 1
iodata3: .RES.H 1

```

Source file a.asm

```

#include    "iodef.h"    /* Read the IO definition file */
        .SECTION      P,CODE,ALIGN=2
        :
        MOVW A, @iodata2 /* Read a value from I/O */
        :

```

Source file b.asm

```

#include    "iodef.h"    /* Read the IO definition file */
        .SECTION      P,CODE,ALIGN=2
        :
        MOVW @iodata2, A /* Write a value from I/O */
        :

```

● I/O section (IO specification)

Data is stored in an area to which various I/O ports have been allocated.

- Common I/O section (IOCOMMON specification)

Data is stored in an area to which various I/O ports have been allocated.

Shared linkage between common I/O sections is created by the linker.

- Direct section (DIR specification)

Data is stored in a direct access area.

- Direct section with initial values (DIRCONST specification)

Initial-value data that does not change is stored in a direct access area.

- Common direct section (DIRCOMMON specification)

Data is stored in a direct access area.

Shared linkage between common direct sections is created by the linker.

8.3 Section Types and Attributes

When section code is output to an object, five attributes (area type, linkage type, read attribute, write attribute, and execute attribute) for each section are set as section information.

These attributes are checked by the linker during linking.

■ Section Types and Attributes

When section code is output to an object, five attributes (area type, linkage type, read attribute, write attribute, and execute attribute) for each section are set as section information.

These attributes are checked by the linker during linking.

Table 8.3-1 lists the attribute for each section type.

Table 8.3-1 Attribute for Each Section Type

Section type	Area type	Linkage type	Read	Write	Execute
CODE	Code	Concatenated linkage	o	x	o
DATA	Data	Concatenated linkage	o	o	x
CONST	Constant	Concatenated linkage	o	x	x
COMMON	Data	Shared linkage	o	o	x
STACK	Stack	Concatenated linkage	o	o	x
IO	Data (I/O)	Concatenated linkage	o	o	x
IOCOMMON	Data (I/O)	Shared linkage	o	o	x
DIR	Direct	Concatenated linkage	o	o	x
DIRCONST	Direct	Concatenated linkage	o	x	x
DIRCOMMON	Direct	Shared linkage	o	o	x

o : Allowed x : Not allowed

Note:

Dummy sections do not have an attribute because their code is not output to an object.

8.4 Section Allocation Patterns

There are two types of section allocation pattern:

- Relative section
- Absolute section

The allocation address for a relative section is determined by the linker.

The allocation address for an absolute section can be specified by the assembler.

■ Section Allocation Patterns

There are two section allocation patterns:

● Relative section (ALIGN specification)

The allocation address for a relative section is determined by the linker. The value of a symbol having an address within a relative section is unknown until linking has been completed. A symbol of this type is called a relative symbol.

A section whose location need not be known can be set as a relative section.

In general, since most programs contain code for relative sections, the linker determines their location addresses.

The linker allocates relative sections to the memory space based on the boundary values specified with ALIGN.

[Example]

```

        .SECTION      PROG, CODE, ALIGN=1
_main:
        PUSHW        IX
        MOVW         A, SP
        MOVW         IX, A
        PUSHW        A
        PUSHW        A

```

● Absolute section (LOCATE specification)

The allocation address for an absolute section can be specified by the assembler.

The linker allocates absolute sections to the memory space starting at the start addresses specified with LOCATE. The value of a symbol having an address within an absolute section is known because the address is determined. A symbol of this type is called an absolute symbol.

An absolute section can be used to define an EIT vector table or IO area.

The following format can be used to specify absolute sections:

LOCATE=address

[Example]

```
        .SECTION      IO,DATA,LOCATE=0x0000
timer_cmd:      /* Timer command address */
        .RES          1
timer_data:     /* Timer data address */
        .RES          1
```

8.5 Section Linkage Methods

Linkage between sections is created by the linker.
There are two types of section linkage: concatenated linkage and shared linkage.
In concatenated linkage, sections having the same name are concatenated in a memory area.
In shared linkage, a memory area shared by sections having the same name is created.

■ Section Linkage Methods

Linkage between sections is created by the linker.
There are two types of section linkage: concatenated linkage and shared linkage.
The linkage method for a section depends on the type of the section.
Table 8.5-1 lists the section types and linkage methods.

Table 8.5-1 Section Types and Linkage Methods

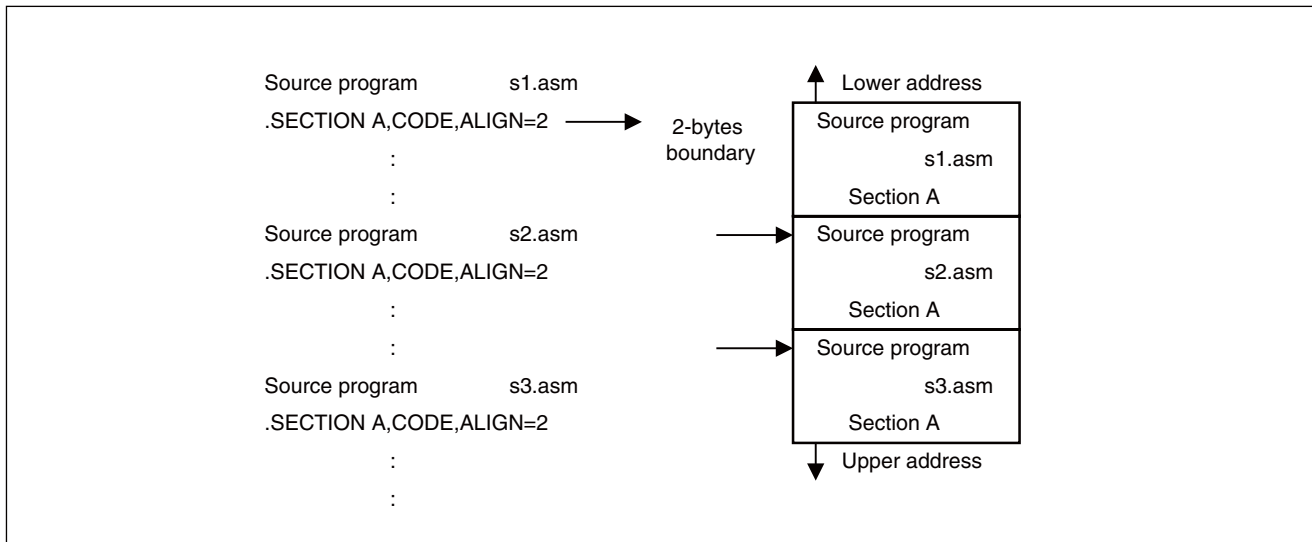
Section type	Linkage method
CODE DATA CONST STACK IO DIR	Concatenated linkage
COMMON IOCOMMON DIRCOMMON	Shared linkage

■ Concatenated Linkage

Sections having the same name, specified in different source programs, are concatenated in a memory area. Note that the same type and same allocation pattern must be specified for these sections.

[Example]

Figure 8.5-1 Example of Concatenated Linkage

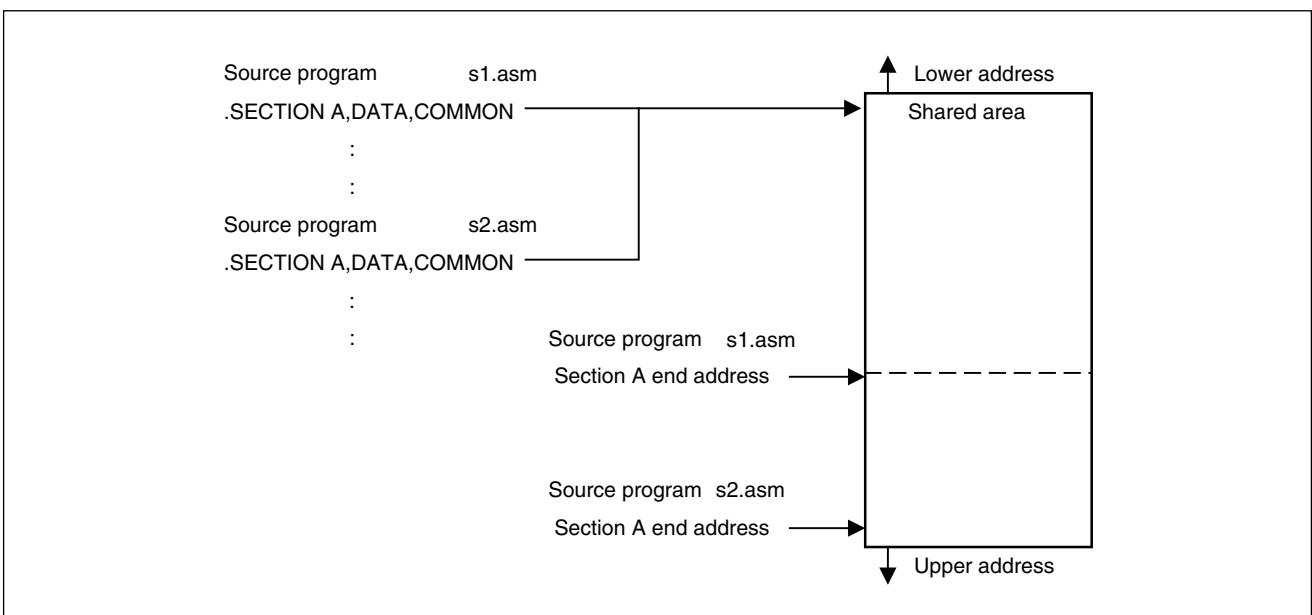


■ Shared Linkage

A memory area shared by sections having the same name, specified in different programs, is created. The size of this area is the size of the largest section.

[Example]

Figure 8.5-2 Example of Shared Linkage



8.6 Multiple Descriptions of a Section

A single source program can contain multiple occurrences of the `.SECTION` instruction, each of which specifies the same section name.

A set of section descriptions specifying the same section name is handled as a single continuous section description.

■ Multiple Descriptions of a Section

A single source program can contain multiple occurrences of the `.SECTION` instruction, each of which specifies the same section name.

A set of section descriptions specifying the same section name is handled as a single continuous section description.

The first occurrence of the `.SECTION` instruction declares the start of a section description. Subsequent occurrences of the `.SECTION` instruction indicate the continuation of the section description.

For the second and subsequent descriptions specifying the same section name, the location counter inherits the value from the previous description.

Multiple occurrences of the `.SECTION` instruction must not specify different section types or section allocation patterns.

By default, the second and subsequent section descriptions with the same name inherit the section types and allocation patterns that have already been defined.

[Example]

```
.SECTION    P,CODE,ALIGN=2
Text 1

.SECTION    D,DATA,ALIGN=4
Text 2

.SECTION    P,CODE
Text 3

.SECTION    D
Text 4
```

The above source program is handled in the same way as the following source program.

```
.SECTION    P,CODE,ALIGN=2
Text 1
Text 3

.SECTION    D,DATA,ALIGN=4
Text 2
Text 4
```


8.7 Setting ROM Storage Sections

This section describes how to write programs or data values that will be stored in ROM. Program code, as well as initial-value data that does not change, can be stored in ROM, then used from ROM. For initial-value data that is allowed to be changed, however, specify that the data is stored in ROM with the `-sc` linker option. Also specify that it should be transferred to RAM at run time so that it can be used.

■ Setting ROM Storage Sections

Programs and data values that will be stored in ROM can be classified as follows:

1. Program code (CODE section)
2. Initial-value data that does not change (CONST section)
3. Initial-value data that is allowed to be changed (DATA section)

The data of 1) or 2) can be stored in ROM, and then used from ROM. Before the data of 3) can be used, however, because it is allowed to be changed, it must have already been transferred from ROM to RAM. Therefore, specify that the data is stored in the ROM area with the linker, and specify that it should be transferred to the RAM area at run time so that it can be used.

Sections that will be used for ROM-to-RAM transfer should be given descriptive names.

The C compiler uses `INIT` as the name of a section for initial-value data that is allowed to be changed. The assembler should also use `INIT`.

■ Transfer of Initial-value Data

The initial-value data that is allowed to be changed is transferred from ROM to RAM as described below.

With the linker, use the `-sc` option to specify the name of sections for ROM-to-RAM transfer. The linker then automatically generates the following symbols for the specified section names:

- `_ROM_section-name`
- `_RAM_section-name`

The symbols indicate the start addresses in the ROM and RAM areas, respectively.

The following example specifies a ROM area (0xE100 to 0xE1FF) and a RAM area (0x0100 to 0x01FF).

It also sets the `INIT` sections for ROM-to-RAM transfer.

```
flnk896s -ro ROM=0xE100/0xE1FF -ra RAM=0x0100/0x01FF -sc @INIT=ROM,INIT=RAM
sample.obj
```

In this case, `_ROM_INIT` and `_RAM_INIT` are generated, then `_ROM_INIT` and `_RAM_INIT` are set to 0xE100 and 0x0100, respectively.

For more information, see the "SOFTUNE Linkage Kit Manual".

The symbols are used to create a ROM-to-RAM transfer program, as shown in the following example. The program is integrated into a startup routine.

[Example]

```

        .IMPORT      _ROM_INIT          /* Start address of the INIT section in ROM */
        .IMPORT      _RAM_INIT          /* Start address of the INIT section in RAM */
        .SECTION     INIT,DATA,ALIGN=4  /* Initial value data section which can be changed */
ptbl:
        .DATA        10,11,12          /* Initial-value data is stored in ROM */
:
        .SECTION     CODE,CODE,ALIGN=2 /* ROM-to-RAM transfer program */
init_copy:
        MOVW         IX,#_ROM_INIT      /* Source (ROM) address */
        MOVW         EP,#_RAM_INIT      /* Destination (RAM) address */
        MOVW         A,#SIZEOF(INIT)    /* Obtain the INIT section size */
        BZ           init_copy_end      /* Test the transfer size for 0 */
init_copy_loop:
        MOV          A,@IX+0            /* Obtain the ROM data */
        MOV          @EP,A             /* Transfer to RAM */
        INCW         EP                /* Increase the source address */
        INCW         IX                /* Increase the destination address */
        XCH          A,T               /* Load the size to accumulator */
        DECW         A                 /* Reduce the size by 1 */
        BNZ          init_copy_loop    /* Repeat until the size becomes 0 */
init_copy_end:

```

CHAPTER 9

MACHINE INSTRUCTIONS

This chapter describes the formats of machine instructions and the rules governing how to write them. For details of machine instructions and their addressing mode, see the instruction manual or programming manual of the relevant CPU.

9.1 Machine Instruction Format

9.2 Operand Field Format

9.1 Machine Instruction Format

This chapter describes the format of machine instructions and the rules governing how to write them.

■ Machine Instruction Format

Machine instructions are interpreted and executed by the CPU to run a program.

For details of machine instructions, see the instruction manual or programming manual of the relevant CPU.

The general format of machine instructions is shown below.

[Format]

[symbol]	operation	[operand[,operand] ...]
----------	-----------	--------------------------

operation: Instruction mnemonic

operand: Addressing mode

[Description]

If the symbol field holds a symbol, the current address is assigned to the symbol.

The operation field holds an instruction mnemonic.

The operand field holds operands necessary for the machine instruction. Operands must be separated with a comma (,).

Each operand specifies an addressing mode.

[Examples]

```
PUSHW    IX
MOVW     A,SP
MOVW     IX,A
PUSHW     A
CALL     _proc0
```

9.2 Operand Field Format

This section describes the format of the operand field.

■ Operand Field Format

[Format]

[operand [,operand] ...]

operand: Addressing mode

[Description]

An addressing mode that can be written in the operand field of a machine instruction is determined according to the machine instruction.

If a machine instruction has more than one operand, they are separated with a comma (,).

For the addressing modes that can be written in the operand field of a machine instruction and their details, see the instruction manual or programming manual of the relevant CPU.

■ Order of Operands

The order in which operands are written is determined by the machine instruction type. The basic rules are as follows:

● Arithmetic and logical operation instructions

In arithmetic and logical operation instructions, an operation is performed between the first and second operands, and the result is stored in the first operand.

First operand <- first operand .op. second operand

[Example]

ADD A,#5 /* A ← A + 5 */

● Transfer instructions

In transfer instructions, a transfer occurs from the second operand to the first operand.

First operand <- second operand

[Example]

MOV A,R5 /* A ← R5 */

CHAPTER 10

ASSEMBLER PSEUDO- INSTRUCTIONS

Unlike machine instructions, assembler pseudo-instructions tell the assembler what to do.

The assembler pseudo-instructions are categorized into the following eight groups:

- Program structure definition instruction
- Address control instruction
- Program linkage instruction
- Symbol definition instruction
- Area definition instruction
- Debugging information output control instruction
- Library file specification instruction
- List output control instruction

This chapter describes the format and function of each assembler pseudo-instruction.

10.1 Scope of Integer Constants Handled by Pseudo-Instructions

10.2 Program Structure Definition Instructions

10.3 Address Control Instructions

10.4 Program Linkage Instructions

10.5 Symbol Definition Instructions

10.6 Area Definition Instructions

10.7 Debugging Information Output Control Instruction

10.8 Library File Specification Instruction

10.9 List Output Control Instructions

10.1 Scope of Integer Constants Handled by Pseudo-Instructions

Pseudo-instructions can specify integer constants having five different sizes: bit (1 bit), byte (8 bits), halfword (16 bits), long word (32 bits), word (16 bits).

If no size is specified, one word (16 bits) is assumed.

■ Scope of Integer Constants Handled by Pseudo-instructions

Pseudo-instructions can specify integer constants having the following five different sizes:

- Bit (1 bit)
- Byte (8 bits)
- Halfword (16 bits)
- Long word (32 bits)
- Word (16 bits).

If no size is specified, one word (16 bits) is assumed.

Table 10.1-1 lists the size specifiers used in pseudo-instructions.

Table 10.1-1 Size Specifiers

Size specifier	Data size
I (bit)	1 bit
B (byte)	8 bits (1 byte)
H (halfword)	16 bits (2 bytes)
L (long word)	32 bits (4 bytes)
W (word)	16 bits (2 bytes)

10.2 Program Structure Definition Instructions

A program structure definition instruction signifies the end of a source program, declares a module name, or defines section information.

■ Program Structure Definition Instructions

There are three different program structure definition instructions:

- `.PROGRAM:` Declares a module name
- `.END:` Signifies the end of a source program
- `.SECTION:` Defines a section

10.2.1 .PROGRAM Instruction

The **.PROGRAM** instruction specifies a module name.

A module is named in accordance with naming rules.

If the **.PROGRAM** instruction is omitted, the main file name of an object file is used as the module name.

■ .PROGRAM Instruction

[Format]

	.PROGRAM	module-name
--	-----------------	-------------

[Description]

The **.PROGRAM** instruction specifies the name of the module.

The module name is determined in accordance with naming rules.

The **.PROGRAM** instruction can be used in a source program only once.

If the **.PROGRAM** instruction is omitted, the main file name of an object file is used as the module name.

If the main file name violates a naming rule, a warning message is output, and any character not allowed in the module name is replaced with an underscore (_).

[Example]

```
.PROGRAM      test_name
```

■ Relationship with Startup Options

If the **-name** option is specified, a name specified in the option is used as the module name.

10.2.2 .END Instruction

The .END instruction signifies the end of a source program.
 The .END instruction can be omitted. If it is omitted, assembly continues until all source programs are assembled.
 A start address can be specified in the .END instruction.

■ .END Instruction

[Format]

	.END	[start-address]
--	------	-----------------

start-address: Expression

[Description]

The .END instruction signifies the end of a source program.
 The .END instruction can be omitted. If it is omitted, assembly continues until all source programs are assembled.
 If a source program follows the .END instruction, it is not assembled.
 If start-address is specified in the .END instruction, it sets the start address of the program.
 The program start address is referred by the SOFTUNE Workbench to load the program, and the corresponding value is set in the program counter.
 If start-address is omitted, no start address is set up.
 start-address must be an absolute or simple relative expression.
 start-address must point within a code section.

[Example]

```

        .SECTION          PROG, CODE, ALIGN=2
start:
        :
        .END              start
    
```

10.2.3 .SECTION Instruction

The **.SECTION** instruction declares the beginning of a section and specifies the type and location format of the section.

■ .SECTION Instruction

[Format]

	<code>.SECTION</code>	<code>section-name[,specifier[,specifier]]</code>
--	-----------------------	---

specifier:	{ section-type section-location-format }
section-type:	{ CODE DATA CONST COMMON STACK DUMMY IO IOCOMMON DIR DIRCOMMON DIRCONST }
section-location-format:	{ ALIGN=boundary-value LOCATE=start-address }
boundary-value:	Expression (absolute expression)
start-address:	Expression (absolute expression)

[Description]

The `.SECTION` instruction declares the beginning of a section and specifies the type and location format of the section.

The section is named in accordance with naming rules.

Only one section-type and section-location-type can be specified in one `.SECTION` instruction.

If section-type is omitted, a code section is assumed.

If section-location-type is omitted, `ALIGN=1` is assumed.

■ Section-type

The section-type operand specifies a section type.

See Section "8.2 Section Types", for details.

- `CODE`A code section is specified.
- `DATA`A data section is specified.
- `CONST`A data section with initial values is specified.
- `COMMON`A common section is specified.
- `STACK`A stack section is specified.
- `DUMMY`A dummy section is specified
- `IO`An I/O section is specified.
- `IOCOMMON`A common I/O section is specified.
- `DIR`A direct section is specified
- `DIRCONST`A direct section with initial values is specified
- `DIRCOMMON`A common direct section is specified

■ Section-location-format

section-location-format specifies how the section is located.

See Section "8.4 Section Allocation Patterns", for details.

● ALIGN=boundary-value

A relative section is specified.

The section is aligned on a specified boundary by the linker.

boundary-value must be an absolute expression.

boundary-value must be 2 raised to the nth power, where n is an integer.

● LOCATE=start-address

An absolute section is specified.

The section is aligned on a specified start address.

start-address must be an absolute expression.

[Examples]

```
.SECTION      P, CODE, ALIGN=1
:
.SECTION      D, DATA, LOCATE=0x1000
:
.SECTION      C, CONST, LOCATE=0x2000
:
.SECTION      V, COMMON, ALIGN=1
:
```

10.3 Address Control Instructions

An address control instruction changes the value in the location counter.

■ Address Control Instructions

There are three different address control instructions:

- `.ALIGN:` Creates alignment on a boundary.
- `.ORG:` Changes the location counter value.
- `.SKIP:` Increments the location counter value.

10.3.1 .ALIGN Instruction

If the value in the location counter is not on a specified boundary, the **.ALIGN** instruction increments the value until it is aligned on the specified boundary. If the value is already on a specified boundary, the **.ALIGN** instruction does nothing.

■ .ALIGN Instruction

[Format]

	<code>.ALIGN</code>	boundary-value
--	---------------------	----------------

boundary-value: Expression (absolute expression)

[Description]

If the value in the location counter is not on a specified boundary, the **.ALIGN** instruction increments the value until it is aligned on the specified boundary. If the value is already on a specified boundary, the **.ALIGN** instruction does nothing.

boundary-value must be an absolute expression.

boundary-value must be a positive value not greater than 0x8000.

[Condition]

Boundary value in <code>.SECTION</code> instruction \geq boundary value in <code>.ALIGN</code>
--

(This condition does not apply to absolute sections.)

[Examples]

```

.SECTION      D,DATA,ALIGN=8 /* (8=2^3) */
.DATA.B      0
.ALIGN       8
.DATA.B      0xff
.ALIGN       4
:

```

10.3.2 .ORG Instruction

The **.ORG** instruction sets the value of a specified expression in the location counter.

■ **.ORG Instruction**

[Format]

	.ORG	expression
--	-------------	------------

[Description]

The **.ORG** instruction sets the value of a specified expression in the location counter.

If the **.ORG** instruction is used within an absolute section, it is impossible to return the location counter to a location before the start address specified (at **LOCATE**) in a **.SECTION** instruction.

The expression specified in the **.ORG** instruction must be an absolute expression or a simple relative expression that has, as its value, a symbol in the same section as this instruction.

[Examples]

```
.SECTION      D,DATA,LOCATE=0x100
.DATA        0
.ORG         0x200
.DATA        2
.ORG         0x300
.DATA        3
:
```


10.3.3 .SKIP Instruction

The **.SKIP** instruction increments the location counter by the value of the specified expression.

■ .SKIP Instruction

[Format]

	.SKIP	expression
--	--------------	------------

[Description]

The **.SKIP** instruction increments the location counter by the value of the specified expression.

The expression must be an absolute expression.

[Examples]

```
.SECTION      D,DATA,ALIGN=2
.DATA.H      0x0505
.SKIP        2
.DATA.H      0x1010
:
```

10.4 Program Linkage Instructions

A program linkage instruction is used to enable programs to share symbols.

A program linkage instruction is used to declare an external definition for a label so that the symbol can be referred by other programs. It is also used to declare an external reference for a symbol in another program so that it can be used in the program in which the program linkage instruction is issued.

■ Program Linkage Instructions

There are three different program linkage instructions:

- `.EXPORT:` Declares an external definition symbol.
- `.GLOBAL:` Declares an external definition/reference symbol.
- `.IMPORT:` Declares an external reference symbol.

10.4.1 .EXPORT Instruction

The **.EXPORT** instruction enables symbols defined in the program in which it is issued to be referred in other programs.

■ .EXPORT Instruction

[Format]

	.EXPORT	symbol[,symbol] ...
--	----------------	---------------------

[Description]

The **.EXPORT** instruction enables symbols defined in the program in which it is issued to be referred in other programs.

The symbols must be defined in the program that contains the **.EXPORT** instruction of interest.

The following two types of symbols can be specified in the **.EXPORT** instruction:

- Symbol with an absolute value
- Symbol with an address

No error is reported if identical symbols are specified.

[Examples]

-----Program 1 -----

```

        .EXPORT      abc1,abc2
        :
abc1:   .EQU         5*3
        :
abc2:   .ADDC        A,R5

```

-----Program 2 -----

```

        .IMPORT      abc1,abc2
        :
        .DATA        abc1
        :
        .DATA        abc2
        :

```

10.4.2 .GLOBAL Instruction

The **.GLOBAL** instruction declares symbols for external definition or reference. If a symbol specified in the **.GLOBAL** instruction is defined in the program that contains this instruction, the symbol is declared for external definition.

■ .GLOBAL Instruction

[Format]

	<code>.GLOBAL</code>	<code>symbol[,symbol] ...</code>
--	----------------------	----------------------------------

[Description]

The **.GLOBAL** instruction declares symbols for external definition or reference. If a symbol specified in the **.GLOBAL** instruction is defined in the program that contains this instruction, the symbol is declared for external definition.

The following two types of external definition symbols can be specified in the **.GLOBAL** instruction:

- Symbol with an absolute value
- Symbols with an address

If a symbol specified in the **.GLOBAL** instruction is not defined in the program that contains this instruction, the symbol is declared for external reference.

No error is reported if identical symbols are specified.

[Examples]

```
.GLOBAL    abc,sub    /* abc is an external definition symbol. */
                                /* sub is an external reference symbol. */

:
abc:    CALL    sub
```

10.4.3 .IMPORT Instruction

The **.IMPORT** instruction declares that symbols specified in this instruction are defined in programs other than the one containing this instruction.

■ .IMPORT Instruction

[Format]

	<code>.IMPORT</code>	<code>symbol[,symbol] ...</code>
--	----------------------	----------------------------------

[Description]

The **.IMPORT** instruction declares that symbols specified in this instruction are defined in programs other than the one containing this instruction.

The symbols specified in the **.IMPORT** instruction must be specified as external in the programs from which they are imported.

No error is reported if identical symbols are specified.

[Examples]

```

-----Program 1 -----
        .IMPORT      xyz1,xyz2
        :
        .DATA        xyz1
        :
        .DATA        xyz2
-----Program 2 -----
        .EXPORT      xyz1,xyz2
        :
xyz1:    .EQU         5*3
        :
xyz2:    ADDC         A,R5

```

10.5 Symbol Definition Instructions

A symbol definition instruction assigns a value to a symbol.

■ Symbol Definition Instructions

There is a symbol definition instruction:

- `.EQU:` Assigns a value to a symbol.

10.5.1 .EQU Instruction

The .EQU instruction assigns the value of a specified expression to a specified symbol.

■ .EQU Instruction

[Format]

symbol	.EQU	expression
--------	------	------------

[Description]

The .EQU instruction assigns the value of a specified expression to a specified symbol.

It is impossible to assign a value to a symbol that has already been defined.

The expression must be an absolute or simple relative expression.

[Examples]

```
TEN:      .EQU      10          /* TEN=10 */
ONE:      .EQU      TEN/10      /* ONE=TEN/10 */
val1:     .DATA     0xFFFF0000
val2:     .EQU      val1+2
```

10.6 Area Definition Instructions

An area definition instruction sets constants in memory and secures memory areas.

■ Area Definition Instructions

There are 18 different area definition instructions:

- `.DATA:` Defines an integer constant.
- `.BIT:` Defines a 1-bit integer constant.
- `.BYTE:` Defines an 8-bit integer constant.
- `.HALF:` Defines a 16-bit integer constant.
- `.LONG:` Defines a 32-bit integer constant.
- `.WORD:` Defines a 16-bit integer constant.
- `.DATAB:` Defines an integer constant block.
- `.FDATA:` Defines a floating-point constant.
- `.FLOAT:` Defines a 32-bit floating-point constant.
- `.DOUBLE:` Defines a 64-bit floating-point constant.
- `.FDATAB:` Defines a floating-point constant block.
- `.RES:` Defines an integer constant area.
- `.FRES:` Defines a floating-point constant area.
- `.SDATA:` Defines a character string.
- `.ASCII:` Defines a character string.
- `.SDATAB:` Defines a character string block.
- `.STRUCT:` Defines the beginning of a structure.
- `.ENDS:` Defines the end of a structure.

10.6.1 .DATA, .BIT, .BYTE, .HALF, .LONG, and .WORD Instructions

The following range definition instructions set the values of specified expressions in memory as stated below:

The .DATA instruction secures memory areas each having the specified size.

The .BIT instruction secures memory areas each having a size of one bit.

The .BYTE instruction secures memory areas each having a size of one byte.

The .HALF instruction secures memory areas each having a size of one halfword.

The .LONG instruction secures memory areas each having a size of one long word.

The .WORD instruction secures memory areas each having a size of one word.

■ .DATA Instruction

[Format]

[symbol]	.DATA.s	expression[.expression] ...
----------	---------	-----------------------------

Size specifier (s): I Bit (1 bit)
 B Byte (8 bits)
 H Halfword (16 bits)
 L Long word (32 bits)
 W Word (16 bits) <default >

[Description]

The .DATA instruction sets the values of specified expressions in memory areas each having the specified size (.s).

If a size specifier (.s) is omitted, one word is assumed.

The expression specified in the instruction can be either absolute or relative.

[Examples]

```
.DATA.B    0x12,0x23,0xa3
.DATA      -1,0xffff
```

■ .BIT Instruction

[Format]

[symbol]	.BIT	expression[.expression] ...
----------	------	-----------------------------

[Description]

The .BIT instruction secures memory areas each having a size of one bit.

This instruction is equivalent to the following definition:

```
.DATA.I    expression[,expression]...
```

■ .BYTE Instruction**[Format]**

[symbol]	.BYTE	expression[.expression] ...
----------	-------	-----------------------------

[Description]

The .BYTE instruction secures memory areas each having a size of one byte (8 bits).

This instruction is equivalent to the following definition:

.DATA.B expression[,expression] ...

■ .HALF Instruction**[Format]**

[symbol]	.HALF	expression[,expression] ...
----------	-------	-----------------------------

[Description]

The .HALF instruction secures memory areas each having a size of one halfword (16 bits).

This instruction is equivalent to the following definition:

.DATA.H expression[,expression] ...

■ .LONG Instruction**[Format]**

[symbol]	.LONG	expression[,expression] ...
----------	-------	-----------------------------

[Description]

The .LONG instruction secures memory areas each having one long word (32 bits).

This instruction is equivalent to the following definition:

.DATA.L expression[,expression] ...

■ .WORD Instruction**[Format]**

[symbol]	.WORD	expression[,expression] ...
----------	-------	-----------------------------

[Description]

The .WORD instruction secures memory areas each having a size of one word (16 bits).

This instruction is equivalent to the following definition:

.DATA.W Expression[,expression] ...

10.6.2 .DATAB Instruction

The .DATAB instruction sets a specified value in a specified number of memory areas each having the specified size.

If no size is specified, 1 word is assumed.

■ .DATAB Instruction

[Format]

[symbol]	.DATAB.s	expression-1,expression-2
----------	----------	---------------------------

Size specifier (s):

- I..... Bit (1 bit)
- B Byte (8 bits)
- H Halfword (16 bits)
- L Long word (32 bits)
- W Word (16 bits) <default >

[Description]

The .DATAB instruction sets a specified value (expression 2) in a specified number (expression 1) of memory areas each having the specified size (.s).

If a size specifier (.s) is omitted, one word is assumed.

Expression 1 must be an absolute expression.

Expression 2 can be either absolute or relative.

[Example]

```
.DATAB.B 4,0x12
```

Note:

If expression 1 in the .DATAB instruction is an excessively large number, it will take a long time to execute the instruction, because the instruction sets the value of expression 2 in memory by a specified number (expression 1) of repetitions. Consequently, the value of expression 1 cannot be greater than 1048575, since an error will be reported.

10.6.3 .FDATA, .FLOAT, and .DOUBLE Instructions

The following range definition instructions set floating-point constants in memory as stated below:

The **.FDATA** instruction secures memory areas each having the size that matches a specified type specifier.

The **.FLOAT** instruction secures memory areas each having the single-precision (4-byte) size.

The **.DOUBLE** instruction secures memory areas each having the double-precision (8-byte) size.

■ .FDATA Instruction

[Format]

[symbol]	.FDATA.t	floating-point-constant[,floating-point-constant] ...
----------	----------	---

Type specifier (t): S.....Single-precision floating-point constant, 32-bit (4-byte) <default >
 D.....Double-precision floating-point constant, 64-bit (8-byte)

[Description]

The **.FDATA** instruction sets floating-point constants in memory areas each having the size that matches a specified type specifier (.t).

Neither an expression nor an integer constant can be specified instead of a floating-point constant.

If a type specifier (.t) is omitted, the single-precision size is assumed.

See Section "7.9 Floating-Point Constants", for floating-point constants.

[Examples]

```
.FDATA.S    2.1e4      /* Single precision */
.FDATA.D    3.2e5      /* Double precision */
.FDATA      4.3e-2     /* Single precision */
.FDATA      0xFFFF0000 /* Single precision */
```

■ .FLOAT Instruction

[Format]

[symbol]	.FLOAT	floating-point-constant[,floating-point-constant] ...
----------	--------	---

[Description]

The **.FLOAT** instruction secures memory areas each having the single-precision (4-byte) size.

This instruction is equivalent to:

```
.FDATA.S    floating-point-constant[,floating-point-constant]...
```

■ .DOUBLE Instruction

[Format]

[symbol]	.DOUBLE	floating-point-constant[,floating-point-constant] ...
----------	---------	---

[Description]

The .DOUBLE instruction reserves memory areas each having the double-precision (8-byte) size.

This instruction is equivalent to:

.FDATA.D floating-point-constant[,floating-point-constant]...

10.6.4 .FDATAB Instruction

The **.FDATAB** instruction sets a specified floating-point constant in a specified number of memory areas each having the size that matches the specified type specifier. If a type specifier is omitted, the single-precision size is assumed.

■ **.FDATAB Instruction**

[Format]

[symbol]	.FDATAB.s	expression,floating-point-constant
----------	-----------	------------------------------------

Type specifier (t): S.....Single-precision floating-point constant, 32-bit (4-byte) <default>
 DDouble-precision floating-point constant, 64-bit (8-byte)

[Description]

The **.FDATAB** instruction sets a specified floating-point constant in a specified number of memory areas each having the size that matches a specified type specifier (.t).
If a type specifier (.t) is omitted, the single-precision size is assumed.
The expression specified in the instruction must be an absolute expression.
Neither an expression nor an integer constant can be specified instead of a floating-point constant.

[Example]

.FDATAB.S 4,0xFF00000
.FDATAB.S 12,0r1.2e10

Note:

If expression specified in the **.FDATAB** instruction is an excessively large number, it will take a long time to execute the instruction, because the instruction sets a floating-point constant in memory by a specified number (expression) of repetitions. Consequently, the value of the expression cannot be greater than 1048575, since an error will be reported.

10.6.5 .RES Instruction

The .RES instruction secures a specified number of memory areas each having the specified size.

The memory areas secured by the .RES instruction do not initially contain any data. If no size is specified, one word is assumed.

■ .RES Instruction

[Format]

[symbol]	.RES.s	expression
----------	--------	------------

Size specifier (s): I..... Bit (1 bit)
 B Byte (8 bits)
 H Halfword (16 bits)
 L Long word (32 bits)
 W Word (16 bits) <default >

[Description]

The .RES instruction secures a specified number (expression) of memory areas each having the specified size (.s).

The memory areas secured by the .RES instruction do not initially contain any data.

If a size specifier (.s) is omitted, one word is assumed.

The expression specified in the instruction must be an absolute expression.

[Examples]

```
.RES.H 2      /* Two 1-halfword areas (16 bits each) */
.RES.B 4      /* Four 1-byte areas (8 bits each) */
```

10.6.6 .FRES Instruction

The .FRES instruction secures a specified number of memory areas each having the size that matches a specified type specifier.
The memory areas secured by the .FRES instruction do not initially contain any data.
If a type specifier is omitted, the single-precision size is assumed.

■ .FRES Instruction

[Format]

[symbol]	.FRES.t	expression
----------	---------	------------

Type specifier (t): S.....Single-precision floating-point constant, 32-bit (4-byte) <default>
 DDouble-precision floating-point constant, 64-bit (8-byte)

[Description]

The .FRES instruction secures a specified number of memory areas each having the size that matches a specified type specifier (.t).
The memory areas secured by the .FRES instruction do not initially contain any data.
If a type specifier (.t) is omitted, the single-precision size is assumed.
The expression specified in this instruction must be an absolute expression.

[Examples]

.FRES.S 2 /* Two single-precision areas (4 bytes each) */
.FRES.D 4 /* Four double-precision areas (8 bytes each) */

10.6.7 .SDATA and .ASCII Instructions

The `.SDATA` and `.ASCII` instructions set specified character strings in memory.

■ .SDATA Instruction

[Format]

[symbol]	<code>.SDATA</code>	character-string[,character-string] ...
----------	---------------------	---

[Description]

The `.SDATA` instruction sets specified character strings in memory.

See Section "7.8 Strings", for how to write character strings.

[Examples]

```
.SDATA    "STR","IN","G"      → |S|T|R|I|N|G|
.SDATA    "EF\tXYZ\0"         → |E|F|09|X|Y|Z|00|
```

■ .ASCII Instruction

[Format]

[symbol]	<code>.ASCII</code>	character-string[,character-string] ...
----------	---------------------	---

[Description]

The `.ASCII` instruction is equivalent to the `.SDATA` instruction. They differ only in instruction name.

[Example]

```
.ASCII    "GHI\r\n"          → |G|H|I|0D|0A|
```

10.6.8 .SDATAB Instruction

The .SDATAB instruction sets a specified character string in the specified number of memory areas.

■ .SDATAB Instruction

[Format]

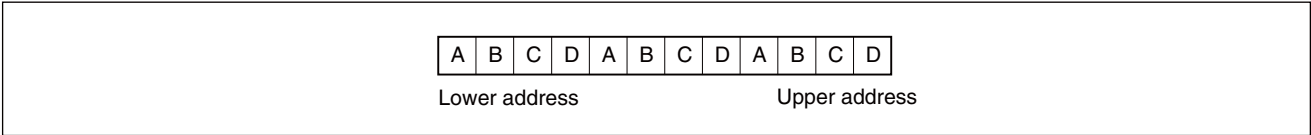
[symbol]	.SDATAB	expression,character-string
----------	---------	-----------------------------

[Description]

The .SDATAB instruction sets a specified character string in the specified number (expression) of memory areas.
The expression specified in this instruction must be an absolute expression.

[Example]

.SDATAB 3,"ABCD"



Note:

If the expression specified in the .SDATAB instruction is an excessively large number, it will take a long time to execute the instruction, because the instruction sets a character string in memory by a specified number (expression) of repetitions. Consequently, the value of the expression cannot be greater than 1048575, since an error will be reported.

10.6.9 .STRUCT and .ENDS Instructions

The **.STRUCT** and **.ENDS** instructions define the name and members of a structure. The beginning (**.STRUCT** instruction) and end (**.ENDS** instruction) of a structure must correspond.

Area definition pseudo-instructions are written as the members of the structure.

■ .STRUCT and .ENDS Instructions

[Format]

structure-tag-name	.STRUCT	
[member-name] :	Area definition pseudo-instruction	Expression
structure-tag-name	.ENDS	

[Description]

The **.STRUCT** and **.ENDS** instructions define the name and members of a structure.

The beginning (**.STRUCT** instruction) and end (**.ENDS** instruction) of a structure must correspond.

Area definition pseudo-instructions are written as the members of the structure.

[Examples]

```

ABC:      .STRUCT
m1:      .BIT      0
m2:      .BYTE     2
ABC:      .ENDS

```

■ Structure Area Definition

[Format]

structure-symbol	structure-tag-name	<[expression[,...]] >
------------------	--------------------	-----------------------

[Description]

The structure tag name functions as if it were an area definition pseudo-instruction for securing an area having the size of the structure.

The operand field specifies the value of each structure member.

Expressions enclosed in angle brackets (< >) can be omitted, but the angle brackets cannot.

Each expression enclosed in angle brackets provides the initial value for the corresponding structure member.

If an area does not need to be initialized, do not write an expression for it; just write a comma (,).

If a member is followed only by members that do not need not to be initialized, no expression need to be specified after the one corresponding to that member.

[Examples]

```

ABC:      .STRUCT
m1:       .BIT      0
m2:       .BYTE     2
ABC:      .ENDS
c:        ABC       <0,2>

```

■ Access to a Structure**[Format]**

structure-symbol.member-name

[Description]

A structure member can be referred by prefixing its name with a combination of the corresponding structure symbol and a period (.).

A structure member has a 16-bit offset within the corresponding structure, and it can be written as a displacement in an expression.

[Examples]

```

ABC:      .STRUCT
m1:       .BIT      0
m2:       .BYTE     2
ABC:      .ENDS
c:        ABC       <0,2 >
          MOV       A,c+m2
          SETB      c.m1

```

10.7 Debugging Information Output Control Instruction

The debugging information output control instruction specifies part of the debugging information that will be output.

If a startup option for debugging information output (-g) is specified, symbol information used in a program is output to the corresponding object.

Specifying ON or OFF in the debugging information output control instruction ensures that only necessary debugging information is output.

Debugging Information Output Control Instruction

Only one debugging information output control instruction is available, which is:

- `.DEBUG:` Specifies part of the debugging information to be output.

.DEBUG Instruction

[Format]

	<code>.DEBUG</code>	<code>{ON OFF}</code>
--	---------------------	-----------------------

ON: Signifies the beginning of debugging information output.

OFF: Signifies the end of debugging information output.

[Description]

The `.DEBUG` instruction specifies part of the debugging information to be output.

If a startup option for debugging information output (-g) is specified for a program, symbol information used in the program is output to the corresponding object.

Specifying ON or OFF in the `.DEBUG` instruction ensures that only necessary debugging information is output.

The `.DEBUG` instruction may be used in a source program any number of times; it is valid whenever it appears.

Debugging information is output for symbols within a range where debugging information output is ON.

Debugging information output is initially ON.

[Examples]

```
.DEBUG      ON
                                     /* Debugging information in this range is output. */
.DEBUG      OFF
                                     /* Debugging information in this range is not output. */
.DEBUG      ON
```

Relationship with Startup Options

The `.DEBUG` information is enabled only if -g is specified.

If -g is not specified or is canceled by -Xg, the `.DEBUG` instruction is disabled; no debugging information is output at all.

10.8 Library File Specification Instruction

The library file specification instruction specifies a library file.

■ Library File Specification Instruction

Only one library file specification instruction is available, which is:

- `.LIBRARY`: Specifies a library file.

■ `.LIBRARY` Instruction

[Format]

	<code>.LIBRARY</code>	<code>"library-file-name"</code>
--	-----------------------	----------------------------------

[Description]

The `.LIBRARY` instruction specifies the name of a library file that the linker will search for.

To specify more than one library file, there must be a `.LIBRARY` instruction for each library.

[Examples]

```
.LIBRARY    "liblo.lib"  
.LIBRARY    "libstd.lib"
```

10.9 List Output Control Instructions

The list output control instructions specify the output format of assembly lists.

■ List Output Control Instructions

There are six different list output control instructions:

- `.FORM`.....Specifies the number of lines per page and the number of character positions per line.
- `.TITLE`Specifies a title.
- `.HEADING`Specifies or changes a title.
- `.LIST`Specifies details of the output format of assembly source lists.
- `.PAGE`Specifies that the page be ejected.
- `.SPACE`Specifies that a blank line be output.

10.9.1 .FORM Instruction

The **.FORM** instruction specifies the number of lines per assembly list page and the number of character positions per line on the assembly list.

The number of lines per page can range between 20 and 255. If 0 is specified, no page eject occurs.

The number of character positions per line can range between 80 and 1023.

The initial values are specified as follows: **.FORM LIN=60,COL=100**

■ **.FORM Instruction**

[Format]

	.FORM	specification[,specification]
--	--------------	-------------------------------

specification: {number-of-lines|number-of-character-positions}

number-of-lines: LIN=expression(absolute-expression){0|20 to 255}

number-of-character-positions: COL=expression(absolute-expression)80 to 1023

[Description]

The **.FORM** instruction specifies the number of lines per assembly list page and the number of character positions per line in the assembly list.

The **.FORM** instruction may be used in a source program any number of times; it is valid whenever it appears.

The expressions specified in the **.FORM** instruction must be absolute expressions.

The number of lines per page can range between 20 and 255. If 0 is specified, no page eject occurs.

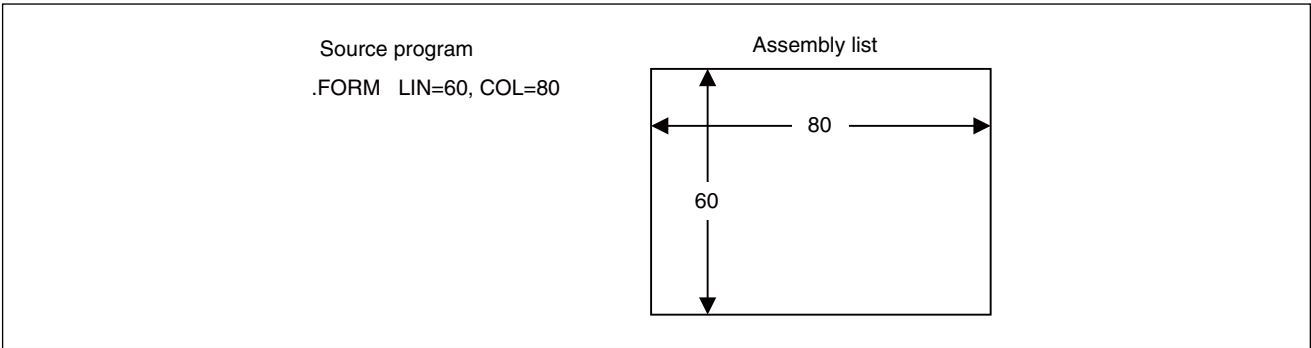
The number of character positions per line can range between 80 and 1023.

The initial values are specified as follows: **.FORM LIN=60,COL=100**

The assembler outputs, with a margin, a list within the specified number of lines and numbers of character positions.

[Example]

Figure 10.9-1 Output Example by .FORM Instruction



■ Relationship with Startup Options

- pl invalidates the specification of a number of lines.
- pw invalidates the specification of a number of character positions.

10.9.2 .TITLE Instruction

The `.TITLE` instruction specifies a title, which will be displayed as a comment in the header on each page of the assembly list.
The specified title text is output to all pages, including the first one.
The `.TITLE` instruction can be written in a source program only once.

■ **.TITLE Instruction**

[Format]

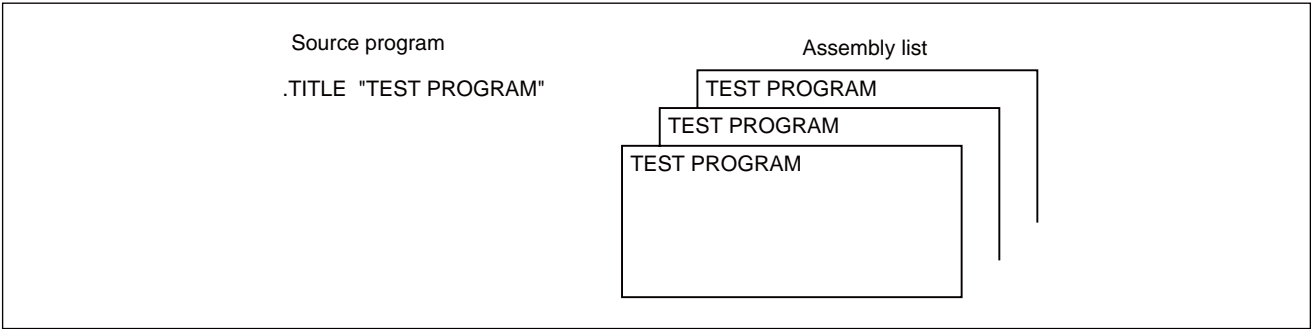
	<code>.TITLE</code>	<code>"title-text"</code>
--	---------------------	---------------------------

[Description]

The `.TITLE` instruction specifies a title, which will be displayed as a comment in the header on each page of the assembly list.
The specified title text is output to all pages, including the first one.
The `.TITLE` instruction can be written in a source program only once.
The title can be up to 60 characters.

[Example]

Figure 10.9-2 Output Example by .TITLE Instruction



10.9.3 .HEADING Instruction

The **.HEADING** instruction specifies a title, which will be displayed as a comment in the header on each page of the assembly list.

The **.HEADING** instruction ejects the page and outputs the specified title text to a new page.

The **.HEADING** instruction may be written in a source program any number of times; it is valid whenever it appears.

■ **.HEADING Instruction**

[Format]

	<code>.HEADING</code>	<code>"title-text"</code>
--	-----------------------	---------------------------

[Description]

The **.HEADING** instruction specifies a title, which will be displayed as a comment in the header on each page of the assembly list.

The **.HEADING** instruction ejects the page and outputs the specified title text to a new page.

If the **.HEADING** instruction corresponds to the first line of a page, it outputs the specified title to this page.

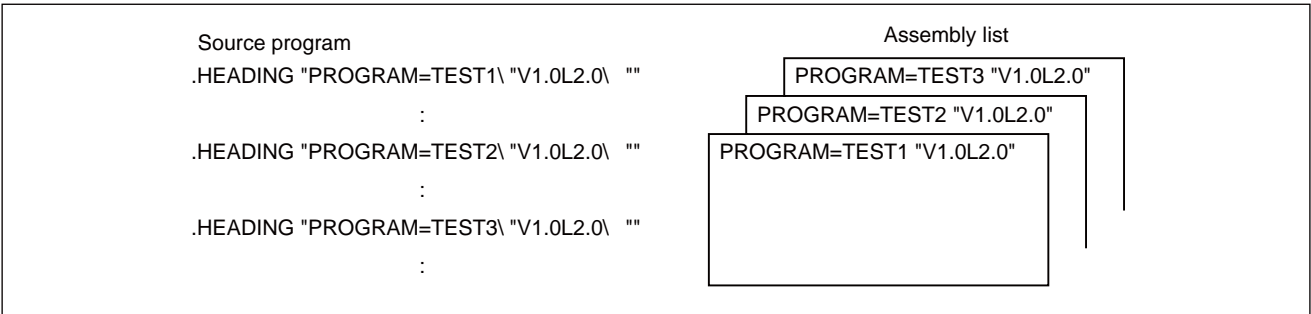
When the **.HEADING** instruction is issued, if **.LIST OFF** has been specified, no title is output. The title specified in the **.HEADING** instruction is output to the first line of the page after **.LIST ON** is specified.

The **.HEADING** instruction may be written in a source program any number of times; it is valid whenever it appears.

The title can be up to 60 characters.

[Examples]

Figure 10.9-3 Output Example by .HEADING Instruction



10.9.4 .LIST Instruction

The .LIST instruction specifies details of the output format of the assembly list. The .LIST instruction may be written in a source program any number of times; it is valid whenever it appears.

If the .LIST instruction specifies ON or OFF, this instruction is not a target for listing. The initial value is .LIST ON,CALL,COND,DEF,EXPOBJ,INC.

■ .LIST Instruction

[Format]

	.LIST	specification[,specification] ...
--	-------	-----------------------------------

- Specification:
- { ON|OFF } Specifies whether to list an assembly source.
 - { CALL|NOCALL } Specifies whether to output macro call instructions to the assembly list.
 - { COND|NOCOND } Specifies whether to output nontext portions to the assembly list.
 - { DEF|NODEF } Specifies whether to output macro instructions and definitions to the assembly list.
 - { EXP|NOEXP|EXPOBJ } Specifies whether to output macro-expanded text to the assembly list.
 - { INC|NOINC } Specifies whether to output include file text to the assembly list.

[Description]

The .LIST instruction specifies details of the output format of the assembly list.

The .LIST instruction may be written in a source program any number of times; it is valid whenever it appears.

If the .LIST instruction specifies ON or OFF, this instruction is not a target for listing.

The initial value is .LIST ON,CALL,COND,DEF,EXPOBJ,INCSTR.

The specifications of the .LIST instruction have the following meanings:

- ON: Specifies to list an assembly source.
- OFF: Specifies not to list an assembly source.
- CALL: Specifies to output macro call instructions to the assembly source list.
- NOCALL: Specifies not to output macro call instructions to the assembly source list.
- COND: Specifies to output nontext portions* to the assembly source list.
- NOCOND: Specifies not to output nontext portions to the assembly source list.
- DEF: Specifies to output macro definitions and instructions to the assembly source list.
- NODEF: Specifies not to output macro definitions and instructions to the assembly source list.
- EXP: Specifies to output macro-expanded text to the assembly source list.
- NOEXP: Specifies not to output macro-expanded text to the assembly source list.
- EXPOBJ: Specifies not to output macro-expanded text to the assembly source list, but specifies to output object code.
- INC: Specifies to output include file text to the assembly source list.
- NOINC: Specifies not to output include file text to the assembly source list.

Note:

* : The term "nontext portion" refer to the if clause portion that is not a target for assembly.

[Example]

```
.LIST  ON
      :
      :      /* This portion is listed. */
      :
.LIST  OFF
      :
      :      /* This portion is not listed. */
      :
.LIST  ON
```

■ Relationship with Startup Options

- If `-linc ON` is specified

The INC/NOINC specification is disabled, and include file text is always listed.

- If `-linc OFF` is specified

The INC/ONINC specification is disabled, and include file text is not listed at all.

- If `-lexp ON` is specified

The EXP/NOEXP/EXPOBJ specification is disabled, and macro-expanded text is always listed.

- If `-lexp OFF` is specified

The EXP/NOEXP/EXPOBJ specification is disabled, and macro-expanded text is not listed at all.

- If `-lexp OBJ` is specified

The EXP/NOEXP/EXPOBJ specification is disabled, and only the object of macro-expanded text is always listed.

10.9.5 .PAGE Instruction

The `.PAGE` instruction updates the page number, and starts outputting the next page of the assembly list.

If the `.PAGE` instruction is on the first line of a page, it is disabled.

The `.PAGE` instruction itself is not listed.

■ **.PAGE Instruction**

[Format]

	<code>.PAGE</code>	
--	--------------------	--

[Description]

The `.PAGE` instruction updates the page number, and starts outputting the next page of the assembly list.

If the `.PAGE` instruction is on the first line of a page, it is disabled.

The `.PAGE` instruction itself is not listed.

[Examples]

Figure 10.9-4 Output Example by `.PAGE` Instruction

Source program	Assembly list	
<code>.DATA 10</code>	<div><code>.DATA 10</code></div> <div><code>.DATA 20</code></div>	Page n
<code>.DATA 20</code>		
<code>.PAGE</code>		
<code>.DATA 30</code>	<div><code>.DATA 30</code></div> <div><code>.DATA 40</code></div>	Page n+1
<code>.DATA 40</code>		

10.9.6 .SPACE Instruction

The **.SPACE** instruction outputs a specified number of blank lines.

The number of blank lines to be output can range between 0 and 255.

The **.SPACE** instruction itself is not listed, but is included in the line count.

■ .SPACE Instruction

[Format]

	.SPACE	number-of-blank-lines
--	---------------	-----------------------

number-of-blank-lines: Expression (absolute expression)

[Description]

The **.SPACE** instruction outputs as many blank lines as specified in the number-of-blank-lines operand.

The expression specified in the **.SPACE** instruction must be an absolute expression.

The number of blank lines to be output can range between 0 and 255.

If the instruction specifies more blank lines than the page size, the excessive blank lines are not output.

The **.SPACE** instruction itself is not listed, but is included in the line count.

[Example]

.SPACE 4

CHAPTER 11

PREPROCESSOR PROCESSING

Preprocessor processing provides text processing functions such as macro expansion, repeat expansion, conditional assembly, macro replacement, and file reading.

These functions allow effective coding of assembly programs, in which similar blocks of text are often used repeatedly.

Each preprocessor instruction conforms to the C compiler preprocessor specifications so that it can be easily assimilated to C instructions.

Some instructions, such as `#macro`, are unique to the assembler, not found in the C compiler.

This chapter explains the functions of the preprocessor as well as each preprocessor function.

- 11.1 Preprocessor
- 11.2 Basic Preprocessor Rules
- 11.3 Preprocessor Expressions
- 11.4 Macro Definitions
- 11.5 Macro Call Instructions
- 11.6 Repeat Expansion
- 11.7 Conditional Assembly Instructions
- 11.8 Macro Name Replacement
- 11.9 `#include` Instruction
- 11.10 `#line` Instruction

CHAPTER 11 PREPROCESSOR PROCESSING

11.11 `#error` Instruction

11.12 `#pragma` Instruction

11.13 No-operation Instruction

11.14 Defined Macro Names

11.15 Differences from the C Preprocessor

11.1 Preprocessor

A preprocessor is generally called a preprocessing program. It is used to process text before it is actually assembled.

This preprocessor provides four main functions:

- Macro definition
- Conditional assembly
- Macro name replacement
- File reading

■ Preprocessor

● Macro definition

There are cases in which the programmer wishes to execute multiple instructions or a certain unit of processing with a single instruction.

In these cases, macro definition is useful.

The text of the instruction string that is to be defined is called the macro body.

When a macro call is made, the macro is expanded into the macro body.

In the macro body, the programmer can write not only machine instructions, pseudo-instructions, and macro names, but formal arguments, #local instructions, and local symbols.

[Example]

#macro get_timer addr, reg

#local loop

loop:

MOV A, #addr

CMP A, #0

BEQ loop

MOV reg, A

#endm

/* get_timer is a macro call instruction */

/* addr and reg2 are formal arguments */

/* loop is a local symbol */

get_timer 0x100, R1 /* Macro call */

__0000000001loop:

MOV A, #0x100

CMP A, #0

BEQ __0000000001loop

MOV R1, A

Macro body

Macro definition

Macro expansion

● Conditional assembly

When, for example, an instruction that is intended to be executed and another one that is not intended to be executed are existent, code as follows:

```
#if      CPU_TYPE=1
        text-1

#else

        text-2

#endif
```

In this example, if CPU_TYPE is equal to 1, text-1 is selected and assembled.

If CPU_TYPE is not equal to 1, text-2 is selected and assembled.

The instructions used for conditional assembly are called conditional assembly instructions.

The conditional assembly instructions are #if, #ifdef, #ifndef, #elif, #else, and #endif instructions.

● Macro name replacement

An important function of the preprocessor is macro name replacement.

For example, if a constant value is to be used, it can be written directly as follows:

```
MOV      A,#0xFF
```

Alternatively, 0xFF can be defined for some meaningful name as follows:

```
#define   IOMASK    0xFF
MOV      A,#IOMASK
```

The latter is easier to understand.

The function that replaces the name IOMASK to 0xFF is called macro replacement.

● File reading

It is useful if the variables and macro names to be shared are stored in a separate file so that the file can be read by another file when it is necessary.

[Example]

File iodef.h

#define	IOMASK	0xFF	/* I/O mask value */
#define	SETCMD	1	/* Data set command */
	:		

File com.asm

#include	"iodef.h"	/* Read defined values */
	:	
MOV	A,R2	
AND	A,#IOMASK	/* Mask the data */
MOV	R2,A	
MOV	A,#SETCMD	
MOV	A,R7	
MOV	@A,T	/* Send the data set command */

11.2 Basic Preprocessor Rules

This section explains how to write programs using preprocessor instructions and explains preprocessor rules.

■ Preprocessor Instruction Format

Each preprocessor instruction must be preceded by a # symbol.

The preprocessor instructions are listed below:

#macro	#local	#exitm	#endm
#repeat	#if	#ifdef	#ifndef
#elif	#else	#endif	#define
#set	#undef	#purge	#include
#line	#error	#pragma	#

■ Comments

A comment can begin in any column.

A semicolon (;) or two slashes (//) start a line comment.

A comment can also be enclosed by /* and */, as in C.

A comment enclosed by /* and */ can appear anywhere.

■ Continuation of a Line

A backslash (\) at the end of a line means that the line continues on the next line.

It is assumed that the beginning of the next line starts at the location of the backslash (\) indicating continuation.

If the backslash (\) is followed by a character other than the line-feed character, the line cannot be continued.

■ Integer Constants

Four types of integer constants are available: binary, octal, decimal, and hexadecimal.

■ Character Constants

A character constant must be enclosed the character in single quotation marks (').

■ Macro Names

Each time a macro name appears in text, the macro name is expanded or replaced by the character string defined for it.

■ Formal Arguments

A formal argument is defined by a macro definition (`#macro`). A macro call instruction can be used to set an argument for the formal argument.

■ Local Symbols

A local symbol automatically generates a unique name at macro expansion.

Thus, if a jump symbol, for example, is defined in a macro body as a local symbol, the symbol will never be defined multiple times no matter how many times the macro is expanded.

11.2.1 Preprocessor Instruction Format

Each preprocessor instruction must be preceded by a # symbol.

Blanks and comments can be written between column 1 and a preprocessor instruction.

When a line comment is written, it is regarded as continuing until the end of the line.

■ Preprocessor Instruction Format

[Format]

```
#preprocessor-instruction "parameter ..."
```

[Description]

Each preprocessor instruction must be preceded by a # symbol.

Blanks and comments can be written between column 1 and a preprocessor instruction.

When a line comment is written, it is regarded as continuing until the end of the line.

Preprocessor instructions, preceded by a #, are not processed by macro expansion.

The preprocessor instructions are listed below:

#macro	#local	#exitm	#endm
#repeat	#if	#ifdef	#ifndef
#elif	#else	#endif	#define
#set	#undef	#purge	#include
#line	#error	#pragma	#

[Examples]

```
#define    LINEMAX        255
#ifndef    OFF
/* OFF */ #define    OFF    0
/* ON  */ #define    ON     -1    /* Not 1 */
#endif
```

11.2.2 Comments

Comments are classified as line comments and range comments.

A line comment begins with a semicolon (;) or two slashes (//).

A comment can also be enclosed by /* and */, as in C. This type of comment is called a range comment.

■ Comments

[Format]

```
/* Range comment */
// Line comment
; Line comment
```

[Description]

A comment can begin in any column.

Comments are classified as line comments and range comments.

A line comment begins with a semicolon (;) or two slashes (//).

A comment can also be enclosed by /* and */, as in C. This type of comment is called a range comment.

A range comment can appear anywhere.

[Examples]

```
/*-----
Comments
-----*/

#define      STRLEN      10      ; Character length
/* test1 */#if   TEST==1      // Test mode 1
      :
/* test2 */#elif TEST==2      /* Special test */
      :
/* end */#endif
```


11.2.3 Continuation of a Line

When a backslash (\) is placed at the end of a line, the line is assumed to continue to the next line.

It is also assumed that the beginning of the next line follows the backslash (\) indicating continuation.

If the backslash (\) is followed by a character other than the line-feed character, the line cannot be continued.

■ Continuation of a Line

[Format]

\Line-feed-character

[Description]

Placing a backslash (\) at the end of a line means that the line continues on the next line.

It is assumed that the beginning of the next line starts at the position of the backslash (\) indicating continuation.

If the backslash (\) is followed by a character other than the line-feed character, the line cannot be continued.

A backslash can also be used to indicate the continuation of comments, character constants, and character strings.

[Examples]

```
.DATA      0x01, 0x02, 0x03, \
           0x04, 0x05, 0x06, ; Comment \
           0x07, 0x08, 0x09

.SDATA     "abcdefghijklnopqrstuvwxyz \
ABCDEFGHIJKLMNPNOPQRSTUVWXYZ"      /* Continuation of a character string */
```

11.2.4 Integer Constants

Four types of integer constants are available: binary, octal, decimal, and hexadecimal. Integer constants are exactly the same as the numeric constants in the assembly phase.

■ Integer Constants

Four types of integer constants are available: binary, octal, decimal, and hexadecimal.

The long-type specification (such as 123L) and the unsigned-type specification (such as 123U) in C are supported.

● Binary constants

A binary constant is an integer constant represented in binary notation.

It must be preceded by a prefix (B' or 0b) or suffix (B).

The prefix (B' or 0b) and suffix (B) can be either uppercase or lowercase character.

[Examples]

B'0101 0b0101 0101B

● Octal constants

An octal constant is an integer constant represented in octal notation.

It must be preceded by a prefix (Q' or 0) or suffix (Q).

The prefix (Q') and suffix (Q) can be either uppercase or lowercase character.

[Examples]

Q'377 0377 377Q

● Decimal constants

A decimal constant is an integer constant represented in decimal notation.

It is preceded by a prefix (D') or suffix (D).

The prefix and suffix for decimal constants can be omitted.

The prefix (D') and suffix (D) can be either uppercase or lowercase character.

[Examples]

D'1234567 1234567 1234567D

● Hexadecimal constants

A hexadecimal constant is an integer constant represented in hexadecimal notation.

It must be preceded by a prefix (H' or 0x) or suffix (H).

The prefix (H' or 0x) and suffix (H) can be either uppercase or lowercase character.

[Examples]

H'ff 0xFF 0FFH

11.2.5 Character Constants

A character constant represents a character value.

In a character constant, a character constant element must be enclosed in single quotation marks (').

Character constants are exactly the same as those in the assembly phase.

■ Character Constants

In a character constant, a character constant element must be enclosed in single quotation marks (').

Character constant elements can be characters, extended representations, octal representations, and hexadecimal representations.

A character constant element can be up to four characters.

Character constants are handled in base-256 notation.

■ Character Constant Elements

● Characters

All characters (including the blank) except the backslash (\) and single quotation mark (') can be independent character constant elements.

[Examples]

'P' '@A' '0A'

● Extended representations

A specific character preceded by a backslash (\) can be a character constant element. This form is called an extended representation. Table 11.2-1 lists the extended representations.

Table 11.2-1 Extended Representations

Character	Character constant element	Value
Line feed character	\n	0x0A
Horizontal tab character	\t	0x09
Backspace character	\b	0x08
Carriage return character	\r	0x0D
Line feed character	\f	0x0C
Backslash	\\	0x5C
Single quotation mark	\'	0x27
Double quotation mark	\"	0x22
Alarm character	\a	0x07
Vertical tab character	\v	0x0B
Question mark	\?	0x3F

Note:

The characters used in extended representations must be lowercase character.

[Examples]

\n \" '\\'

● Octal representations

The bit pattern of a character code is written directly to represent single-byte data. An octal representation is one to three octal digits preceded by a backslash (\).

[Examples]

Character constant element	Bit pattern
\0'	b'00000000
\377'	b'11111111
\53'	b'00101011
\0123'	b'00001010 => 'Divided into \012' and '3'

● Hexadecimal representations

The bit pattern of a character code is written directly to represent single-byte data. A hexadecimal representation is character x (lowercase) and one or two hexadecimal digits preceded by a backslash (\).

[Examples]

Character constant element	Bit pattern
<code>'\x0'</code>	<code>b'00000000</code>
<code>'\xff'</code>	<code>b'11111111</code>
<code>'\x2B'</code>	<code>b'00101011</code>
<code>'\x0A5'</code>	<code>b'00001010</code> => Divided into <code>'\x0A'</code> and <code>'5'</code>

11.2.6 Macro Names

Each time a macro name appears in text, the macro name is expanded or replaced by the character string defined for it.

In C, a macro name can also be called an identifier.

■ Macro Name Rules

- A macro name can be up to 255 characters in length.
- A macro name must begin with an alphabetic character or an underscore (_).
- The second and subsequent characters of a macro name must be alphanumeric characters, number, or underscores (_).
- Macro name characters are case-sensitive.

[Examples]

A Zabcde ppTRUE 123456789

■ Macro Name Types

● Defined macro

Refers to a macro name defined by the #define or #set instruction.

A defined macro name can appear anywhere in text. Each time it appears, it is replaced by the character string defined for it.

[Examples]

```
#define TRUE 1
#define FALSE 0
#define add(a,b) (a)+(b)
/* TRUE, FALSE, and add are macro names */
```

● Macro call instruction

Refers to a macro name defined by the #macro instruction.

Only blanks and comments can be written between the beginning of a line and a macro instruction.

If a line comment is written, the comment continues until the end of the line.

A macro call instruction is expanded into the defined text.

[Example]

```
#macro max a,b,c
:
#endm
/* mac is a macro name */
```

11.2.7 Formal Arguments

A formal argument is defined by a macro definition (the #macro instruction). A macro call instruction is used to set an argument for the formal argument.


■ Formal Argument Naming Rules

Formal arguments must conform to the macro naming rules given in Section "11.2.6 Macro Names".


■ Formal Argument Replacement Rules

Formal arguments can be replaced in a macro body only.
Formal arguments have no effect after macro expansion ends.

[Example]

#macro mv reg1, reg2
 MOV R0, reg2  Macro body
 MOV reg2, reg1

#endm

 mv R1, R2 /* Macro call */
 MOV R0, R2  Macro expansion
 MOV R2, R1

 MOV reg1, R0 /* Because this is outside the macro body, reg1 is not treated */
 /* as a formal argument and therefore is not replaced. */

11.2.8 Local Symbols

A local symbol automatically generates a unique name at macro expansion. Thus, if a jump symbol, for example, is defined in a macro body as a local symbol, the symbol will never be defined multiple times no matter how many times the macro is expanded.

■ Local Symbol Naming Rules

Local symbols must conform to the macro naming rules given in Section "11.2.6 Macro Names".

■ Local Symbol Replacement Rules

Local symbols can be replaced in a macro body only.

A local symbol is generated in the following format:

```
__nnnnn local-symbol
```

A local symbol begins with two underscores (`_`), which are followed by a 5-digit number.

The 5-digit number is incremented by 1 in the range from 00001 to 65535 each time a macro call is made.

The 5-digit number is followed by a user-specified local symbol name.

A local symbol has no effect after the macro expansion ends.

[Example]

#macro	get_timer	addr, reg	
#local	loop		<div> <div>loop is defined as a local symbol by the #local instruction</div> <div>Macro body</div> </div>
loop:			
	MOV	A, #addr	
	CMP	A, #0	
	BEQ	loop	
	MOV	reg, A	
#endm			
	get_timer	0x0100, R1	/* Macro call */
__00001loop:			<div> <div>Macro expansion</div> <div>The loop portion is replaced by __00001loop</div> </div>
	MOV	A, 0x010	
	CMP	A, #0	
	BEQ	__00001loop	
	MOV	R1, A	
	BRA	loop	/* Because this is outside the macro body, loop is not treated */
			/* as a local symbol and therefore is not replaced. */

11.3 Preprocessor Expressions

Preprocessor expressions are used with the `#if`, `#elif`, `#set`, and `#repeat` instructions. The operators used in expressions conform to constant expressions in C.

■ Preprocessor Expressions

The following terms can be used in an expression:

- Integer constants
- Character constants
- Macro names
- Formal arguments (in macro bodies only)

Macro names and formal arguments are replaced before being used as terms.

If an expression contains an undefined macro name, it is evaluated with the macro name being replaced by 0.

Preprocessor expressions can be constant expressions only.

The relative symbols, absolute symbols, EQU symbols, and section symbols in the assembly phase cannot be used.

[Example]

```
#if      (MODE & 0xff) + 1 > 3
      :
#endif
```

■ Preprocessor Expression Evaluation Precision

Operation expressions are evaluated as 32 bits. Please be careful operation expressions exceeding 32 bits is not guaranteed (no error results, however).

Relational and equivalence expressions are regarded as equal to 1 if evaluated as true and 0 if evaluated as false.

■ Preprocessor Operators

Operators are used in an expression.

The operators that can be used in an expression are as follows:

- Unary operators

!	Logical NOT	Used in true/false decision
~	NOT	Used in bit decision
+	Positive	
-	Negative	
- Binary operators

*	Multiplication	
/	Division	
%	Remainder	
+	Addition	
-	Subtraction	
<<	Left arithmetic shift	
>>	Right arithmetic shift	
<	Relational operator	Less than
<=	Relational operator	Less than or equal to
>	Relational operator	Greater than
>=	Relational operator	Greater than or equal to
==	Relational operator	Equal to
!=	Relational operator	Not equal to
&	Bit AND	
^	Bit XOR	
	Bit OR	
&&	Logical AND	
	Logical OR	

■ Preprocessor Operator Precedence

Table 11.3-1 indicates the preprocessor operator precedence.

Table 11.3-1 Preprocessor Operator Precedence

Precedence	Operator	Associativity	Applicable expression
1	()	Left	Parentheses
2	! ~ + -	Right	Unary expression
3	* / %	Left	Multiplication expression
4	+ -	Left	Addition expression
5	<< >>	Left	Shift expression
6	< <= > >=	Left	Relational expression
7	== !=	Left	Equivalence expression
8	&	Left	Bit AND expression
9	^	Left	Bit XOR expression
10		Left	Bit OR expression
11	&&	Left	Logical AND expression
12		Left	Logical OR expression

11.4 Macro Definitions

A macro definition consists of a **#macro** instruction, macro body, and **#endm** instruction.

When a macro call is made, the specified macro name is expanded into the macro body defined by the macro definition.

■ Macro Definitions

[Format]

```
#macro macro-name, [formal-argument[, formal-argument] ... ]  
    Macro body  
#endm
```

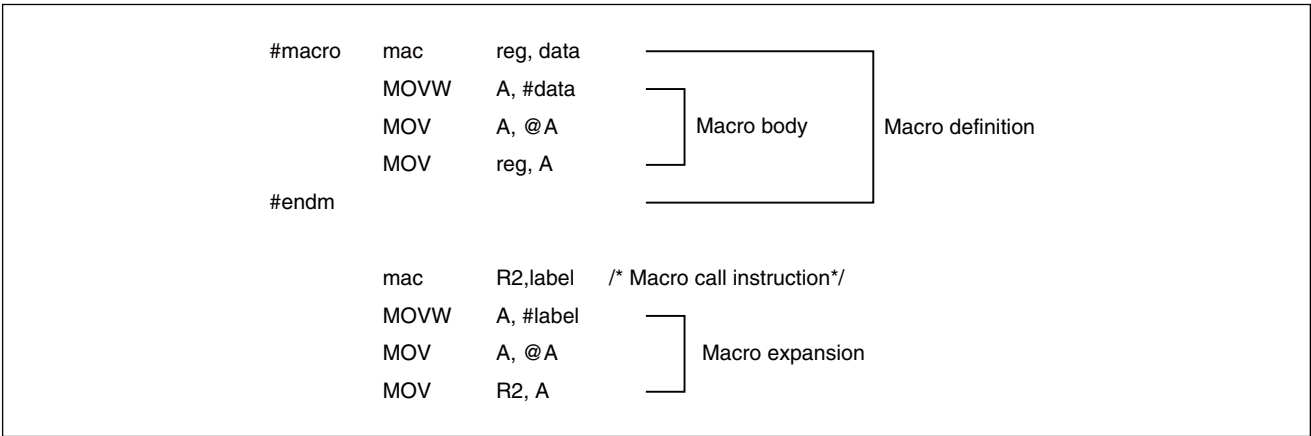
[Description]

A macro definition consists of a **#macro** instruction, macro body, and **#endm** instruction.
The text between the **#macro** and **#endm** instructions is called the macro body.
The macro body is registered in a macro description, with the macro name as a keyword. When the macro name appears, the macro body is expanded into the corresponding macro name.
The macro name used as a keyword is called the macro call instruction.
Expansion into a macro body is called macro expansion.

■ Macro Definition Rules

Defining another macro in a macro body is not possible.
A macro body can be up to about 60K bytes in size. Macro bodies exceeding this size cannot be registered.

[Example]



11.4.1 #macro Instruction

The **#macro** instruction declares the beginning of a macro definition and defines a macro name and formal argument(s).

■ #macro Instruction

[Format]

```
#macro macro-name, [formal-argument[. formal-argument] ... ]
```

[Description]

The **#macro** instruction declares the beginning of a macro definition and defines a macro name and formal arguments.

The macro name specified with the **#macro** instruction is used as a macro call instruction.

When the macro call instruction is used, the macro name is expanded into the defined macro body.

■ #macro Instruction Rules

- The definition that starts with the **#macro** instruction must end with the **#endm** instruction.
- Two or more formal arguments with the same name cannot be specified with the **#macro** instruction.
- The formal arguments specified with this instruction are valid within the corresponding macro body only.
- When a pattern that is the same as a formal argument is found, it is immediately replaced.
- If a formal argument is the same as a macro name or local symbol, the replacement of the formal argument has precedence.
- Formal arguments are optional.

[Example]

```
#macro    mac      r1, r2, data
          MOV      A, data
          MOV      r2, r1
          MOV      r1, A
#endm

          mac      R2, R7, label      /* Macro call instruction*/
          MOV      A, label
          MOV      R7, R2
          MOV      R2, A
```

Macro body

Macro expansion

11.4.2 #local Instruction

The symbol generated by the **#local** instruction is called a local symbol.
A local symbol automatically generates a unique name each time a macro call is made.

■ #local Instruction

[Format]

#local local-symbol[. local-symbol] ...

[Description]

The **#local** instruction defines a local symbol.
The symbol generated by the **#local** instruction is called a local symbol.
A local symbol automatically generates a unique name each time a macro call is made.
A local symbol generates a unique name so that it is not defined multiple times.
For an explanation of the local symbol generation rules, see Section "11.2.8 Local Symbols".

■ #local Instruction Rules

- The **#local** instruction can be used in a macro body only.
- Any number of local symbols can be specified.
- Two or more local symbols with the same name cannot be specified.
- The local symbols defined by the **#local** instruction in a macro body are valid in that macro body only.
- When the same pattern as a local symbol is found, a unique name is immediately generated.
- If a local symbol is the same as a macro name or formal argument, the replacement of the formal argument has precedence.

[Example]

#macro busyloop data
#local label
 MOVW A, #data
label:
 DECW A
 BNE label
#endm

Macro body

busyloop 10 /* Macro call instruction*/
 MOVW A, #10
__00001label:
 DECW A
 BNE __00001label

Macro expansion

11.4.3 #exitm Instruction

The #exitm instruction forcibly terminates macro or repeat expansion.

■ #exitm Instruction

[Format]

#exitm

[Description]

The #exitm instruction forcibly terminates macro or repeat expansion.

■ #exitm Instruction Rules

- The #exitm instruction can be used in a macro body only.
- The #exitm instruction has no effect on a conditional assembly instruction.
- If macro or repeat expansions are nested, each #exitm instruction terminates the corresponding expansion only; it does not terminate the other expansions.
- Any number of #exitm instructions can be used in a macro body.

[Example]

#macro mac cnt
 NOP
#if cnt >= 5
#exitm
#endif
 MOV A, #cnt
#endm

Macro body

mac 4 /* Macro call instruction*/
NOP
MOV A, #4 Macro expansion

mac 5 /* Macro call instruction*/
NOP Macro expansion

11.4.4 #endm Instruction

The #endm instruction declares the end of a macro definition.
The #endm instruction also terminates the expansion text of repeat expansion.

■ #endm Instruction

[Format]

#endm

[Description]

The #endm instruction declares the end of a macro definition.
The #endm instruction also terminates the expansion text of repeat expansion.
Thus, the #endm instruction must always be used together with the #macro or #repeat instruction.

[Example]

#macro mac a,b
 .DATA a,b

#endm
#repeat 3
 NOP

#endm
 NOP
 NOP
 NOP

Repeat expansion

11.5 Macro Call Instructions

When the macro name defined by the `#macro` instruction is found, the macro is expanded.

This function is called a macro call. The macro name is called a macro call instruction.

■ Macro Call Instruction

[Format]

`macro-call-instruction [argument[. argument] ...]`

[Description]

When the macro name defined by the `#macro` instruction is found, the macro is expanded.

This function is called a macro call. The macro name is called a macro call instruction.

■ Macro Call Instruction Rules

- Enter the macro name used as a macro call in the instruction field.
- The macro instruction must be defined before it can be used.
- If an argument contains a comma (,), it must be enclosed in parentheses () or angle brackets < >.
 - If an argument is enclosed in parentheses, the parentheses are treated as part of the argument.
 - If an argument is enclosed in angle brackets, the angle brackets are not treated as part of the argument.
- The number of arguments specified with a macro call instruction must be equal to the number of arguments in the corresponding macro definition. If the arguments specified with the macro call instruction are fewer, null characters are assumed to specify for the missing arguments.
- To specify a null character as an argument, write two commas (,) or write a pair of angle brackets < >.

11.6 Repeat Expansion

Repeat expansion contains the **#repeat** and **#endm** instructions.
In expansion text, write the text to be repeated.
Immediately after the **#endm** instruction, repeat expansion repeats the expansion text the number of times specified by the iteration.

■ Repeat Expansion

[Format]

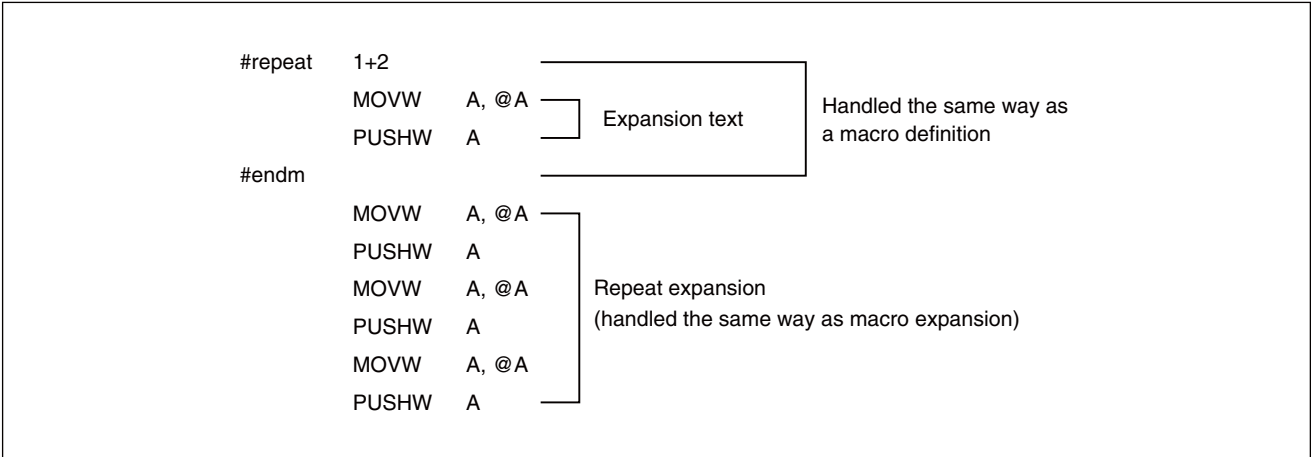
```
#repeat    iteration
           expansion-text
#endm
```

Repeat iteration: Preprocessor expression

[Description]

Repeat expansion contains the **#repeat** and **#endm** instructions.
For expansion-text, write the text to be repeated.
Immediately after the **#endm** instruction, repeat expansion repeats the expansion text the number of times specified by the iteration.
The text between the **#repeat** and **#endm** instructions is handled the same way as a macro definition. Repeat expansion is handled the same way as macro expansion.
Therefore, if, the output of a macro definition is controlled with a list control instruction, repeat expansion is processed the same way.
Expansion text can be up to 60K bytes in size. Any expansion text exceeding this size cannot be registered.
The **#local** instruction cannot be used in expansion text.

[Example]



■ #repeat Instruction

[Format]

#repeat Repeat iteration

Repeat iteration: Preprocessor expression

[Description]

The #repeat instruction declares the beginning of expansion text.

The expansion text is repeated the number of times specified by the iteration.

■ #repeat Instruction Rules

- A definition that starts with the #repeat instruction must end with the #endm instruction.
- If the iteration is 0 or less, nothing is repeated.

11.7 Conditional Assembly Instructions

The conditional assembly instructions are used to select, on the basis of a condition, the text that is to be assembled.

Between the `#if`, `#ifdef`, or `#ifndef` instruction and the `#endif` instruction is the `#if` clause. The `#if` clause contains the text subject to conditional assembly.

■ Conditional Assembly Instructions

[Format]

```
#if instruction|#ifdef instruction|#ifndef instruction
    text
[#else instruction|#elif instruction]
    text
#endif instruction
```

[Description]

Between the `#if`, `#ifdef`, or `#ifndef` instruction and the `#endif` instruction is an if clause. The if clause contains the text subject to conditional assembly.

The `#else` or `#elif` instruction can be used in the if clause.

An if clause can contain another if clause, a feature referred to as nesting of if clauses.

Six conditional assembly instructions are available:

- `#if` instruction
- `#ifdef` instruction
- `#ifndef` instruction
- `#else` instruction
- `#elif` instruction
- `#endif` instruction

11.7.1 #if Instruction

The **#if** instruction declares the beginning of an if clause.

If the conditional expression is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the conditional expression is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

■ #if Instruction

[Format]

#if conditional-expression

conditional-expression: Preprocessor expression

[Description]

The **#if** instruction declares the beginning of an if clause.

The conditional expression is false if it is equal to 0, and true if not equal to 0.

If the conditional expression is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the conditional expression is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

An if clause that starts with the **#if** instruction must end with the **#endif** instruction. Thus, the **#if** and **#endif** instructions must always be paired.

[Examples]

```
#define      ABC 1
#if         ABC == 1
            .DATA      0

#endif

/* Because the conditional expression of the #if instruction is true, */
/* .DATA 0 is assembled. */
#if         0
            .DATA      100

#endif

/* Because the conditional expression of the #if instruction is false, */
/* .DATA 100 is not assembled. */
```

11.7.2 #ifdef Instruction

The **#ifdef** instruction declares the beginning of an if clause.

The if clause is true if the macro name specified in the if clause has been defined and false if it has not been defined.

If the if clause is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the if clause is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

■ #ifdef Instruction

[Format]

```
#ifdef macro-name
```

[Description]

The **#ifdef** instruction declares the beginning of an if clause.

The if clause is true if the macro name specified in the if clause has been defined and false if it has not been defined.

If the if clause is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the if clause is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

An if clause that starts with the **#ifdef** instruction must end with the **#endif** instruction. Thus, the **#ifdef** and **#endif** instructions must always be paired.

[Examples]

```
#define      ON
#ifdef      ON
            .DATA      0

#endif

/* Because the macro name (ON) specified with the #ifdef instruction has been defined, */
/* .DATA 0 is assembled. */
#ifdef      OFF
            .DATA      100

#endif

/* Because the macro name (OFF) specified with the #ifdef instruction has not been */
/* defined, .DATA 100 is not assembled. */
```

11.7.3 #ifndef Instruction

The **#ifndef** instruction declares the beginning of an if clause.

The if clause is false if the macro name specified in the if clause has been defined and true if it has not been defined.

If the if clause is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the if clause is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

The true/false decision of the **#ifndef** instruction is the opposite from that of the **#ifdef** instruction.

■ #ifndef instruction

[Format]

```
#ifndef macro-name
```

[Description]

The **#ifndef** instruction declares the beginning of an if clause.

The if clause is false if the macro name specified in the if clause has been defined and true if it has not been defined.

If the if clause is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the if clause is false, the text between this instruction and the corresponding **#else**, **#elif**, or **#endif** instruction is not assembled.

The true/false decision of the **#ifndef** instruction is the opposite from that of the **#ifdef** instruction.

An if clause that starts with the **#ifndef** instruction must end with the **#endif** instruction. Thus, the **#ifndef** and **#endif** instructions must always be paired.

[Examples]

```
#define      ON
#ifndef     ON
        .DATA      0

#endif

/* Because the macro name (ON) specified with the #ifndef instruction has been */
/* defined, .DATA 0 is not assembled. */

#ifndef     OFF
        .DATA      100

#endif

/* Because the macro name (OFF) specified with the #ifndef instruction has not been */
/* defined, .DATA 100 is assembled. */
```

11.7.4 #else Instruction

The #else instruction can be used in an if clause.
The #else instruction inverts previous assembly condition.

■ #else Instruction

[Format]

#else

[Description]

The #else instruction can be used in an if clause.
The #else instruction inverts previous assembly condition.
If the condition specified in the if clause starting with the #if, #ifdef, or #ifndef instruction is true, the text between the #else instruction and the corresponding #endif instruction is not assembled.
If the condition specified in the if clause starting with the #if, #ifdef, or #ifndef instruction is false, the text between the #else instruction and the corresponding #endif instruction is assembled.

[Examples]

```
#define      NUM      3
#if         NUM == 3
    .SDATA    "ABC"

#else
    .SDATA    "DEF"    /* This is assembled. */
#endif
#ifdef      NUM
    .SDATA    "=="     /* This is assembled. */
#else
    .SDATA    "==="
#endif
#ifndef     NUM
    .SDATA    "NO"
#else
    .SDATA    "OK"     /* This is assembled. */
#endif
```


11.7.5 #elif Instruction

The **#elif** instruction can be used in an if clause.

The **#elif** instruction has the same function as that of the **#else** and **#if** instructions used together.

Thus, the **#elif** instruction is valid only if the assembly condition is false.

■ #elif Instruction

[Format]

#elif conditional-expression

conditional-expression: Preprocessor expression

[Description]

The **#elif** instruction can be used in an if clause.

The **#elif** instruction has the same function as that of the **#else** and **#if** instructions used together.

Thus, the **#elif** instruction is valid only if the assembly condition is false.

● If the assembly condition is true

The text between this instruction and the corresponding **#endif** instruction is not assembled.

● If the assembly condition is false

The conditional expression (another assembly condition) is evaluated.

If the expression is true, the text between this instruction and the next conditional assembly instruction is assembled.

If the expression is false, the text between this instruction and the corresponding **#else**, **#elif** or **#endif** instruction is not assembled.

[Examples]

```
#define      NUM      3
#if         NUM==1
    .SDATA    "ABC"

#elif      NUM==2
    .SDATA    "DEF"

#elif      NUM==3
    .SDATA    "GHI"      /* This is assembled */

#endif
#ifdef     NUM
    .SDATA    "==="      /* This is assembled */

#elif      NUM==3
    .SDATA    "****"

#endif
```

```
#ifndef      NUM
            .SDATA    "NO"
#elif       NUM==3
            .SDATA    "OK"      /* This is assembled */
#endif

/* If the conditions are false, as shown below, nothing is assembled */
#if         NUM==10
            .SDATA    "?????"
#elif       NUM==20
            .SDATA    "$$$$$$"
#endif
```

11.7.6 #endif Instruction

The **#endif** instruction indicates the end of conditional assembly.
If conditional assembly instructions are nested, each **#endif** instruction is valid only for the most internal instruction in **#if**,**#ifdef**,or **#ifndef** instruction.

■ **#endif** Instruction

[Format]

#endif

[Description]

The **#endif** instruction indicates the end of conditional assembly.
If conditional assembly instructions are nested, each **#endif** instruction is valid only for the most internal instruction in **#if**,**#ifdef**,or **#ifndef** instruction.

[Examples]

```
#ifndef    _IODEF_
#define    _IODEF_                /* This is assembled */
#define    VER            2        /* This is assembled */
#if       VER == 1
#define    IOBUFNUM        10
#elif     VER == 2
#define    IOBUFNUM        15        /* This is assembled */
#elif     VER == 3
#define    IOBUFNUM        22
#endif    /* Indicates the end of "if VER == 1"
#ifdef    IOCH
#undef     IOCH
#endif    /* Indicates the end of #ifdef IOCH */
#define    IOCH            4        /* This is assembled */
#endif    /* Indicates the end of #ifndef _IODEF_ */
```

11.8 Macro Name Replacement

The **#define** instruction defines a macro name. Each time the macro name appears, it is replaced by the character string defined for it.

The **#set** instruction defines a numeric value for a macro name. The result of evaluating an expression is set.

The **#undef** instruction deletes a macro name.

■ Macro Name Replacement

Each time a macro name appears in text, the macro name is replaced by the character defined it. This is referred to as macro replacement.

Macro replacement is valid for any macro names defined by the **#define** and **#set** instructions when the macro names appear in text.

■ Macro Replacement Rules

Formal argument and local symbol replacement also conforms to these rules.

Macro replacement is not valid for the following.

- Preprocessor names
- Characters in comments
- Characters in character strings

[Examples]

```
#define      ABC      define
#ABC        NUM      10      /* #ABC is not replaced by #define */
#define      MSG      Fujitsu
/* MSG */                      /* MSG is not replaced by Fujitsu */
.SDATA      "MSG"      /* .SDATA "MSG" is not replaced by "Fujitsu" */
```

- A macro name can be indicated explicitly by placing a backslash (\) in front of it. Usually, the backslash (\) can be omitted.

[Examples]

```
#define      MD      FPU
#define      SYM      start
.PROGRAM MD      /* Replaced by .PROGRAM FPU */
MD\SYM:          /* Replaced by FPU start: */
```

- If the character string resulting from macro replacement contains another macro name, macro replacement is repeated. Macro replacement can be repeated up to 255 times.

[Examples]

```
#define      NUM      10
#define      ANUM      (NUM+2)
#define      BNUM      ANUM*3
.DATA      BNUM      /* Replaced by .DATA(10+2)*3 */
```

11.8.1 #define Instruction

The **#define** instruction defines a character string for a macro name.

When the macro name is found in text, the macro name is immediately replaced by the defined character string.

Two types of **#define** instruction are available: parameter-less **#define** instruction and parameter-attached **#define** instruction.

■ Parameter-less #define Instruction

[Format]

```
#define macro-name character-string-to-be-defined
```

[Description]

The parameter-less **#define** instruction defines a macro name without an argument.

The character string is defined for the macro name.

When the macro name is found in text, the macro name is immediately replaced by the defined character string.

If no character string is omitted, a null character is defined.

The **#define** instruction cannot change a character string that has already been defined for a macro name.

[Examples]

#define	DB	.DATA.B
#define	DW	.DATA.W
	DB	0, 2
	DW	0xffffffff

■ Parameter-attached #define Instruction

[Format]

```
#define macro-name(formal-argument[,formal-argument] ... ) character-string-to-be-defined
```

[Description]

The parameter-attached #define instruction defines a macro name with an arguments.

The macro name must be immediately followed by a left parenthesis "(". There must be no blanks or comments between the macro name and the left parenthesis.

The parenthesis must be followed by the formal arguments and by a right parenthesis ")", and finally by the character that is being defined.

When the macro name is found in text, a format check is performed first, then the formal arguments are replaced by the corresponding arguments and the macro name is expanded.

A macro name with arguments is replaced in the following format:

macro-name (argument, [argument] ...)

The number of arguments must be equal to the number of formal arguments.

If no character string is specified, a null character is defined.

The #define instruction cannot change a character string that has already been defined for a macro name.

[Examples]

```
#define      NUM      10
#define      eq(a, b)  a==b
#define      ne(a, b)  a!=b
               .DATA      eq(NUM, 10)          /* Replaced by .DATA (10==10) */
               .DATA      ne(5, NUM)           /* Replaced by .DATA (5!=10) */
               .DATA      eq(ne(5, NUM), 1)     /* Replaced by .DATA ((5!=10)==1) */
```

11.8.2 Replacing Formal Macro Arguments by Character Strings (# Operator)

The # operator replaces the argument corresponding to a formal argument with a character string.

■ Replacing Formal Macro Arguments with Character Strings (# Operator)

[Format]

#formal-argument

[Description]

The # operator can be used in the character string to be defined in a parameter-attached #define instruction.

The # operator replaces the argument corresponding to a formal argument with a character string.

Any blanks before and after the argument are deleted before being replaced by the character string.

[Example]

```
#define      MDL (name) #name
            .SDATA  MDL(test)      /* Replaced by .SDATA test */
```

11.8.3 Concatenating the Characters to be Replaced by Macro Replacement (## Operator)

The ## operator concatenates the characters before and after it.

■ Concatenating the Characters to be Replaced by Macro Replacement (## Operator)

[Format]

character##character

[Description]

The ## operator can be used in the character string defined in a #define instruction.

In macro replacement, when the ## operator appears in character string that is being defined, the characters before and after the ## operator are concatenated, and the ## operator removed.

With the argument-attached #define instruction, if the ## operator is immediately preceded or succeeded by a formal argument in the character string, the formal argument is replaced by the corresponding real argument before being concatenated.

The character string resulting from concatenation by the ## operator is subject to macro replacement.

[Example]

#define	abcd	10	
#define	val	ab##cd	
	.DATA	val	/* Replaced by .DATA 10 */
#define	val2(x)	x##cd	
	.DATA	val2(ab)	/* Replaced by .DATA 10 */

11.8.4 #set Instruction

The **#set** instruction evaluates an expression and defines the result for a macro name as a decimal constant.

■ **#set Instruction**

[Format]

#set macro-name expression

expression: Preprocessor expression

[Description]

The **#set** instruction evaluates an expression and defines the result for a macro name as a decimal constant.

The **#set** instruction can be executed for the same macro name as many times as possible.

The difference from the **#define** instruction is that the **#set** instruction allows the macro name to be used as a variable.

[Examples]

```
#set      CNT      1
#repeat   3
          .DATA    CNT
#set      CNT      CNT+1
#endm

          .DATA    1
          .DATA    2
          .DATA    3
```

Repeat expansion

If the second **#set** instruction is replaced by a **#define** instruction (**#define CNT CNT+1**), **CNT** cannot be replaced correctly by macro replacement, causing an error.

11.8.5 #undef Instruction

The **#undef** instruction deletes the specified macro name.

■ #undef Instruction

[Format]

#undef macro-name

[Description]

The **#undef** instruction deletes the specified macro name.

This instruction is not valid for formal arguments or local symbols.

Once a macro name is deleted, it can be redefined by the **#define** instruction.

[Example]

```
#define      ABC      100*2
            .DATA      ABC      /* Replaced by .DATA 100*2 */
#undef      ABC
#define      ABC      "***ABC***"
            .SDATA      ABC      /* Replaced by .SDATA "***ABC***" */
```

11.8.6 #purge Instruction

#purge deletes all macro names.

■ #purge Instruction

[Format]

#purge

[Description]

#purge deletes all macro names.
This instruction is not valid for formal arguments or local symbols.
Once the macro names are deleted, they can be redefined by the #define instruction.

[Example]

```
#define      ABC      100*2
#define      DEF      200*3
      .DATA      ABC      /* Replaced by .DATA 100*2 */
      .DATA      DEF      /* Replaced by .DATA 200*3 */

#purge
#define      ABC      "***ABC***"
      .SDATA      ABC      /* Replaced by .SDATA "***ABC***" */
```

11.9 #include Instruction

The **#include** instruction reads the specified file to include it in the source program.

■ #include Instruction

[Format]

#include <file-name>	[Format 1]
#include "file-name"	[Format 2]
#include file-name	[Format 3]

[Description]

The **#include** instruction reads the specified file to include it in the source program.

The file included by the **#include** instruction is called an include file.

An include file can include another file using the **#include** instruction, a feature called nesting files.

Nesting is possible up to eight levels.

Depending on the format used, the **#include** statement searches for a file through different directories.

Note:

If the file name specified with the **#include** instruction is a relative path name, it is handled as being relative to the directory containing the source file.

■ File Search for Format 1

If format 1 is used, the instruction searches for the file through the following directories in the indicated order until the file is found:

1. Directory specified by the **-I** start-time option
2. Directory specified by the **INC896** environment variable
3. Include directory in the development environment
 - %FETOOL%\LIB\869\INCLUDE

■ File Search for Formats 2 and 3

If format 2 or 3 is used, the instruction searches for the file through the following directories in the indicated order until the file is found:

1. An attempt is made to access the file with the specified file name.
2. Directory specified by the -I start-time option
3. Directory specified by the INC896 environment variable
4. Include directory in the development environment
 - %FETOOL%\LIB\869\INCLUDE

[Examples]

```
#include    <stdio.h >
#include    "stype.h"
#include    stype.h
#include    <sys\iodef.h >
#include    ". . \iodef.h"
#include    \usr\local\iodef.h
```

11.10 #line Instruction

The **#line** instruction changes the line number of the next line to the specified line number.

■ #line Instruction

[Format]

<code>#line line-number [file-name]</code>
--

file-name: Character string

[Description]

The **#line** instruction changes the line number of the next line to the specified line number.

If a file name is specified, the file name is also changed to this file name.

[Example]

```
#line        1000 "test.asm"
```

As a result, the line number of the line following the **#line** instruction line is changed to 1000, and the file name is changed to "test.asm".

11.11 #error Instruction

The **#error** instruction sends the specified message to the standard output as an error message.

After the **#error** instruction has been executed, no processing is performed.

■ #error Instruction

[Format]

<code>#error error-message</code>

[Description]

The **#error** instruction sends the specified message to the standard output as an error message.

After the **#error** instruction has been executed, no processing is performed.

[Example]

```
#error    Test program miss!
```

11.12 #pragma Instruction

The #pragma instruction does nothing.

■ #pragma Instruction

[Format]

#pragma character-string

[Description]

The #pragma instruction is provided for compatibility with the C preprocessor.

This instruction has no effect on the assembler, and the assembler performs no action.

11.13 No-operation Instruction

The no-operation instruction does nothing.

■ No-operation Instruction

[Format]

#

[Description]

The # symbol is treated as a no-operation instruction provided it is followed by a line-feed character only.

The no-operation instruction performs no action.

[Examples]

#

#

#

.SECTION CODE

:

11.14 Defined Macro Names

Defined macro names are reserved.

They cannot be deleted by the #undef instructions.

■ Defined Macro Names

● __LINE__

This macro name is replaced by the decimal line number of the current source line.

[Example] If the current source line number is 101

```
.DATA      __LINE__      /* Replaced by .DATA 101 */
```

● __FILE__

This macro name is replaced by the current source file name.

[Example] If the current source file name is t1.asm

```
.SDATA      __FILE__      /* Replaced by .SDATA "t1.asm" */
```

● __DATE__

This macro name is replaced by the date of assembly in the following format:

"Mmm dd yyyy"

where Mmm is an abbreviation for the month name, dd is the day, and yyyy is the year.

[Example] Assembled in August 7, 1966

```
.SDATA      __DATE__      /* Replaced by .SDATA "Aug 7 1966" */
```

● __TIME__

This macro name is replaced by the time of assembly in the following format:

"hh:mm:ss"

where hh is hours, mm is minutes, and ss is seconds.

[Example] Assembled at 12:34:56

```
.SDATA      __TIME__      /* Replaced by .SDATA "12:34:56" */
```

● __FASM__

This macro name is replaced by the decimal constant 1.

[Example]

```
.DATA      __FASM__      /* Replaced by .DATA 1 */
```

● __CPU_8L__

This macro name is replaced by the decimal constant 1.

This macro name is valid for the F²MC-8L Family only.

- `__CPU_8FX__`

This macro name is replaced by the decimal constant 1.

This macro name is valid for the F²MC-8FX Family only.

■ Defined Macro Name

- `defined (macro-name)`

This macro name is replaced by the decimal constant 1 if the specified macro name has been defined, and by the decimal constant 0 if the macro name has not been defined.

[Examples]

	<code>.DATA</code>	<code>defined(ABC)</code>	<code>/* .Replaced by .DATA0 */</code>
<code>#define</code>	<code>ABC</code>		
	<code>DATA</code>	<code>defined(ABC)</code>	<code>/* .Replaced by .DATA1 */</code>

11.15 Differences from the C Preprocessor

This section explains the differences between the assembler's preprocessor and the C preprocessor.

■ Differences from the C Preprocessor

The following eight functions are provided by the assembler's preprocessor, but not by the C preprocessor:

- `#macro` instruction
- `#local` instruction
- `#exitm` instruction
- `#endm` instruction
- `#repeat` instruction
- `#set` instruction
- `#purge` instruction
- `__FASM__` defined macros

The function that is not the same in the assembler's preprocessor and the C preprocessor is:

- `#pragma` instruction
 - Assembler's preprocessor: Does nothing.
 - C preprocessor: See the C language manual.

CHAPTER 12

ASSEMBLER PSEUDO-MACHINE INSTRUCTIONS

The assembler supports the use of assembler pseudo machine instructions.

A set of machine instructions for each MCU can be specified as a single machine instruction. This type of instruction is called an assembler pseudo machine instruction.

This chapter describes the formats and functions of the assembler pseudo machine instructions.

12.1 Assembler Pseudo Machine Instructions for the F²MC-8L/8FX Family

12.1 Assembler Pseudo Machine Instructions for the F²MC-8L/8FX Family

In the F²MC-8L/8FX family, three types of assembler pseudo machine instruction can be used.

Another type of branch instruction can also be used to calculate the distance to a branch destination and to allocate the optimum code.

■ Assembler Pseudo Machine Instructions for the F²MC-8L/8FX Family

In the F²MC-8L/8FX family, the following types of assembler pseudo machine instructions can be used:

- Branch pseudo machine instructions
- Operation pseudo machine instructions
- Miscellaneous pseudo machine instructions

In addition, the following type of branch instruction can be used to calculate the distance to a branch destination and to allocate the optimum code:

- Optimum allocation branch pseudo machine instruction

12.1.1 Branch Pseudo Machine Instructions

The following branch pseudo machine instructions can be used:

BV, BNV, BLE, BGT, BLS, BHI

■ Branch Pseudo Machine Instructions

Table 12.1-1 lists the branch pseudo machine instructions.

Table 12.1-1 Branch Pseudo Machine Instructions

Format	Condition	Expansion	Remark
BV label	Branch if V=1	BN \$+7 BLT label JMP \$+5 BGE label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +122 bytes).
BNV label	Branch if V=0	BN \$+7 BGE label JMP \$+5 BLT label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +122 bytes).
BLE label	Branch if (Z or N or V)=1	BEQ label BLT label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +125 bytes).
BGT label	Branch if (Z or N or V)=0	BEQ \$+4 BGE label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +127 bytes).
BLS label	Branch if (Z or N)=1	BEQ label BLO label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +125 bytes).
BHI label	Branch if (Z and N)=0	BEQ \$+4 BHS label	A branch is allowed within the range of (next instruction address -128 bytes) to (next instruction address +127 bytes).

12.1.2 Operation Pseudo Machine Instructions

The following operation pseudo machine instructions can be used:
NOTA, TESTA

■ Operation Pseudo Machine Instructions

Table 12.1-2 lists the operation pseudo machine instructions that can be used.

Table 12.1-2 Operation Pseudo Machine Instructions

Format	Flag change	Expansion	Remark
NOTA	Z, N	XOR A, #0FFH	No A or T operation is performed.
TESTA	Z, N, V, C	CMP A, #00H	Whether the accumulator is 0 is determined.

12.1.3 Miscellaneous Pseudo Machine Instructions

The following miscellaneous pseudo machine instructions can be used:
PUSHAT, POPAT, SETRP, SETRPA, SETDB, SETDBA

■ Miscellaneous Pseudo Machine Instructions

Table 12.1-3 lists the miscellaneous pseudo machine instructions that can be used.

Table 12.1-3 Miscellaneous Pseudo Machine Instructions (1 / 2)

Format	Flag change	Expansion	Remarks
PUSHAT		PUSHW A XCHW A, T PUSHW A	After the execution of this instruction, the contents of A and T are exchanged.
POPAT		PUSHW A XCHW A, T POPW A	
SETRP #imm		MOVW A, PS MOVW A, #07FFH ANDW A MOVW A, #(imm << 11) ORW A MOVW PS, A	With this instruction, the contents of A and T are not saved. This expansion is only F ² MC-8L family.
		MOV A, 0x78 AND A, #3 OR A, #(imm << 3) MOV 0x78, A	With this instruction, the contents of A and T are not saved. This expansion is only F ² MC-8FX family.

Table 12.1-3 Miscellaneous Pseudo Machine Instructions (2 / 2)

Format	Flag change	Expansion	Remarks
SETRPA		MOVW A, #001FH ANDW A PUSHW A MOVW A, PS MOVW A, #07FFH ANDW A XCHW A, T POPW A CLRC ROLC A ROLC A ROLC A SWAP ORW A MOVWA PS, A	With this instruction, the contents of A and T are not saved. This expansion is only F ² MC-8L family.
		CLRC ROLC A ROLC A ROLC A MOV A,0x78 AND A,#3 OR A MOV 0x78,A	With this instruction, the contents of A and T are not saved. This expansion is only F ² MC-8FX family.
SETDB #imm		MOV A,0x78 AND A,#0xF8 OR A,#(imm & 7) MOV 0x78,A	With this instruction, the contents of A and T are not saved. This format is only F ² MC-8FX family.
SETDBA		AND A,#7 MOV A,0x78 AND A,#0xF8 OR A MOV 0x78,A	With this instruction, the contents of A and T are not saved. This format is only F ² MC-8FX family.

12.1.4 Optimum Allocation Branch Pseudo Machine Instructions

Optimum allocation branch pseudo machine instructions are used to calculate the distance to a branch destination, to determine whether a relative branch is allowed, and then to allocate the optimum code.

The following optimum allocation branch pseudo machine instruction can be used:
Bcc16

■ Optimum Allocation Branch Pseudo Machine Instructions

Optimum allocation branch pseudo machine instructions are used to calculate the distance to a branch destination, to determine whether a relative branch is allowed, and then to allocate the optimum code.

Table 12.1-4 lists the optimum allocation branch pseudo machine instructions that can be used.

Table 12.1-4 Optimum Allocation Branch Pseudo Machine Instruction (1 / 3)

Branch expansion instruction	Operation	Distance	Generated instruction
BZ16 label	Branch if (Z)=1	-128 to +127	BZ label
		Others	BNZ \$+5 JMP label
BNZ16 label	Branch if (Z)=0	-128 to +127	BNZ label
		Others	BZ \$+5 JMP label
BEQ16 label	Branch if (Z)=1	-128 to +127	BEQ label
		Others	BNE \$+5 JMP label
BNE16 label	Branch if (Z)=0	-128 to +127	BNE label
		Others	BEQ \$+5 JMP label
BC16 label	Branch if (C)=1	-128 to +127	BC label
		Others	BNC \$+5 JMP label
BNC16 label	Branch if (C)=0	-128 to +127	BNC label
		Others	BC \$+5 JMP label
BLO16 label	Branch if (C)=1	-128 to +127	BLO label
		Others	BHS \$+5 JMP label

Table 12.1-4 Optimum Allocation Branch Pseudo Machine Instruction (2 / 3)

Branch expansion instruction	Operation	Distance	Generated instruction
BHS16 label	Branch if (C)=0	-128 to +127	BHS label
		Others	BLO \$+5 JMP label
BN16 label	Branch if (N)=1	-128 to +127	BN label
		Others	BP \$+5 JMP label
BP16 label	Branch if (N)=0	-128 to +127	BP label
		Others	BN \$+5 JMP label
BLT16 label	Branch if (V) xor (N)=1	-128 to +127	BLT label
		Others	BGE \$+5 JMP label
BGE16 label	Branch if (V) xor (N)=0	-128 to +127	BGE label
		Others	BLT \$+5 JMP label
BBC16 bit,label	Branch if (bit)=0	-128 to +127	BBC bit,label
		Others	BBS \$+6 JMP label
BBS16 bit,label	Branch if (bit)=1	-128 to +127	BBS label
		Others	BBC \$+6 JMP label
BV16 bit,label	Branch if (V)=1	-128 to +127	BN \$+7 BLT label JMP \$+5 BGE label
		Others	BN \$+7 BGE \$+10 JMP label BLT \$+5 JMP label
BNV16 bit,label	Branch if (V)=0	-128 to +127	BN \$+7 BGE label JMP \$+5 BLT label
		Others	BN \$+7 BLT \$+10 JMP label BGE \$+5 JMP label

Table 12.1-4 Optimum Allocation Branch Pseudo Machine Instruction (3 / 3)

Branch expansion instruction	Operation	Distance	Generated instruction	
BLE16 bit,label	Branch if (Z) or ((N) or (V))=1	-128 to +127	BEQ	label
		Others	BLT	label
BGT16 label	Branch if (Z) or ((N) or (V))=0	-128 to +127	BEQ	label
		Others	BGE	\$+5
BLS16 label	Branch if (Z) or (C)=1	-128 to +127	JMP	label
		Others	BEQ	\$+4
BHI16 label	Branch if (Z) or (C)=0	-128 to +127	BHS	\$+5
		Others	JMP	label

CHAPTER 13

STRUCTURED INSTRUCTIONS

**Structured instructions are provided to perform structured programming.
This chapter explains the structured instructions.**

13.1 Overview of Structured Instructions

13.2 Structured Program Instructions

13.3 Expressions (Assignment Expressions, Operation and Assignment Expressions, Increment/Decrement Expressions)

13.1 Overview of Structured Instructions

Structured instructions are provided to perform structured programming.

■ Overview of Structured Instructions

The structured instructions are provided to perform structured programming.

[Example]

```

        .program          "Sample_for_8L"
        .import           func
        .import           paral
        .section          segment1,code

func1:
        .(r0 = #0)                // Assignment expression
        .while (r0 <= #10)        // Structured program instruction
            .(paral < <+ #1)      // Operation and assignment expression
            .call              func
            .(r0++)              // Increment expression
        .endw
        ret
        .end

```


13.2 Structured Program Instructions

The structured program instructions consist of five types of control syntax instructions and two control transfer instructions. The five types of control syntax instructions are 2-process selection, multiple-process selection, computed repetition, at-end condition repetition, and execution condition repetition. The two control transfer instructions are the break and continue instructions.

Table 13.2-1 lists the structured program instructions.

■ Overview of Structured Program Instructions

The structured program instructions consist of five types of control syntax instructions and two control transfer instructions. The five types of control syntax instructions are 2-process selection, multiple-process selection, computed repetition, at-end condition repetition, and execution condition repetition. The two control transfer instructions are the break and continue instructions.

Table 13.2-1 lists the structured program instructions.

Table 13.2-1 Structured Program Instructions

Structured program instruction	Purpose
.if	2-process selection
.else	
.endif	
.switch	Multiple-process selection
.case	
.default	
.endsw	
.for	Computed repetition
.endf	
.repeat	At-end condition repetition
.until	
.while	Execution condition repetition
.endw	
.break	Break
.continue	Continue

13.2.1 Conditional Expressions in a Structured Program Instruction

In a structured program instruction, conditional expressions used to control the program can be written.

■ Conditional Expressions in a Structured Program Instruction

Three types of conditional expressions are provided: condition code expressions, relational conditional expressions, and bit conditional expressions.

A compound conditional expression, created by linking up to eight conditional expressions of above three types, can also be written.

● Condition code expression

Condition codes based on the bit indicating the CPU condition can be used in a condition code expression.

Condition codes must be enclosed by angle brackets (< >).

Table 13.2-2 lists the conditional expressions that can be used in a condition code expression.

Table 13.2-2 Conditional Expressions that can be Used in a Condition Code Expression

Z (EQ)	Z==1	NZ (NE)	Z==0
C (LO)	C==1	NC (NS)	C==0
N	N==1	P	N==0
V	V==1	NV	V==0
LT	(V or N)==1	GE	(V or N)==0
LE	((V xor N) or Z)==1	GT	((V xor N) or Z) ==0
LS	(C or Z) ==1	HI	(C or Z) ==0

[Example]

```
.IF (<NE >)
    mov a, r1
.else
    mov a, r2
.endif
```

● Relational conditional expression

A relational conditional expression has two operands. The relational condition is determined according to the CPU condition set by the results of comparing the operands.

Table 13.2-3 lists the conditional expressions that can be used in a relational conditional expression.

Table 13.2-3 Conditional Expressions that can be Used in a Relational Conditional Expression

==	Equal to	!=	Not equal to
<	Less than	<=	Less than or equal to
>	Greater than	>=	Greater than or equal to

To perform comparison between unsigned values under the relational conditions of <, <=, >, and >=, append .u to the relational expression.

Table 13.2-4 list the terms that can be used as an operand in relational conditional expressions.

Table 13.2-4 Operands that can be Used in a Relational Conditional Expressions

Left term	Right term
A	dir/#d8/@EP/@IX+off/Ri
@EP	#d8
@IX+off	
Ri	

[Example]

```
.if (A <= #0)
    call func
.endif
```

● Bit conditional expression

Bit conditional expressions use the bit status of an operand to evaluate the condition.

The system evaluates the condition as true when the bit status is 1 and as false when the bit status is 0.

When an exclamation mark (!) is appended, the logic is inverted.

[Example]

```
.while (bit_flag)
    call func
.endw
```

● Compound conditional expression

A compound conditional expression consisting of up to eight conditional expressions of the three types already described uses logical OR (||) or logical AND (&&) to evaluate the condition.

Note that the zero flag (Z) is updated in a bit conditional expression when a condition code expression is used after the bit expression.

Logical OR and logical AND cannot be used together as compound conditions in a compound conditional expression.

A compound conditional expression can be used in the following three types of structured program instructions:

- 2-process selection syntax (.if syntax)
- At-end condition repetition syntax (.repeat syntax)
- Execution condition repetition syntax (.while syntax)

[Example]

```
.while (bit_flag1 && bit_flag2)
    call func
.endw
```

13.2.2 Generation Rules for Structured Program Instructions

This section describes rules for generating symbols and statements with the structured program instructions.

■ Rules for Generating Symbols

A structured program instruction generates a symbol in the following format:

Structured program instruction	symbol format
2-process selection	_Innnnn
Multiple-process selection	_Snnnnn
Computed repetition	_Fnnnnn
At-end condition repetition	_Rnnnnn
Execution condition repetition	_Wnnnnn

A symbol begins with two underscores (__), which are followed by a 5-digit decimal value.

The 5-digit value is incremented by one each time a symbol is generated. Values from 00001 to 65535 can be generated.

■ Rules for Generating Statements

When a machine instruction is generated by a structured program instruction, it is preceded by a single tab code as shown below, regardless of any nesting in the structured program instruction.

The system also outputs a tab code between the machine instruction and its operand.

Tab code	Machine instruction	Tab code	Operand	New-line code
----------	---------------------	----------	---------	---------------

When a symbol is defined, a colon (:) is added so that only the symbol, starting in the first column on the line, is generated.

name-of-generated-symbol

13.2.3 Format for Structured Program Instructions

This section describes the basic format for structured program instructions. For details of each structured program instruction, see the sections from Section "13.2.4 2-process Selection Syntax", to Section "13.2.9 Control Transfer Instructions".

■ Format for Structured Program Instructions

[Format]

```
.instruction (conditional-expression)
```

[Description]

● instruction

Specify a structured program instruction such as .if or .while.

".instruction" represents a single instruction. The period (.) and instruction must not be separated.

For details of each structured program instruction, see sections from Section "13.2.4 2-process Selection Syntax", to Section "13.2.9 Control Transfer Instructions".

● conditional-expression

Specify one of the following expressions: a condition code expression, a relational conditional expression, or a bit conditional expression.

■ Format for Conditional Expressions

● Condition code expression

```
( <condition-code> )
```

● Relational conditional expression

```
(op1 relational-expression op2)
```

● Bit conditional expression

```
([!]op1)
```

When an exclamation mark (!) is appended, the logic is inverted (the condition is true when the bit is 0).

13.2.4 2-process Selection Syntax

The 2-process selection syntax generates an instruction that executes process-1 when the conditional expression is true and executes process-2 when the conditional expression is false.

A condition code expression, relational conditional expression, or bit conditional expression can be written as a conditional expression.

■ 2-process Selection Syntax

[Format]

```
.if (conditional-expression)
    process-1
[.else]
    process-2
.endif
```

[Description]

The 2-process selection syntax generates an instruction that executes process-1 when the conditional expression is true and executes process-2 when the conditional expression is false.

A condition code expression, relational conditional expression, or bit conditional expression can be written as conditional-expression.

[Example]

```
.if ( <ne>)                                BEQ16      __I00001
    mov  a,r0                               mov        a,r0
.else                                       JMP        __I00002
    mov  a,r1                               __I00001:   mov        a,r1
.endif                                     __I00002:
```

13.2.5 Multiple-Process Selection Syntax

The multiple-process selection syntax compares the switch and case operands. If these operands are equal, the process following the .case statement is executed. Otherwise, the syntax generates an instruction that performs the comparison in the subsequent .case statement.

The .case statement can be used any number of times.

To exit a multiple-process selection syntax, use a .break statement.

■ Multiple-process Selection Syntax

[Format]

```
.switch (operand-1)
.case (operand-n)
    process
    [.break]
[.default]
    process
.endsw
```

[Description]

The multiple-process selection syntax compares the switch operand (operand-1) and case operand (operand-n). If these operands are equal, the process following the .case statement is executed. Otherwise, the syntax generates an instruction that performs the comparison in the subsequent .case statement.

A .case statement can be used any number of times.

To exit a multiple-process selection syntax, use a .break statement.

Table 13.2-5 list the items that can be used as an operand.

Table 13.2-5 Items that can be Used as an Operand

switch operand	case operand
A	dir/#d8/@EP/@IX+off/Ri
dir	#d8
@EP	
@IX+off	
Ri	

[Example]

.switch (A)	
.case (#0)	CMP A,#0
	BNE16 __S00002
mov a,r0	mov a,r0
.break	JMP __S00001
.case (#1)	__S00002:
	CMP A,#1
	BNE16 __S00003
mov a,r1	mov a,r1
.break	JMP __S00001
.default	__S00003:
mov a,r0	mov a,r0
.endsw	__S00001:

13.2.6 Computed Repetition Syntax

The computed repetition syntax generates an instruction that executes a process repeatedly while the condition is true.

Whether to increment or decrement the counter is specified by a sign prefixed to op4. The process is executed only if the condition is true in the first iteration.

The condition is evaluated by comparing op1 and op3.

■ Computed Repetition Syntax

[Format]

```
.for (op1=op2, relational-expression op3,[{+|-}]op4)
    process
.endf
```

[Description]

The computed repetition syntax generates an instruction that executes a process repeatedly while the condition is true.

Whether to increment or decrement the counter is specified by a sign prefixed to op4. If the sign is omitted, the counter is incremented.

The process is executed only if the condition is true in the first iteration.

The condition is evaluated by comparing op1 and op3.

Table 13.2-6 list the items that can be used as an operand.

Table 13.2-6 Items that can be Used as an Operand

op1	op2	op3	op4
A	dir #d8 @EP @IX+off Ri @A	dir #d8 @EP @IX+off Ri	Dir #d8 @EP @IX+off Ri
dir	#d8	#d8	
@EP			
@IX+off			
Ri			

[Example]

```

.for (A=#0, <=.u #255, +1)
    incw ep
.endf

MOV A,#0
JMP    __F00001
__F00002:
    incw    ep
__F00003:
    ADDC    A,#1
__F00001:
    CMP     A,#255
    BHI16   __F00001
__F00004:

```

13.2.7 At-end Condition Repetition Syntax

The at-end condition repetition syntax generates an instruction that executes a process repeatedly until the condition is true.

A condition code expression, relational conditional expression, or bit conditional expression can be used as the conditional expression.

In the at-end condition repetition syntax, since the condition is evaluated after the process is executed, the process is executed at least once even though the condition is true in the first iteration.

■ At-end Condition Repetition Syntax

[Format]

```
.repeat
    process
.until (conditional-expression)
```

[Description]

The at-end condition repetition syntax generates an instruction that executes a process repeatedly until the condition is true.

A condition code expression, relational conditional expression, or bit conditional expression can be used as the conditional expression.

In the at-end condition repetition syntax, since the condition is evaluated after the process is executed, the process is executed at least once even though the condition is true in the first iteration.

[Example]

.repeat	__R00001:
decw ep	decw ep
.until (<eq>)	__R00002:
	BNE16 __R00001
	__R00003:

13.2.8 Execution Condition Repetition Syntax

The execution condition repetition syntax generates an instruction that executes a process while the condition is true.

A condition code expression, relational conditional expression, or bit conditional expression can be used as the conditional expression.

In the execution condition repetition syntax, since the condition is evaluated before the process is executed, the process is never executed if the condition is not true in the first iteration.

■ Execution Condition Repetition Syntax

[Format]

```
.while (conditional-expression)
    process
.endw
```

[Description]

The execution condition repetition syntax generates an instruction that executes a process while the condition is true.

A condition code expression, relational conditional expression, or bit conditional expression can be used as the conditional expression.

In the execution condition repetition syntax, since the condition is evaluated before the process is executed, the process is never executed if the condition is not true in the first iteration.

[Example]

```
.while ( <le>)
                                JMP      __W00002
                                __W00001:
                                incw     ep
                                incw     ep
                                __W00002:
                                BLE16    __W00001
                                __W00003:
```

13.2.9 Control Transfer Instructions

Control transfer instructions exit from the syntax or transfer control to the repetition in a repetition syntax.

A `.break` instruction exits from the innermost syntax that includes the instruction.

A `.continue` instruction transfers control to the repetition in the innermost repetition syntax that includes the instruction.

■ **.break Instruction**

[Format]

`.break`

[Description]

A `.break` instruction exits from the innermost syntax that includes the instruction.
This instruction can be used in a multiple-process selection syntax or any repetition syntax.

[Example]

```
.while ( <le>)                                JMP      __W00002
__W00001:
    incw  ep                                incw      ep
    .if ( <ne>)                            BEQ16    __I00001
        .break                            JMP      __W00003
    .endif                                __I00001:
    .endw                                __W00002:
                                        BLE16    __W00001
                                        __W00003:
```

■ **.continue Instruction**

[Format]

`.continue`

[Description]

A `.continue` instruction transfers control to the repetition in the innermost repetition syntax that includes the instruction.
This instruction can be used in any repetition syntax.

[Example]

.while (<le>)	JMP	__W00002
	__W00001:	
incw ep	incw	ep
.if (<ne>)	BEQ16	__I00001
.continue	JMP	__W0002
.else	JMP	__I00002
	__I00001:	
.break	JMP	__W00003:
.endif	__I00002:	
.endw	__W00002:	
	BLE16	__W00001
	__W00003:	

13.3 Expressions (Assignment Expressions, Operation and Assignment Expressions, Increment/Decrement Expressions)

The assembler supports expressions in addition to the structured program instructions. Expressions are classified as assignment expressions, operation and assignment expressions, and increment/decrement expressions.

■ Overview of Expressions

The assembler supports expressions in addition to the structured program instructions.

Table 13.3-1 lists expression types.

Table 13.3-1 Expression Types

Assignment expression	Assigns the right term to the left term.
Operation and assignment expression	Performs arithmetic operation on the right and left terms and assigns the result to the right term.
Increment/decrement expression	Increments or decrements a term by one and makes the result the value of the term.

Table 13.3-2 lists available expressions.

Table 13.3-2 Expression

Expression	Purpose
.(op1 = op2)	Assignment expression
.(op1 += op2)	Operation and assignment expression
.(op1 -= op2)	
.(op1 *= op2)	
.(op1 /= op2)	
.(op1 %= op2)	
.(op1 &= op2)	
.(op1 = op2)	
.(op1 ^= op2)	
.(op1 <<= op2)	
.(op1 >= op2)	
.(op++)	Increment expression
.(op--)	Decrement expression

13.3.1 Format for Expressions

This section describes the basic format for expressions.

For details of each expression, see the sections from Section "13.3.2 Assignment Expressions", to Section "13.3.4 Increment/Decrement Expressions".

■ Format for Expressions

[Format]

.(expression)

[Description]

Use one of the following expressions for expression: an assignment expression, operation and assignment expression, increment expression, or decrement expression.

An expression must start with a period (.) and an opening parenthesis "(", and end with a closing parenthesis ")".

For details of each expression, see the sections from Section "13.3.2 Assignment Expressions", to Section "13.3.4 Increment/Decrement Expressions".

13.3.2 Assignment Expressions

Assignment expressions generate an instruction that assigns the right term to the left term.

It is possible to make assignments to multiple variables (up to eight) at the same time if the variables have the same data size as an assigned value. In this case, an assignment to a 16-bit variable and an assignment to an 8-bit variable cannot be specified together. If an 8-bit value is assigned to a 16-bit register, the high-order 8 bits are undefined.

■ Assignment Expression

[Format]

.(op1[.data-size] = op2[.data-size] = ... = op9)

data-size: { b|w }

[Description]

Assignment expressions generate an instruction that assigns the right term to the left term.

It is possible to make assignments to multiple variables (up to eight) at the same time if the variables have the same data size as the assigned value.

When a value is assigned to multiple variables at the same time, an assignment to a 16-bit variable and an assignment to an 8-bit variable cannot be specified together.

If an 8-bit value is assigned to a 16-bit register, the high-order 8 bits are undefined.

[Example]

.(R0=R1=#0)MOV A,#0
MOV R1,A
MOV R0,A

■ Default Data Size

When assignment is performed, the default data size depends on the target variable.

Table 13.3-3 shows the correspondence between the target variable and the default data size.

Table 13.3-3 Correspondence Between the Target Variable and the Default Data Size

Target variable	Data size
EP, IX, PS, SP	16bits
Other than the above	8 bits

■ Items that can be Used as a Term

Almost all items, including registers and symbols that can be used as assembler operands, can be written as a term in an assignment expression.

A value to be assigned and a target variable must not be the same.

Table 13.3-4 lists the items that can be used in an assignment expression.

Table 13.3-4 Items that can be Used in an Assignment Expression

8-bit assignment		
Item	Target variable	Assigned value
Register	A, T, R0 to R7, EP, IX, PS, SP*	A, T, R0 to R7, EP, IX, PS, SP*
Register (indirect address)	@A*, @IX+offset, @EP	@A*, @IX+offset, @EP
Address	Symbols and address value	Symbols and address value
Immediate value		#numeric, #symbol
16-bit assignment		
Register	A, T, EP, IX, PS, SP*	A, T, EP, IX, PS, SP*
Register (indirect address)	@A*, @IX+offset, @EP	@A*, @IX+offset, @EP
Address	Symbols and address value	Symbols and address value
Immediate value		#numeric, #symbol

The items indicated by an asterisk (*) cannot be used when making an assignment to multiple variables.

13.3.3 Operation and Assignment Expressions

Operation and assignment expressions generate an instruction that assigns the operation result of right and left terms to the left term.

■ Operation and Assignment Expression

[Format]

.(op1[.data-size] operation-and-assignment-symbol op2)

data-size: {b|w}

[Description]

Operation and assignment expressions generate an instruction that assigns the operation result of right and left terms to the left term.

Table 13.3-5 lists operators that can be used as an operation and assignment symbol.

Table 13.3-5 Operation and Assignment Operations that can be Used

Operator	8-bit operation and assignment	16-bit operation and assignment	Remarks
Arithmetic operation	+=	+=	
	-=	-=	
	*=	*=	Operation of unsigned values
	/=		Operation of unsigned values
	%=		Operation of unsigned values
Logical operation	&=	&=	
	=	=	
	^=	^=	
Shift operation	<<=		The right term is an immediate value from #1 to #7.
	>>=		The right term is an immediate value from #1 to #7.

■ Default Data Size

When an operation and assignment is performed, the default data size depends on the left term.

Table 13.3-6 shows the correspondence between the left term and the default data size.

Table 13.3-6 Correspondence between the Left Term in an Operation and Assignment Expression and the Default Data Size

Target variable	Data size
EP, IX	16 bits
Other than the above	8 bits

■ Items that can be Used as a Term

Almost all items, including registers and symbols that can be used as assembler operands, can be written as a term in an operation and assignment expression.

Table 13.3-7 lists the items that can be used.

Table 13.3-7 Items that can be Used in an Operation and Assignment Expression

8-bit assignment		
Item	Left term	Right term
Register	A, T, R0 to R7, EP, IX	A, T, R0 to R7, EP, IX
Register (indirect address)	@A*, @IX+offset, @EP	@A*, @IX+offset, @EP
Address	Symbols and address value	Symbols and address value
Immediate value		#numeric, #symbol
16-bit assignment		
Register	A, T, EP, IX	A, T, EP, IX
Register (indirect address)	@A*, @IX+offset, @EP	@A*, @IX+offset, @EP
Address	Symbols and address value	Symbols and address value
Immediate value		#numeric, #symbol

[Example]

.(@IX-0 += @IX-1)

.(_sym -= _sym+1)

.(EP *= #16)

.(@EP /= #7)

.(@EP %= #7)

.(A &= #0x80)

.(A |= #0xF0)

.(A ^= #0xFF)

.(R0 <<= #2)

.(@A >>= #3)

```

CLRC
MOV    A,@IX-0
ADDC   A,@IX-1
MOV    @IX-0,A
CLRC
MOV    A,_sym
MOV    A,_sym+1
SUBC   A
MOV    _sym,A
MOVW   A,EP
MOVW   A,#16
MULU   A
MOVW   EP,A
MOV    A,#7
MOVW   A,@EP
XCHW   A,T
DIVU   A
MOVW   @EP,A
MOV    A,#7
MOVW   A,@EP
XCHW   A,T
DIVU   A
XCHW   A,T
MOVW   @EP,A
AND    A,#0x80
OR     A,#0xF0
XOR    A,#0xFF
MOV    A,R0
ROL    A
ROL    A
AND    A,#0xFC
CLRC
MOV    R0,A
MOV    A,@A
RORC   A
RORC   A
RORC   A
AND    A,#0x1F
CLRC
XCH    A,T
MOV    @A,T

```

13.3.4 Increment/Decrement Expressions

Increment/decrement expressions generate an instruction that increments or decrements a term.

When an 8-bit value is incremented or decremented, values other than R0 to R7 are handled as 16-bit values.

■ Increment/Decrement Expressions

[Format]

```
.(op[.data-size]{++|--})
```

data-size: {b|w}

[Description]

Increment/decrement expressions generate an instruction that increments or decrements a term.

Write ++ for an increment expression and -- for a decrement expression.

When an 8-bit value is incremented or decremented, values other than R0 to R7 are handled as 16-bit values. Therefore, if the result of incrementing or decrementing exceeds eight bits, flags are not affected.

[Example]

.(EP++)	INCW	EP
.(@EP--)	MOV	A,@EP
	DECW	A
	MOV	@EP,A

■ Default Data Size

The default data size in increment/decrement expressions depends on the term.

Table 13.3-8 shows the correspondence between the term and the default data size.

Table 13.3-8 Correspondence between the Term in an Increment/decrement Expression and the Default Data Sized

Target variable	Data size
EP, IX	16 bits
Other than the above	8 bits

■ Terms that can be Used

As a term of an increment/decrement expression, almost all items including registers and symbols that can be used as assembler operands can be described.

Table 13.3-9 lists the items that can be used.

Table 13.3-9 Items that can be Used in an Increment/decrement Expression

Item	8 bits	16 bits
Register	A, T, R0 to R7, EP, IX	A, T, EP, IX
Register (indirect address)	@A, @IX+offset, @EP	@A, @IX+offset, @EP
Address	Symbols and address values	Symbols and address values

APPENDIX

The appendix explain error messages and note restrictions that must be observed.

APPENDIX A Error Messages

APPENDIX B Restrictions

APPENDIX C The Acquisition Method of an Extended Direct Access
Area Bank Number

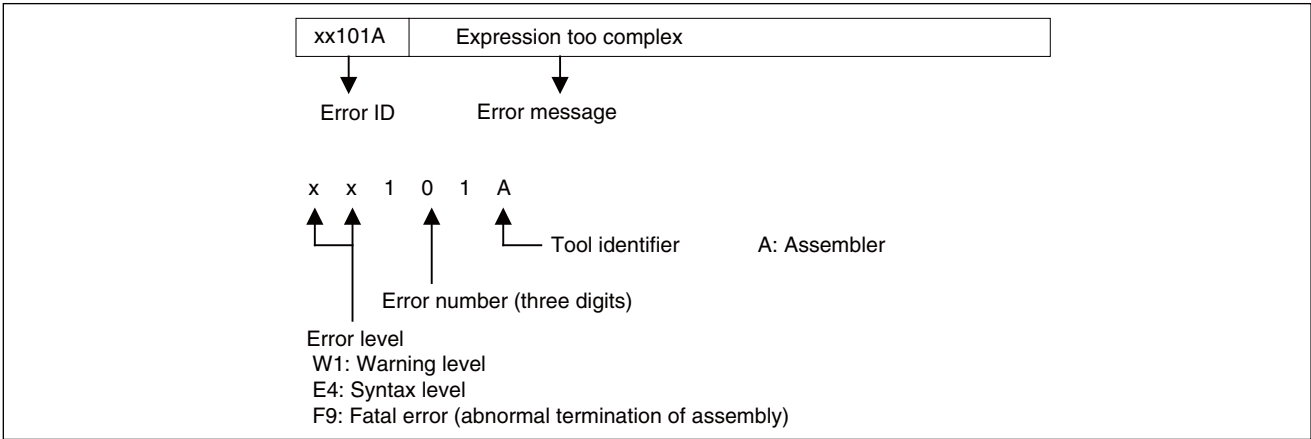
APPENDIX D Difference in Specification of SOFTUNE Assembler
(FASM896S) and Old Assembler (ASM96)

APPENDIX A Error Messages

The assembler displays the error messages below.

■ Format of Error Messages

Figure A-1 Format of Error Message



Note:
Supplementary explanations are provided underneath some error messages.

■ Error Messages

E4101A	Expression too complex
--------	------------------------

[Program action]
Stops the expression evaluation.

E4102A	Missing expression(s)
--------	-----------------------

[Program action]
Stops the expression evaluation.

W1103A	Divide by zero
--------	----------------

[Program action]
Stops the expression evaluation.

E4104A	No terms in parentheses
--------	-------------------------

[Program action]

Stops the expression evaluation.

W1105A	Illegal term in expression
--------	----------------------------

[Program action]

Stops the expression evaluation.

E4106A	Unbalanced parentheses in expression
--------	--------------------------------------

[Program action]

Stops the expression evaluation.

E4107A	Syntax error
--------	--------------

[Program action]

Ignores the instruction.

E4109A	Nothing macro-name
--------	--------------------

[Program action]

Ignores the instruction.

E4110A	Nothing include file-name
--------	---------------------------

[Program action]

Ignores the instruction.

E4111A	Cannot open include file
--------	--------------------------

[Program action]

Ignores the instruction.

E4112A	Nested include file exceeds 8
--------	-------------------------------

The number of nested include files must not exceed 8.

[Program action]

Ignores the instruction.

E4114A	Nested macro-call exceeds 255
--------	-------------------------------

The number of nested macro calls must not exceed 255.

[Program action]

Ignores the instruction.

E4115A	Changed level exceeds 255
--------	---------------------------

The number of nested changed levels must not exceed 255.

[Program action]

Changes the macro names to null characters.

E4116A	Invalid value
--------	---------------

The integer constant includes illegal characters.

[Program action]

Ignores only the illegal characters.

E4117A	Macro name duplicate definition
--------	---------------------------------

[Program action]

Ignores the instruction.

W1118A	Argument duplicate definition
--------	-------------------------------

[Program action]

Ignores the argument.

W1119A	Local-symbol duplicate definition
--------	-----------------------------------

[Program action]

Ignores the local symbol.

W1120A	Too many arguments
--------	--------------------

[Program action]

Ignores the extra arguments.

W1121A	Not enough arguments
--------	----------------------

[Program action]

Creates as many null-character arguments as are required.

E4122A	Missing '('
--------	-------------

[Program action]

Ignores the instruction.

E4123A	Unterminated macro-name ')'
--------	-----------------------------

[Program action]

Ignores the instruction.

E4124A	Unterminated comment
--------	----------------------

[Program action]

Appends */ to the comment.

W1125A	Unterminated '
--------	----------------

[Program action]

Ends the character constant at the end of the line.

W1126A	Unterminated "
--------	----------------

[Program action]

Ends the character string at the end of the line.

W1127A	Unterminated '>'
--------	------------------

[Program action]

Appends > to the include file.

W1128A	Value overflow
--------	----------------

The specified number exceeds 32 bits in length.

[Program action]

Accepts only the lower 32 bits.

W1129A	Name too long
--------	---------------

The name exceeds 255 characters in length.

[Program action]

Ignores the extra characters.

W1130A	String too long
--------	-----------------

The character string exceeds 4095 characters in length.

[Program action]

Ignores the extra characters.

W1131A	Defined-string too long
--------	-------------------------

The defined character string exceeds 4095 characters in length.

[Program action]

Ignores the extra characters.

W1132A	Expanded-string too long
--------	--------------------------

The character string resulting from macro expansion exceeds 4095 characters in length.

[Program action]

Ignores the extra characters.

W1133A	Logical-lines too long
--------	------------------------

The logical lines, including continuation lines, exceeds 4095 characters in length.

[Program action]

Ignores the extra characters.

W1134A	Meaningless description
--------	-------------------------

[Program action]

Ignores the description.

E4135A	Has no #if-statement
--------	----------------------

[Program action]

Ignores the instruction.

E4136A	Has no #macro-statement
--------	-------------------------

[Program action]

Ignores the instruction.

W1137A	#endif expected
--------	-----------------

[Program action]

Assumes that a #endif statement has been written.

E4138A	#endm expected
--------	----------------

[Program action]

Assumes that a #endm statement has been written.

E4140A	Not used macro-statement
--------	--------------------------

[Program action]

Ignores the instruction.

E4141A	Too long macro-body
--------	---------------------

[Program action]

Ignores the macro definition.

W1142A	Meaningless .CASE
--------	-------------------

[Program action]

Ignores the .CASE instruction.

W1143A	Meaningless .DEFAULT
--------	----------------------

[Program action]

Ignores the .DEFAULT instruction.

W1144A	.CASE statement not permitted here
--------	------------------------------------

[Program action]

Ignores the .CASE instruction.

W1145A	.ENDSW without .CASE statement
--------	--------------------------------

[Program action]

Continues processing assuming that the .CASE instruction has not been written.

W1146A	#PURGE not permitted here
--------	---------------------------

[Program action]

Ignores the #PURGE instruction.

W1147A	Data width is not permitted to indicate to last item. Ignored this suffix
--------	---

[Program action]

Ignores the specification.

E4148A	.BREAK not in structured block
--------	--------------------------------

[Program action]

Ignores the specification.

E4149A	.CONTINUE not in structured block
--------	-----------------------------------

[Program action]

Ignores the specification.

E4150A	Missing angle bracket
--------	-----------------------

[Program action]

Ignores the specification.

E4151A	Conditional expression overflow
--------	---------------------------------

[Program action]

Ignores the specification.

E4152A	&& and conditional expression exist
--------	--

[Program action]

Ignores the specification.

E4153A	Illegal structured block order
--------	--------------------------------

[Program action]

Ignores the specification.

E4154A	Source item and destination item is the same
--------	--

[Program action]

Ignores the specification.

E4155A	Number of item is overflowed
--------	------------------------------

[Program action]

Ignores the specification.

E4501A	Missing expression(s)
--------	-----------------------

[Program action]

Ignores the instruction.

E4502A	Out of section
--------	----------------

[Program action]

Creates a section defined by .SECTION CODE, CODE, ALIGN=1.

E4503A	Invalid directive (instruction-name)
--------	--------------------------------------

[Program action]

Ignores the instruction.

E4504A	Invalid word (detailed information)
--------	-------------------------------------

[Program action]

Ignores the coding up to the next delimiter.

W1505A	Name too long
--------	---------------

The identifier exceeds 255 characters in length.

[Program action]

Ignores the extra characters.

E4506A	Missing string terminator (")
--------	-------------------------------

[Program action]

Ends the character string at the end of the line.

E4507A	Expression too complex
--------	------------------------

[Program action]

Stops the expression evaluation.

E4510A	Value overflow (detailed information)
--------	---------------------------------------

The specified number exceeds 32 bits in length.

[Program action]

Accepts only the lower 32 bits.

E4511A	Missing string terminator (')
--------	-------------------------------

[Program action]

Ends the character constant at the end of the line.

E4512A	Divide by zero
--------	----------------

[Program action]

Assigns 0 to the value of the expression.

E4513A	Expression too complex
--------	------------------------

[Program action]

Stops the expression evaluation.

E4514A	Register not permitted in expression (register-name)
--------	--

[Program action]

Stops the expression evaluation.

E4515A	No terms in parentheses
--------	-------------------------

[Program action]

Stops the expression evaluation.

E4516A	Illegal term in expression (detailed information)
--------	---

[Program action]

Stops the expression evaluation.

E4517A	Unbalanced parentheses in expression
--------	--------------------------------------

[Program action]

Stops the expression evaluation.

E4518A	Cannot out this operator (detailed information)
--------	---

[Program action]

Assigns 0 to the value of the expression.

E4519A	Register list symbol not permitted in expression (detailed information)
--------	---

[Program action]

Stops the expression evaluation.

E4520A	String too long
--------	-----------------

The character string exceeds 4095 characters in length.

[Program action]

Ignores the extra characters.

E4521A	Structured definitions. Invalid directive
--------	---

[Program action]

Ignores the pseudo-instruction.

E4522A	Structured definitions. Invalid instruction
--------	---

[Program action]

Ignores the machine instruction.

E4524A	Duplicate declaration (symbol-name)
--------	-------------------------------------

The symbol has already been declared by a .GLOBAL, .EXPORT, or .IMPORT instruction.

[Program action]

Ignores this declaration.

E4525A	Duplicate definition (symbol-name or section-name)
--------	--

[Program action]

Ignores the symbol definition.

E4526A	Cannot declare (symbol-name)
--------	------------------------------

A symbol cannot be declared by a .GLOBAL, .EXPORT, or .IMPORT instruction.

[Program action]

Ignores the declaration.

W1527A	Undefined symbol:treats as an external reference symbol (symbol-name)
--------	---

[Program action]

Treats the symbol as an external reference symbol.

E4528A	Terms other than the section symbol are described in the size operator (symbol-name)
--------	--

[Program action]

Stops the expression evaluation.

E4529A	External reference symbol is described in the size operator: Please make and deal with an empty section by the name (symbol-name)
--------	---

[Program action]

Stops the expression evaluation.

E4530A	Invalid symbol field (detailed information)
--------	---

[Program action]

Ignores the specification up to the end of the line.

E4531A	Not an absolute expression
--------	----------------------------

The expression includes an external reference symbol or absolute symbol.

[Program action]

Ignores the instruction.

E4532A	Not complex relocatable expression
--------	------------------------------------

The expression includes multiple external reference symbols or absolute symbols.

[Program action]

Ignores the instruction.

E4533A	Forward reference symbol is described in expression
--------	---

A forward reference symbol cannot be used in an expression in an instruction.

[Program action]

Ignores the instruction.

E4534A	Syntax error in operand (detailed information)
--------	--

[Program action]

Ignores the instruction.

W1535A	Meaningless description (detailed information)
--------	--

[Program action]

Ignores the meaningless specification.

E4536A	Duplicate directive (detailed information)
--------	--

[Program action]

Uses the first specified instruction.

E4537A	Reserved word cannot define symbol (detailed information)
--------	---

[Program action]

Ignores the specification up to the end of the line.

E4538A	Reserved word cannot be used as a section name (detailed information)
--------	---

[Program action]

Ignores the instruction.

E4539A	Size operator is described in expression
--------	--

A size operator cannot be used in an expression in an instruction.

[Program action]

Ignores the instruction.

E4540A	Conflicting section attribute (parameter-name)
--------	--

[Program action]

Ignores only the parameter name.

W1541A	Value out of range
--------	--------------------

[Program action]

The program masks an operational result (value) of an equation described in the operand, in accordance with the operand size.

See section "7.11 Expressions" for details.

The assembler outputs object files.

[Supplementary explanation]

When a list file output specification option (-l) is specified, the assembler outputs a list file.

The operand code generate the list file is a value obtained by masking the operation result in accordance with the operand size.

This message appears when the result of the operational result of the equation described in the operand exceeds that operand size, when the -OVFW is specified.

See section "7.11.2 Range of Operand Values" for details.

E4541A	Value out of range
--------	--------------------

[Program action]

The program masks an operational result (value) of an equation described in the operand, in accordance with the operand size.

See section "7.11 Expressions" for details.

The assembler does not outputs object files.

[Supplementary explanation]

When a list file output specification option (-l) is specified, the assembler outputs a list file.

The operand code output to the list file is a value obtained by masking the operation result in accordance with the operand size.

This message appears when the result of the operational result of the equation described in the operand exceeds that operand size, when the -XOVFW is specified.

E4542A	Invalid keyword (keyword-name)
--------	--------------------------------

[Program action]

Ignores only the specified keyword.

E4543A	Invalid kind of register
--------	--------------------------

[Program action]

Ignores the instruction.

E4544A	Invalid register list (detailed information)
--------	--

[Program action]

Ignores the instruction.

W1545A	Meaningless symbol field (detailed information)
--------	---

[Program action]

Ignores the symbol field.

E4548A	Missing keyword
--------	-----------------

[Program action]

Ignores the instruction.

E4550A	Location counter overflow
--------	---------------------------

[Program action]

Continues processing.

W1551A	Missing .END directive
--------	------------------------

[Program action]

Assembles the program up to the end of the file.

E4552A	Invalid value
--------	---------------

[Program action]

Ignores the instruction.

E4553A	Missing (=) behind keyword (keyword-name)
--------	---

[Program action]

Ignores only the specified keyword.

E4554A	Duplicate keyword (keyword-name)
--------	----------------------------------

[Program action]

Uses the first specified keyword.

W1555A	Too long module-name
--------	----------------------

The module name exceeds 255 characters in length.

[Program action]

Ignores the extra characters.

E4556A	Missing symbol field
--------	----------------------

[Program action]

Ignores the instruction.

E4557A	Starting address out of section
--------	---------------------------------

[Program action]

Ignores the starting address.

E1558A	Starting address not in code section
--------	--------------------------------------

[Program action]

Ignores the starting address for dummy sections. For other sections, the assembler sets the starting address.

E4559A	Conflicting size-suffix (detailed information)
--------	--

The operand size is different from the operation size.

[Program action]

Uses the operation size.

E4561A	Floating value underflow (detailed information)
--------	---

[Program action]

Sets the floating-point constant to +0.

E4562A	Floating value overflow (detailed information)
--------	--

[Program action]

Sets the floating-point constant to the maximum value that can be represented with the precision, and that has the specified sign.

E4565A	Not a power of 2
--------	------------------

[Program action]

Accepts the specification.

W1566A	Bigger than alignment size of .SECTION directive
--------	--

[Program action]

Accepts the specified value.

E4567A	Not enough operands
--------	---------------------

[Program action]

Ignores the instruction.

W1568A	Invalid word in module name
--------	-----------------------------

The module name is invalid.

[Program action]

Replaces each of the illegal characters with an underscore (_).

E4570A	Invalid section name (section-name)
--------	-------------------------------------

[Program action]

Ignores the instruction.

E4571A	Smaller value than beginning address set with LOCATE value of .SECTION directive cannot be set
--------	--

[Program action]

Ignores the instruction.

E4572A	Bigger value than boundary value set with ALIGN value of .SECTION directive cannot be set
--------	---

[Program action]

Ignores the instruction.

E4573A	Has no statement
--------	------------------

[Program action]

Ignores the instruction.

W1574A	It cannot be guaranteed to arrange in a proper boundary. :Please set ALIGN value of .SECTION directive in 2 or more
--------	---

[Program action]

Ignores the instruction.

W1575A	It cannot be guaranteed to arrange in a proper boundary. :Please set ALIGN value of .SECTION directive in 4 or more
--------	---

[Program action]

Ignores the instruction.

W1576A	It cannot be guaranteed to arrange in a proper boundary. :Please set ALIGN value of .SECTION directive in 8 or more
--------	---

[Program action]

Ignores the instruction.

E4600A	Invalid operation mnemonic (instruction-name)
--------	---

[Program action]

Ignores the instruction and creates a NOP instruction.

E4605A	Invalid Operation-suffix
--------	--------------------------

[Program action]

Ignores the subsequent operation fields.

E4606A	Invalid Option-suffix in operation field (detailed information)
--------	---

[Program action]

Ignores only the operation option.

W1608A	Conflicting Option-suffix in operation field (detailed information)
--------	---

[Program action]

Uses the first specified operation option.

E4616A	Not enough operands
--------	---------------------

[Program action]

Ignores the instruction and creates a NOP instruction.

E4617A	Too many operands (detailed information)
--------	--

[Program action]

Ignores the extra operands.

E4619A	Unbalanced parentheses (in operand (number))
--------	--

[Program action]

Ignores the instruction and creates a NOP instruction.

E4622A	Format-suffix not permitted (detailed information)
--------	--

[Program action]

Ignores the specified format.

E4625A	Invalid Size-suffix (character-string)
--------	--

[Program action]

Ignores the specified size.

E4626A	Size-suffix not permitted
--------	---------------------------

[Program action]

Ignores the specified size.

E4629A	Syntax error (operand-number, detailed information)
--------	---

[Program action]

Ignores the instruction and creates a NOP instruction.

E4639A	Addressing mode not permitted (in operand (number))
--------	---

[Program action]

Ignores the instruction and creates a NOP instruction.

E4651A	Floating point data too short (in operand (number))
--------	---

[Program action]

Adds as many 0s as required to the data.

E4652A	Floating point data too long (in operand (number))
--------	--

[Program action]

Ignores the extra data.

E4654A	Illegal instruction start address (detailed information)
--------	--

The instruction start address is an odd address.

[Program action]

Changes the instruction start address to an even address.

Note: Specify an even address as an instruction start address.

E4656A	Nothing operand (detailed information)
--------	--

[Program action]

Ignores the instruction and creates a NOP instruction.

E4698A	Number of include files exceeds 32766
--------	---------------------------------------

[Program action]

Stops assembly.

E4699A	Source-line exceeds 65535 lines
--------	---------------------------------

[Program action]

Resets the source line number to 0 to continue processing; however, object code for the extra source lines is not created.

W1701A	Cannot optimize for insufficient memory
--------	---

[Program action]

The assembler does not optimize normal branch instructions.

E4703A	Cannot put machine-code
--------	-------------------------

[Program action]

Continues processing.

W1704A	It is possible to optimize. :The back and forth instruction is replaced.
--------	--

[Program action]

Changes the sequence of this and the preceding instructions.

W1705A	It is possible to optimize. :Changes to another instruction.
--------	--

[Program action]

Changes the instruction to the optimum instruction.

W1706A	It is possible to optimize. :The instruction is deleted.
--------	--

[Program action]

Deletes the instruction.

W1707A	It is possible to optimize. :The instruction is newly generated.
--------	--

[Program action]

Creates a new instruction.

W1710A	Instruction is written after .END
--------	-----------------------------------

[Program action]

Ignores the code following the .END instruction.

W1711A	Location address backed by .ORG
--------	---------------------------------

[Program action]

Continues processing with the specified address.

W1712A	This structure field cannot initialized
--------	---

[Program action]

Ignores the specification.

W1713A	Too many initialize data
--------	--------------------------

[Program action]

Ignores the extra initialization data.

W1714A	Section type error
--------	--------------------

[Program action]

Ignores the symbol attribute and creates an instruction.

E4715A	Section type error
--------	--------------------

[Program action]

Ignores the specification.

E4716A	.ENDS expected
--------	----------------

[Program action]

Terminates abnormally.

W1717A	Mismatch .ENDS
--------	----------------

[Program action]

Ignores the specification.

E4750A	Terms other than the bit symbol are described in the bit operator
--------	---

[Program action]

Ignores the specification.

E4756A	Invalid address modifier
--------	--------------------------

[Program action]

Ignores the specification.

E4758A	Input to @A can not be written in this expression
--------	---

[Program action]

Ignores the specification.

E4759A	@EP and EP or @IX and IX exist in this expression
--------	---

[Program action]

Ignores the specification.

W1800A	Not 2 bytes attribute
--------	-----------------------

[Program action]

Sets the lowest bit of the value to 0.

W1801A	Not 4 bytes attribute
--------	-----------------------

[Program action]

Sets the lower two bits of the value to 0.

E4802A	Invalid register
--------	------------------

[Program action]

Ignores the instruction and creates a NOP instruction.

E4803A	Fixed point of double precision is not supported
--------	--

[Program action]

Ignores the instruction and creates a NOP instruction.

W1806A	DIVU is detected
--------	------------------

[Program action]

Continues processing.

[Supplementary explanation]

The operation of F²MC-8FX family by a DIVU instruction is different from that of the conventional F²MC-8L family. Therefore, when using the F²MC-8L family assembler source in the F²MC-8FX family, make sure that there are no problems in the area where the DIVU instruction is used. This message appears when the DIVU instruction is detected in the assembler source when the -div_check option is specified. If you do not need the message, specify the -Xdiv_check option. For details on the -div_check option, see section "4.7.2 -div_check, -Xdiv_check".

F9860A	CPU information not found (CPU-name)
--------	--------------------------------------

[Program action]

Stop the Assembly process.

Note: CPU information specified by the -cpu option is not registered in the CPU information file.

Check the CPU MB number specified by the -cpu option again.

If there is no mistake in the specification, contact Fujitsu Limited.

F9861A	Mismatch CPU information file version
--------	---------------------------------------

[Program action]

Stop the Assembly process.

Note: The CPU information file version that was read is old, and then the information required by this Assembler is not included.

Re-install the Assembler Pack.

F9901A	Insufficient memory (error-ID)
--------	--------------------------------

[Program action]

Terminates abnormally.

Note: Increase memory capacity.

F9902A	Internal error (error-ID)
--------	---------------------------

The error is due to a contradiction in the internal processing of the assembler.

[Program action]

Terminates abnormally.

Note: Make a note of the error ID, and contact our development section.

F9903A	File Write Error (file-type)
--------	------------------------------

[Program action]

Terminates abnormally.

Note: Check that the disk capacity is sufficient, and that the file is not write-protected.

F9904A	File read error (file-type)
--------	-----------------------------

Some of the source files or work files cannot be read.

[Program action]

Terminates abnormally.

Note: Check that no source files or work files have been forcibly deleted during assembly.

F9905A	Cannot open message file
--------	--------------------------

[Program action]

Terminates abnormally.

Note: Set an environmental variable, or check for a message file.

F9907A	Structured nest level overflow
--------	--------------------------------

[Program action]

Terminates abnormally.

F9951A	Source filename not specified
--------	-------------------------------

[Program action]

Terminates abnormally.

F9952A	Cannot open file (file-name)
--------	------------------------------

[Program action]

Terminates abnormally.

F9953A	Invalid option name (option-name)
--------	-----------------------------------

[Program action]

Terminates abnormally.

F9954A	Invalid value (option-name)
--------	-----------------------------

[Program action]

Terminates abnormally.

F9955A	Invalid sub-option name (option-name)
--------	---------------------------------------

[Program action]

Terminates abnormally.

F9956A	Invalid option description (option-name)
--------	--

[Program action]

Terminates abnormally.

F9959A	Nested option file exceeds 8 (detailed information)
--------	---

[Program action]

Terminates abnormally.

F9960A	Too many file
--------	---------------

Multiple files cannot be assembled.

[Program action]

Terminates abnormally.

F9961A	-cpu option not specified
--------	---------------------------

[Program action]

Terminates abnormally.

APPENDIX B Restrictions

The following restrictions apply when the assembler is used.

■ Restrictions Related to Preprocessor Operation

- The number of nested include files must not exceed 8.
- The size of a macro body must not exceed 64512 bytes.
- The number of nested macro calls must not exceed 256.
- The number of dummy arguments specified in a macro definition must not exceed 32767.
- An unlimited number of macro names can be registered (The maximum number depends on the size of the available memory area).
- The number of nested structured control statements must not exceed 40.
- The number of conditional expressions in a compound conditional expression of a structured control statement must not exceed 8.
- The number of substitution destination items specified in a expression with which group substitution of a structured instruction is described must not exceed 8.

■ Restrictions Related to Assembly

- The number of lines in a source file to be assembled must not exceed 65535.
- An unlimited number of symbols can be registered (The maximum number depends on the size of the available memory area).
- The number of characters allowed on one line, including continuation lines, must not exceed 4095.
- The value of expression-1 in .DATAB and .FDATAB instructions must not exceed 1048575.

■ Restrictions on Option Files

- The number of characters contained on one line in an option file must not exceed 4095.
- The number of nested option files must not exceed 8.

■ Other Restrictions

- The main file name of a source file must not be longer than 248 characters.
- No two-byte characters can be used in a module name.

APPENDIX C The Acquisition Method of an Extended Direct Access Area Bank Number

The acquisition method of the bank number of the extended direct access area which can be used with F²MC-8FX family is explained.

■ The Acquisition Method of an Extended Direct Access Area Bank Number

In order to acquire an extended direct access area bank number from the symbol defined as the extended direct access area, please use the broad view defined by the following header files.

- Header file: %FETOOL%\lib\896\include\extdir.inc
- Macro name: EDIRBANK()

[example]

```
#include      <extdir.h >
:
.IMPORT      DIR0_a
:
.SECTION      CODE, CODE, ALIGN=1
SETDB      #EDIRBANK(DIR0_a)
MOV      A, X:DIR0_a
:
```

APPENDIX D Difference in Specification of SOFTUNE Assembler (FASM896S) and Old Assembler (ASM96)

The difference in the specification of a SOFTUNE assembler (FASM896S) and an old edition assembler (ASM96) is explained.

- D.1 Assembler Language Basic Rules
- D.2 Expression Processing
- D.3 Pseudo Instruction
- D.4 Macro Processing
- D.5 Structured Control Instruction
- D.6 Machine Instruction
- D.7 Difference in Command Line
- D.8 Environmental Variables
- D.9 Options
- D.10 Alleviation of Restrictions

D.1 Assembler Language Basic Rules

This section explains the difference in the source line syntax for FASM896S and ASM96.

■ Statement Format

The assembler 1 line format is as shown below.

- Input for each field omitted when not required
- Each field delimited by one or more spaces, or one tab
- Difference in comment field and continuation display field

[FASM896S]

Symbol field	Operation field	Operand field	Comment field	Continuation display field
--------------	-----------------	---------------	---------------	----------------------------

The max. length of one line is 4095 characters.

[ASM96]

Symbol field	Operation field	Operand field	Comment field
--------------	-----------------	---------------	---------------

The max. length of one line is 127 characters.

■ Comment Field

FASM896S	ASM96
A comment begins with a semicolon (;) or 2 slashes (//). The comment must continue until the end of this line.	A comment begins with a semicolon (;). The comment must continue until the end of this line.
Like in C, a comment enclosed by /* */ can be input.	

■ Continuation Display Field

FASM896S	ASM96
It is possible to continue this line by entering a backslash (\) immediately before a line feed.	No continuation lines can be specified.

■ Character Set

The following characters can be used at source input.

In addition, all characters including Kanji code can be entered in the comment field.

Type	Character
Alphabetic character (uppercase letter)	A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z
Alphabetic character (lowercase letter)	a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z
Numeric character	0,1,2,3,4,5,6,7,8,9
Special character	+ - * / () ; : ' " , _ . @ # \$! % & < > = \ space tab

The special characters "~" and "^" can be used for FASM896S, but not for ASM96.

■ Name (Symbol, Label)

When defining / inputting a name, use the following definition.

	FASM896S	ASM96
Name rule	Input begins with an alphabetic character or an underscore (_). Subsequently, alphabetic characters, numerical characters, and underscores are used.	
Maximum character string length	255 characters	31 characters
Case sensitive enabled/ disabled	Enabled	Can be selected using optional specification

■ Input of Location Counter Symbol

	FASM896S	ASM96
Character used	\$	*

■ Input of Integer Constant

	FASM896S	ASM96
Binary input	B'nnnnnnnn 0nnnnnnnnB 0Bnnnnnnnn	B'nnnnnnnn 0nnnnnnnnB
Octal input	Q'nnnnnnnn nnnnnnnnQ 0nnnnnnnn	Q'nnnnnnnn nnnnnnnnQ
Decimal input	D'nnnnnnnn nnnnnnnnD nnnnnnnn	D'nnnnnnnn nnnnnnnnD nnnnnnnn
Hexadecimal input	H'nnnnnnnn nnnnnnnnH 0Xnnnnnnnn	H'nnnnnnnn nnnnnnnnH

■ Character Constant

	FASM896S	ASM96
Input	Character enclosed in single quotes	Character enclosed in double quotes
Maximum character length	4	2
Control code input	Allowed	Not allowed
Extended notation	Special character can be specified immediately after a backslash (\) symbol.	To enter a double quote, enter it twice in a row.
Others	C-type control character can be specified (\n, \t, etc.)	Numerical information cannot be input.

■ Character String Constant

	FASM896S	ASM96
Input	Character are enclosed in double quotes.	
Control code input	Allowed	Not allowed
Extended notation	Special characters can be specified immediately after a backslash (\) symbol.	To enter a double quote, enter it twice in a row.
Others	C-type control characters can be specified.	Numerical information cannot be input.

■ Floating-Point Constants

	FASM896S	ASM96
Input format	[Or][+ -]{.d d[.d]}[e[+ -]d] [F'][+ -]{d d[.d]}[e[+ -]d] H'h d:decimal 0xh h:hexadecimal	Not allowed
Input example	0r-1.34e-20 F'-2.5 -1.2e10	

D.2 Expression Processing

Some kinds of operators used in expressions and the difference in an operation priority are explained.

■ Operator List

Function	FASM896S	ASM96
Priority change	()	
Negation of logical value	!	NOT
Bit reversion	~	
Unary plus	+	
Unary minus	-	
Bit address extraction	bitadr	%
Bit position extraction	Bitpos	!
Section size extraction	sizeof	segsz
Multiplication	*	
Division	/	
Remainder calculation	%	
Addition	+	
Subtraction	-	
Left shift	<<	SHL
Right shift	>>	SHR
Less than (comparison)	<	LT
Equal to or less than (comparison)	<=	LE
Larger than (comparison)	>	GT
Equal to or larger than (comparison)	>=	GE
Equal to (comparison)	==	EQ
Unmatch (comparison)	!=	NE
Bit AND	&	AND
Bit XOR	^	XOR
Bit OR		OR
Logical AND	&&	
Logical OR		

■ Difference in Operator Priority

Function	FASM896S	ASM96
Priority change	1	1
Negation of logical value	2	2
Bit reversion		
Unary plus		2
Unary minus		
Bit address extraction		
Bit position extraction		
Section size extraction		
Multiplication	3	3
Division		
Remainder calculation		
Addition	4	4
Subtraction		
Left shift	5	5
Right shift		
Less than (comparison)	6	6
Equal to or less than (comparison)		
Larger than (comparison)		
Equal to or larger than (comparison)		
Equal to (comparison)	7	7
Unmatch (comparison)		
Bit AND	8	8
Bit XOR	9	9
Bit OR	10	10
Logical AND	11	
Logical OR	12	

■ Other Notes

Function	FASM896S	ASM96
Arithmetic precision	32-bit length with / without sign	16-bit length without sign
Sign	With / without sign	Without sign
Support of data with type	Provided	Not provided

D.3 Pseudo Instruction

The differences in pseudo instruction is shown below.

■ List of Differences in Pseudo Instruction

Table D.3-1 List of Differences in Pseudo Instruction (1 / 2)

Function	FASM896S	ASM96
Object name setting	.PROGRAM	NAME
Source input end declaration	.END	END
External reference symbol declaration	.IMPORT	EXTRN / EXTERN
External definition symbol declaration	.EXPORT	PUBLIC
Instruction area definition	.SECTION CODE (*1)	CSEG
I/O related area definition	.SECTION DIR (*1)	DIRSEG
Data area definition	.SECTION DATA (*1)	DSEG
Stack area definition	.SECTION STACK (*1)	SSEG
Area definition end	No input required	ENDS
Bit-unit data definition	.DATA.I	DBIT
Byte-unit data definition	.DATA.B or .SDATA (*2)	DB
2-byte unit data definition	.DATA.H	DW
4-byte unit data definition	.DATA.L	-
Bit-unit area reservation	.RES.I	-
Byte-unit area reservation	.RES.B	RBIT
2-byte unit area reservation	.RES.H	RB
4-byte unit area reservation	.RES.L	RW
Single-precision floating decimal data definition	.FDATA.S	-
Double-precision floating decimal data definition	.FDATA.D	-
Single-precision floating decimal area reservation	.FRES.S	-
Double-precision floating decimal area reservation	.FRES.D	-
Structure declaration	.STRUCT ~ .ENDS	STRUCT ~ ENDS
Symbol definition	.EQU	EQU
Symbol definition	-	SET

Table D.3-1 List of Differences in Pseudo Instruction (2 / 2)

Function	FASM896S	ASM96
Boundary alignment	.ALIGN	-
Location adjustment	.ORG	ORG
List format control	.LIST	LIST
List output suppression	.LIST	LISTOFF
List output specification	.LIST	LISTON
List page break specification	.PAGE	PAGE
List title specification	.HEADING	TITLE
Subtitle specification	-	SUBTTL

*1: The actual correspondence varies depending on the attributes to be added (ABS, WORD, COMMON, etc.)

*2: One is selected depending on whether character string data or numerical data is defined.

■ Definition of Section (segment)

The FASM896S section is defined according to the rule described by "10.2.3 .SECTION Instruction".

The corresponding patterns are shown below.

Function	FASM896S	ASM96
Code division definition	.SECTION section name, CODE, ALIGN=1	section name CSEG
	.SECTION section name, CODE, LOCATE=0	section name CSEG ABS
Data division definition	.SECTION section name, DATA, ALIGN=1	section name DSEG
	.SECTION section name, DATA, LOCATE=0	section name DSEG ABS
	.SECTION section name, DATA, ALIGN=1	section name DSEG BYTE
	.SECTION section name, CONST, ALIGN=1	section name DSEG ROM
	.SECTION section name, COMMON, ALIGN=1	section name DSEG COMMON
DIR division definition	.SECTION section name, DIR, ALIGN=1	section name DIRSEG
	.SECTION section name, DIR, LOCATE=0	section name DIRSEG ABS
	.SECTION section name, DIR, ALIGN=1	section name DIRSEG BYTE
	.SECTION section name, DIRCOMMON, ALIGN=1	section name DIRSEG COMMON
Stack area definition	.SECTION section name, STACK, ALIGN=1	SSEG

Note:

In ASM96, description of a section name is omissible.

■ Character String Data Definition

FASM896S	ASM96
.SDATA "ABS\n"	DB "ABC", H'0d, H'0a
.DATA.B D'00, D'05 .SDATA "ABCDE" .DATA.B D'00	DB D'00, D'05, "ABCDE", D'00

D.4 Macro Processing

The difference in macro instruction is shown below.

■ Difference in Macro Instruction

Function	FASM896S	ASM96
Macro definition start	#MACRO	&MACRO
Local symbol setting	#LOCAL	&LOCAL
Macro symbol setting	#SET	&SET
Cancels a macro symbol	#PURGE	&PURGE
Iterative macro definition	#REPEAT	&REPEAT
Conditional macro	#IF	&IF
True		&THEN
False		&ELSE
Conditional macro end	#ENDIF	&ENDIF
Macro definition end	#ENDM	&ENDM
File read	#INCLUDE	&INCLUDE

D.5 Structured Control Instruction

The differences in structured control instruction are shown below.

■ Structured Control Instruction

The structured control instructions that can be used for FASM896S and ASM96 are the same.

Input of condition code conditional expressions is also the same for FASM896S and ASM96.

The input of comparison conditional expressions has the following differences.

	FASM896S		ASM96	
	With sign	Without sign	With sign	Without sign
Equal to	==		Z(EQ)	
Not equal to	!=		NZ(NE)	
Less than	<	<.U	LT	LO
Equal to or less than	<=	<=.U	LE	LS
Larger than	>	>.U	GT	HI
Equal to or larger than	>=	>=.U	GE	

D.6 Machine Instruction

Input of machine instructions is specified by the hardware manual. Only operand field input is changed.

■ Machine Instruction

Input of machine instructions is specified by the hardware manual. Only operand field input is changed.

For the difference in operand input, see "D.1 Assembler Language Basic Rules".

D.7 Difference in Command Line

The difference in command line is shown below.

■ Assembler Activation Format

FASM896S	ASM96
fasm896S [options]... [file name]	asm96 file name [options]
	asm96 @option file

■ Option Specification for Assembler

Functions	FASM896S	ASM96
Position to specify options	Any position is allowed	After specification of file name
Specification of option file	-f option file	@option file
Specification of two or more option files	Allowed	Not allowed

D.8 Environmental Variables

The difference in environmental variables is shown below.

■ Environmental Variables

Table D.8-1 Environmental Variables

	FASM896S	ASM96
Specification of working directory	TMP	TMP
Specification of include file detection path	INC896	INC96
Specification of installation path	FETOOL	-
Specification of display character code	FELANG	-
Specification of default option file store path	OPT896	-

D.9 Options

The difference in options is shown below.

■ Options

Table D.9-1 Options (1 / 2)

	FASM896S	ASM96
Object file name specification	-o	O
Object file output suppression	-Xo	NO
Debug information output	-g/-Xg	DEB,DBG,LO,HI/NDB
Working directory specification	Environmental variables are used	WD pathname
Case sensitive enabled / disabled	Always enabled	CP/NCP
Warning message output level specification	-W[0-3]	-
Module name specification	-name module	-
List file output	-l/-Xl -lf filename	L [filename]
List 1 page line count	-pl [0 20-255]	PL [40-128]
List 1 line character count	-pw [80-255]	PW [80-136]
List page break processing suppression	-pl 0	-
Information list output	-linf [on off]	-
Source list output	-lsrc [on off]	-
Section list output	-lsec [on off]	-
Cross reference list output	-lcros [on off]	XR/NXR
Include file list output control	-linc [on off]	ICL/NCL
Macro IF statement list control	-	CD/NCD
Macro definition division list control	-	DEF/NDEF
List control for macro call and structured control instruction line	-	CL/NCL
List control for expanded division of structured control instruction	-	STR/NSTR
List output control for macro expanded division	-lexp [on off]	EXP/NEXP
Does not start preprocessor	-p	-

Table D.9-1 Options (2 / 2)

	FASM896S	ASM96
Start only preprocessor	-P/Pf filename	-
Macro name specification	-D name[=def]	-
Macro name cancel	-U name	-
Display of include path	-H	-
Outputting comments in preprocessor output	-C	-
Error information file output enable / disable	-	-[NO]EP
Status code file output	-	-RC
Help display	-help	-
Target CPU specification	-cpu	-
Default option file input suppression	-Xdof	-
Activation message display / no display	-V/-XV	-

D.10 Alleviation of Restrictions

The difference in restrictions (limited processing) is shown below.

■ Restrictions

	FASM896S	ASM96
1 line character count	4095 characters	128 characters
Source file line count	65535 characters	32000 characters
Symbol count	Unlimited	32000 symbols
Name length	255 characters	31 characters
Number of macro names that can be registered	Unlimited	Macro definition count : 512 Local symbol count : 256
Macro nest	256 levels	8 levels

INDEX

**The index follows on the next page.
This is listed in alphabetic order.**

Index

Symbols

# Operator	
Replacing Formal Macro Arguments with Character Strings (# Operator)	227
## Operator	
Concatenating the Characters to be Replaced by Macro Replacement (## Operator)	228
#define	
Parameter-attached #define Instruction	226
Parameter-less #define Instruction	225
#elif	
#elif Instruction	221
#else	
#else Instruction	220
#endif	
#endif Instruction	223
#endm	
#endm Instruction	212
#error	
#error Instruction	235
#exitm	
#exitm Instruction	211
#exitm Instruction Rules	211
#if	
#if Instruction	217
#ifdef	
#ifdef Instruction	218
#ifndef	
#ifndef Instruction	219
#include	
#include Instruction	232
#line	
#line Instruction	234
#local	
#local Instruction	210
#local Instruction Rules	210
#macro	
#macro Instruction	209
#macro Instruction Rules	209
#pragma	
#pragma Instruction	236
#purge	
#purge Instruction	231
#repeat	
#repeat Instruction	215
#repeat Instruction Rules	215
#set	
#set Instruction	229

#undef	
#undef Instruction	230
.ALIGN	
.ALIGN	82
.ALIGN Instruction	155
.ASCII	
.ASCII	89
.ASCII Instruction	173
.BIT	
.BIT	85
.BIT Instruction	165
Break	
Break Instruction	266
.BYTE	
.BYTE	85
.BYTE Instruction	166
.continue	
.continue Instruction	266
.DATA	
.DATA	85
.DATA Instruction	165
.DATAB	
.DATAB	86
.DATAB Instruction	167
.DEBUG	
.DEBUG	90
.DEBUG Instruction	177
.DOUBLE	
.DOUBLE	87
.DOUBLE Instruction	169
.END	
.END	80
.END Instruction	151
.ENDS	
.STRUCT and .ENDS Instructions	175
.EQU	
.EQU	84
.EQU Instruction	163
.EXPORT	
.EXPORT	83
.EXPORT Instruction	159
.FDATA	
.FDATA	87
.FDATA Instruction	168
.FDATAB	
.FDATAB	87
.FDATAB Instruction	170
.FLOAT	
.FLOAT	87

.FLOAT Instruction.....	168	.RES	
.FORM		.RES.....	88
.FORM	92	.RES Instruction	171
.FORM Instruction.....	180	.SDATA	
.FRES		.SDATA	89
.FRES	88	.SDATA Instruction.....	173
.FRES Instruction.....	172	.SDATAB	
.GLOBAL		.SDATAB	89
.GLOBAL	83	.SDATAB Instruction	174
.GLOBAL Instruction.....	160	.SECTION	
.HALF		.SECTION	81
.HALF	86	.SECTION Instruction.....	152
.HALF Instruction.....	166	.SKIP	
.HEADING		.SKIP	82
.HEADING	93	.SKIP Instruction	157
.HEADING Instruction.....	183	.SPACE	
.IMPORT		.SPACE	94
.IMPORT	83	.SPACE Instruction.....	188
.IMPORT Instruction.....	161	.STRUCT	
.LIBRARY		.STRUCT and .ENDS Instructions	175
.LIBRARY	91	.TITLE	
.LIBRARY Instruction	178	.TITLE	92
.LIST		.TITLE Instruction.....	182
.LIST	93	.WORD	
.LIST Instruction.....	184	.WORD	86
.LONG		.WORD Instruction.....	166
.LONG	86	_check	
.LONG Instruction	166	-div_check	49
.ORG		-Xdiv_check.....	49
.ORG	82		
.ORG Instruction.....	156	Numerics	
.PAGE		2-process Selection Syntax	
.PAGE	93	2-process Selection Syntax	259
.PAGE Instruction.....	187	8FX	
.PROGRAM		Assembler Pseudo Machine Instructions for the	
.PROGRAM.....	80	F ² MC-8L/8FX Family	242
.PROGRAM Instruction	150		

INDEX

A	
Absolute Values	
Absolute Values	117
Acquisition Method	
The Acquisition Method of an Extended Direct Access	
Area Bank Number	303
Activation Format	
Assembler Activation Format	318
Address Control Instructions	
Address Control Instructions	154
Addressing Specifiers	
Addressing Specifiers	118
Allocation Patterns	
Section Allocation Patterns	131, 136
Area Definition Instructions	
Area Definition Instructions	164
Arguments	
Formal Arguments	194
Replacing Formal Macro Arguments with Character	
Strings (# Operator)	227
Arithmetic Operators	
Arithmetic Operators	123
Assembler	
Assembler Activation Format	318
Option Specification for Assembler	318
Overview (Assembler Syntax)	5
Overview (SOFTUNE Assembler)	4
Assembler Pseudo Machine Instructions	
Assembler Pseudo Machine Instructions for the	
F ² MC-8L/8FX Family	242
Assembly	
Assembly Phase	4
Restrictions Related to Assembly	302
Assignment	
Assignment Expression	270
Operation and Assignment Expression	272
At-end Condition Repetition Syntax	
At-end Condition Repetition Syntax	264
B	
Backward Reference Symbols	
Forward Reference Symbols and Backward Reference	
Symbols	105
Bank Number	
The Acquisition Method of an Extended Direct Access	
Area Bank Number	303
Binary	
Binary Constants	106
Bit Addresses	
Bit Addresses	118
Bit Position	
Bit Position Number Operator	
(BITPOS Operator)	124
Bit Symbol Address Operator	
Bit Symbol Address Operator	
(BITADR Operetor)	124
BITADR	
Bit Symbol Address Operator	
(BITADR Operetor)	124
BITPOS	
Bit Position Number Operator	
(BITPOS Operator)	124
Bitwise Operators	
Bitwise Operators	122
Boolearn	
Boolearn Values	122
Branch Instructions	
Optimization of Branch Instructions	66
Branch Pseudo Machine Instructions	
Branch Pseudo Machine Instructions	243
C	
-C	
-C	46
C Preprocessor	
Differences from the C Preprocessor	240
Character	
Character Constant	307
Character Constant Elements	199
Character Constants	108, 193, 199
Character Set	
Character Set	102, 306
Character String	
Character String Constant	308
Character String Data Definition	314
-cif	
-cif	64
-cmsg	
-cmsg	56
Command	
fasm896s Command Lines	16
Comment	
Comment Field	101, 305
Comments	128, 193, 196
Comments Described in an Option File	22
Composition	
Composition	70
Computed Repetition Syntax	
Computed Repetition Syntax	262
Concatenated Linkage	
Concatenated Linkage	139
Conditional Assembly Instructions	
Conditional Assembly Instructions	216
Conditional Expressions	
Conditional Expressions in a Structured Program	
Instruction	254

- Format for Conditional Expressions 258
- Constant**
 - Binary Constants..... 106
 - Character Constant 307
 - Character Constant Elements..... 199
 - Character Constants 108, 193, 199
 - Character String Constant 308
 - Data Format of Double-precision Floating-point Constants 113
 - Decimal Constants 106
 - Floating-Point Constants..... 308
 - Hexadecimal Constants 106
 - Input of Integer Constant 307
 - Integer Constants 106, 193, 198
 - Notation for Floating-point Constants 111
 - Octal Constants..... 106
 - Range of the Representable Floating-point Constants 114
 - Scope of Integer Constants Handled by Pseudo-instructions..... 148
 - Data Format of Single-precision Floating-point Constants 113
- Continuation**
 - Continuation Field 101
 - Continuation of a Line 193
- Continuation Display**
 - Continuation Display Field 305
- cpu**
 - cpu 59
- Cross-reference**
 - Cross-reference List 96
- cwno**
 - cwno..... 57
- D**
- D**
 - D..... 43
- Data Format**
 - Data Format of Double-precision Floating-point Constants 113
 - Data Format of Single-precision Floating-point Constants 113
- Data Size**
 - Default Data Size 270, 273, 275
- Debugging**
 - Options Related to Objects and Debugging 29, 30
- Debugging Information Output Control Instruction**
 - Debugging Information Output Control Instruction 177
- Decimal**
 - Decimal Constants 106
- Decrement Expressions**
 - Increment/Decrement Expressions 275
- Default**
 - Default Data Size.....273, 275
- Default Operation**
 - About Default Operation49
- Definition**
 - Character String Data Definition314
 - Macro Definition Rules208
 - Macro Definitions.....208
 - Structure Area Definition175
- Development Environment**
 - Directory Structure of the Development Environment13
- Direct Access Area**
 - The Acquisition Method of an Extended Direct Access Area Bank Number303
- Directory Structure**
 - Directory Structure of the Development Environment13
- div**
 - div_check49
- DIVU Instruction**
 - About a DIVU Instruction Check Function.....49
- Double Precision**
 - Data Format of Double-precision Floating-point Constants113
 - Specification of Single or Double Precision.....112
- E**
- Elements**
 - Character Constant Elements199
- Environmental Variables**
 - Environmental Variables319
- Error**
 - Error Display78
- Error Messages**
 - Error Messages.....278
 - Format of Error Messages278
- Error Number**
 - Error Number W1551A.....53
 - Error Number W1711A.....53
- Evaluation Precision**
 - Preprocessor Expression Evaluation Precision205
- Execution Condition Repetition Syntax**
 - Execution Condition Repetition Syntax265
- Expression**
 - Assignment Expression270
 - Format for Expressions269
 - Increment/Decrement Expressions275
 - Operation and Assignment Expression.....272
 - Overview of Expressions268
- Expression Syntax**
 - Expression Syntax115

INDEX

Expression Types
 Expression Types 116

Extended Direct Access Area
 The Acquisition Method of an Extended Direct Access
 Area Bank Number..... 303

External Reference
 External Reference Values 117

F

-f 52

F²MC
 Assembler Pseudo Machine Instructions for the
 F²MC-8L/8FX Family 242

fasm896
 fasm896s Command Lines 16

FELANG
 FELANG..... 9

FETOOL
 FETOOL 8

Field
 Comment Field 101, 305
 Continuation Display Field..... 305
 Continuation Field..... 101
 Operand Field 101
 Operation Field 100
 Symbol Field 100

File
 Specifying a File 17

File Name
 Format for Specifying a File Name 18, 19
 Specifying a File Name with Components Omitted
 18, 20

File Search
 File Search for Format 1 232
 File Search for Formats 2 and 3 233

Floating-Point Constants
 Floating-Point Constants 308

Floating-point Constants
 Data Format of Double-precision Floating-point
 Constants..... 113
 Data Format of Single-precision Floating-point
 Constants..... 113
 Notation for Floating-point Constants..... 111
 Range of the Representable Floating-point Constants
 114

Formal Argument
 Formal Argument Naming Rules 203
 Formal Argument Replacement Rules 203
 Formal Arguments..... 194

Formal Macro Arguments
 Replacing Formal Macro Arguments with Character
 Strings (# Operator)..... 227

Format
 Assembler Activation Format..... 318
 File Search for Formats 2 and 3 233
 Format for Conditional Expressions 258
 Format for Specifying a File Name 18, 19
 Format for Structured Program Instructions 258
 Header Format..... 73
 Machine Instruction Format 144
 Operand Field Format 145
 Preprocessor Instruction Format 193, 195
 Section Description Format..... 130
 Section-location-format 153
 Statement Format..... 100, 305

Format 1
 File Search for Format 1 232

Format 2
 File Search for Formats 2 and 3 233

Format 3
 File Search for Formats 2 and 3 233

Forward Reference Symbols
 Forward Reference Symbols and Backward Reference
 Symbols..... 105

G

-g 32

H

-H 45

Header Format
 Header Format..... 73

-help
 -help 58

Hexadecimal
 Hexadecimal Constants 106

I

-I 44

INC896
 INC896..... 11

Include File
 Include File 79

Increment
 Increment/Decrement Expressions 275

Information
 Information List..... 72, 74

Initial-value
 Transfer of Initial-value Data 141

Instruction
 #elif Instruction 221
 #else Instruction..... 220

#endif Instruction	223	About a DIVU Instruction Check Function.....	49
#endm Instruction	212	Area Definition Instructions.....	164
#error Instruction	235	Assembler Pseudo Machine Instructions for the F ² MC-8L/8FX Family	242
#exitm Instruction	211	Branch Pseudo Machine Instructions	243
#exitm Instruction Rules	211	Conditional Assembly Instructions	216
#if Instruction	217	Conditional Expressions in a Structured Program Instruction.....	254
#ifdef Instruction	218	Debugging Information Output Control Instruction	177
#ifndef Instruction.....	219	Diffierence in Macro Instruction	315
#include Instruction.....	232	Format for Structured Program Instructions.....	258
#line Instruction	234	Label Definition Instructions	162
#local Instruction	210	Library File Specification Instruction	178
#local Instruction Rules	210	List of Differences in Pseudo Instruction.....	312
#macro Instruction	209	List Output Control Instructions	179
#macro Instruction Rules	209	Machine Instruction	317
#pragma Instruction	236	Macro Call Instruction	213
#purge Instruction	231	Macro Call Instruction Rules	213
#repeat Instruction	215	Miscellaneous Pseudo Machine Instructions.....	245
#repeat Instruction Rules	215	No-operation Instruction	237
#set Instruction	229	Operation Pseudo Machine Instructions	244
#undef Instruction	230	Optimization of Branch Instructions	66
.ALIGN Instruction	155	Optimum Allocation Branch Pseudo Machine Instructions	247
.ASCII Instruction	173	Overview of Structured Instructions	252
.BIT Instruction	165	Overview of Structured Program Instructions	253
.Break Instruction.....	266	Parameter-attached #define Instruction	226
.BYTE Instruction.....	166	Parameter-less #define Instruction.....	225
.continue Instruction.....	266	Preprocessor Instruction Format.....	193, 195
.DATA Instruction	165	Program Linkage Instructions	158
.DATAB Instruction.....	167	Program Structure Definition Instructions	149
.DEBUG Instruction.....	177	Pseudo-instructions for which an Expression Containing the Size Operator cannot be Specified.....	126
.DOUBLE Instruction.....	169	Scope of Integer Constants Handled by Pseudo- instructions	148
.END Instruction.....	151	Structured Control Instruction.....	316
.EQU Instruction.....	163	Integer	
.EXPORT Instruction	159	Input of Integer Constant	307
.FDATA Instruction	168	Integer Constants	106, 193, 198
.FDATAB Instruction.....	170	Integer Constants	
.FLOAT Instruction.....	168	Scope of Integer Constants Handled by Pseudo- instructions	148
.FORM Instruction	180	K	
.FRES Instruction.....	172	-kanji	
.GLOBAL Instruction.....	160	-kanji.....	60
.HALF Instruction.....	166	L	
.HEADING Instruction.....	183	-l	
.IMPORT Instruction.....	161	-l.....	34
.LIBRARY Instruction	178	Label	
.LIST Instruction.....	184	Name (Symbol,Label)	306
.LONG Instruction	166		
.ORG Instruction.....	156		
.PAGE Instruction.....	187		
.PROGRAM Instruction	150		
.RES Instruction.....	171		
.SDATA Instruction	173		
.SDATAB Instruction.....	174		
.SECTION Instruction	152		
.SKIP Instruction	157		
.SPACE Instruction	188		
.STRUCT and .ENDS Instructions.....	175		
.TITLE Instruction	182		
.WORD Instruction	166		

INDEX

Label Definition Instructions	
Label Definition Instructions	162
-LBA	
-LBA	61
-lcros	
-lcros	37
-lexp	
-lexp	38
-lf	
-lf	34
Library File Specification Instruction	
Library File Specification Instruction	178
-linc	
-linc	38
Line	
Continuation of a Line	197
-linf	
-linf	36
Linkage	
Concatenated Linkage	139
Shared Linkage	139
Linkage Methods	
Section Linkage Methods	138
List Output Control Instructions	
List Output Control Instructions	179
Listing	
Options Related to Listing	29, 33
Local Symbol	
Local Symbol Naming Rules	204
Local Symbol Replacement Rules	204
Local Symbols	194
Location	
Section-location-format	153
Location Counter	
Input of Location Counter Symbol	306
Location Counter Symbols	107
Logical Operators	
Logical Operators	122
-lsec	
-lsec	37
-lsrc	
-lsrc	37

M

Machine Instruction	
Machine Instruction	317
Machine Instruction Format	144
Macro	
Defined Macro Name	239
Defined Macro Names	238
Macro Definition Rules	208
Macro Definitions	208
Macro Name Replacement	224

Macro Name Rules	202
Macro Name Types	202
Macro Names	193
Macro Replacement Rules	224
Replacing Formal Macro Arguments with Character Strings (# Operator)	227

Macro Call Instruction	
Macro Call Instruction	213
Macro Call Instruction Rules	213

Macro Instruction	
Difference in Macro Instruction	315

Macro Replacement	
Concatenating the Characters to be Replaced by Macro Replacement (## Operator)	228

Miscellaneous Pseudo Machine Instructions	
Miscellaneous Pseudo Machine Instructions	245

Module	
Obtaining the Size of a Section in another Module	126

Multiple Descriptions	
Multiple Descriptions of a Section	140

Multiple-process Selection Syntax	
Multiple-process Selection Syntax	260

N

Name	
Name (Symbol,Label)	306
Name Classification	103
Operators for Calculating Values from Names	123

-name	
-name	54

Naming Rules	
Formal Argument Naming Rules	203
Naming Rules	103
Local Symbol Naming Rules	204

No-operation Instruction	
No-operation Instruction	237

Number Operator	
Bit Position Number Operator (BITPOS Operator)	124

O

-o	
-o	31

Objects	
Options Related to Objects and Debugging	29, 30

Octal	
Octal Constants	106

Operand	
Operand Field	101
Operand Field Format	145

Order of Operands.....	145	Default Option File.....	23
Range of Operand Values	121	Option File.....	21
Operation		Restrictions on Option Files.....	302
Operation and Assignment Expression	272	Overview	
Operation Field.....	100	Overview (Assembler Syntax)	5
Operation Pseudo Machine Instructions		Overview (SOFTUNE Assembler)	4
Operation Pseudo Machine Instructions	244	-OVFW	
Operator		-OVFW	62
Arithmetic Operators	123	P	
Bit Position Number Operator		-P	
(BITPOS Operator)	124	-P	42
Bit Symbol Address Operator		-p	
(BITADR Operetor)	124	-p	41
Bitwise Operators	122	Parameter-attached #define	
Logical Operators	122	Parameter-attached #define Instruction	226
Operators for Calculating Values from Names		Parameter-less #define	
.....	123	Parameter-less #define Instruction.....	225
Precedence of Operators	127	-Pf	
Preprocessor Operator Precedence	207	-Pf.....	42
Preprocessor Operators.....	206	Phase	
Pseudo-instructions for which an Expression		Assembly Phase	4
Containing the Size Operator cannot be		Preprocessor Phase	4
Specified	126	-pl	
Relational Operators.....	122	-pl	35
Replacing Formal Macro Arguments with Character		Precedence	
Strings (# Operator).....	227	Preprocessor Operator Precedence.....	207
Size Operator.....	105	Precision	
Operator List		Precision in Operations on Expressions.....	116
Operator List	309	Preprocessor Expression Evaluation Precision	
Operator Priority		205
Difference in Operator Priority	310	Specification of Single or Double Precision.....	112
OPT896		Preprocessor	
OPT896	12	Differences from the C Preprocessor	240
Optimization		Options Related to the Preprocessor	29, 40
Optimization of Branch Instructions	66	Preprocessor.....	191
Optimum Allocation Branch Pseudo Machine		Preprocessor Expression Evaluation Precision	
Instructions		205
Optimum Allocation Branch Pseudo Machine		Preprocessor Expressions	205
Instructions.....	247	Preprocessor Instruction Format.....	193, 195
Option		Preprocessor Operations	77
Option Specification for Assembler	318	Preprocessor Operator Precedence.....	207
Options	320	Preprocessor Operators	206
Options Related to Listing	29, 33	Preprocessor Phase	4
Options Related to Objects and Debugging.....	29, 30	Restrictions Related to Preprocessor Operation	
Options Related to the Preprocessor	29, 40	302
Other Options	29, 50	Priority	
Relationship with Start-time Options	71	Difference in Operator Priority	310
Relationship with Startup Options		Program Linkage Instructions	
.....	150, 177, 181, 186	Program Linkage Instructions	158
Rules for Startup Options.....	26	Program Structure Definition Instructions	
Startup Options.....	27	Program Structure Definition Instructions	149
Target-dependent Options.....	29	Pseudo Instruction	
Target-dependent options.....	47	List of Differences in Pseudo Instruction.....	312
Option File			
Comments Described in an Option File	22		

INDEX

Pseudo Machine	
Branch Pseudo Machine Instructions	243
Miscellaneous Pseudo Machine Instructions	245
Operation Pseudo Machine Instructions	244
Optimum Allocation Branch Pseudo Machine Instructions	247
Assembler Pseudo Machine Instructions for the F ² MC-8L/8FX Family	242
Pseudo-instructions	
Pseudo-instructions for which an Expression Containing the Size Operator cannot be Specified	126
Scope of Integer Constants Handled by Pseudo-instructions	148
-pw	
-pw	35
R	
Reference Symbols	
Forward Reference Symbols and Backward Reference Symbols	105
Relational Operators	
Relational Operators	122
Relative Values	
Relative Values	117
Repeat Expansion	
Repeat Expansion	214
Repetition Syntax	
At-end Condition Repetition Syntax	264
Computed Repetition Syntax	262
Execution Condition Repetition Syntax	265
Replacement	
Concatenating the Characters to be Replaced by Macro Replacement (## Operator)	228
Macro Name Replacement	224
Macro Replacement Rules	224
Replacement Rules	
Formal Argument Replacement Rules	203
Local Symbol Replacement Rules	204
Reserved Words	
Reserved Words	104
Restrictions	
Other Restrictions	302
Restrictions	322
Restrictions on Option Files	302
Restrictions Related to Assembly	302
Restrictions Related to Preprocessor Operation	302
ROM Storage	
Setting ROM Storage Sections	141
Rule	
#exitm Instruction Rules	211
#local Instruction Rules	210
#macro Instruction Rules	209

#repeat Instruction Rules	215
Formal Argument Naming Rules	203
Formal Argument Replacement Rules	203
Local Symbol Naming Rules	204
Local Symbol Replacement Rules	204
Macro Call Instruction Rules	213
Macro Definition Rules	208
Macro Name Rules	202
Macro Replacement Rules	224
Naming Rules	103
Rules for Generating Statements	257
Rules for Generating Symbols	257
Rules for Startup Options	26

S

-sa	
-sa	48
Section	
Definition of Section (segment)	313
Multiple Descriptions of a Section	140
Obtaining the Size of a Section in another Module	126
Section Allocation Patterns	131, 136
Section Description Format	130
Section Linkage Methods	138
Section List	95
Section Types	131, 132
Section Types and Attributes	135
Section-location-format	153
Section-type	152
Setting ROM Storage Sections	141
Section Size	
Section Size Determination (SIZEOF Operator)	125
Section Values	
Section Values	118
Segment	
Definition of Section (segment)	313
Shared Linkage	
Shared Linkage	139
Single	
Specification of Single or Double Precision	112
Single Precision	
Data Format of Single-precision Floating-point Constants	113
Size Operator	
Pseudo-instructions for which an Expression Containing the Size Operator cannot be Specified	126
Size Operator	105
SIZEOF	
Section Size Determination (SIZEOF Operator)	125

SOFTUNE	
Overview (SOFTUNE Assembler).....	4
Source List	
Source List	76
Start-time Options	
Relationship with Start-time Options	71
Startup Options	
Relationship with Startup Options	150, 177, 181, 186
Rules for Startup Options.....	26
Startup Options.....	27
Statement	
Rules for Generating Statements.....	257
Statement Format	100
Statement Format	
Statement Format	305
Strings	
Strings	110
Structure	
Access to a Structure	176
Structure Area Definition.....	175
Structured Control Instruction	
Structured Control Instruction	316
Structured Instructions	
Overview of Structured Instructions.....	252
Structured Program Instruction	
Conditional Expressions in a Structured Program	
Instruction	254
Format for Structured Program Instructions	258
Overview of Structured Program Instructions	253
Symbol	
Forward Reference Symbols and Backward Reference	
Symbols	105
Input of Location Counter Symbol.....	306
Local Symbols	194
Location Counter Symbols.....	107
Name (Symbol,Label)	306
Rules for generating Symbols.....	257
Symbol Field	100
Syntax	
2-process Selection Syntax.....	259
At-end Condition Repetition Syntax	264
Computed Repetition Syntax.....	262
Execution Condition Repetition Syntax	265
Expression Syntax.....	115
Multiple-process Selection Syntax	260
Overview (Assembler Syntax).....	5
T	
-tab	39
-target-dependent	
Target-dependent Options	29, 47
Term	
Items that can be Used as a Term	271, 273
Terms that can be Used	276
Term Types	117
Termination	
Termination Code	24
TMP	10
Transfer	
Transfer of Initial-value Data	141
U	
-U	43
V	
-V	55
W	
-w	53
X	
-Xcmsg	56
-Xcwno	57
-Xdiv	49
-Xdof	51
-Xg	32
-Xl	34
-XLBA	61
-Xo	31
-Xsa	48
-XV	55

CM81-00208-3E

FUJITSU SEMICONDUCTOR • CONTROLLER MANUAL
F²MC-8L/8FX FAMILY
SOFTUNE™ ASSEMBLER MANUAL
for V3

March 2008 the third edition

Published **FUJITSU LIMITED** Electronic Devices
Edited Strategic Business Development Dept.
