

FR FAMILY SOFTUNE™ ASSEMBLER MANUAL for V6

FR FAMILY SOFTUNE™ ASSEMBLER MANUAL for V6

FUJITSU MICROELECTRONICS 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 are developing application programs using the FR family microprocessor. Read this manual thoroughly.

■ Trademarks

SOFTUNE is a trademark of FUJITSU MICROELECTRONICS LIMITED.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the U.S. and other countries.

The company names and brand names herein are the trademarks or registered trademarks of their respective owners.

■ Configuration of This Manual

This manual consists of two parts and an appendix.

PART1 OPERATION

Part 1 explains how to use the SOFTUNE assembler.

PART2 SYNTAX

Part 2 describes the syntax and format for writing assembly source programs.

APPENDIX

The two appendixes 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 MICROELECTRONICS device; FUJITSU MICROELECTRONICS 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 MICROELECTRONICS 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 MICROELECTRONICS or any third party or does FUJITSU MICROELECTRONICS warrant non-infringement of any third-party's intellectual property right or other right by using such information. FUJITSU MICROELECTRONICS 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 MICROELECTRONICS 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

| | | |
|------------------|-----------------------------------------------------------------------------------------------|-----------|
| PART1 | 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 | INC911 | 11 |
| 2.5 | OPT911 | 12 |
| 2.6 | Directory Structure of the Development Environment | 13 |
| CHAPTER 3 | STARTUP METHOD | 15 |
| 3.1 | fasm911s 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 Allowed 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 | -O | 48 |
| 4.7.2 | FPU Information Options (-FPU, -XFPU) | 49 |
| 4.7.3 | -cpu | 50 |
| 4.7.4 | -cif | 51 |
| 4.8 | Other Options | 52 |
| 4.8.1 | -Xdof | 53 |
| 4.8.2 | -f | 54 |
| 4.8.3 | -w | 55 |
| 4.8.4 | -name | 56 |
| 4.8.5 | -V, -XV | 57 |
| 4.8.6 | -cmsg, -Xcmsg | 58 |
| 4.8.7 | -cwno, -Xcwno | 59 |
| 4.8.8 | -help | 60 |
| 4.8.9 | -UDSW, -XUDSW | 61 |
| 4.8.10 | -OVFW, -XOVFW | 62 |
| 4.8.11 | -reglist_check, -Xreglist_check | 65 |
| 4.8.12 | -CO | 67 |

CHAPTER 5 OPTIMIZATION CODE CHECK FUNCTIONS 69

| | | |
|-------|-----------------------------------------------------------------------------|----|
| 5.1 | Optimization Code Check Functions of fasm911s | 70 |
| 5.1.1 | Optimization Code Check Levels | 71 |
| 5.1.2 | Forward Reference Symbol Optimization Function | 72 |
| 5.1.3 | Optimization of LDI Instructions | 73 |
| 5.1.4 | Optimization of Branch Instructions | 74 |
| 5.1.5 | Optimization of LDI:20 and LDI:32 Instructions | 76 |
| 5.1.6 | Optimization that Prevents Interlocks Caused by Register Interference | 77 |
| 5.1.7 | Optimization that Replaces Normal Branch Instructions | 79 |
| 5.1.8 | Optimization that Replaces Delayed Branch Instructions | 81 |

CHAPTER 6 ASSEMBLY LIST 83

| | | |
|-------|------------------------------------------------------------|----|
| 6.1 | Composition | 84 |
| 6.2 | Page Format | 85 |
| 6.3 | Information List | 87 |
| 6.4 | Source List | 89 |
| 6.4.1 | Preprocessor and Optimization Code Check Processings | 90 |
| 6.4.2 | Error Display | 91 |
| 6.4.3 | Include File | 92 |
| 6.4.4 | .END, .PROGRAM, .SECTION | 93 |
| 6.4.5 | .ALIGN, .ORG, .SKIP | 94 |
| 6.4.6 | .EXPORT, .GLOBAL, .IMPORT | 95 |
| 6.4.7 | .EQU, .REG | 96 |
| 6.4.8 | .DATA, .BYTE, .HALF, .LONG, .WORD, .DATAB | 97 |

| | | |
|-------------------|-----------------------------------------------------------------|------------|
| 6.4.9 | .FDATA, .FLOAT, .DOUBLE, .FDATAB | 99 |
| 6.4.10 | .RES, .FRES | 101 |
| 6.4.11 | .SDATA, .ASCII, .SDATAB | 102 |
| 6.4.12 | .DEBUG | 103 |
| 6.4.13 | .LIBRARY | 104 |
| 6.4.14 | .FORM, .TITLE, .HEADING, .LIST, .PAGE, .SPACE | 105 |
| 6.5 | Section List | 107 |
| 6.6 | Cross-reference List | 108 |
| PART2 | SYNTAX | 109 |
| CHAPTER 7 | BASIC LANGUAGE RULES | 111 |
| 7.1 | Statement Format | 112 |
| 7.2 | Character Set | 114 |
| 7.3 | Names | 115 |
| 7.4 | Forward Reference Symbols and Backward Reference Symbols | 117 |
| 7.5 | Integer Constants | 118 |
| 7.6 | Location Counter Symbols | 119 |
| 7.7 | Character Constants | 120 |
| 7.8 | Strings | 122 |
| 7.9 | Floating-Point Constants | 123 |
| 7.10 | Data Formats of Floating-Point Constants | 125 |
| 7.11 | Expressions | 127 |
| 7.11.1 | Terms | 129 |
| 7.11.2 | Range of Operand Value | 130 |
| 7.11.3 | Operators | 131 |
| 7.11.4 | Values Calculated from Names | 133 |
| 7.11.5 | Precedence of Operators | 135 |
| 7.12 | Register Lists | 136 |
| 7.13 | Comments | 137 |
| CHAPTER 8 | SECTIONS | 139 |
| 8.1 | Section Description Format | 140 |
| 8.2 | Section Types | 142 |
| 8.3 | Section Types and Attributes | 144 |
| 8.4 | Section Allocation Patterns | 145 |
| 8.5 | Section Linkage Methods | 146 |
| 8.6 | Multiple Descriptions of a Section | 148 |
| 8.7 | Setting ROM Storage Sections | 149 |
| CHAPTER 9 | MACHINE INSTRUCTIONS | 151 |
| 9.1 | Machine Instruction Format | 152 |
| 9.2 | Operand Field Format | 153 |
| CHAPTER 10 | ASSEMBLER PSEUDO-INSTRUCTIONS | 155 |
| 10.1 | Scope of Integer Constants Handled by Pseudo-Instructions | 156 |
| 10.2 | Program Structure Definition Instructions | 157 |

| | | |
|--------|----------------------------------------------------------|-----|
| 10.2.1 | .PROGRAM Instruction | 158 |
| 10.2.2 | .END Instruction | 159 |
| 10.2.3 | .SECTION Instruction | 160 |
| 10.3 | Address Control Instructions | 162 |
| 10.3.1 | .ALIGN Instruction | 163 |
| 10.3.2 | .ORG Instruction | 164 |
| 10.3.3 | .SKIP Instruction | 165 |
| 10.4 | Program Linkage Instructions | 166 |
| 10.4.1 | .EXPORT Instruction | 167 |
| 10.4.2 | .GLOBAL Instruction | 168 |
| 10.4.3 | .IMPORT Instruction | 169 |
| 10.5 | Symbol Definition Instructions | 170 |
| 10.5.1 | .EQU Instruction | 171 |
| 10.5.2 | .REG Instruction | 172 |
| 10.6 | Area Definition Instructions | 173 |
| 10.6.1 | .DATA, .BYTE, .HALF, .LONG, and .WORD Instructions | 174 |
| 10.6.2 | .DATAB Instruction | 176 |
| 10.6.3 | .FDATA, .FLOAT, and .DOUBLE Instructions | 177 |
| 10.6.4 | .FDATAB Instruction | 179 |
| 10.6.5 | .RES Instruction | 180 |
| 10.6.6 | .FRES Instruction | 181 |
| 10.6.7 | .SDATA and .ASCII Instructions | 182 |
| 10.6.8 | .SDATAB Instruction | 183 |
| 10.6.9 | .STRUCT and .ENDS Instructions | 184 |
| 10.7 | Debugging Information Output Control Instruction | 186 |
| 10.8 | Library File Specification Instruction | 187 |
| 10.9 | List Output Control Instructions | 188 |
| 10.9.1 | .FORM Instruction | 189 |
| 10.9.2 | .TITLE Instruction | 190 |
| 10.9.3 | .HEADING Instruction | 191 |
| 10.9.4 | .LIST Instruction | 192 |
| 10.9.5 | .PAGE Instruction | 194 |
| 10.9.6 | .SPACE Instruction | 195 |

CHAPTER 11 PREPROCESSOR PROCESSING 197

| | | |
|--------|---------------------------------------|-----|
| 11.1 | Preprocessor | 198 |
| 11.2 | Basic Preprocessor Rules | 200 |
| 11.2.1 | Preprocessor Instruction Format | 201 |
| 11.2.2 | Comments | 202 |
| 11.2.3 | Continuation of a Line | 203 |
| 11.2.4 | Integer Constants | 204 |
| 11.2.5 | Character Constants | 205 |
| 11.2.6 | Macro Names | 207 |
| 11.2.7 | Formal Arguments | 208 |
| 11.2.8 | Local Symbols | 209 |
| 11.3 | Preprocessor Expressions | 210 |
| 11.4 | Macro Definitions | 212 |

| | | |
|-------------------|--------------------------------------------------------------------------------------|------------|
| 11.4.1 | #macro Instruction | 213 |
| 11.4.2 | #local Instruction | 214 |
| 11.4.3 | #exitm Instruction | 215 |
| 11.4.4 | #endm Instruction | 216 |
| 11.5 | Macro Call Instructions | 217 |
| 11.6 | Repeat Expansion | 218 |
| 11.7 | Conditional Assembly Instructions | 220 |
| 11.7.1 | #if Instruction | 221 |
| 11.7.2 | #ifdef Instruction | 222 |
| 11.7.3 | #ifndef Instruction | 223 |
| 11.7.4 | #else Instruction | 224 |
| 11.7.5 | #elif Instruction | 225 |
| 11.7.6 | #endif Instruction | 227 |
| 11.8 | Macro Name Replacement | 228 |
| 11.8.1 | #define Instruction | 229 |
| 11.8.2 | Replacing Formal Macro Arguments by Character Strings (# Operator) | 231 |
| 11.8.3 | Concatenating the Characters to be Replaced by Macro Replacement (## operator) | 232 |
| 11.8.4 | #set Instruction | 233 |
| 11.8.5 | #undef Instruction | 234 |
| 11.8.6 | #purge Instruction | 235 |
| 11.9 | #include Instruction | 236 |
| 11.10 | #line Instruction | 237 |
| 11.11 | #error Instruction | 238 |
| 11.12 | #pragma Instruction | 239 |
| 11.13 | No-operation Instruction | 240 |
| 11.14 | Defined Macro Names | 241 |
| 11.15 | Differences from the C Preprocessor | 243 |
| CHAPTER 12 | ASSEMBLER PSEUDO MACHINE INSTRUCTIONS | 245 |
| 12.1 | Assembler Pseudo Machine Instructions | 246 |
| APPENDIX | | 253 |
| APPENDIX A | Error Messages | 254 |
| APPENDIX B | Restrictions | 278 |
| INDEX..... | | 279 |

PART1 OPERATION

Part 1 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 FR families. The assembler processing consists of two phase: the preprocessor phase and assembly phase.

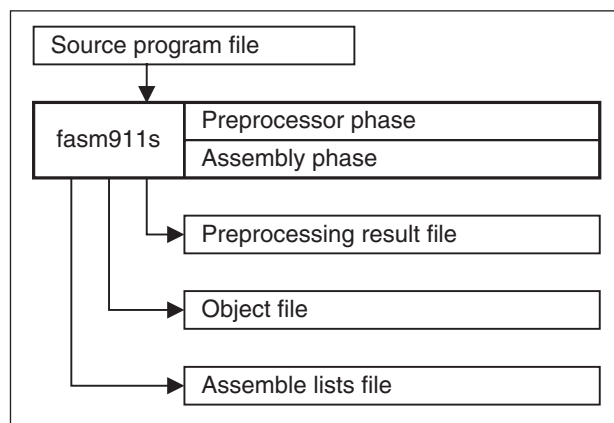
■ Overview

This Assembler assembles the source programs described with the FR family assembly language. Also, it outputs re-locatable objects and assembly lists.

There are two main processes of the assembler; the pre-process phase and the assembly phase.

See Figure 1.1-1 for the configuration of the Assembler.

Figure 1.1-1 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 preprocessor specification as one of its function specifications, the header file can be shared with C when preprocessor processing is performed.

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

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

The results of preprocessing can be stored in 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

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

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

- Comment description shared with C

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 preprocessor specification.

The header file can therefore be shared with C when preprocessor processing is performed.

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

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 INC911

2.5 OPT911

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 specified, the system assumes the parent directory of the directory that contains the assembler that has been started as the installation directory.

■ 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 specified, the system assumes the parent directory of the directory that contains the assembler that has been started (location-of-directory-containing-assembler\..) as the installation directory.

[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]

| |
|------------------------------------------|
| <code>SET FELANG={ASCII EUC SJIS}</code> |
|------------------------------------------|

[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 INC911

INC911 specifies an include path.

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

This environment variable can be omitted.

■ INC911

[Format]

| |
|-----------------|
| SET INC911=path |
|-----------------|

[Description]

INC911 specifies 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 startup option (-I). If no include file is found, the system then searches the path set for INC911.

This environment variable can be omitted.

[Example]

```
SET INC911=E:\INCLUDE
```

2.5 OPT911

**OPT911 specifies the directory that contains the default option file.
This environment variable can be omitted.**

■ OPT911

[Format]

| |
|----------------------|
| SET OPT911=directory |
|----------------------|

[Description]

OPT911 specifies the directory that contains the default option file.

For an explanation of the default option file, see Section "3.6 Default Option File".

These environment variables can be omitted.

If these variables are not specified, 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\911\FASM911.OPT

[Example]

SET OPT911=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 directories and files:

- %FETOOL%\BIN

This is a load module directory.

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

- %FETOOL%\LIB

This is a library directory.

This directory contains appended files such as libraries.

- %FETOOL%\LIB\911

These directories contain the MCU libraries.

These directories contain message files, library files, and include files.

- %FETOOL%\LIB\911\INCLUDE

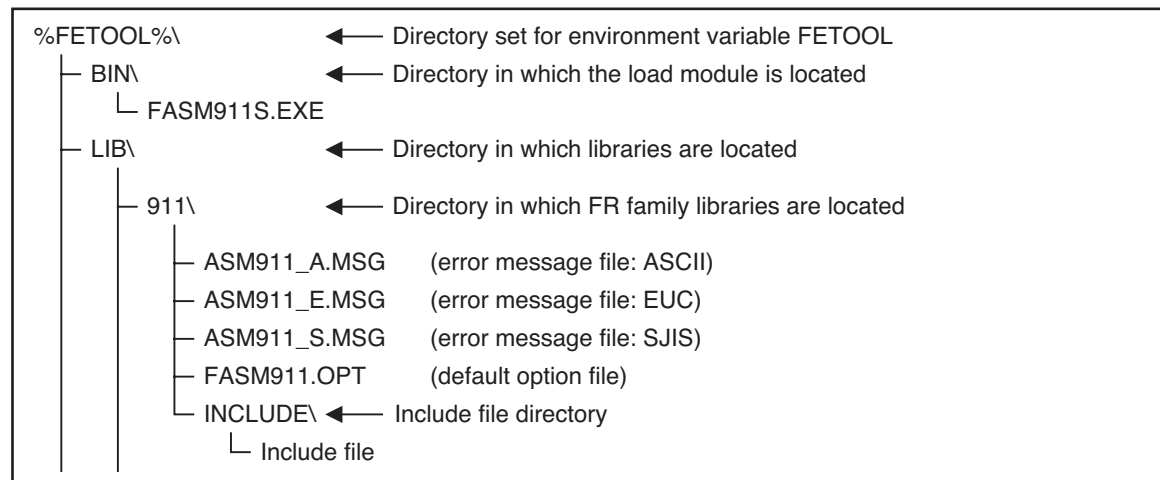
These directories contain the MCU include files.

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 The Directory Structure for the Development Environment



CHAPTER 3

STARTUP METHOD

This chapter describes how to start the assembler.

The assembler startup commands are as follows:

- fasm911s

3.1 fasm911s Commands

3.2 Specifying a File

3.3 Handling of File Names

3.4 Option File

3.5 Comments Allowed in an Option File

3.6 Default Option File

3.7 Termination Code

3.1 fasm911s Commands

The fasm911s commands are described in the following format:

fasm911s [startup-option] ... [file-name]

■ fasm911s Command Lines

[Format]

```
fasm911s [startup-option] ... [file-name]
```

[Description]

A startup option and a file name can be specified on the fasm911s 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 fasm911s commands identify a startup option from 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 fasm911s command described in the file.

This function allows the fasm911s 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 fasm911s command, a startup option list (help message) is output.

The fasm911s commands support the default option file function.

The fasm911s command described in the default option file is executed first.

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

[Example]

```
fasm911s -f def.opt -l prog.asm
```

```
fasm911s -f def.opt 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 |
|-----------------------|----------------------|
| fas911s test | test.asm |
| fas911s test. | test. |
| fas911s D:\WORK\test | D:\WORK\test.asm |
| fas911s ..\FR\abc.src | ..\FR\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 the assembler handles file names.

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 the assembler handles a file name 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> "<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 (.).

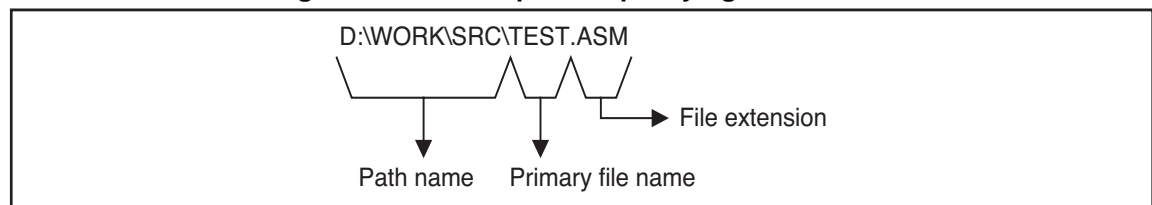
<path-name> and <file-extension> can be omitted.

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 Windows version is used, <drive-name> is included in <path-name>.

[Example]

Figure 3.3-1 Example of Specifying a File Name



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-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 a Partly Omitted File Name

| Omitted component | | Handling |
|-------------------|-------------------------------------|----------|
| Path name | | Current |
| Extension | Source file extension | .asm |
| | Object file extension | .obj |
| | List file extension | .lst |
| | Option file extension | .opt |
| | Preprocessing result file extension | .as |

[Example]

```
fasm911s TEST -o D:\WORK\SRC\TEST -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 fasm911s commands in a file so that they can be specified as a batch. With this function, frequently used 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 fasm911s commands in a file so that they can be specified as a batch. With this function, frequently used 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.

fasm911s commands can be described in an option file.

Comments can be placed in an option file.

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

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

[Example]

Figure 3.4-1 Example of Specifying an Option File

```
option file: def.opt
-I D:\usr\include
-D SMAP
-I
```

```
fasm911s -V -f def.opt test
```

The above example is interpreted as follows:

```
fasm911s -V -I D:\usr\include -D SMAP -I test
```

3.5 Comments Allowed 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 Allowed 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]

```
#
# FR80 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

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.

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 |
|--------------|--------------------------|
| fasm911s | FASM911.OPT |

The default option file is referenced as described below.

- When environment variable OPT911 is set
The system references the file under the directory set for environment variable OPT911.
%OPT911%\FASM911.OPT
 - When environment variable OPT911 is not set
The system references the default option file under the development environment directory.
%FETOOL%\LIB\911\FASM911.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 control assembly processing.
The startup options are identified by an option symbol.
The option 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]

```
fas911s -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) fasm911s -C -name prog test.asm -l
2) fasm911s 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]

```
fas911s -lf t1 test.asm -Xl
```

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

4.2 Startup Option List

Table 4.2-1 lists the startup options.

■ 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. | |
| -I path | Specifies an include path. | |
| -H | Outputs an include path. | Not output |
| -C | Specifies whether to leave comments in the preprocessor output. | Comments are not left. |

Table 4.2-1 Startup Options (2 / 2)

| Specification format | Function overview | Initial value |
|-----------------------------------|---------------------------------------------------------------------------------|----------------|
| Target-dependent options | | |
| -O [0-2] | Specifies the optimization code check level. | 0 |
| -FPU [0-15] | Specifies the FPU channel number. | No FPU |
| -XFPU | Specifies that the FPU is not connected. | |
| -cpu MB-number | Specifies the target chip. | |
| -cif CPU-information file name | Specifies a CPU information file to be referred. | |
| Other options | | |
| -Xdof | Cancels a default option file. | 2 |
| -f filename | Specifies an option file. | |
| -w [0-3] | Specifies the output level of warning messages. | Not displayed |
| -name module-name | Specifies a module name. | |
| -V | Outputs a startup message. | |
| -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 |
| -UDSW | Warning when referring to undefined symbol. | Outputs |
| -XUDSW | Warning output is inhibited when referring to an undefined symbol. | |
| -OVFW | Specifies to generate a code as a WARNING level for overflow. | |
| -XOVFW | Specifies not to generate a code as an error level for overflow. | ERROR |
| -reglst_check | Specification of duplication specification check for register list. | Check |
| -Xreglst_check | Suppression specification of duplication specification check for register list. | |
| -CO | Specification of output FR/FR80 common object. | |

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 the functions 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:

- -oSpecifies an object file name.
- -XoCreates no object file.
- -gOutputs debugging information.
- -XgCancels 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]

```
fasn911s ex1 -o ex1_a
```

■ -Xo

[Format]

```
-Xo
```

[Description]

-Xo creates no object file.

[Example]

```
fasn911s 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]

```
iasm911s c_test -g
```

■ -Xg

[Format]

-Xg

[Description]

-Xg outputs no debugging information to an object file.

[Example]

```
iasm911s 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 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, -Xl

-l creates an assembly list file.

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

-Xl creates no assembly list file.

If -l, -lf, nor -Xl is specified, no assembly list file 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]

```
fasml 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 extension .lst.

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

[Example]

```
fasml test -lf\asm\src
```

■ -Xl

[Format]

```
-Xl
```

[Description]

-Xl creates no assembly list file.

[Example]

```
fasml test -Xl
```

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]

```
fasn911s 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]

```
fasn911s 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 each of these lists is output.

- **linf** specifies whether an information list is output.
- **lsrc** specifies whether a source list is output.
- **lsec** specifies whether a section list is output.
- **lcros** specifies whether 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 an information list is output.

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

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

[Example]

```
fasm911s 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 a source list is output.

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

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

[Example]

```
fasm911s 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 a section list is output.

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

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

[Example]

```
fasn911s 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 a cross-reference list is output.

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

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

[Example]

```
fasn911s 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 or lowercase can be used to specify ON or OFF.

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

[Example]

```
fasn911s 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: The text of a macro expansion section is not output to the list. Only the object code is output to the list. <default>

[Description]

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

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

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

[Example]

```
fasn911s 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]

| |
|--------------------------|
| <code>-tab {0-32}</code> |
|--------------------------|

[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]

```
fasm911s 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 Does 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 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 reduces required 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]

```
fas911s 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]

```
fas911s 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 file-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]

```
fas911s test -Pf \fas\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 specified without "=defined-character-string", 1 is defined.

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

This option can be specified more than once.

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

[Example]

```
fas911s test -D OS_type=3 -D Windows
```

■ -U

[Format]

```
-U macro-name
```

[Description]

-U cancels a macro name specified using -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]

```
fas911s 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]

| |
|------------------------------|
| <code>-I include-path</code> |
|------------------------------|

[Description]

-I specifies an include path.

Specify a path to search for an include file specified using 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".

[Example]

```
fasn911s test -I \include -I \FR
```


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 not found when the path was searched.

[Example]

```
fasn911s test -I \include -I \FR -H
```

4.6.6 -C

-C leaves all comments and blank characters during preprocessor processing. If this option is not specified, a comment and series of blank characters are replaced with a single blank character.

■ -C

[Format]

| |
|----|
| -C |
|----|

[Description]

-C leaves all comments and blank characters during preprocessor processing.

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

Omitting this option reduces the processing load in the assembly phase.

[Example]

```
fas911s test -C
```

4.7 Target-Dependent Options

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

■ Target-dependent Options

The following five target-dependent options are supported:

- -OSpecifies the check level of the optimization code.
- -FPUSpecifies the FPU channel number.
- -XFPUSpecifies that no FPU is connected.
- -cpuSpecifies the target chip.
- -cifSpecifies a CPU information file to be referred.

4.7.1 -O

-O specifies the check level for machine instruction optimization code.

■ -O

[Format]

| |
|-------------------------------|
| <code>-O [check-level]</code> |
|-------------------------------|

check-level: 0 to 2

[Description]

-O specifies the check level for machine instruction optimization code.

For details of checking optimization code, see "CHAPTER 5 OPTIMIZATION CODE CHECK FUNCTIONS".

If check-level is omitted, 0 is the default.

If this option is not specified, 0 is the default.

[Example]

```
fasn911s test -O 2
```

4.7.2 FPU Information Options (-FPU, -XFPU)

The FPU information options are options related to the FPU.

-FPU specifies that the FPU is connected and simultaneously specifies the number of the channel to which the FPU is connected.

-XFPU specifies that the FPU is not connected.

If neither **-FPU** nor **-XFPU** is specified, it is assumed that the FPU is not connected.

It is an effective option only when Target CPU is FR family.

■ FPU Information Options (-FPU and -XFPU)

The following two FPU information options are supported:

- **-FPU**..... Specifies that the FPU is connected.
- **-XFPU**..... Specifies that the FPU is not connected.

■ -FPU

[Format]

```
-FPU [channel-number]
```

channel-number: 0 to 15

[Description]

-FPU specifies that the FPU is connected.

channel-number indicates the number of the channel to which the FPU is connected.

If channel-number is omitted, 0 is the default.

If this option is not specified, the FPU is not connected, disabling the use of FPU instructions.

It is an effective option only when target CPU is FR family.

It is invalid when target CPU is FR80 family.

[Example]

```
fasm911s test -FPU 0
```

■ -XFPU

[Format]

```
-XFPU
```

[Description]

-XFPU specifies that the FPU is not connected.

Therefore, FPU instructions cannot be used.

[Example]

```
fasm911s test -XFPU
```

4.7.3 -cpu

-cpu specifies the target chip.

Specify the name of the product to be used as the target chip.

■ -cpu

[Format]

| |
|--------------------------|
| <code>-cpu target</code> |
|--------------------------|

target: name of the product to be used

[Description]

-cpu specifies the target chip.

Specify the name of the product to be used as target.

[Example]

```
iasm911s test -cpu MB91101
```

```
iasm911s test -cpu MB91307
```

4.7.4 -cif

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

■ -cif

[Format]

| |
|--------------------------------------------|
| <code>-cif CPU-information-filename</code> |
|--------------------------------------------|

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

[Description]

Specify a CPU information file that SOFTUNE Tools reference.

[Example]

```
fasmg911s test -cpu MB91101 -cif "C:\Softune6\lib\911\911.csv"
```

Note:

SOFTUNE Tools get CPU information by referring the CPU information file. Reference to the different CPU information file 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\911\911.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 so that each tool can refer the same CPU information file.

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 is output.
- -Xcwno Specifies 0 as the termination code when a warning is output.
- -help Displays a help message.
- -UDSW Warning when referring to undefined symbol.
- -XUDSW Warning output is inhibited when referring to an undefined symbol.
- -OVFW Specifies to generate a code as a WARNING level for overflow.
- -XOVFW Specifies not to generate a code as an ERROR level for overflow.
- -reglst_check..... Specification of duplication specification check for register list.
- -Xreglst_check.. Suppression specification of duplication specification check for register list.
- -CO Specification of output FR/FR80 common object.

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 Section "3.6 Default Option File".

[Example]

```
fasn911s test -Xdof
```

4.8.2 -f

-f reads the specified option file.

The fasm911s commands can be placed in an option file.

■ -f

[Format]

| |
|----------------------------------|
| <code>-f option-file-name</code> |
|----------------------------------|

[Description]

-f reads the specified option file.

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

The fasm911s commands can be placed in an option file.

Multiple option files can be specified.

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

[Example]

```
fas911s 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:

| 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]

```
fasm911s test -w 0
```

Note:

The following warnings are output when the warning level is set to 3.

W1551A: A warning is output if there is no .END instruction at the end of the source file.

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

■ -name

[Format]

| |
|--------------------------------|
| <code>-name module-name</code> |
|--------------------------------|

[Description]

-name specifies a module name.

A specified module name must comply with naming rules.

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

[Example]

```
fasn911s test -name prog
```

Note:

When an inappropriate character string is specified for the module name with this option, a warning message is output with the last line of the source lines.

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]

```
fasn911s test -V
```

■ -XV

[Format]

-XV

[Description]

-XV cancels the display of a startup message.

[Example]

```
fasn911s 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, an exit message is not displayed.

■ -cmsg

[Format]

`-cmsg`

[Description]

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

[Example]

```
fasn911s test -cmsg
```

■ -Xcmsg

[Format]

`-Xcmsg`

[Description]

-Xcmsg cancels the display of a termination message.

[Example]

```
fasn911s 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 displayed 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]

```
fasn911s test -cwno
```

■ -Xcwno

[Format]

| |
|--------|
| -Xcwno |
|--------|

[Description]

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

[Example]

```
fasn911s test -cwno -Xcwno
```

4.8.8 -help

-help displays the startup option list.
This list is referred to as 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]

```
fas911s test -help
```


4.8.9 -UDSW, -XUDSW

-UDSW displays a warning message when referring to an undefined symbol.

-XUDSW does not display a warning message when referring to an undefined symbol.

If neither -UDSW nor -XUDSW is specified, a warning message is displayed when referring to an undefined symbol.

■ -UDSW

[Format]

| |
|-------|
| -UDSW |
|-------|

[Description]

This option displays a warning message when referring to an undefined symbol.

[Example]

```
fasn911s test -UDSW
```

■ -XUDSW

[Format]

| |
|--------|
| -XUDSW |
|--------|

[Description]

This option does not display a warning message when referring to an undefined symbol.

[Example]

```
fasn911s test -XUDSW
```

4.8.10 -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 assemble list is output when an assemble list output specification option (-l) is specified.

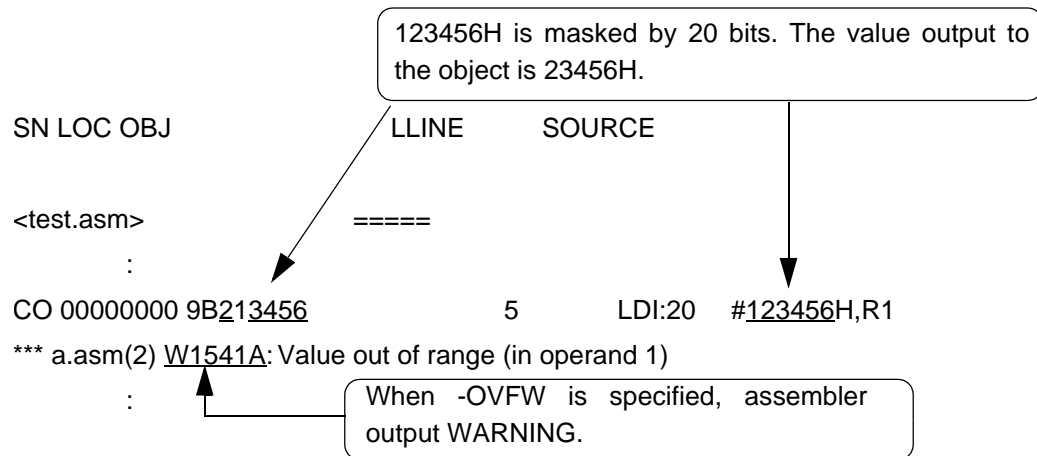
The following shows an example of this option, and an example of an output list.

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

[Example]

```
fasn911s test -OVFW -l
```

- Output list



■ -XOVFW

[Format]

-XOVFW (Default)

[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 an error message. (W1541A: Value out of range.)
- Does not output an object file.
- Masks the operand operation result in accordance with the operand size, and sets only lower bits to output a code to the assemble 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.

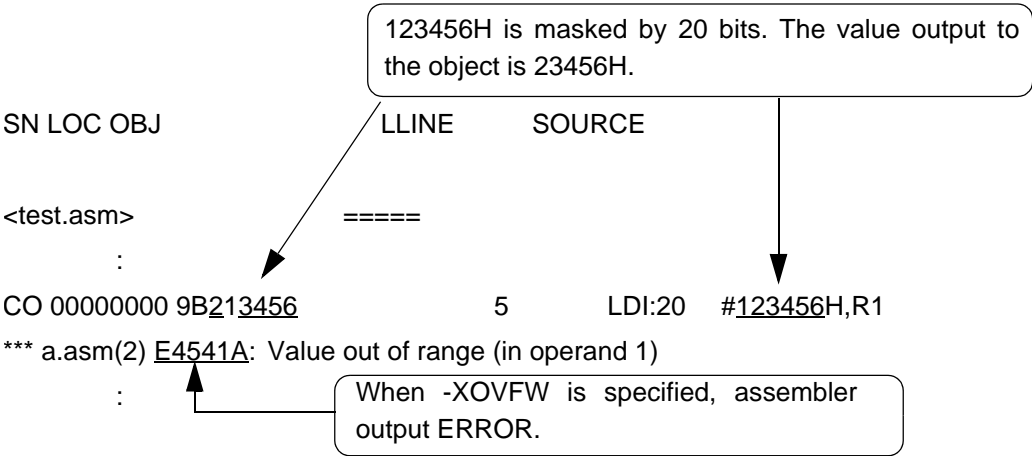
To output an assembly list, specify the assembly list output specification option (-l).

The assemble list is also output when an error occurs.

[Example]

```
fasm911s test -XOVFW -l
```

- Output list



4.8.11 -reglst_check, -Xreglst_check

-reglst_check is an option for outputting warning when there are duplication specification for the register list.

-Xreglst_check is an option for suppressing **-reglst_check**.

■ -reglst_check

[Format]

```
-reglst_check (Default)
```

[Description]

When this option is specified a warning is displayed if there are duplications in the register list.
(W1546A: Duplication in the specification of the register list (Register name)).

[Example 1 Specified by ","]

```
STM (R1, R2, R3, R1)
*** sample.asm(2) W1546A: Duplication in the specification of the register list (R1)
```

[Example 2 Specified by "-"]

```
STM (R1-R5, R3-R6)
*** sample.asm(3) W1546A: Duplication in the specification of the register list (R3,R4,R5)
```

[Example 3 Specified by "-" and ","]

```
STM (R1-R5, R3,R6,R7)
*** sample.asm(4) W1546A: Duplication in the specification of the register list (R3)
```

[Example 4 Specified by a multiple of register list symbols]

```
WKREG: .REG (R1-R4)
WKREG2: .REG (R3-R6)
STM (WKREG, WKREG2)
*** sample.asm(8) W1546A: Duplication in the specification of the register list (R3,R4)
```

[Example 5 Specified by a register list symbol and ","]

```
WKREG: .REG (R1-R4)
STM (WKREG,R2,R4,R7)
*** sample.asm(8) W1546A: Duplication in the specification of the register list (R2,R4)
```

[Example 6 Specified by a register list symbol and "-"]

```
WKREG: .REG (R1-R4)
STM (WKREG,R2-R5,R7)
*** sample.asm(8) W1546A: Duplication in the specification of the register list (R2,R3,R4)
```

[Example 7 Duplication specified in register list symbol]

```
WKREG: .REG (R1,R4,R7,R3-R7)
```

```
*** sample.asm(8) W1546A: Duplication in the specification of the register list (R4,R7)
```

■ **-Xreglst_check**

[Format]

| |
|----------------|
| -Xreglst_check |
|----------------|

[Description]

When this option is specified, the -reglst_check option is suppressed.

This option is the default setting.

4.8.12 -CO

FR/FR80 common object can linked with either target CPU of FR and FR80.

■ -CO

[Format]

| |
|-----|
| -CO |
|-----|

[Description]

FR/FR80 common object can linked with either target CPU of FR and FR80.

When -CO option is specified, the error occurs by the instruction shown in the Table 4.8-1.

Table 4.8-1 Incompatibility of FR and FR80 Instructions

| Instructions incompatibility | FR | FR80 |
|------------------------------|----|------|
| LDRES @Ri+,#u4 | ○ | × |
| STRES #u4,@Ri | ○ | × |
| COPOP #u4,#CC,CRj,CRi | ○ | × |
| COPLD #u4,#CC,Rj,CRi | ○ | × |
| COPST #u4,#CC,CRj,Ri | ○ | × |
| COPSV #u4,#CC,CRj,Ri | ○ | × |
| SRCH0 Ri | × | ○ |
| SRCH1 Ri | × | ○ |
| SRCHC Ri | × | ○ |

○: Compatible ✕: Incompatible

- When -CO option is specified

The following error occurs by the instruction shown in the Table 4.8-1.

E4601A:Unusable operation mnemonic with common object (instruction-name)

- When -CO option is not specified

-Target CPU is FR

The following error occurs by the instruction for FR80 shown in the Table 4.8-1.

E4600A: Invalid operation mnemonic (instruction-name)

-Target CPU is FR80

The following error occurs by the instruction for FR shown in the Table 4.8-1.

E4600A: Invalid operation mnemonic (instruction-name)

The following errors occur regardless of presence specified of -CO option, when the instruction that doesn't exist neither FR or FR80 is described.

E4600A: Invalid operation mnemonic (instruction-name)

Note:

-cpu option cannot omit.

Please specify -cpu option when you make a FR/FR80 common object.

[Example]

```
fasm911s -cpu MB91101 common_module.asm -CO
```

```
fasm911s -cpu MB91680 common_module.asm -CO
```


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 fasm911s

5.1 Optimization Code Check Functions of fasm911s

For the fasm911s, those portions of a program that can be optimized, provided optimization does not disturb program operation, are located.

For optimization other than branch instruction optimization, a warning message is displayed.

■ Optimization Code Check Functions for the fasm911s

For the fasm911s, the following six kinds of optimizable instructions are located:

- Optimization of LDI instructions
- Optimization of branch instructions
- Optimization of LDI:20 and LDI:32 instructions
- Optimization that prevents interlocks caused by register interference
- Optimization that replaces normal branch instructions
- Optimization that replaces delayed branch instructions

5.1.1 Optimization Code Check Levels

Table 5.1-1 lists optimization code check processing to be detected and their corresponding check levels.

A check level is specified using the start-time option -O.

If a check level is not specified, the default is 0.

■ Check Levels and Optimization Code Check Processing

Table 5.1-1 Check Levels and Optimization Code Check Processing

| Check level | Optimization of LDI instructions and optimization of branch instructions | Optimization of LDI:20 and LDI:32 instructions | Optimization that prevents register-interference-originated interlocks | Optimization that replaces normal branch instructions | Optimization that replaces delayed branch instructions |
|-------------|--------------------------------------------------------------------------|------------------------------------------------|------------------------------------------------------------------------|-------------------------------------------------------|--------------------------------------------------------|
| 0 | Y | - | - | - | - |
| 1 | Y | N | N | - | - |
| 2 | Y | N | N | N | N |

Y: Optimization is performed.

N: The optimization code check displays a warning message.

- : An optimization code check is not performed.

5.1.2 Forward Reference Symbol Optimization Function

The assembler provides a forward reference symbol optimization function.

■ Forward Reference Symbol Optimization Function

The assembler provides a forward reference symbol optimization function.

This optimization function operates only on the following instructions:

- LDI instruction: LDI
- Extended branch instruction: CALL20 BRA20 Bcc20 CALL20:D BRA20:D Bcc20:D
CALL32 BRA32 Bcc32 CALL32:D BRA32:D Bcc32:D

Because the optimization function performs processing at high speed, some requirements are imposed on the immediate value of the operand and the description of the branch address expression as explained below. The symbols mean both forward reference and backward reference.

● LDI instruction

- The symbol represents an address that has an absolute value.
- The symbol is included in the same section as the machine instructions.
The symbol is included in a section other than the code sections.
- The format of the expression applies to one of the following:
 - symbol
 - symbol + offset-value
 - symbol - offset-value

[Example]

```
.SECTION P, CODE, ALIGN=2
LDI   #fwd_no+3, R2
.SECTION D, DATA, LOCATE=0x1000
fwd_no: .DATA 1, 2, 3
```

● Extended branch instruction

- The symbol represents an address that has an absolute or relative value.
- The symbol is included in the same section as the machine instructions.
- The format of the expression applies to one of the following:
 - symbol
 - symbol + offset-value
 - symbol - offset-value

[Example]

```
.SECTION P, CODE, ALIGN=2
BRA32 #label, R2
:
label:  ADD   #4, R1
```

In ordinary use, these requirements do not matter, and most instructions can be optimized.

5.1.3 Optimization of LDI Instructions

Optimization is performed on LDI instructions (except LDI:8, LDI:20, and LDI:32 instructions).

■ Optimization of LDI Instructions

Optimization is always performed.

Optimization is performed on LDI instructions (except LDI:8, LDI:20, and LDI:32 instructions).

If the immediate value expression contains a forward reference symbol, the maximum number of bits is usually selected as explained in Section "7.4 Forward Reference Symbols and Backward Reference Symbols".

[Example When forward reference symbol optimization is not performed]

```
.SECTION P, CODE, ALIGN=2
LDI #fwd_no, R2 --> LDI:32 #fwd_no, R2 (A 32-bit instruction is created.)
.SECTION D, DATA, LOCATE=0x1000
fwd_no: .DATA 1, 2, 3
```

[Example When forward reference symbol optimization is performed]

```
.SECTION P, CODE, ALIGN=2
LDI #fwd_no, R2 ---> LDI:20 #fwd_no, R2 (A 20-bit instruction is created.)
.SECTION D, DATA, LOCATE=0x1000
fwd_no: .DATA 1, 2, 3
```

As shown in the above example, the optimum instruction format is selected for the LDI instruction according to the value of forward reference symbol fwd_no. (In the above case, since the value of fwd_no is 0x1000, the LDI:20 instruction is selected.)

Note :

For optimization, the symbol must satisfy some requirements. For details, see Section "5.1.2 Forward Reference Symbol Optimization Function".

5.1.4 Optimization of Branch Instructions

Optimization operates on branch instructions. The optimum instruction is created based on the distance to the branch destination label.

The following instructions are optimized (20-bit extended branch instructions, 32-bit extended branch instructions)

- Instructions to be optimized: CALL20 BRA20 Bcc20 CALL20:D BRA20:D Bcc20:D CALL32 BRA32 Bcc32 CALL32:D BRA32:D Bcc32:D

■ Optimization of Branch Instructions

Optimization is always performed.

Optimization operates on 20-bit extended branch instructions or 32-bit extended branch instructions.

First, the distance to the branch destination is calculated by the following expression:

$$\text{distance} = \text{branch-destination-label} - \text{current-location} - 2$$

Then the optimal instruction for the distance is created.

Table 5.1-2 shows the relationships between the distance and instructions that are created.

If the branch destination label is an external reference symbol, the distance is classified as "other".

The following can be specified for the condition-specifying portion "cc".

| Specifying condition | Condition | Specifying condition | Condition |
|----------------------|-----------|----------------------|------------------------|
| EQ | (Z)=1 | NV | (V)=0 |
| NE | (Z)=0 | LT | (V) or (N)=1 |
| BC | (C)=1 | GE | (V) or (N)=0 |
| NC | (C)=0 | LE | ((V) xor (N)) or (Z)=1 |
| N | (N)=1 | GT | ((V) xor (N)) or (Z)=0 |
| P | (N)=0 | LS | (C) or (Z)=1 |
| V | (V)=1 | HI | (C) or (Z)=0 |

Note :

Some requirements are imposed on optimization. For details, see Section "5.1.2 Forward Reference Symbol Optimization Function".

Table 5.1-2 Relationships between Distance and Created Instructions in Branch Instructions

| Extended branch instruction | Distance | Created instruction |
|-----------------------------|-----------------|--------------------------------------------------------|
| CALL20 label, Ri | -0x800 ~ +0x7fe | CALL label |
| | Other | LDI:20 #label, Ri CALL @Ri |
| CALL20:D label, Ri | -0x800 ~ +0x7fe | CALL:D label |
| | Other | LDI:20 #label, Ri CALL:D @Ri |
| BRA20 label, Ri | -0x100 ~ +0xfe | BRA label |
| | Other | LDI:20 #label, Ri JMP @Ri |
| BRA20:D label, Ri | -0x100 ~ +0xfe | BRA:D label |
| | Other | LDI:20 #label, Ri JMP:D @Ri |
| Bcc20 label, Ri (*) | -0x100 ~ +0xfe | Bcc label |
| | Other | Bxcc false LDI:20 #label, Ri JMP @Ri false: |
| Bcc20:D label, Ri (*) | -0x100 ~ +0xfe | Bcc label |
| | Other | Bxcc false LDI:20 #label, Ri JMP:D @Ri false: |
| CALL32 label, Ri | -0x800 ~ +0x7fe | CALL label |
| | Other | LDI:32 #label, Ri CALL @Ri |
| CALL32:D label, Ri | -0x800 ~ +0x7fe | CALL:D label |
| | Other | LDI:32 #label, Ri CALL:D @Ri |
| BRA32 label, Ri | -0x100 ~ +0xfe | BRA label |
| | Other | LDI:32 #label, Ri JMP @Ri |
| BRA32:D label, Ri | -0x100 ~ +0xfe | BRA:D label |
| | Other | LDI:32 #label, Ri JMP:D @Ri |
| Bcc32 label, Ri (*) | -0x100 ~ +0xfe | Bcc label |
| | Other | Bxcc false LDI:32 #label, Ri JMP @Ri false: |
| Bcc32:D label, Ri (*) | -0x100 ~ +0xfe | Bcc label |
| | Other | Bxcc false LDI:32 #label, Ri JMP:D @Ri false: |

*: xcc represents the condition opposite to cc.

5.1.5 Optimization of LDI:20 and LDI:32 Instructions

Optimization operates on LDI:20 and LDI:32 instructions.

If the instruction can be replaced with one having a smaller immediate value, it is replaced and a warning message is displayed.

The immediate value, however, must not include a forward reference symbol, and must be an absolute value.

The following instructions are optimized.

- Instructions optimized: LDI:20, LDI:32

■ Optimization of LDI:20 and LDI:32 Instructions

Optimization operates on LDI:20 and LDI:32 instructions.

If the instruction can be replaced with one having a smaller immediate value, it is replaced and a warning message is displayed.

The immediate value, however, must not include a forward reference symbol, and must be an absolute value.

- LDI:20 #I20,Ri

If the value of i20 is in the range 0 to 0xff, the instruction is changed to an LDI:8 instruction.

- LDI:32 #I32,Ri

If the value of i32 is in the range 0 to 0xff, the instruction is changed to an LDI:8 instruction.

If the value of i32 is in the range 0x100 to 0xffff, the instruction is changed to an LDI:20 instruction.

[Example]

| Before optimization | After optimization |
|---------------------|--------------------|
| LDI:20 #0,R0 | LDI:8 #0,R0 |
| LDI:20 #0xff,R0 | LDI:8 #0xff,R0 |
| LDI:32 #0,R0 | LDI:8 #0,R0 |
| LDI:32 #0xff,R0 | LDI:8 #0xff,R0 |
| LDI:32 #0xffff,R0 | LDI:20 #0xffff,R0 |

5.1.6 Optimization that Prevents Interlocks Caused by Register Interference

Optimization operates on an instruction for which an interlock occurs because of register interference. It is checked the relation of before and after of instruction. If the instruction can be placed before the preceding instruction, it is placed there to avoid an interlock, and a warning message is displayed.

The following instructions are optimized.

- Instructions optimized: LD, LDUH, LDUB, LEAVE, ANDCCR, ORCCR, DIV1
MOV Ri, PS
DMOV @dir10, R13
DMOVB @dir8, R13

■ Optimization that Prevents Interlocks Caused by Register Interference

When a register hazard occurs, FR/FR80 controls pipelining by applying an interlock, avoiding the hazard. Note that if there is no register interference, there is no hazard occurs and no interlock is applied.

Optimization operates on a instruction for which an interlock occurs because of register interference. It is checked the relation of before and after of instruction. If the instruction can be placed before the preceding instruction, it is placed there to avoid an interlock, and a warning message is displayed.

Optimization operates on instructions whose number of machine cycles is represented by "b" in the *Programming Manual*.

[Example 1 Instructions that are replaced]

| Before optimization | After optimization | Comment |
|-----------------------------------------------|-----------------------------------------------|----------------------------------------------------------------------------|
| ADDN #4, R1 LD @R4, R0 CMP #0, R0 | LD @R4, R0 ADDN #4, R1 CMP #0, R0 | R0 in LD causes register interference. LD and ADDN are replaced. |
| ST R1, @R2 DMOV @dir10, R13 MOV R13, R0 | DMOV @dir10, R13 ST R1, @R2 MOV R13, R0 | R13 in DMOV causes register interference. DMOV and ST are replaced. |
| ADDN #4, R1 LEAVE MOV R14, R0 | LEAVE ADDN #4, R1 MOV R14, R0 | R14 in LEAVE causes register interference. LEAVE and ADDN are replaced. |

[Example 2 Instructions that are not replaced]

| Before optimization | After optimization | Comment |
|-----------------------------------------|--------------------|-----------------------------------------------------------------------------------------------------------------------|
| ADDN #4, R1 LD @R4, R0 CMP #0, R2 | ← | Instructions are not replaced because no register interference occurs. |
| ST R2, @R4 LD @R4, R0 CMP #0, R0 | ← | Although R0 in LD causes register interference, instructions cannot be replaced because @R4 in LD is specified in ST. |
| ADDN #4, R1 LEAVE MOV @R15+, RP | ← | R15 in LEAVE does not cause an interlock. |

5.1.7 Optimization that Replaces Normal Branch Instructions

Optimization operates on a normal branch instruction. It is checked the relation of before and after of instruction. If the instruction can be placed before the preceding instruction, a warning message is displayed, and it is changed to a delayed branch instruction. In addition, the preceding instruction is moved to the delay slot. This processing loads the instruction into the delay slot, maintain execution speed. The following instructions are optimized.

- Instructions optimized: **JMP, RET**
CALL, BRA, Bcc
CALL20, BRA20, Bcc20
CALL32, BRA32, Bcc32

■ Optimization that Replaces Normal Branch Instructions

When an instruction is determined to be a branch instruction, FR/FR80, which is performing pipelining, has already read the next instruction.

In branching, a normal branch instruction cancels an instruction that is being read unnecessarily while it is being executed. This reduces execution speed.

Optimization operates on a normal branch instruction. It is checked the relation of before and after of instruction. If the instruction can be placed before the preceding instruction, a warning message is displayed, and it is changed to a delayed branch instruction. In addition, the preceding instruction is moved to the delay slot.

This processing loads the instruction into the delay slot, maintaining execution speed.

For details of delayed branches, see the *Programming Manual*.

[Example 1 Instructions that are replaced]

| Before optimization | After optimization | Comment |
|---------------------------|-----------------------------|----------------------------------------------------------------|
| ADDN #4, R0 BRA label | BRA:D label ADDN #4, R0 | BRA is changed to BRA:D. ADDN is moved to the delay slot. |
| ADDN #4, R0 CALL label | CALL:D label ADDN #4, R0 | CALL is changed to CALL:D. ADDN is moved to the delay slot. |
| ADDN #4, R0 BEQ label | BEQ:D label ADDN #4, R0 | BEQ is changed to BEQ:D. ADDN is moved to the delay slot. |

[Example 2 Instructions that are not replaced]

| Before optimization | After optimization | Comment |
|-----------------------------------------------------|--------------------|-----------------------------------------------------------------------------------------------------------------------|
| CMP #4,R0 BEQ label | ← | Instructions cannot be replaced because an instruction that toggles a flag precedes a conditional branch instruction. |
| MUL R2,R3 BRA label | ← | Instructions cannot be replaced because an instruction before a branch instruction cannot be moved to the delay slot. |
| CALL:D label1 ADDN #4,R0 BRA label2 | ← | Instructions cannot be replaced because the delay slot precedes a branch instruction. |
| label1: BRA label2 | ← | Instructions cannot be replaced because a label precedes a branch instruction. |

5.1.8 Optimization that Replaces Delayed Branch Instructions

Optimization operates on a delayed branch instruction. If there is a NOP instruction in the delay slot, and the delayed branch instruction can be placed before the preceding instruction, the NOP instruction is deleted, and a warning message displayed. In addition, the preceding instruction is moved to the delay slot.

If the instructions cannot be replaced, the delayed branch instruction is changed to a normal branch instruction, and the NOP instruction is deleted. A warning message is also displayed.

If the delay slot holds an instruction other than a NOP instruction, the optimization is not performed.

The following instructions are optimized.

- Instructions optimized: JMP:D, RET:D
 CALL:D, BRA:D, Bcc:D
 CALL20:D, BRA20:D, Bcc20:D
 CALL32:D, BRA32:D, Bcc32:D
-

■ Optimization that Replaces Delayed Branch Instructions

When an instruction is determined to be a branch instruction, FR/FR80, which is performing pipelining, has already read the next instruction.

The delayed branch instruction must execute the read instruction by storing it in the delay slot.

For this reason, since the next instruction is inevitably executed during branching, the program must be written very carefully.

Generally, while a program is being developed, a NOP instruction is always placed in the delay slot. When the program is closer to completion, the NOP instruction is deleted.

Optimization operates on a delayed branch instruction. If a NOP instruction is in the delay slot, and the delayed branch instruction can be placed before the preceding instruction, the NOP instruction is deleted, and a warning message is displayed. In addition, the preceding instruction is moved to the delay slot.

If the instructions cannot be replaced, the delayed branch instruction is changed to a normal branch instruction, and the NOP instruction is deleted. A warning message is also displayed.

If the delay slot holds an instruction other than a NOP instruction, the optimization is not performed.

For details of delayed branches, see the *Programming Manual*.

[Example 1 Instructions that are replaced]

| Before optimization | After optimization | Comment |
|--------------------------------------------|-------------------------------------|----------------------------------------------------|
| ADDN #4,R0 BRA:D label NOP | BRA:D label ADDN #4,R0 | ADDN is moved to the delay slot. NOP is deleted |
| MOV R1,R2 CALL:D label NOP | CALL:D label MOV R1,R2 | MOV is moved to the delay slot. NOP is deleted. |
| ADDN #4,R0 BEQ:D label NOP | BEQ:D label ADDN #4,R0 | ADDN is moved to the delay slot. NOP is deleted |

[Example 2 Instructions that are not replaced]

| Before optimization | After optimization | Comment |
|----------------------------------------------------------------|---------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ADDN #4,R0 CALL:D label LDI:8 #4,R4 | ← | There is no optimization because an instruction other than a NOP instruction is in the delay slot. |
| MUL R2,R3 BRA:D label NOP | MUL R2,R3 BRA label | Instructions cannot be replaced because the instruction before the branch instruction cannot be placed in the delay slot. The delayed branch instruction is therefore changed to a normal branch instruction, and the NOP instruction is deleted. |
| CALL:D label1 ADDN #4,R0 BRA:D label2 NOP | CALL:D label1 ADDN #4,R0 BRA label2 | The instructions cannot be replaced because the delay slot precedes a branch instruction. The delayed branch instruction is therefore changed to a normal branch instruction, and the NOP instruction is deleted. |
| CMP #4,R0 BEQ:D label NOP | CMP #4,R0 BEQ label | The instructions cannot be replaced because the instruction before the conditional branch instruction is a flag-dependent instruction. The delayed branch instruction is therefore changed to a normal branch instruction, and the NOP instruction is deleted. |
| label1: BRA:D label2 NOP | label1: BRA label2 | The instructions cannot be replaced because a label precedes a branch instruction. The delayed branch instruction is therefore changed to a normal branch instruction, and the NOP instruction is deleted. |

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, and option files.

- **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, object data and other information are displayed.

- **Section list**

The section list consists of the names and attributes of and other data about sections 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

With the fasm911s, whether to display a list can be specified 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 between lists and start-time options.

Table 6.1-1 Relationship between 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 forming an assembly list is shown below. In fasm911s, the size of this format can be specified by a startup option or pseudo-instruction.

When using start-time options, specify the number of lines with `-pl`, and the number of columns 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 forming an assembly list is shown below. In fasm911s, the size of this format can be specified by a startup option or pseudo-instruction.

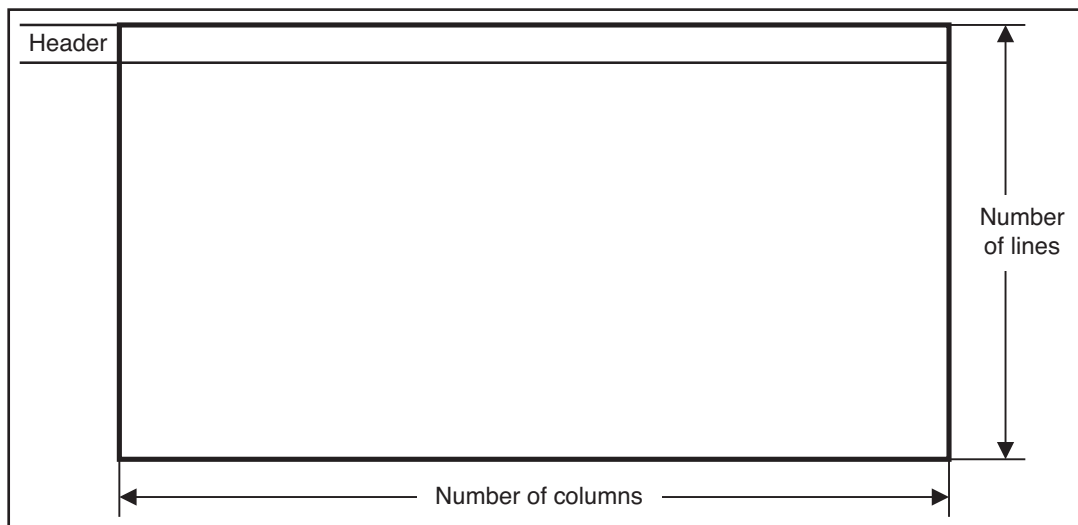
When using start-time options, specify the number of lines with `-pl`, and number of columns with `-pw`.

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



| | Number of lines | Number of columns |
|---------------------------|-----------------|-------------------|
| Range of specified values | 0.20 to 255 | 80 to 1023 |
| Initial value | 60 | 100 |

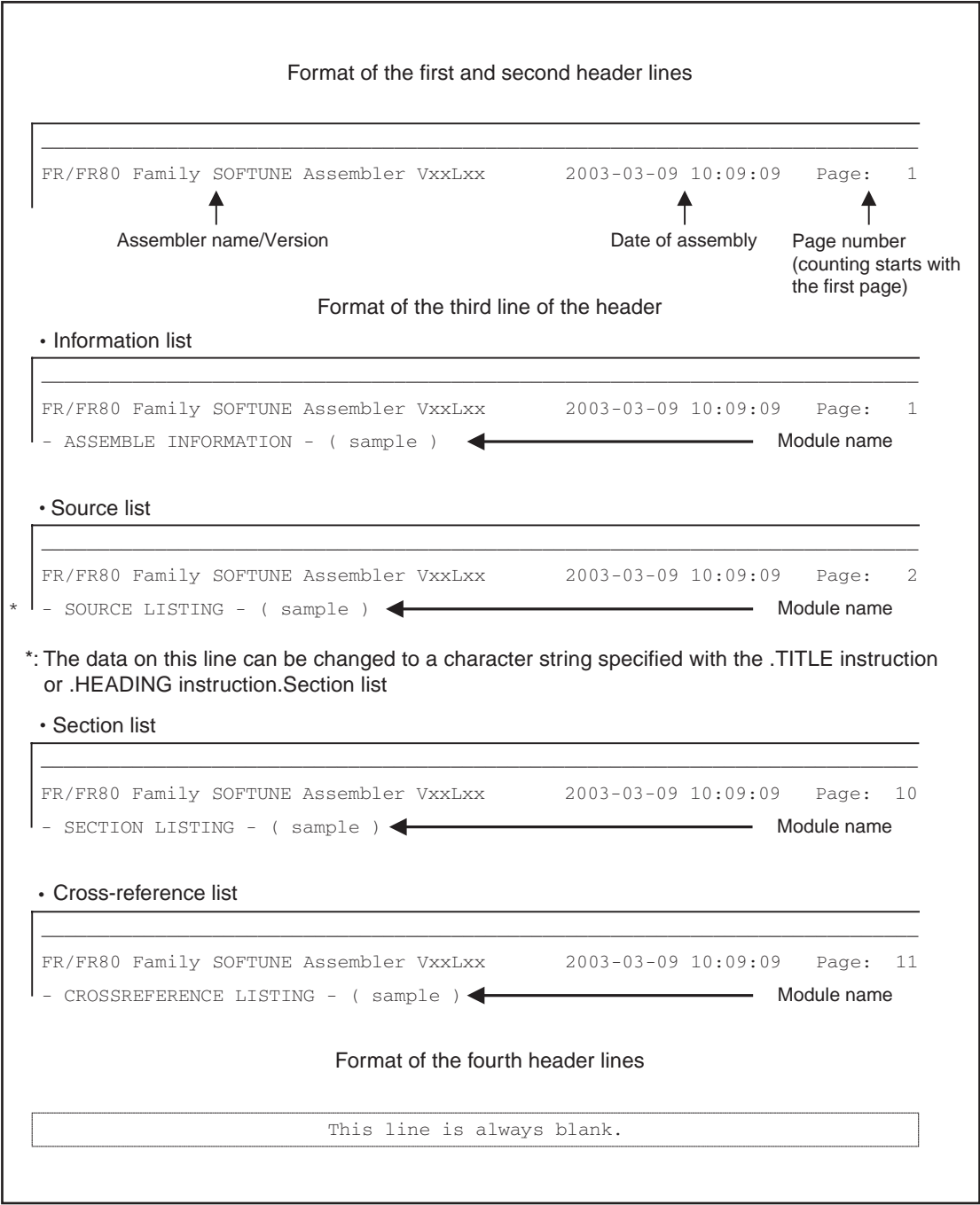
Note :

When 0 is specified as the number of lines, there are no page breaks.

■ Header Format

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

Figure 6.2-2 Header Format



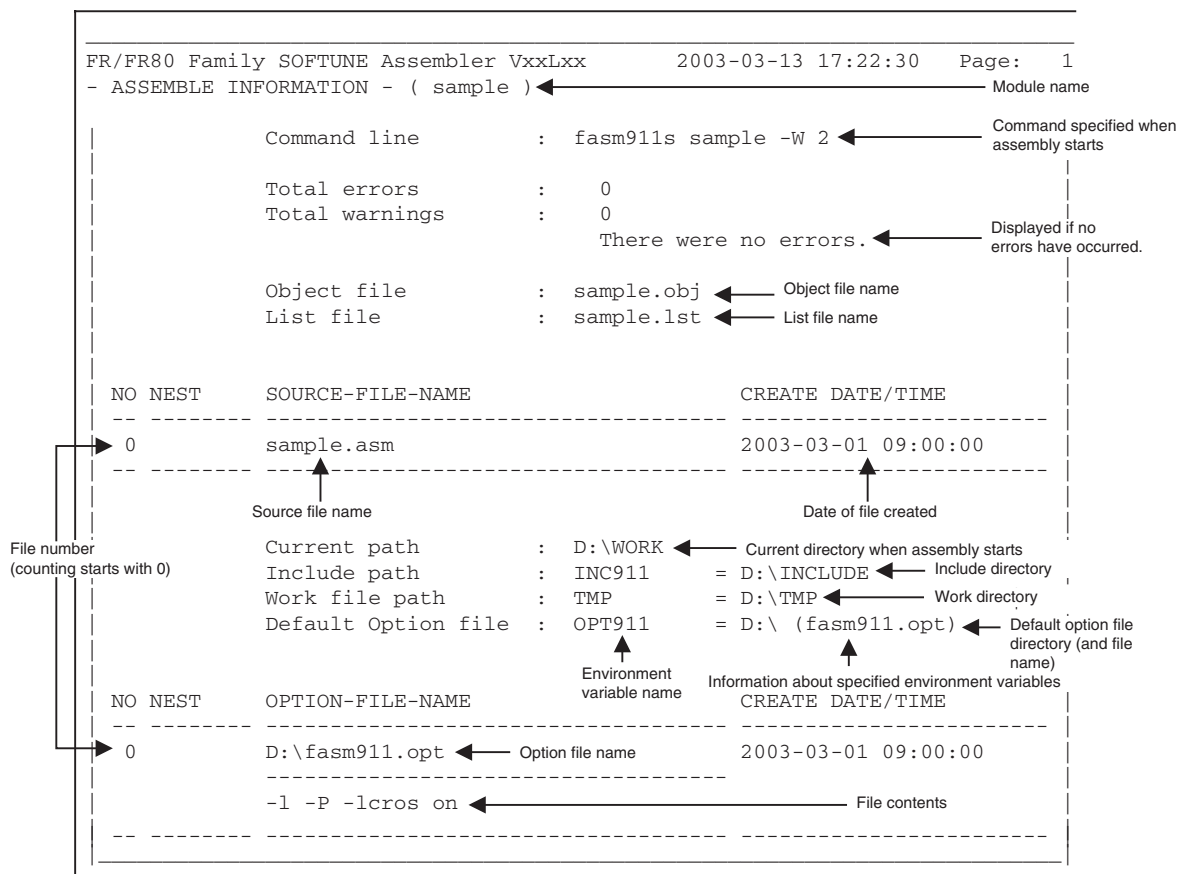
6.3 Information List

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 [Form 1]



[Format 2]

Figure 6.3-2 Information List [Form 2]

FR/FR80 Family SOFTUNE Assembler VxxLxx2003-03-13 17:22:30Page: 1

- ASSEMBLE INFORMATION - (sample)Module name

Command line : fasm911s -l sample -Xg

** Total errors : 2 (First Line: 22)

** Total warnings : 3 (First Line: 7)

Object file : -None-

List file : sample.lst

Asterisks are displayed if any errors or warnings have occurred.

If any errors or warnings have occurred, the number of each and the list line number of the line of the first occurrence for the error or warning is found are displayed.

| 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 nets (The number of asterisks represents the nesting depth.)

Current path : D:\WORK

Include path : INC911 = -None- = D:\TOOL\LIB\911\INCLUDE\

Work file path : TMP = D:\TMP

Default Option file : OPT911 = -None- = D:\TOOL\LIB\911\ (fasm911.opt)

If no option file errors exist, no file contents are displayed.

88

PART 1 OPERATION

6.4 Source List

The source list consists of 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

| FR/FR80 Family SOFTUNE Assembler VxxLxx | | | 2003-03-13 17:22:30 | Page: 2 |
|-----------------------------------------|----------|-------------------|---------------------|------------------------------|
| - SOURCE LISTING - (sample) | | | | |
| SN | LOC | OBJ | LLINE | SOURCE |
| <sample.asm> | | | ===== | |
| DA | 00000000 | -----<DATA>----- | 1 | .SECTION DATA, DATA |
| DA | 00000000 | 00ABCDEF 00000123 | 2 | .DATA.W 0xABCDEF, 0x123 |
| DA | 00000008 | 0020 0010 | 3 | .DATA.H 32, 16 |
| CO | 00000000 | -----<CODE>----- | 4 | .SECTION CODE, CODE, ALIGN=4 |
| CO | 00000000 | 9FA0 | 5 | NOP |
| CO | 00000002 | 9FA0 | 6 | NOP |
| CO | 00000004 | 9FA0 | 7 | NOP |
| | | | 8 | .IMPORT imp |
| CO | 00000006 | 9F8100000000 | 9 | LDI #rel, R1 |
| CO | 0000000C | 9F8300000000 | 10 | LDI #imp, R3 |

↑

First two characters of the section name

↑

Location value
(hexadecimal number)

↑

Object code value
(hexadecimal number)

↑

The attribute of the value included in the object code.
The order of priority for the displayed attribute is as follows.
I: External reference value
S: Section value
R: Relative value
Blank: Absolute value

↑

List line number
These numbers correspond to
the list line numbers in the object file.
: 10 decimal digits.

↑

Source program
If the source program
will not fit on one line,
it is displayed on two lines or more.

6.4.1 Preprocessor and Optimization Code Check Processings

If any operation is performed on a line by the preprocessor or by the optimization code check functions, a symbol is displayed for the line.

■ Preprocessor and Optimization Code Check Processings

Figure 6.4-2 Preprocessor and Optimization Code Check Processings

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|----------|-------|----------------------|
| : | : | : | : | : |
| | | | 6 | #ifdef DEF1 |
| | | | 7 | X .DATA 0 /* then */ |
| | | | 8 | #else |
| DA | 00000004 | 00000001 | 9 | .DATA 1 /* then */ |
| | | | 10 | #endif |

Information about operations performed by the preprocessor or optimization code check functions

Blank: Normal

Preprocessor

- X: The line has not been assembled.
- &: Macro expansion is performed for this line.

Optimization code check

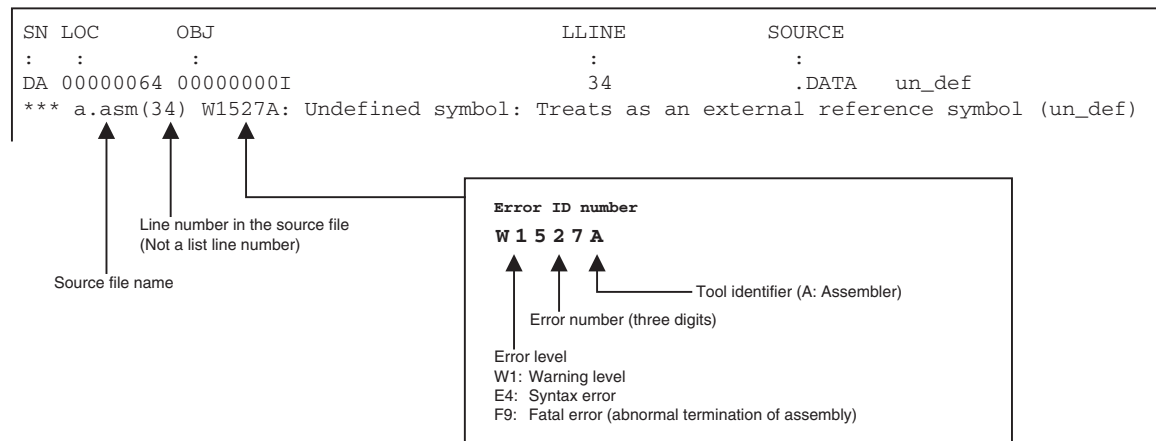
- x: The instruction has been deleted by optimization.
- C: The instruction has been replaced with another instruction by optimization.
- O: A new instruction has been created by optimization.
- V: The order of this and the next instruction have been switched by optimization.
(This symbol always appears with A.)
- A: The order of this and the preceding instruction have been changed by optimization.
(The symbol always appears with V.)

6.4.2 Error Display

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

■ Error Display

Figure 6.4-3 Error Display

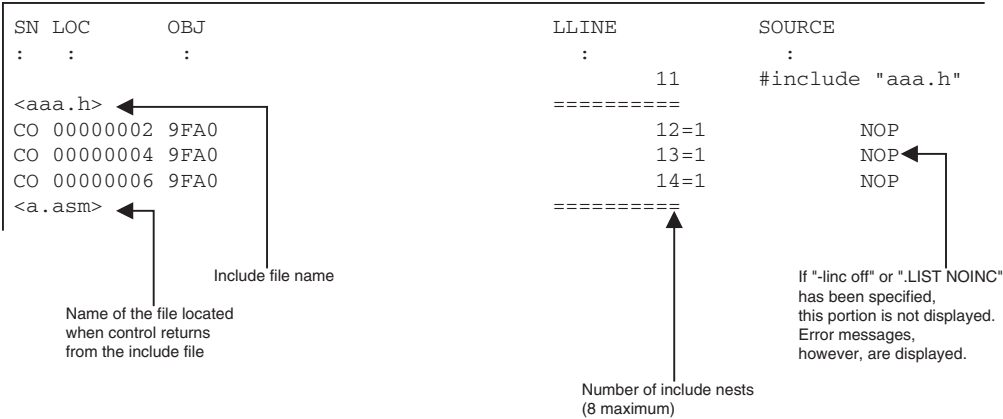


6.4.3 Include File

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



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

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-------------|-------|------------------------------|
| : | : | : | : | : |
| | | | 25 | .END |
| or | | => 00000300 | 25 | .END 0x300 /*start address*/ |
| or | | => 00000010 | 25 | .END start /*start address*/ |

If specified, a start address is displayed in hexadecimal.
 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

| 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 in folding style

■ .SECTION

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|------------------|-------|----------------------------------|
| : | : | : | : | : |
| SE | 00000000 | -----<SEC1>----- | 4 | .SECTION SEC1, CODE, ALIGN=4 |
| : | : | : | : | : |
| SE | 00000010 | -----<SEC2>----- | 19 | .SECTION SEC2, CODE, LOCATE=0x10 |

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

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

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|----------------------|-------|----------|
| : | : | : | : | : |
| CO | 00000004 | - 00000001 [3] < [4] | 16 | .ALIGN 4 |

Location after the ALIGN instruction

Location before the ALIGN instruction

Specified ALIGN value (decimal number)

Offset value to the ALIGN boundary (decimal number)

■ .ORG

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|-----|-------|------------|
| : | : | : | : | : |
| CO | 00000500 | | 29 | .ORG 0x500 |

Location after the .ORG instruction

■ .SKIP

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|-----------------|-------|----------|
| : | : | : | : | : |
| CO | 0000002A | - 00000020 [10] | 25 | .SKIP 10 |

Location after the SKIP instruction

Location before the SKIP instruction

Specified SKIP value (decimal number)

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

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|--------------------|
| : | : | : | : | : |
| | | | 21 | .EXPORT exp1, exp2 |

■ .GLOBAL

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|--------------------|
| : | : | : | : | : |
| | | | 8 | .GLOBAL imp1, exp1 |

■ .IMPORT

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|--------------|
| : | : | : | : | : |
| | | | 6 | .IMPORT imp1 |

6.4.7 .EQU, .REG

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

- **.EQU:** Assigns a value to a symbol.
- **.REG:** Assigns a value to a register list symbol.

■ .EQU

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|------------|-------|-------------------|
| : | : | : | : | : |
| | | = 00000100 | 7 | sym01: .EQU 0x100 |

↑
Set symbol value (hexadecimal number)

■ .REG

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|----------|-------|---------------------------|
| : | : | : | : | : |
| | | = (3E3F) | 3 | reg1: .REG (r0-r5,r9-r13) |

↑
Register list value set as register symbol value
(hexadecimal number)

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

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

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

■ .DATA

| SN | LOC | OBJ | | LLINE | SOURCE |
|----|----------|----------------------|------------------------------------------------------------------------------------------------------|-------|--------------------|
| : | : | : | This indicates that boundary alignment has been performed. The format is the same as that of .ALIGN. | : | : |
| SE | 00000000 | 10 | | 8 | .DATA.B 0x10 |
| SE | 00000001 | 20 10 | | 9 | .DATA.B 0x20,16 |
| SE | 00000003 | 00R 00S | | 10 | .DATA.B REL01,SEC1 |
| SE | 00000006 | - 00000005 [1] < [2] | | 11 | .DATA.H 0x10 |
| SE | 00000006 | 0010 | | | |
| SE | 00000008 | 0020 0010 | | 12 | .DATA.H 32,16 |
| SE | 0000000C | 0000R 0000S | | 13 | .DATA.H REL01,SEC1 |
| SE | 00000010 | 00000010 | | 14 | .DATA.L 0x10 |
| SE | 00000014 | 00000020 00000010 | | 15 | .DATA.L 0x20,0x10 |
| SE | 0000001C | 00000000R 00000000S | | 16 | .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

■ .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.

■ .DATAB

| SN | LOC | OBJ | This indicates that boundary alignment has been performed. The format is the same as that of .ALIGN. | | LLINE | SOURCE | |
|----|----------|----------------------|------------------------------------------------------------------------------------------------------|--|-------|----------|---------|
| : | : | : | | | : | | |
| SE | 00000000 | [2] 10 | | | 8 | .DATAB.B | 2,0x10 |
| SE | 00000002 | [16] 0020 | | | 9 | .DATAB.H | 16,0x20 |
| SE | 00000024 | - 00000022 [2] < [4] | | | 10 | .DATAB.L | 2,0x10 |
| SE | 00000024 | [2] 00000010 | | | | | |
| SE | 0000002C | [2] 00000000 | | | R 11 | .DATAB.L | 2,REL01 |
| SE | 00000034 | [2] 00 | | | S 12 | .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 constant blocks (floating-point numbers).

■ .FDATA

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|-------------------|-------|-----------------------------|
| : | : | : | : | : |
| SE | 00000000 | 3F800000 40133333 | 6 | .FDATA.S 0r1.0,0r2.3 |
| SE | 00000008 | 3FF0000000000000 | 7 | .FDATA.D 0r1.0 |
| SE | 00000010 | 11112222 | 8 | .FDATA.S 0x11112222 |
| SE | 00000014 | 1111222233334444 | 9 | .FDATA.D 0x1111222233334444 |

↑
Location

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

Note :

"00000024 - 00000022 [2] < [4]" indicates that boundary alignment has been performed. The format is the same as that of .ALIGN.

■ .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

| SN | LOC | OBJ | L | LINE | SOURCE |
|----|----------|----------------------|---|------|-------------------|
| : | : | : | : | : | : |
| SE | 00000000 | [2] 3F800000 | | 8 | .FDATAB.S 2,0r1.0 |
| SE | 00000008 | [2] 3FF0000000000000 | | 9 | .FDATAB.D 2,0r1.0 |

Location

Repeat count
(decimal number)

Data value
(hexadecimal number)

Note :

"00000024 - 00000022 [2] < [4]" indicates that boundary alignment has been performed. The format is the same as that of .ALIGN.

6.4.10 .RES, .FRES

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

- **.RES:** Area definition instruction (no data values: integer)
- **.FRES:** Area definition instruction (no data values: floating-point number)

■ .RES

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|-------------------|-------|--------------|
| : | : | : | : | : |
| SE | 00000000 | [2]B | 8 | .RES.B 2 |
| SE | 00000002 | [5]H | 9 | .RES.H 5 |
| SE | 00000010 | - 0000000C[4]<[8] | 10 | .RES.L 3+1-1 |
| SE | 00000010 | [3]L | | |

This indicates that boundary alignment has been performed. The format is the same as that of .ALIGN.

Location

Repeat count (decimal number)

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

■ .FRES

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|------|-------|-----------|
| : | : | : | : | : |
| SE | 00000000 | [2]S | 8 | .FRES.S 2 |
| SE | 00000008 | [2]D | 9 | .FRES.D 2 |

Location

Repeat count (decimal number)

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

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

| SN | LOC | OBJ | LLINE | SOURCE |
|----|----------|-------------------------------|-------|--------------------------------------|
| : | : | : | : | : |
| SE | 00000000 | 61 62 63 64 65 66 67 68 69 6A | 6 | .SDATA "abcdefghijkmnopqrstuv\ |
| | | 6B 6C 6D 6E 6F 70 71 72 73 74 | 7 | wxy1234567890" /*continuation-line*/ |
| | | 75 76 09 77 78 79 31 32 33 34 | | |
| | | 35 36 37 38 39 30 | | |
| SE | 00000024 | 31 32 33 FF 31 32 33 | 8 | .SDATA "123\xff123", "12345\t\n" |
| SE | 0000002B | 31 32 33 34 35 09 0A | | |
| SE | 00000039 | 31 32 33 09 31 32 33 | 9 | .SDATA "123\t123", " ", "\"", \" |
| SE | 00000039 | | 10 | "1234567890" |
| SE | 00000039 | 22 | | |
| SE | 0000003A | 31 32 33 34 35 36 37 38 39 30 | | |
| SE | 00000044 | | 11 | .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

| SN | LOC | OBJ | Repeat count (decimal number) | LLINE | SOURCE |
|----|----------|-----|-------------------------------|-------|----------------------------------------|
| : | : | : | : | : | : |
| SE | 000000A0 | [2] | | 7 | .SDATAB 2,"" /*null character-string*/ |
| SE | 000000A0 | [5] | 31 32 33 34 35 36 37 38 | 8 | .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

| 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**

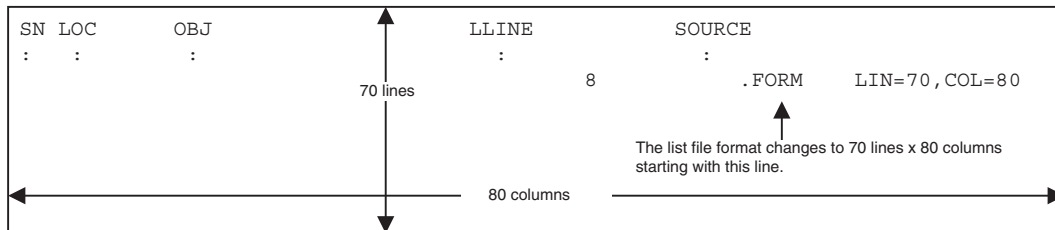
| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|------------------------|
| : | : | : | : | : |
| | | | 10 | .LIBRARY "library.lib" |

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:** displaying blank lines.

■ .FORM



■ .TITLE

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|-------------------------|
| : | : | : | : | : |
| | | | 6 | .TITLE "SAMPLE PROGRAM" |

The title is displayed on all pages starting with the first page.

| FR/FR80 Family SOFTUNE Assembler VxxLxx | | | | 2003-03-09 10:09:09 Page: 2 |
|-----------------------------------------|-----|-----|-------|----------------------------------------|
| SAMPLE PROGRAM | | | | Character string specified with .TITLE |
| SN | LOC | OBJ | LLINE | SOURCE |

■ .HEADING

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|---------------------------|
| : | : | : | : | : |
| | | | 6 | .HEADING "SAMPLE PROGRAM" |

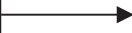
Page break occurs at the line where .HEADING is described.

| FR/FR80 Family SOFTUNE Assembler VxxLxx | | | | 2003-03-09 10:09:09 Page: 2 |
|-----------------------------------------|-----|-----|-------|------------------------------------------|
| SAMPLE PROGRAM | | | | Character string specified with .HEADING |
| SN | LOC | OBJ | LLINE | SOURCE |

■ .LIST

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|----------------------------|
| : | : | : | : | : |
| | | | 6 | |
| | | | 7 | ; .LIST OFF is on the next |
| | | | | line (eighth line). |
| | | | 10 | .LIST ON |
| | | | 11 | |

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.)



■ .PAGE


| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|-----------------------------|
| : | : | : | : | : |
| | | | 5 | ; .PAGE is on the next line |
| | | | | (sixth 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

| SN | LOC | OBJ | LLINE | SOURCE |
|----|-----|-----|-------|---------------------------|
| : | : | : | : | : |
| | | | 5 | ; .SPACE 2 is on the next |
| | | | | line (sixth line). |
| | | | | |
| | | | 7 | ; |

The specified number of blank lines are created.



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

| FR/FR80 Family SOFTUNE Assembler VxxLxx | | | 2003-03-09 10:09:09 | | Page: 10 |
|-----------------------------------------|-----------------|----------|---------------------|-----|-----------------|
| - SECTION LISTING - (sample) | | | Module name | | |
| NO | SECTION-NAME | SIZE | ATTRIBUTES | | |
| 0 | sec1 | 00000008 | CODE | REL | ALIGN=2 |
| 1 | sec02 | 00000008 | CODE | REL | ALIGN=2 |
| 2 | on | 00000004 | CODE | REL | ALIGN=2 |
| 3 | sec05 | 0000006C | CODE | REL | ALIGN=2 |
| 4 | sec06 | 00000020 | DATA | REL | ALIGN=4 |
| 5 | sec07 | 00000004 | CONST | REL | ALIGN=4 |
| 6 | sec08 | 00000004 | COMMON | REL | ALIGN=4 |
| 7 | sec09 | 00000012 | STACK | REL | ALIGN=4 |
| | sec10 | 00000010 | DUMMY | | |
| 8 | sec11 | 00000004 | CODE | REL | ALIGN=4 |
| 9 | sec12 | 00000400 | CODE | ABS | LOCATE=00000100 |
| 10 | sec13 | 00000000 | CODE | REL | ALIGN=4 |
| 11 | sec14 | 00000010 | DATA | ABS | LOCATE=00000000 |
| 12 | sec15 | 00000010 | CONST | REL | ALIGN=4 |
| 13 | sec16 | 00000020 | COMMON | ABS | LOCATE=00000010 |
| 14 | sec17 | 00000028 | STACK | REL | ALIGN=4 |

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

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

Section size
(hexadecimal number)

Section type

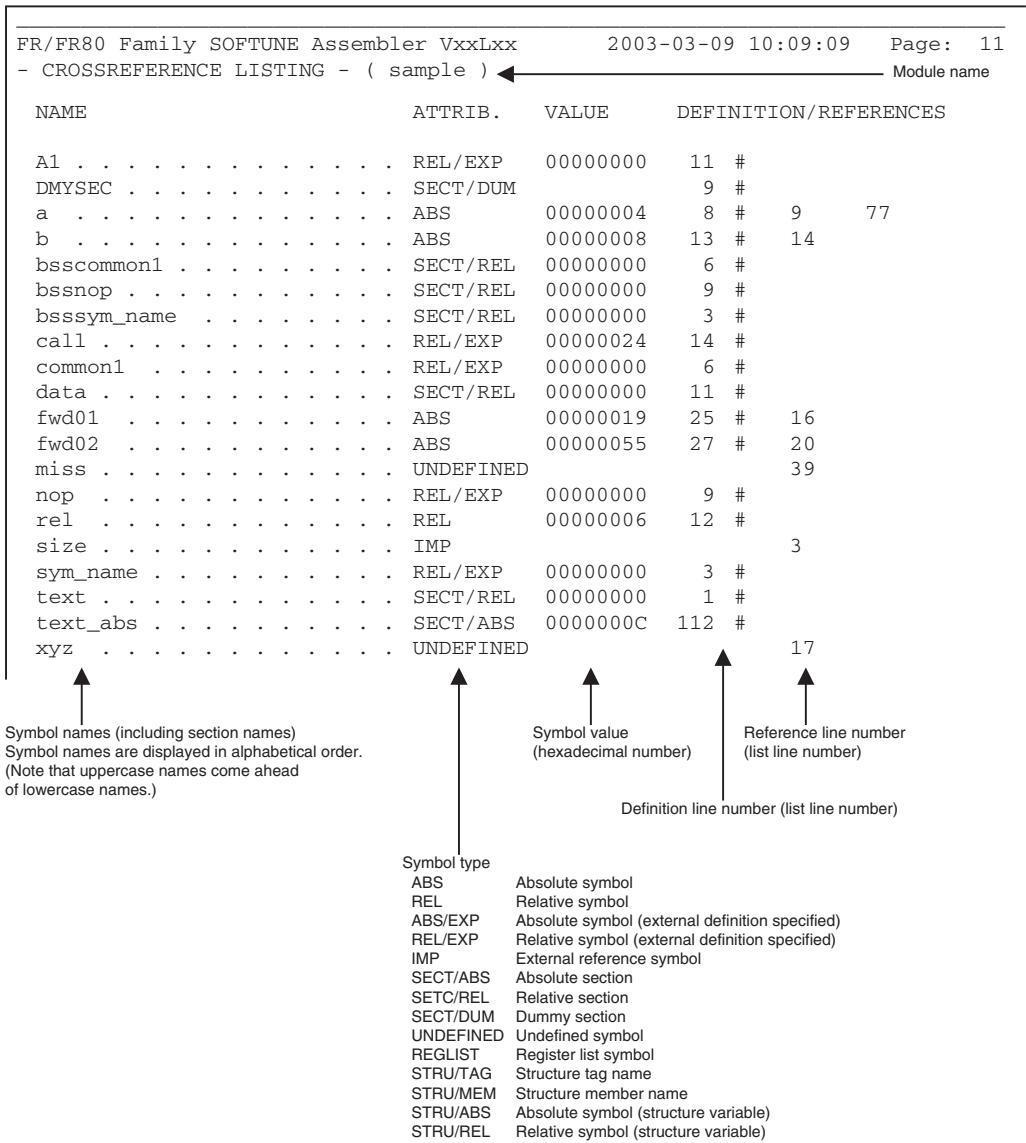
Section placement format
ALIGN values are displayed for relative sections.
LOCATE values are displayed for absolute sections.
Blanks are displayed for dummy sections.

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



PART2 SYNTAX

Part 2 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 7

BASIC LANGUAGE RULES

This chapter describes writing and rules that create the program using the 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 Register Lists
- 7.13 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

■ 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.

■ 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
LD @ (R14, 4), 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
-----*/
        ST    RP, @-SP    ; Comment
        ENTER /* Comment */ #20 // 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   "abcdefghijklmnopqrstuvwxy \
ABCDEF GHIJKLMNOPQRSTUVWXYZ" /* String continuation */
```

7.2 Character Set

The following 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 |

Notes:

- 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, register list symbols, and macro names are used.

The following are reserved words:

- **Register names:** These vary with the architecture.
 - **Size operators :** `SIZEOF`
-

■ Naming Rules

- 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 separated the uppercase from lowercase.

■ 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.

● Register list symbols

Register list symbols are defined with the `.REG` instruction. They represent register lists.

They are assigned to symbols.

● 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 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 R15, AC, FP, SP, PS, TBR, RP, SSP, USP, MDH, MDL, CR0 to CR15 |
|---------------------------------------------------------------------|

● Operators

SIZEOF

● Other reserved names

The following 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 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.
- When a forward reference symbol is used as an operand for a machine instruction, instruction code is usually generated with the maximum number of bits for the range of selectable bits. For example, when `LDI # forward-reference-symbol,Ri` is written, `LDI:32` is selected. In most of these cases, however, even though the code contains forward reference symbols, the optimum code is generated because the assembler performs optimization for forward reference symbols.

For more information, see Section "5.1.2 Forward Reference Symbol Optimization Function".

■ 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 can use either uppercase or lowercase.

[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 can use either uppercase or lowercase.

[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 can use either uppercase or lowercase.

[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 can use either uppercase or lowercase.

[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]

```
BEQ    $+4
CALL   label-$
```

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, extended representation, 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' "
```

- Extended representation

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

Character constants of this type are called extended representation.

Table 7.7-1 lists the extended representation.

Table 7.7-1 Extended Representation

| 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 lowercase letters can be used in extended representation.

[Example]

```
'\n'   '\n'   '\n\\'
```

- Octal representation

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

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

[Example]

| Character constant | Bit pattern |
|--------------------|------------------------------------------|
| '\0' | b'00000000 |
| '\377' | b'11111111 |
| '\53' | b'00101011 |
| '\0123' | b'00001010 → Divided into '\012' and '3' |

- Hexadecimal representation

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

Each hexadecimal representation starts with a backslash (\), and contains character x (lowercase) one or two hexadecimal digits.

[Example]

| Character constant | Bit pattern |
|--------------------|------------------------------------------|
| '\x0' | b'00000000 |
| '\xff' | b'11111111 |
| '\x2B' | b'00101011 |
| '\x0A5' | b'00001010 → Divided into '\x0A' and '5' |

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 representation (\").

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

[Example]

```
"ABCD\n"
"012345\n\0"
"\xff Fujitsu\tMicroelectronics\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]</code> | <code>[+ -]</code> | <code>{.d d [.[d]]}</code> | <code>[e [[+ -] d]]</code> | |
| <code>[F']</code> | <code>[+ -]</code> | <code>{.d d [.[d]]}</code> | <code>[e [[+ -] d]]</code> | <code>d is a decimal number.</code> |

[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 is a hexadecimal number.</code> |

[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.

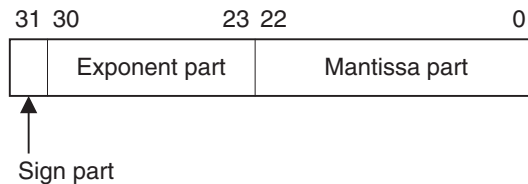
The following two data formats are provided for floating-point constants:

- 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.

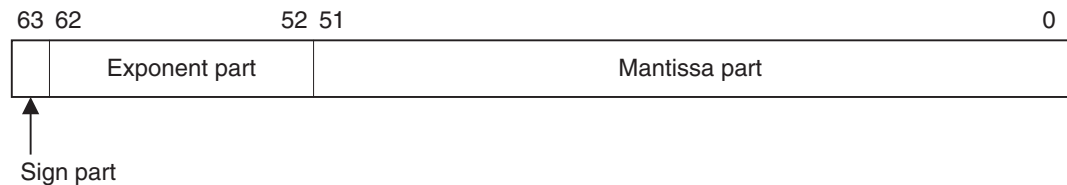
■ Data Format of Single-precision Floating-point Constants

The data format of single-precision floating-point constants includes 1 bit in the sign part, 8 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 1 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 notation 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 expressions are not supported.

■ Expression Types

Expressions are calculated 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 of 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.

■ 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.

■ 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.

7.11.2 Range of Operand Value

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. (Refer to Section "4.8.10 -OVFW, -XOVFW".)

This section explains the range of the operand value.

■ Range of Operand Values

Table 7.11-1 lists examples of operand value range.

If the operational result is outside of the range in the table below (Table 7.11-1), the assembler will display a warning or an error.

For details on each operand, see the Hardware Manual.

Table 7.11-1 Example Ranges of Operand Values

| Operand Types | Range of Values |
|----------------------------------------|------------------------------------------------------------------|
| 4-bit immediate value (i4) | 0 to 15 (0 when expanded), or -16 to -1 (negative when expanded) |
| Signed 8-bit immediate value (i8, s8) | -128 to 255 |
| Signed 10-bit immediate value (s10) | -512 to 508 |
| Signed 20-bit immediate value (i20) | -524288 to 1048575 |
| Signed 32-bit immediate value (i32) | -2147483648 to 4294967295 |
| Unsigned 4-bit immediate value (u4) | 0 to 15 |
| Unsigned 8-bit immediate value (u8) | 0 to 255 |
| Unsigned 10-bit immediate value (u10) | 0 to 1020 |
| Unsigned 6-bit address value (udisp6) | 0 to 60 |
| Signed 8-bit address value (disp6) | -128 to 127 |
| Signed 9-bit address value (disp9) | -256 to 254 |
| Signed 10-bit address value (disp10) | -512 to 508 |
| Unsigned 8-bit address value (dir8) | 0 to 255 |
| Unsigned 9-bit address value (dir9) | 0 to 510 |
| Unsigned 10-bit address value (dir10) | 0 to 1020 |
| Signed 9-bit branch address (label9) | -128 to 254 |
| Signed 12-bit branch address (label12) | -2048 to 2046 |

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: `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 following 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".

- `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:

- **SIZEOF:** Size operator

■ Section Size Extraction (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 operation 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=4

.SECTION PROGRAM, CODE, ALIGN=4
LDI #SIZEOF(ABC), R1
LDI #SIZEOF(DMY), R2

.SECTION DATA, DATA, ALIGN=4
.DATA SIZEOF(PROGRAM), SIZEOF(DMY)

.SECTION DMY, DUMMY, ALIGN=4
```

■ 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)

■ 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 P,CODE,ALIGN=2
LDI #SIZEOF(S),R1        /* 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 7.11-2 shows the precedence of operators.

Table 7.11-2 Precedence of Operators

| Precedence | Operator | Associativity | Target expression |
|------------|-------------------|---------------|---------------------------|
| 1 | () | Left-to-right | Parenthesized expression |
| 2 | ! ~ + - 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_1: .data  imp & 0xff
```

7.12 Register Lists

In a register list, zero or more general-purpose registers can be defined. When defining two or more registers, separate them with a comma (.).

■ Register Lists

[Format]

`([register-specification [,register-specification] ...])`

register-specification: {register|range|register-list-symbol}

register: General-purpose register for each target architecture

range: register-register

register-list-symbol: Symbol to which a register list is assigned

[Description]

In a register list, zero or more general-purpose registers can be defined.

When defining two or more registers, separate them with a comma (.).

When a range is specified, register numbers must be listed in ascending order.

A register list symbol can be specified.

Before a register list symbol can be used, however, it must have already been defined.

[Example]

```

STM    (R1,R2,R7 to R9)
WKREG: .REG  (R1 to R4)
LDM    (WKREG)
LDM    (WKREG,R8,R10)
```

7.13 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 */ .DAT 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|IO|IOCOMMON}
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 8 section types, which are listed below.
By default, CODE is assumed.

- CODECode section
- DATAData section
- CONSTData section with initial values
- COMMONCommon section
- STACKStack section
- DUMMYDummy section
- IOI/O section
- IOCOMMON....Common I/O section

■ Section Allocation Patterns

One of the following section allocation patterns is specified:

● ALIGNRelative section

An allocation address in memory is determined by the linker.

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

● LOCATEAbsolute section

A section is allocated starting at the specified start address.

By default, ALIGN=2 is used.

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

[Example]

```
.SECTION PROG, CODE, ALIGN=8
/* Section name: PROG                */
/* Section type: Code section         */
/* Section attribute: Relative section (boundary value 8) */
:
.SECTION VAL, DATA, LOCATE=0x1000
/* Section name: VAL                  */
/* Section type: Data section          */
/* Section attribute: 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 it.

■ Section Types

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

The following explains general data types to be used for the section types:

● Code section (CODE specification)

Program code is stored.

Machine instructions are usually used.

[Example]

```
.SECTION PROG, CODE, ALIGN=2
start_program:
    LDI:32 #_data_init, R0
    CALL @R0
```

● Data section (DATA specification)

Data without initial values is stored.

The .RES and .FRES instructions are usually used.

[Example]

```
.SECTION VAL, DATA, ALIGN=4
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=4
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=4
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=4
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=256
        .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
        :
        DMOVH @iodata2,R13 /* Read a value from IO */
        :

```

Source file b.asm

```

#include "iodef.h" /* Read the IO definition file */
        .SECTION P,CODE,ALIGN=2
        :
        DMOVH R13,@iodata2 /* Write a value to IO */
        :

```

- 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.

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 |

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=2
__main:
    ST    RP, @-SP
    ENTER    #20
```

● 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=0x100
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 space.
In shared linkage, a memory space 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.

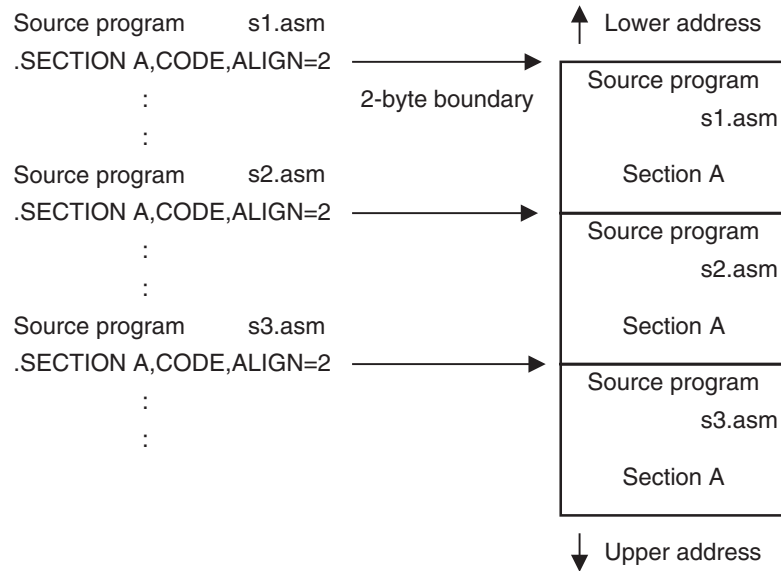
Table 8.5-1 Section Types and Linkage Methods

| Section type | Linkage method |
|--------------------------------------|----------------------|
| CODE DATA CONST STACK IO | Concatenated linkage |
| COMMON IOCOMMON | Shared linkage |

■ Concatenated Linkage

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

[Example]

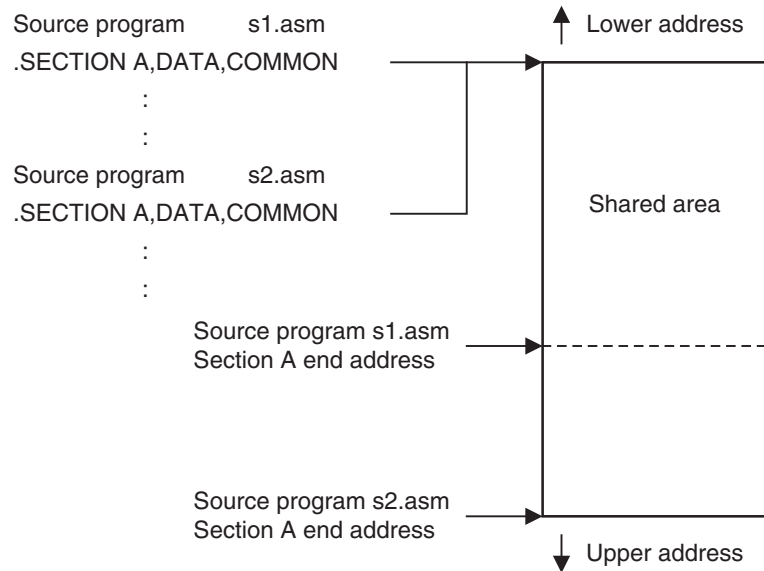


Shared Linkage

A memory space shared by sections having the same name, specified in different source programs, is created.

The size of this area is the size of the largest section.

[Example]



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

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 (0x10000 to 0x1ffff) and a RAM area (0x20000 to 0x2ffff). It also sets the `INIT` sections for ROM-to-RAM transfer.

```
flnk911s -ro ROM=0x10000/0x1ffff -ra RAM=0x20000/0x2ffff
        -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 0x10000 and 0x20000, 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 /* Section for initial-value data that is allowed to 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:
        LDI    #_ROM_INIT,R0      /* Transfer source (ROM) address */
        LDI    #_RAM_INIT,R1      /* Transfer destination (RAM) address */
        LDI    SIZEOF(INIT),R13    /* Obtain the INIT section size */
        CMP    #0,R13             /* Compare the transfer size for 0 */
        BEQ:D  init_copy_end
init_copy_loop:
        ADD    #-1,R13            /* Reduce the size (R13) by 1 */
        LDUB   @(R13,R0),R12      /* Obtain the ROM data */
        BNE:D  init_copy_loop     /* Repeat until the size (R13) becomes 0 */
        STB    R12,@(R13,R1)     /* Transfer to RAM (delayed instruction) */
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 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 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]

```
ST      RP, @-SP
ENTER   #20
CALL32  _proc0, R12
```

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 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:

● Operation instructions

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

First operand .op. second operand -> second operand

[Example]

ADD R2,R5 /* R5 + R2 --> R5 */

● Transfer instructions

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

First operand -> second operand

[Example]

MOV R2,R5 /* R2 --> 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 four different sizes: byte (8 bits), halfword (16 bits), long word (32 bits), word (32 bits).

If a size is not specified, a word is assumed.

■ Scope of Integer Constants Handled by Pseudo-instructions

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

- Byte (8 bits)
- Halfword (16 bits)
- Long word (32 bits)
- Word (32 bits)

If no size is specified, one word is assumed.

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

Table 10.1-1 Size Specifiers

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

| | | |
|--|-----------------------|--------------------------|
| | <code>.PROGRAM</code> | <code>module-name</code> |
|--|-----------------------|--------------------------|

[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
```

■ Relationships 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 debugger 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]

```

        .PROGRAM   test_name
        .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]

| | | |
|--|-----------------|----------------------------------------------|
| | .SECTION | section-name[,specification[,specification]] |
|--|-----------------|----------------------------------------------|

- specification : {section-type|section-location-format}
- section-type : {CODE|DATA|CONST|COMMON|STACK|DUMMY|IO|IOCOMMON}
- 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-format can be specified in one **.SECTION** instruction.

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

If the section placement format is omitted, **ALIGN = 2** is specified.

■ 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.

■ 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=4
:
.SECTION D, DATA, LOCATE=0x1000
:
.SECTION C, CONST, LOCATE=0x2000
:
.SECTION V, COMMON, ALIGN=4
:
```

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]

| | | |
|--|---------------|----------------|
| | .ALIGN | 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.

The expression must be an absolute expression.

The following restrictions are placed on the absolute expression.

- The value must be 2 raised to the nth power (where n is an integer) and not greater than 0x80000000.

[Condition]

| |
|--------------------------------------------------------------------------------------|
| Boundary value in .SECTION instruction \geq boundary value in .ALIGN |
|--------------------------------------------------------------------------------------|

(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]

| | | |
|--|-------------------|------------|
| | <code>.ORG</code> | 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 (`LOCATE` specification) 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 symbols 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 symbol 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 symbol defined in the program in which it is issued to be referred in other programs.

The symbol must be defined in the program that contains the **.EXPORT** instruction of interest.

The following two types of symbol can be specified in the **.EXPORT** instruction:

- symbol with an absolute value
- symbol with an address

No error is reported if identical symbol are specified.

[Examples]

```

-----Program 1 -----
        .EXPORT  abc1,abc2
        :
abc1:   .EQU      5*3
        :
abc2:   ADD       R1,R5
-----Program 2 -----
        .IMPORT  abc1,abc2
        :
        .DATA    abc1
        :
        .DATA    abc2
        :

```

10.4.2 .GLOBAL Instruction

The **.GLOBAL** instruction declares symbol 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 symbol 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 symbol can be specified in the **.GLOBAL** instruction:

- symbol with an absolute value
- symbol 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 symbol 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]

| | | |
|--|----------------|---------------------|
| | .IMPORT | symbol[,symbol] ... |
|--|----------------|---------------------|

[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:    ADD     R1,R5

```

10.5 Symbol Definition Instructions

A symbol definition instruction assigns a value to a symbol.

■ Symbol Definition Instructions

There are two different symbol definition instructions:

- `.EQU` : Assigns a value to a symbol.
- `.REG` : Assigns a value to a register list 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
```

10.5.2 .REG Instruction

The .REG instruction assigns the contents of a specified register list to a specified symbol.

Only general-purpose registers can be specified in the .REG instruction.

A symbol to which the contents of a register list is assigned is referred to as a register list symbol.

A register list symbol that has only one register defined is referred to as a single-register symbol.

■ .REG Instruction

[Format]

| | | |
|--------|------|----------------------------------------------------------|
| symbol | .REG | ([register-specification[,register-specification] ...]) |
|--------|------|----------------------------------------------------------|

register-specification: {register|range-specification|register-list-symbol}

register: Determined by the target MCU.

range-specification: register-register

register-list-symbol: Symbol to which the contents of a register list are assigned

[Description]

The .REG instruction assigns the contents of a specified register list to a specified symbol.

Only general-purpose registers can be specified in the .REG instruction.

A symbol to which the contents of a register list are assigned is referred to as a register list symbol.

A register list symbol that has only one register defined is referred to as a single-register symbol.

A single-register symbol can be used as a general-purpose register.

See Section "7.12 Register Lists", for how to write a register list symbol.

[Examples]

```
REGLIST: .REG    (R0,R7 to R11)
REG1:   .REG     (R1)
        LDM      (REGLIST)    /* Equivalent to LDM(R0,R7-R11) */
        LDI      #10,REG1     /* Equivalent to LDI#10,R1 */
```


10.6 Area Definition Instructions

An area definition instruction secures constants and areas in memory areas.

■ Area Definition Instructions

There are 17 different area definition instructions:

- `.DATA:` Defines an 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 32-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, .BYTE, .HALF, .LONG, and .WORD Instructions

The following area 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 **.BYTE** instruction secures memory areas each having a size of one byte.

The **.HALF** instruction secures memory areas each having a size of one half word.

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 specification (s): B..... Byte (8 bits)
 H Half word (16 bits)
 L Long word (32 bits)
 W Word (32 bits) <<default>>

[Description]

The **.DATA** instruction sets the values of specified expressions in memory areas each having the specified size (.s).

If a size specification (.s) is omitted, one word is assumed.

The expression specified in the instruction can be either absolute or relative expression.

[Examples]

```
.DATA.B 0x12, 0x23, 0xa3
.DATA   -1, 0xffffffff
```

■ .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 (32 bits).

The bit length of word is 32 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 specification (s): BByte (8 bits)
 HHalf word (16 bits)
 LLong word (32 bits)
 W Word (32 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 specification (.s) is omitted, one word is assumed.
Expression 1 must be an absolute expression.
Expression 2 can be either absolute or relative expression.

[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 area by a specified number (expression 1) of repetitions.

10.6.3 .FDATA, .FLOAT, and .DOUBLE Instructions

The following area 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 specification.

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 specification (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 specification (.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 the following definition:

```
.FDATA.S floating-point-constant[,floating-point-constant]...
```

■ .DOUBLE Instruction

[Format]

| | | |
|----------|---------|-------------------------------------------------------|
| [symbol] | .DOUBLE | floating-point-constant[,floating-point-constant] ... |
|----------|---------|-------------------------------------------------------|

[Description]

The .DOUBLE instruction secures memory areas each having the double-precision (8-byte) size.

This instruction is equivalent to the following definition:

.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.t | expression, 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 **.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, 0xFFFF00000
.FDATAB.S 12, 0r1.2e10
```

Note:

If the 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 area by a specified number (expression) of repetitions.

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): BByte (8 bits)
 HHalf word (16 bits)
 L.....Long word (32 bits)
 WWord (32 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>>

D..... Double-precision floating-point constant, 64-bit (8-byte)

[Description]

The **.FRES** instruction secures a specified number (expression) 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 area.

■ .SDATA Instruction

[Format]

| | | |
|----------|--------|-----------------------------------------|
| [symbol] | .SDATA | character-string[,character-string] ... |
|----------|--------|-----------------------------------------|

[Description]

The .SDATA instruction sets specified character strings in memory area.

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] | .ASCII | 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"

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | A | B | C | D | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|

Lower address

Upper address

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 area by a specified number (expression) of repetitions.

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
a:    .BYTE 0
b:    .WORD 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 be initialized, no expression need to be specified after the one corresponding to that member.

[Examples]

```

ABC:  .STRUCT
a:    .BYTE 0
b:    .WORD 2
ABC:  .ENDS
c:    ABC <0, 2>

```

■ Access to a Structure

[Format]

| |
|-------------------------------------------|
| <code>structure-symbol.member-name</code> |
|-------------------------------------------|

[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
a:    .BYTE 0
b:    .WORD 2
ABC:  .ENDS
c:    ABC <0,2>
      LDI:32 #c.a, R0
      LDUB @R0, R1
```

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 stop 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 symbol 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
```

Relationships with Startup Options

The `.DEBUG` instruction 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.

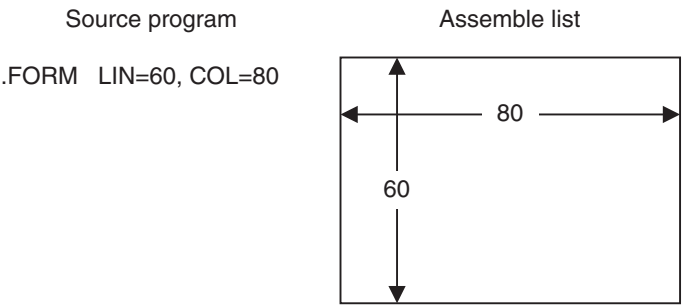
The number of lines per page can range between 20 and 255. If 0 is specified, no page eject occurs.

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]



■ Relationships 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

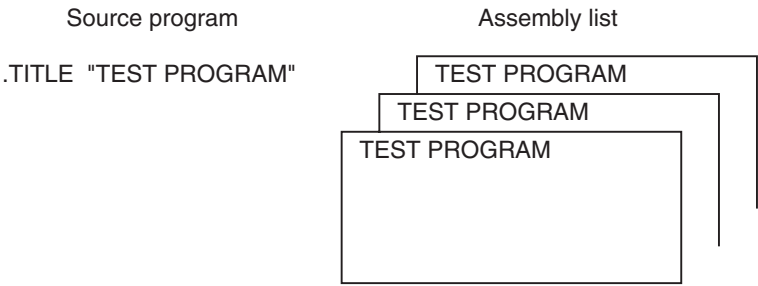
[Format]

| | | |
|--|--------|--------------|
| | .TITLE | "title-text" |
|--|--------|--------------|

[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]



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]

| Source program | Assembly list |
|--------------------------------------------------|-------------------------------------|
| <code>.HEADING "PROGRAM=TEST1\V1.0L2.0\""</code> | |
| <code>:</code> | <div>PROGRAM=TEST3 "V1.0L2.0"</div> |
| <code>.HEADING "PROGRAM=TEST2\V1.0L2.0\""</code> | <div>PROGRAM=TEST2 "V1.0L2.0"</div> |
| <code>:</code> | <div>PROGRAM=TEST1 "V1.0L2.0"</div> |
| <code>.HEADING "PROGRAM=TEST3\V1.0L2.0\""</code> | |
| <code>:</code> | |

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 itself is not output. The initial value is **.LIST ON,CALL,COND,DEF,EXPOBJ,INC**.

■ **.LIST Instruction**

[Format]

| | | |
|--|--------------------|-----------------------------------|
| | <code>.LIST</code> | specification[,specification] ... |
|--|--------------------|-----------------------------------|

- Specification: {ON|OFF}Specifies whether to output an assembly source list.
- {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.
- {STR|NOSTR}Specifies whether to output expansion text of structure control instruction 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,INC,STR**. 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.
- **STR:** Specifies to output expansion text of structure control instruction to the assembly source list.
- **NOSTR:** Specifies not to output expansion text of structure control instruction to the assembly source list.

[Examples]

```
.LIST ON
:           /* This portion is listed. */
.LIST OFF
:           /* This portion is not listed. */
.LIST ON
```

*:The term "nontext portion" refers to the if clause portion that is not a target for assembly.

■ Relationships 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/NOINC 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]

| | | |
|--|-------|--|
| | .PAGE | |
|--|-------|--|

[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]

| Source program | Assembly list | |
|----------------|---------------|----------|
| | | Page n |
| .DATA 10 | .DATA 10 | |
| .DATA 20 | .DATA 20 | |
| .PAGE | | |
| .DATA 30 | .DATA 30 | Page n+1 |
| .DATA 40 | .DATA 40 | |

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.

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 instruction.

- 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
- 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.

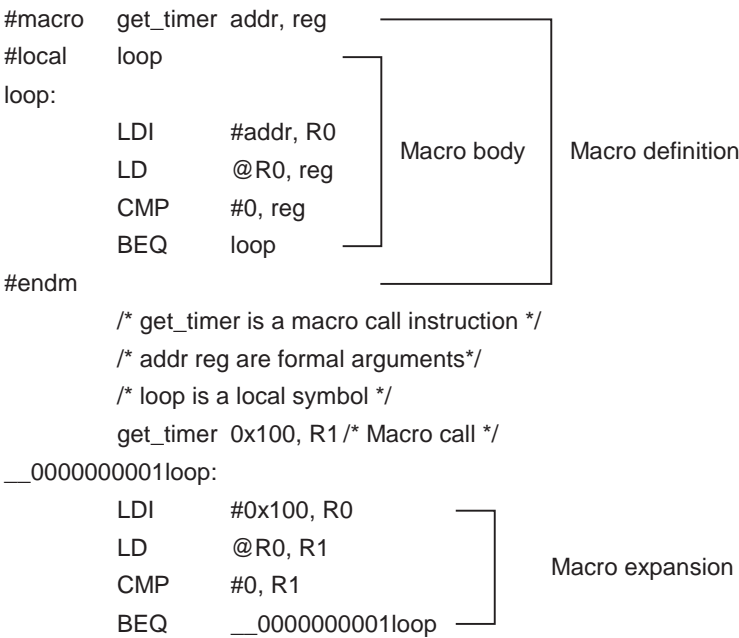
In these cases, macro definition is useful.

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]



● Conditional assembly

When, for example, an instruction is to be executed if a condition is true and another instruction is to be executed if it is false, 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:

```
LDI      #0xFF, R0
```

Alternatively, 0xFF can be defined for some descriptive name as follows:

```
#define  IOMASK, 0xFF
LDI      #IOMASK, R0
```

The latter is easier to understand.

The function that replaces the name IOMASK with 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 included by another file when necessary.

[Example]

File iodef.h

```
#define  IOMASK    0xFF    /* I/O mask value */
#define  SETCMD    1      /* Set data command */
:
```

File com.asm

```
#include "iodef.h"        /* Read defined values */
:
LDI  #IOMASK,R0
AND  R0,R2    /* Mask the data */
LDI  #SETCMD,R0
ST   r0,@R7   /* Send the Set Data 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 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 instruction). A macro call instruction can be used to set an argument for the formal argument.

■ Local Symbols

A symbol 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 replacement.

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 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   "abcdefghijklmnopqrstuvwxy \
ABCDEFHIJKLMNOPQRSTUVWXYZ" /* 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.

[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.

[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.

[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.

[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.

[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 |
|----------------------------|-----------------------------------------------|
| ' \x0' | b'00000000 |
| ' \xff' | b'11111111 |
| ' \x2B' | b'00101011 |
| ' \x0A5' | b'00001010 --> ' Divided into ' \x0A' and '5' |

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 must begin with an alphabetic character or an underscore (_).
- The second and subsequent characters of a macro name must be alphanumeric characters 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 call 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
/* max 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
    ST reg2, @-SP
    MOV reg1, reg2
#endm

mv R1, R2 /* Macro call */
ST R2, @-SP
MOV R1, R2

LD @+SP,reg1 /* Because this is outside the macro body, reg1 is not */
/* treated as a formal argument and is therefore 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:

__nnnnnnnnnn local-symbol

A local symbol begins with two underscores (_), which are followed by a 10-digit number.

The 10-digit number is incremented by 1 in the range from 0000000001 to 4294967295 each time a macro call is made.

The 10-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
loop:
    LDI    #addr, R0
    LD     @R0, reg
    CMP    #0, reg
    BEQ    loop
#endm

get_timer 0x100, R1 /* Macro call */
__0000000001loop:

    LDI    #0x100, R0
    LD     @R0, R1
    CMP    #0, R1
    BEQ    __0000000001loop
    BRA    loop    /* Because this is outside the macro body, loop is not treated */
                  /* as a local symbol and is therefore not replaced. */
```

loop is defined as a local symbol by the #loop instruction.

Macro body

Macro expansion

The loop portion is replaced by __0000000001loop.

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 Operation Precision

Expression operation is used 32-bit. Operation 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-operator 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
```

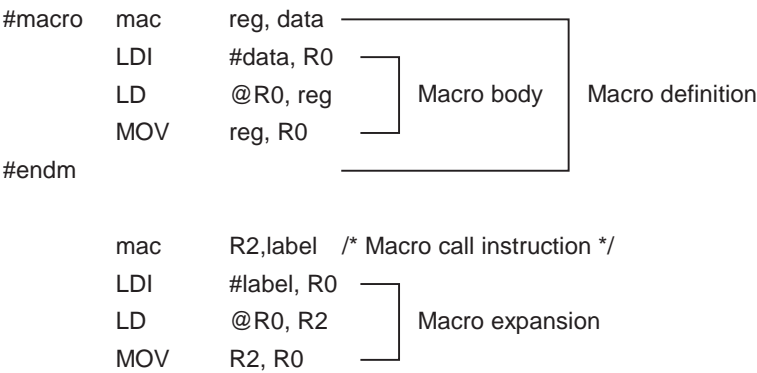
[Description]

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 name is expanded into the corresponding macro body.
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.

[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
LDI #data, r2
LD @r2, r1
MOV r1, r2
#endm

mac R2, R7, label /* Macro call instruction */
LDI #label, R7
LD @R7, R2
MOV R2, R7

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] ...
```

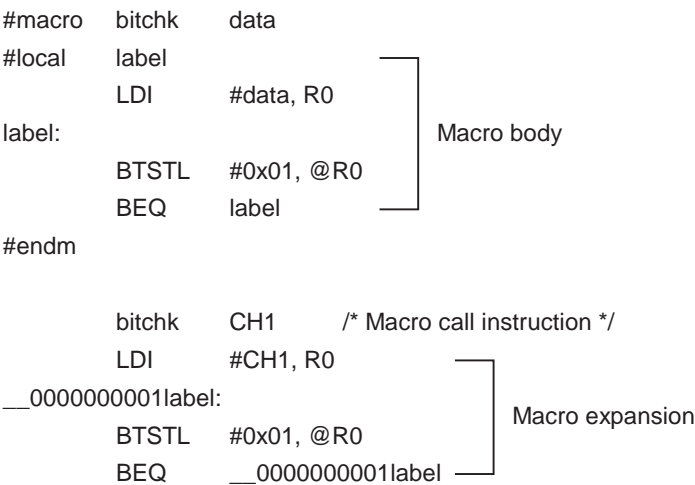
[Description]

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 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]



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

 LDI #cnt, R1

#endm

Macro body

mac 4 /* Macro call instruction */
NOP
LDI #4, R1

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 name 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 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 for the missing arguments.
- To specify a null character as an argument, write two commas (,,) or write a pair of angle brackets <>.

[Examples]

```
#macro  mac      a, b
        LD      a, b
        CALL    @b
#endm
```

```
mac      @R3, R1      /* Macro call instruction */
LD      @R3, R1
CALL    @R1           ☐ Macro expansion
```

```
mac      @(R14, 16), R1 /* Macro call instruction */
LD      @(R14, 16), R1
CALL    @R1           ☐ Macro expansion
```

```
mac      <@(20, SP)>, R5 /* Macro call instruction */
LD      @(20, SP), R5
CALL    @R5           ☐ Macro expansion
```

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]

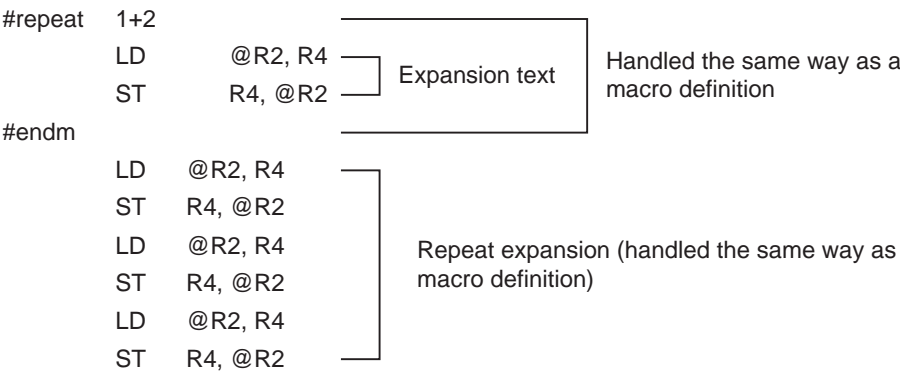
```
#repeat  iteration
        expansion-text
#endm
```

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.
If, therefore, the output of a macro definition is controlled with a list control instruction, repeat expansion is processed the same way.
The `#local` instruction cannot be used in expansion text.

[Example]



■ #repeat instruction

[Format]

#repeat Repeat expansion

iteration: Preprocessor expression

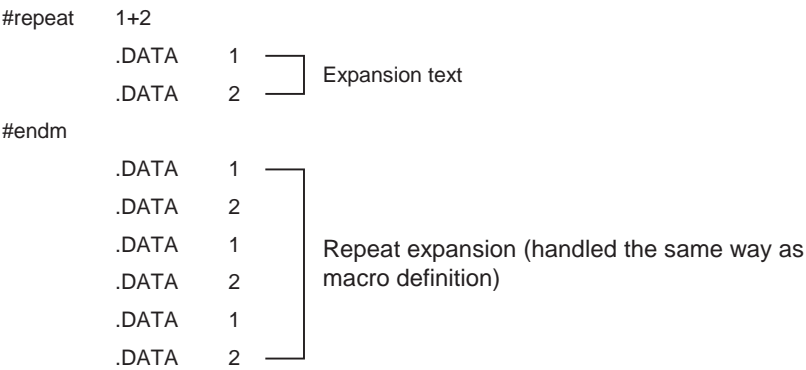
[Description]

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.

[Example]



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 an 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]

| |
|-----------------------------------------|
| <code>#if conditional-expression</code> |
|-----------------------------------------|

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 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 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 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 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 with the **#if**, **#ifdef**, or **#ifndef** is true, it becomes the inverted false and the text between this instruction and the corresponding **#endif** instruction is not assembled.

If the condition specified in the if clause with the **#if**, **#ifdef**, or **#ifndef** is false, it becomes the inverted true and the text between this instruction and the corresponding **#endif** instruction is assembled.

[Examples]

```
#define      NUM      3
#if          NUM == 3
    .SDATA    "ABC"    /* This is assembled. */
#else
    .SDATA    "DEF"
#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 is evaluated and assembly condition is determined again.

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.

Multiple **#elif** instructions can be written in the if clause.

[Examples]

```
#define      NUM      3
#if         NUM==1
    .SDATA    "ABC"

#elif      NUM==2
    .SDATA    "DEF"

#elif      NUM==3
    .SDATA    "GHI"      /* This is assembled */

#elif      NUM==4
    .SDATA    "JKL"

#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 is nested, each **#endif** instruction is valid for the corresponding **#if**, **#ifdef**, or **#ifndef** instruction only.

■ #endif Instruction

[Format]

```
#endif
```

[Description]

The **#endif** instruction indicates the end of conditional assembly.

If conditional assembly is nested, each **#endif** instruction is valid for the corresponding **#if**, **#ifdef**, or **#ifndef** instruction only.

[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 string defined for 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 instruction names
 - Characters in comments
 - Characters in character strings

[Examples]

```
#define ABC define
#ABC NUM 10 /* #ABC is not replaced by #define */
#define MSG SOFTUNE
/* MSG */ /* MSG is not replaced by SOFTUNE */
.SDATA"MSG" /* .SDATA "MSG" is not replaced by .SDATA "SOFTUNE" */
```

- 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 FPUstart: */
```

- 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.

[Example]

```
#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: argument-less **#define** instruction and parameter-attached **#define** instruction.

■ Argument-less #define Instruction

[Format]

```
#define macro-name character-string-to-be-defined
```

[Description]

The argument-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 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      DB      .DATA.B
#define      DW      .DATA.W
              DB      0,2
              DW      0xffffffff
```

■ Argument-attached #define Instruction

[Format]

```
#define macro-name(formal-argument[,formal-argument] ... )
character-string-to-be-defined
```

[Description]

The argument-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 string 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 necessary.

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 and 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 instruction deletes all macro names.

■ #purge Instruction

[Format]

#purge

[Description]

#purge instruction deletes all macro names.
This instruction is not valid for formal arguments and 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 reference 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 INC911 environment variable
3. Include directory in the development environment
 - %FETOOL%\LIB\911\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. First, 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 INC911 environment variable
4. Include directory in the development environment
 - %FETOOL%\LIB\911\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]

| |
|---------------------------------------|
| <code>#pragma character-string</code> |
|---------------------------------------|

[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 in the character string.

[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_FR__`

This macro name is replaced by the decimal constant 1.

It is effective only when MB number of FR family is specified as the -cpu option.

- `__CPU_FR80__`

This macro name is replaced by the decimal constant 1.

It is effective only when MB number of FR80 family is specified as the -cpu option.

■ 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]

```

        .DATA    defined(ABC)    /* .Replaced by .DATA0 */
#define ABC
        .DATA    defined(ABC)    /* .Replaced by .DATA1 */

```

Note:

The character "`__`" means two underscores (`__`).

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.

Note:

The character "`__`" means two underscores (`__`).

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

12.1 Assembler Pseudo Machine Instructions

Table 12.1-1 lists the assembler pseudo machine instructions.

■ Assembler Pseudo Machine Instructions

Table 12.1-1 contains the following items:

● Mnemonic

Mnemonics of assembler pseudo machine instructions are listed.

● Machine cycles

The number of machine cycles for each instruction is listed.

- a: This indicates a memory access cycle that may be prolonged with the Ready function.
- b: This indicates a memory access cycle that may be prolonged with the Ready function. When the next instruction refers a register that will be used for an LD operation, however, an interlock is applied, and the number of execution cycles is incremented by 1.
- c: When the next instruction reads from or write to R15, the SSP, or the USP, an interlock is applied, and the number of execution cycles is incremented by 1 and became 2.
- d: When the next instruction refers the MDH/MDL, an interlock is applied, and the number of execution cycles is incremented by 1 and became 2.

The minimum number of cycles is one for each of a, b, c, and d.

● Flag change

| Flag change | | Flag meaning | |
|-------------|-----------|--------------|---------------|
| C | Change | N | Negative flag |
| - | No change | Z | Zero flag |
| 0 | Clear | V | Overflow flag |
| 1 | Set | C | Carry flag |

● Operation

Instruction operations are listed.

Table 12.1-1 Assembler Pseudo Machine Instructions (1 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|---------------------|----------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| ADD #s5, Ri | 1 | C C C C | $Ri + s5 \rightarrow Ri$ When s5 is a negative value, the ADD2 instruction is specified. |
| ADDN #s5, Ri | 1 | - - - - | $Ri + s5 \rightarrow Ri$ When s5 is a negative value, the ADDN2 instruction is specified. |
| CMP #s5, Ri | 1 | C C C C | $Ri - s5$ When s5 is a negative value, the CMP2 instruction is specified. |
| BAND #u8, @Ri | - | - - - - | $[Ri] \&=u8$ Either of the following instructions is generated, depending on the value of u8. if((u8&0x0F)!=0x0F) The BANDL instruction is generated. if((u8&0xF0)!=0xF0) The BANDH instruction is generated. |
| BOR #u8, @Ri | - | - - - - | $[Ri] =u8$ Either of the following instructions is generated, depending on the value of u8. if((u8&0x0F)!=0) The BORL instruction is generated. if((u8&0xF0)!=0) The BORH instruction is generated. |
| BEOR #u8, @Ri | - | - - - - | $[Ri] ^=u8$ Either of the following instructions is generated, depending on the value of u8. if((u8&0x0F)!=0) The BEORL instruction is generated. if((u8&0xF0)!=0) The BEORH instruction is generated. |
| DIV Ri | - | - C - C | $MDL/Ri \rightarrow MDL, MDL\%Ri \rightarrow MDH$ DIVOS, 32 DIV1s, DIV2, DIV3, and DIV4 are generated. The code length is 72 bytes. |
| DIVU Ri | - | - C - C | $MDL/Ri \rightarrow MDL, MDL\%Ri \rightarrow MDH$ Unsigned DIVOU and 32 DIV1s are generated. The code length is 66 bytes. |
| LSL #u5, Ri | 1 | C C - C | $Ri \ll u5 \rightarrow Ri$ Logical shift When u5 is 16 or greater, the LSL2 instruction is specified. |
| LSR #u5, Ri | 1 | C C - C | $Ri \gg u5 \rightarrow Ri$ Logical shift When u5 is 16 or greater, the LSR2 instruction is specified. |

Table 12.1-1 Assembler Pseudo Machine Instructions (2 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|--------------------------|----------------|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| ASR #u5, Ri | 1 | C C - C | Ri >> s5 → Ri Arithmetic shift When u5 is 16 or greater, the ASR2 instruction is specified. |
| LDI #{i8 i20 i32}, Ri | - | - - - - | {i8 i20 i32} → Ri When the value is an absolute value, the assembler selects i8, i20, or i32 as an optimum choice. In this case, however, the following conditions apply: 1)symbols are included in the same section, or in any section other than a code section. 2)The expression format is one of the following: symbol symbol+offset value symbol-offset value Only when both of conditions 1) and 2) are met is the optimization function activated, and the optimum instruction format selected by the assembler. Otherwise, i32 is selected. When the value is a relative value, external reference value, or section value, i32 is selected. One of the following instructions is specified with i8, i20, or i32: When i8 is selected: LDI:8 #i8, Ri When i20 is selected: LDI:20 #i20, Ri When i32 is selected: LDI:32 #i32, Ri |
| LDM (reglist) | - | - - - - | [R15++] → reglist Load multiple registers (R0 to R15) Either of the following instructions is generated, depending on the registers specified with reglist. R0 to R7: The LDM0 instruction is generated. R8 to R15:The LDM1 instruction is generated. |
| STM (reglist) | - | - - - - | reglist → [--R15] Store multiple registers (R0 to R15) Either of the following instructions is generated, depending on the registers specified with reglist. R0 to R7: The STM0 instruction is generated. R8 to R15: The STM1 instruction is generated. |

Table 12.1-1 Assembler Pseudo Machine Instructions (3 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|-------------------------|----------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| CALL20 label20, Ri | - | - - - - | <p>Next instruction address → RP, label20 → PC</p> <p>Instructions are generated as shown below:</p> <p>1) Label20-\$-2 → disp</p> <p>2) -0x800 <= disp <= +0x7FE</p> <p> CALL label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:20 #label20, Ri</p> <p> JMP @Ri</p> |
| BRA20 label20, Ri | - | - - - - | <p>Label20 → PC Ri: Work register</p> <p>Instructions are generated as shown below:</p> <p>1) Label20-\$-2 → disp</p> <p>2) -0x100 <= disp <= +0xFE</p> <p> BRA label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:20 #label20, Ri</p> <p> JMP @Ri</p> |
| Bcc20 label20, Ri | - | - - - - | <p>if (condition) label20 → PC</p> <p> Ri: Work register</p> <p>For information about the condition specification (cc), see Section "5.1.4 Optimization of Branch Instructions" in Part 1.</p> <p>Instructions are generated as shown below:</p> <p>1) label20-\$-2 → disp</p> <p>2) -0x100 <= disp <= +0xFE</p> <p> Bcc label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> Bxcc false</p> <p> LDI:20 #label20, Ri</p> <p> JMP @Ri</p> <p> false:</p> <p>Note: xcc and cc are mutually exclusive.</p> |
| CALL20:D label20, Ri | - | - - - - | <p>Next instruction address → RP, label20 → PC</p> <p>Instructions are generated as shown below:</p> <p>1) Label20-\$-2 → disp</p> <p>2) -0x800 <= disp <= +0x7FE</p> <p> CALL:D label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:20 #label20, Ri</p> <p> CALL:D @Ri</p> |

Table 12.1-1 Assembler Pseudo Machine Instructions (4 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|--------------------------|----------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| BRA20:D label20, Ri | - | - - - - | <p>Label20 → PC Ri: Work register</p> <p>Instructions are generated as shown below:</p> <p>1) Label20-\$-2 → disp</p> <p>2) -0x100 <= disp <= +0xFE</p> <p> BRA:D label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:20 #label20, Ri</p> <p> JMP:D @Ri</p> |
| Bcc20:D label20, Ri | - | - - - - | <p>if (condition) label20 → PC</p> <p> Ri: Work register</p> <p>For information about the condition specification (cc), see Section "5.1.4 Optimization of Branch Instructions", in Part I.</p> <p>Instructions are generated as shown below:</p> <p>1) Label20-\$-2 → disp</p> <p>2) -0x100 <= disp <= +0xFE</p> <p> Bcc:D label20</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> Bxcc false</p> <p> LDI:20 #label20, Ri</p> <p> JMP:D @Ri</p> <p>false:</p> <p>Note: xcc and cc are mutually exclusive.</p> |
| CALL32 label32, Ri | - | - - - - | <p>Next instruction address → RP,</p> <p>label32 → PC</p> <p>Instructions are generated as shown below:</p> <p>1) Label32-\$-2 → disp</p> <p>2) -0x800 <= disp <= +0x7FE</p> <p> CALL label32</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:32 #label32, Ri</p> <p> CALL @Ri</p> |
| BRA32 label32, Ri | - | - - - - | <p>Label32 → PC Ri: Work register</p> <p>Instructions are generated as shown below:</p> <p>1) Label32-\$-2 → disp</p> <p>2) -0x100 <= disp <= +0xFE</p> <p> BRA label32</p> <p>3)When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> LDI:32 #label32, Ri</p> <p> JMP @Ri</p> |

Table 12.1-1 Assembler Pseudo Machine Instructions (5 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|-------------------------|----------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| Bcc32 label32, Ri | - | - - - - | <p>if (condition) label32 → PC Ri: Work register For information about the condition specification (cc), see Section "5.1.4 Optimization of Branch Instructions", in Part 1. Instructions are generated as shown below: 1) Label32-\$-2 → disp 2) -0x100 <= disp <= +0xFE Bcc label32 3) When the value is out of the range shown in 1) and 2) or contains an external reference: Bxcc false LDI:32 #label32, Ri JMP @Ri false: Note: xcc and cc mutually are exclusive.</p> |
| CALL32:D label32, Ri | - | - - - - | <p>Next instruction address → RP, label32 → PC Instructions are generated as shown below: 1) Label32-\$-2 → disp 2) -0x800 <= disp <= +0x7FE CALL:D label32 3) When the value is out of the range shown in 1) and 2) or contains an external reference: LDI:32 #label32, Ri CALL:D @Ri</p> |
| BRA32:D label32, Ri | - | - - - - | <p>Label32 → PC Ri: Work register Instructions are generated as shown below: 1) Label32-\$-2 → disp 2) -0x100 <= disp <= +0xFE BRA:D label32 3) When the value is out of the range shown in 1) and 2) or contains an external reference: LDI:32 #label32, Ri JMP:D @Ri</p> |

Table 12.1-1 Assembler Pseudo Machine Instructions (6 / 6)

| Mnemonic | Machine cycles | Flag change | Operation |
|--------------------------|----------------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | N Z V C | |
| Bcc32:D label32, Ri | - | - - - - | <p>if (condition) label32 → PC Ri: Work register</p> <p>For information about the condition specification (cc), see Section "5.1.4 Optimization of Branch Instructions", in Part 1.</p> <p>Instructions are generated as shown below:</p> <p>1) Label32-\$-2 → disp 2) -0x100 ≤ disp ≤ +0xFE Bcc:D label32</p> <p>3) When the value is out of the range shown in 1) and 2) or contains an external reference:</p> <p> Bxcc false LDI:32 #label32, Ri JMP:D @Ri false:</p> <p>Note: xcc and cc are mutually exclusive.</p> |

APPENDIX

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

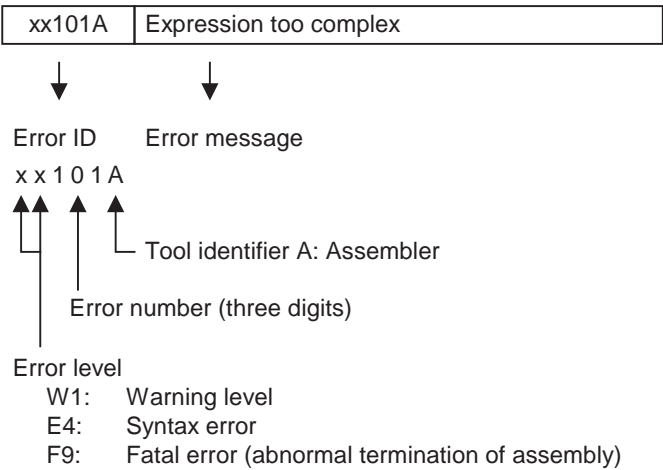
APPENDIX A Error Messages

APPENDIX B Restrictions

APPENDIX A Error Messages

The assembler displays the error messages below.

■ Format of Error Messages



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 replaced levels must not exceed 255.

[Program action]

Replaced 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 this definition of 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.

| | |
|--------|-------------------------|
| 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.

| | |
|--------|-------------------|
| 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=2".

| | |
|--------|--------------------------------------|
| 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.

| | |
|--------|-------------------------------|
| 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.

| | |
|--------|-------------------------------------------|
| 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.

APPENDIX A Error Messages

| | |
|--------|-----------------------------------------------------|
| E4533A | Forward reference symbol is described in expression |
|--------|-----------------------------------------------------|

A forward reference symbol cannot be used in an expression of 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 of 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 a 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 (-I) 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 Value" for details.

| | |
|--------|--------------------|
| E4541A | Value out of range |
|--------|--------------------|

[Program action]

The program masks an operational result (value) of a equation described in the operand, in accordance with the operand size.

See section "7.11 Expressions" for details.

The assembler does not output object files.

[Supplementary Explanation]

When a list file output specification option (-I) 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.

| | |
|--------|-----------------------------------------------------------------------|
| W1546A | Duplication in the specification of the register list (register name) |
|--------|-----------------------------------------------------------------------|

[Program action]

The program ignores duplication registers and continues processing.

[Supplementary Explanation]

This message appears when registers are specified redundantly to the register list.

This message appears when the -reglst_check option is specified.

See section "4.8.11 -reglst_check, -Xreglst_check" for details on -reglst_check.

| | |
|--------|-----------------|
| 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.

| | |
|--------|----------------------|
| E4556A | Missing symbol field |
|--------|----------------------|

[Program action]

Ignores the instruction.

| | |
|--------|---------------------------------|
| E4557A | Starting address out of section |
|--------|---------------------------------|

[Program action]

Ignores the starting address.

| | |
|--------|--------------------------------------|
| W1558A | 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 underline (_).

| | |
|--------|-------------------------------------|
| 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.

| | |
|--------|-------------------------------------------------------------------|
| E4601A | Unusable operation mnemonic with common object (instruction-name) |
|--------|-------------------------------------------------------------------|

[Program action]

Ignores the instruction and creates a NOP instruction.

[Supplementary Explanation]

This error occurs by the instruction shown in the Table A-1, when specification of output FR/FR80 common object option (-CO).

Please see Section "4.8.12 -CO" for the method of outputting FR/FR80 common object.

Table A-1 Incompatibility of FR and FR80 Instructions

| Instructions Incompatibility | FR | FR80 |
|------------------------------|----|------|
| LDRES @Ri+,#u4 | ○ | × |
| STRES #u4,@Ri | ○ | × |
| COPOP #u4,#CC,CRj,CRi | ○ | × |
| COPLD #u4,#CC,Rj,CRi | ○ | × |
| COPST #u4,#CC,CRj,Ri | ○ | × |
| COPSV #u4,#CC,CRj,Ri | ○ | × |
| SRCH0 Ri | × | ○ |
| SRCH1 Ri | × | ○ |
| SRCHC Ri | × | ○ |

○: Compatible ✕: Incompatible

| | |
|--------|--------------------------|
| 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 instruction option.

| | |
|--------|---------------------------------------------------------------------|
| W1608A | Conflicting Option-suffix in operation field (detailed information) |
|--------|---------------------------------------------------------------------|

[Program action]

Uses the first specified instruction 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 specification.

| | |
|--------|---------------------------|
| 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..

| | |
|--------|-----------------------------------------|
| 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.

APPENDIX A Error Messages

| | |
|--------|----------------|
| W1717A | Mismatch .ENDS |
|--------|----------------|

[Program action]

Ignores the specification.

| | |
|--------|--------------------------|
| E4756A | Invalid address modifier |
|--------|--------------------------|

[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..

| | |
|--------|------------------------------------------|
| W1854A | Conflicting operands in this instruction |
|--------|------------------------------------------|

[Program action]

Ignores and creates a code.

| | |
|--------|--------------------------|
| W1855A | Not implemented register |
|--------|--------------------------|

[Program action]

Ignores and creates a code.

| | |
|--------|--------------------------------------|
| 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 Microelectronics 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.

APPENDIX A Error Messages

| | |
|--------|-----------------------------|
| 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.

| | |
|--------|-------------------------------|
| 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.

| | |
|--------|--------------------------------------------------------------------|
| W1999A | If use this instruction about I/O or FIFO port. May be get problem |
|--------|--------------------------------------------------------------------|

[Program action]

Assumes that the instruction will not cause a failure, and continues processing.

APPENDIX B Restrictions

The following restrictions apply when the assembler is used.

■ Restrictions Related to Preprocessor Processing

- The number of nested include files must not exceed 8.
- The number of nested macro calls must not exceed 256.
- The number of dummy arguments specified in a macro definition must not exceed 32,767.
- An unlimited number of macro names can be registered (The maximum number depends on the size of the available memory area).

■ Restrictions Related to Assembly Processing

- An unlimited number of symbol can be registered (The maximum number depends on the size of the available memory area).

■ 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.

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) | 231 |
| ## Operator | |
| Concatenating the Characters to be Replaced by Macro Replacement (## Operator) | 232 |
| #define | |
| Argument-attached #define Instruction | 229 |
| Argument-less #define Instruction | 229 |
| #elif | |
| #elif Instruction | 225 |
| #else | |
| #else Instruction | 224 |
| #endif | |
| #endif Instruction | 227 |
| #endm | |
| #endm Instruction | 216 |
| #error | |
| #error Instruction | 238 |
| #exitm | |
| #exitm Instruction | 215 |
| #exitm Instruction Rules | 215 |
| #if | |
| #if Instruction | 221 |
| #ifdef | |
| #ifdef Instruction | 222 |
| #ifndef | |
| #ifndef Instruction | 223 |
| #include | |
| #include Instruction | 236 |
| #line | |
| #line Instruction | 237 |
| #local | |
| #local Instruction | 214 |
| #local Instruction Rules | 214 |
| #macro | |
| #macro Instruction | 213 |
| #macro Instruction Rules | 213 |
| #pragma | |
| #pragma Instruction | 239 |
| #purge | |
| #purge Instruction | 235 |
| #repeat | |
| #repeat Instruction Rules | 219 |
| #set | |
| #set Instruction | 233 |

| | |
|--------------------------------------|-----|
| #undef | |
| #undef Instruction | 234 |
| .ALIGN | |
| .ALIGN | 94 |
| .ALIGN Instruction | 163 |
| .ASCII | |
| .ASCII | 102 |
| .ASCII Instruction | 182 |
| .BYTE | |
| .BYTE | 97 |
| .BYTE Instruction | 174 |
| .DATA | |
| .DATA | 97 |
| .DATA Instruction | 174 |
| .DATAB | |
| .DATAB | 98 |
| .DATAB Instruction | 176 |
| .DEBUG | |
| .DEBUG | 103 |
| .DEBUG Instruction | 186 |
| .DOUBLE | |
| .DOUBLE | 99 |
| .DOUBLE Instruction | 178 |
| .END | |
| .END | 93 |
| .END Instruction | 159 |
| .ENDS | |
| .STRUCT and .ENDS Instructions | 184 |
| .EQU | |
| .EQU | 96 |
| .EQU Instruction | 171 |
| .EXPORT | |
| .EXPORT | 95 |
| .EXPORT Instruction | 167 |
| .FDATA | |
| .FDATA | 99 |
| .FDATA Instruction | 177 |
| .FDATAB | |
| .FDATAB | 100 |
| .FDATAB Instruction | 179 |
| .FLOAT | |
| .FLOAT | 99 |
| .FLOAT Instruction | 177 |
| .FORM | |
| .FORM | 105 |
| .FORM Instruction | 189 |
| .FRES | |
| .FRES | 101 |
| .FRES Instruction | 181 |

| | | | |
|-------------------------------------|-----|-------------------------|-----|
| .GLOBAL | | .TITLE | |
| .GLOBAL | 95 | .TITLE | 105 |
| .GLOBAL Instruction..... | 168 | .TITLE Instruction..... | 190 |
| .HALF | | .WORD | |
| .HALF | 97 | .WORD | 97 |
| .HALF Instruction..... | 174 | .WORD Instruction..... | 175 |
| .HEADING | | | |
| .HEADING | 105 | | |
| .HEADING Instruction..... | 191 | | |
| .IMPORT | | | |
| .IMPORT | 95 | | |
| .IMPORT Instruction..... | 169 | | |
| .LIBRARY | | | |
| .LIBRARY | 104 | | |
| .LIBRARY Instruction | 187 | | |
| .LIST | | | |
| .LIST | 106 | | |
| .LIST Instruction..... | 192 | | |
| .LONG | | | |
| .LONG..... | 97 | | |
| .LONG Instruction | 175 | | |
| .ORG | | | |
| .ORG | 94 | | |
| .ORG Instruction..... | 164 | | |
| .PAGE | | | |
| .PAGE | 106 | | |
| .PAGE Instruction..... | 194 | | |
| .PROGRAM | | | |
| .PROGRAM..... | 93 | | |
| .PROGRAM Instruction | 158 | | |
| .REG | | | |
| .REG | 96 | | |
| .REG Instruction | 172 | | |
| .RES | | | |
| .RES | 101 | | |
| .RES Instruction..... | 180 | | |
| .SDATA | | | |
| .SDATA..... | 102 | | |
| .SDATA Instruction | 182 | | |
| .SDATAB | | | |
| .SDATAB | 102 | | |
| .SDATAB Instruction..... | 183 | | |
| .SECTION | | | |
| .SECTION | 93 | | |
| .SECTION Instruction | 160 | | |
| .SKIP | | | |
| .SKIP | 94 | | |
| .SKIP Instruction | 165 | | |
| .SPACE | | | |
| .SPACE | 106 | | |
| .SPACE Instruction | 195 | | |
| .STRUCT | | | |
| .STRUCT and .ENDS Instructions..... | 184 | | |

INDEX

- A**
 - Absolute
 - Absolute Values 129
 - Address
 - Address Control Instructions 162
 - ALIGN
 - .ALIGN 94
 - .ALIGN Instruction 163
 - Area
 - Area Definition Instructions 173
 - Argument-attached
 - Argument-attached #define Instruction 229
 - Argument-less
 - Argument-less #define Instruction 229
 - Arithmetic
 - Arithmetic Operators 132
 - ASCII
 - .ASCII 102
 - .ASCII Instruction 182
 - Assembler
 - Assembler Pseudo Machine Instructions 246
 - Assembly
 - Assembly Phase 4
 - Restrictions Related to Assembly Processing 278
- B**
 - Backward
 - Forward Reference Symbols and Backward Reference Symbols 117
 - Binary
 - Binary Constants 118
 - Bitwise
 - Bitwise Operators 131
 - Boolean
 - Boolean Values 131
 - Branch
 - Optimization of Branch Instructions 74
 - Optimization that Replaces Delayed Branch Instructions 81
 - Optimization that Replaces Normal Branch Instructions 79
 - BYTE
 - .BYTE 97
 - .BYTE Instruction 174
- C**
 - C
 - C 46
 - C
 - Differences from the C Preprocessor 243
 - Character
 - Character Constant Elements 205
 - Character Constants 120, 200, 205
 - Character Set 114
 - Concatenating the Characters to be Replaced by Macro Replacement (## Operator) 232
 - Replacing Formal Macro Arguments with Character Strings (# Operator) 231
- Check**
 - Check Levels and Optimization Code Check Processing 71
 - Optimization Code Check Functions for the fasm911s 70
 - Preprocessor and Optimization Code Check Processings 90
- cif**
 - cif 51
- cmsg**
 - cmsg 58
- Command**
 - fasm911s Command Lines 16
- Comment**
 - Comment Field 113
 - Comments 137, 200, 202
 - Comments Allowed in an Option File 22
- Components Omitted**
 - Specifying a File Name with Components Omitted 18, 20
- Composition**
 - Composition 84
- Concatenated**
 - Concatenated Linkage 146
- Concatenating**
 - Concatenating the Characters to be Replaced by Macro Replacement (## Operator) 232
- Conditional**
 - Conditional Assembly Instructions 220
- Constant**
 - Binary Constants 118
 - Character Constant Elements 205
 - Character Constants 120, 200, 205
 - Data Format of Double-precision Floating-point Constants 125
 - Data Format of Single-precision Floating-point Constants 125
 - Decimal Constants 118
 - Hexadecimal Constants 118
 - Integer Constants 118, 200, 204
 - Notation for Floating-point Constants 123
 - Octal Constants 118
 - Range of the Representable Floating-point Constants 126
- Continuation**
 - Continuation Field 113
 - Continuation of a Line 200
 - Continuation of Line 203

| | | |
|----------------------------------------------------|--------|--|
| Counter | | |
| Location Counter Symbols | 119 | |
| -cpu | | |
| -cpu | 50 | |
| Cross-reference | | |
| Cross-reference List | 108 | |
| -cwno | | |
| -cwno | 59 | |
| D | | |
| -D | | |
| -D | 43 | |
| Data | | |
| .DATA | 97 | |
| .DATA Instruction | 174 | |
| Data Format of Double-precision Floating-point | | |
| Constants | 125 | |
| Data Format of Single-precision Floating-point | | |
| Constants | 125 | |
| DATAB | | |
| .DATAB | 98 | |
| .DATAB Instruction | 176 | |
| DEBUG | | |
| .DEBUG | 103 | |
| .DEBUG Instruction | 186 | |
| Debugging | | |
| Debugging Information Output Control Instruction | | |
| | 186 | |
| Options Related to Objects and Debugging | | |
| | 29, 30 | |
| Decimal | | |
| Decimal Constants | 118 | |
| Default | | |
| Default Option File | 23 | |
| define | | |
| Argument-attached #define Instruction | 229 | |
| Argument-less #define Instruction | 229 | |
| Defined | | |
| Defined Macro Name | 242 | |
| Defined Macro Names | 241 | |
| Definition | | |
| Area Definition Instructions | 173 | |
| Macro Definition Rules | 212 | |
| Macro Definitions | 212 | |
| Program Structure Definition Instructions | 157 | |
| Structure Area Definition | 184 | |
| Symbol Definition Instructions | 170 | |
| Development Environment | | |
| Directory Structure of the Development | | |
| Environment | 13 | |
| Differences | | |
| Differences from the C Preprocessor | 243 | |
| Directory | | |
| Directory Structure of the Development Environment | | |
| | 13 | |
| DOUBLE | | |
| .DOUBLE | 99 | |
| .DOUBLE Instruction | 178 | |
| Specification of Single or Double Precision | 124 | |
| Double-precision | | |
| Data Format of Double-precision Floating-point | | |
| Constants | 125 | |
| E | | |
| elif | | |
| #elif Instruction | 225 | |
| else | | |
| #else Instruction | 224 | |
| END | | |
| .END | 93 | |
| .END Instruction | 159 | |
| endif | | |
| #endif Instruction | 227 | |
| endm | | |
| #endm Instruction | 216 | |
| EQU | | |
| .EQU | 96 | |
| .EQU Instruction | 171 | |
| Error | | |
| #error Instruction | 238 | |
| Error Display | 91 | |
| Error Messages | 254 | |
| Format of Error Messages | 254 | |
| exitm | | |
| #exitm Instruction | 215 | |
| #exitm Instruction Rules | 215 | |
| EXPORT | | |
| .EXPORT | 95 | |
| .EXPORT Instruction | 167 | |
| Expression | | |
| Expression Syntax | 127 | |
| Expression Types | 127 | |
| External | | |
| External Reference Values | 129 | |
| F | | |
| -f | | |
| -f | 54 | |
| fasm911s | | |
| fasm911s Command Lines | 16 | |
| Optimization Code Check Functions for the | | |
| fasm911s | 70 | |
| FDATA | | |
| .FDATA | 99 | |
| .FDATA Instruction | 177 | |

INDEX

| | | |
|------------------------------------------------------------------|----------|--|
| FDTAB | | |
| .FDTAB..... | 100 | |
| .FDTAB Instruction | 179 | |
| FELANG | | |
| FELANG..... | 9 | |
| FETOOL | | |
| FETOOL | 8 | |
| Field | | |
| Comment Field | 113 | |
| Continuation Field..... | 113 | |
| Operand Field | 112 | |
| Operand Field Format..... | 153 | |
| Operation Field | 112 | |
| Symbol Field | 112 | |
| File | | |
| Comments Allowed in an Option File..... | 22 | |
| Default Option File..... | 23 | |
| File Search for Format 1 | 236 | |
| File Search for Formats 2 and 3 | 236 | |
| Format for Specifying a File Name | 18, 19 | |
| Include File..... | 92 | |
| Library File Specification Instruction | 187 | |
| Option File | 21 | |
| Specifying a File | 17 | |
| Specifying a File Name with Components Omitted | 18, 20 | |
| FLOAT | | |
| .FLOAT | 99 | |
| .FLOAT Instruction | 177 | |
| Floating-point | | |
| Data Format of Double-precision Floating-point Constants..... | 125 | |
| Data Format of Single-precision Floating-point Constants..... | 125 | |
| Notation for Floating-point Constants..... | 123 | |
| Range of the Representable Floating-point Constants | 126 | |
| FORM | | |
| .FORM..... | 105 | |
| .FORM Instruction | 189 | |
| Formal | | |
| Formal Argument Naming Rules | 208 | |
| Formal Argument Replacement Rules | 208 | |
| Formal Arguments..... | 200 | |
| Format | | |
| Data Format of Double-precision Floating-point Constants..... | 125 | |
| Data Format of Single-precision Floating-point Constants..... | 125 | |
| File Search for Format 1 | 236 | |
| File Search for Formats 2 and 3 | 236 | |
| Format for Specifying a File Name | 18, 19 | |
| Format of Error Messages | 254 | |
| Header Format | 86 | |
| Machine Instruction Format | 152 | |
| Operand Field Format | 153 | |
| Preprocessor Instruction Format | 200, 201 | |
| Section Description Format..... | 140 | |
| Statement Format..... | 112 | |
| Forward | | |
| Forward Reference Symbol Optimization Function | 72 | |
| Forward Reference Symbols and Backward Reference Symbols..... | 117 | |
| -FPU | | |
| -FPU..... | 49 | |
| FPU Information Options (-FPU and -XFPU)..... | 49 | |
| FRES | | |
| .FRES | 101 | |
| .FRES Instruction | 181 | |
| G | | |
| -g | | |
| -g | 32 | |
| GLOBAL | | |
| .GLOBAL | 95 | |
| .GLOBAL Instruction | 168 | |
| H | | |
| -H | | |
| -H..... | 45 | |
| HALF | | |
| .HALF | 97 | |
| .HALF Instruction..... | 174 | |
| Header | | |
| Header Format..... | 86 | |
| HEADING | | |
| .HEADING | 105 | |
| .HEADING Instruction..... | 191 | |
| -help | | |
| -help | 60 | |
| Hexadecimal | | |
| Hexadecimal Constants | 118 | |
| I | | |
| -I | | |
| -I | 44 | |
| if | | |
| #if Instruction | 221 | |
| ifdef | | |
| #ifdef Instruction | 222 | |
| ifndef | | |
| #ifndef Instruction..... | 223 | |
| IMPORT | | |
| .IMPORT | 95 | |
| .IMPORT Instruction | 169 | |

| | |
|--------------------------------------|------------------------------------------------|
| INC911 | 172 |
| INC911 | 180 |
| Include | 182 |
| #include Instruction | 183 |
| Include File | 160 |
| Information | 165 |
| Information List | 195 |
| Initial-value | 184 |
| Transfer of Initial-value Data | 190 |
| Instruction | 175 |
| #elif Instruction | 162 |
| #else Instruction | 173 |
| #endif Instruction | 229 |
| #endm Instruction | 229 |
| #error Instruction | 246 |
| #exitm Instruction | 220 |
| #exitm Instruction Rules | 186 |
| #if Instruction | 187 |
| #ifdef Instruction | 188 |
| #ifndef Instruction | 152 |
| #include Instruction | 217 |
| #line Instruction | 217 |
| #local Instruction | 240 |
| #local Instruction Rules | 74 |
| #macro Instruction | 73 |
| #macro Instruction Rules | 76 |
| #pragma Instruction | 81 |
| #purge Instruction | 79 |
| #repeat Instruction Rules | 200, 201 |
| #set Instruction | 166 |
| #undef Instruction | 157 |
| .ALIGN Instruction | 170 |
| .ASCII Instruction | Integer |
| .BYTE Instruction | Integer Constants |
| .DATA Instruction | Scope of Integer Constants Handled by |
| .DATAB Instruction | Pseudo-instructions |
| .DEBUG Instruction | |
| .DOUBLE Instruction | L |
| .END Instruction | -l |
| .EQU Instruction | -lcros |
| .EXPORT Instruction | LDI |
| .FDATA Instruction | Optimization of LDI Instructions |
| .FDATAB Instruction | Optimization of LDI:20 and LDI:32 Instructions |
| .FLOAT Instruction | |
| .FORM Instruction | -lexp |
| .FRES Instruction | -lexp |
| .GLOBAL Instruction | -lf |
| .HALF Instruction | -lf |
| .HEADING Instruction | |
| .IMPORT Instruction | |
| .LIBRARY Instruction | |
| .LIST Instruction | |
| .LONG Instruction | |
| .ORG Instruction | |
| .PAGE Instruction | |
| .PROGRAM Instruction | |

INDEX

| | | |
|-----------------------------------------------------------------------------------------|--------|--|
| Library | | |
| .LIBRARY | 104 | |
| .LIBRARY Instruction | 187 | |
| Library File Specification Instruction | 187 | |
| -linc | | |
| -linc | 38 | |
| Line | | |
| #line Instruction | 237 | |
| Continuation of a Line | 200 | |
| Continuation of Line | 203 | |
| -linf | | |
| -linf | 36 | |
| Linkage | | |
| Concatenated Linkage | 146 | |
| Program Linkage Instructions | 166 | |
| Section Linkage Methods | 146 | |
| Shared Linkage | 147 | |
| List | | |
| .LIST | 106 | |
| .LIST Instruction | 192 | |
| Cross-reference List | 108 | |
| Information List | 85, 87 | |
| List Output Control Instructions | 188 | |
| Register Lists | 136 | |
| Section List | 107 | |
| Source List | 89 | |
| Listing | | |
| Options Related to Listing | 29, 33 | |
| Local | | |
| #local Instruction | 214 | |
| #local Instruction Rules | 214 | |
| Local Symbol Naming Rules | 209 | |
| Local Symbol Replacement Rules | 209 | |
| Local Symbols | 200 | |
| Location | | |
| Location Counter Symbols | 119 | |
| Logical | | |
| Logical Operators | 131 | |
| LONG | | |
| .LONG | 97 | |
| .LONG Instruction | 175 | |
| -lsec | | |
| -lsec | 37 | |
| -lsrc | | |
| -lsrc | 36 | |
| M | | |
| Machine | | |
| Machine Instruction Format | 152 | |
| Macro | | |
| #macro Instruction | 213 | |
| #macro Instruction Rules | 213 | |
| Concatenating the Characters to be Replaced by Macro Replacement (## Operator) | 232 | |
| Defined Macro Name | 242 | |
| Defined Macro Names | 241 | |
| Macro Call Instruction | 217 | |
| Macro Call Instruction Rules | 217 | |
| Macro Definition Rules | 212 | |
| Macro Definitions | 212 | |
| Macro Name Replacement | 228 | |
| Macro Name Rules | 207 | |
| Macro Name Types | 207 | |
| Macro Names | 200 | |
| Macro Replacement Rules | 228 | |
| Replacing Formal Macro Arguments with Character Strings (# Operator) | 231 | |
| Messages | | |
| Error Messages | 254 | |
| Format of Error Messages | 254 | |
| Module | | |
| Obtaining the Size of a Section in Another Module | 134 | |
| Multiple | | |
| Multiple Descriptions of a Section | 148 | |
| N | | |
| Name | | |
| Name Classification | 115 | |
| -name | | |
| -name | 56 | |
| Naming | | |
| Naming Rules | 115 | |
| No-operation | | |
| No-operation Instruction | 240 | |
| Notation | | |
| Notation for Floating-point Constants | 123 | |
| O | | |
| -O | | |
| -O | 48 | |
| -o | | |
| -o | 31 | |
| Objects | | |
| Options Related to Objects and Debugging | 29, 30 | |
| Obtaining | | |
| Obtaining the Size of a Section in Another Module | 134 | |
| Octal | | |
| Octal Constants | 118 | |
| Operand | | |
| Operand Field | 112 | |
| Operand Field Format | 153 | |
| Operation | | |
| Operation Field | 112 | |

- Operators**
 Operators for Calculating Values from Names 132
- OPT911**
 OPT911 12
- Optimization**
 Check Levels and Optimization Code Check
 Processing 71
 Forward Reference Symbol Optimization Function
 72
 Optimization Code Check Functions for the fasm911s
 70
 Optimization of Branch Instructions 74
 Optimization of LDI Instructions 73
 Optimization of LDI:20 and LDI:32 Instructions
 76
 Optimization that Prevents Interlocks Caused by
 Register Interference 77
 Optimization that Replaces Delayed Branch
 Instructions..... 81
 Optimization that Replaces Normal Branch
 Instructions..... 79
 Preprocessor and Optimization Code Check
 Processings..... 90
- Option**
 Comments Allowed in an Option File 22
 Default Option File 23
 FPU Information Options (-FPU and -XFPU) 49
 Option File 21
 Options Related to Listing 29, 33
 Options Related to Objects and Debugging
 29, 30
 Options Related to the Preprocessor 29, 40
 Other Options 29, 52
 Relationship with Start-time Options 84
 Relationships with Startup Options
 186, 189, 193
 Restrictions on Option Files 278
 Rules for Startup Options..... 26
 Startup Options..... 27
 Target-dependent Options 29, 47
- Order**
 Order of Operands..... 153
- ORG**
 .ORG 94
 .ORG Instruction..... 164
- Other**
 Other Options 29, 52
 Other Restrictions 278
- Overview**
 Overview 4, 5
- OVFW**
 -OVFW 62
- P**
 -P
 -P42
 -p
 -p41
PAGE
 .PAGE106
 .PAGE Instruction194
 -Pf
 -Pf.....42
Phase
 Assembly Phase4
 Preprocessor Phase4
 -pl
 -pl35
pragma
 #pragma Instruction239
Precedence
 Precedence of Operators135
Precision
 Precision in Operations of Expressions128
Preprocessor
 Differences from the C Preprocessor243
 Options Related to the Preprocessor29, 40
 Preprocessor.....198
 Preprocessor and Optimization Code Check
 Processings90
 Preprocessor Expression Operation Precision210
 Preprocessor Expressions210
 Preprocessor Instruction Format.....200, 201
 Preprocessor Operator Precedence.....211
 Preprocessor Operators210
 Preprocessor Phase4
 Restrictions Related to Preprocessor
 Processing.....278
Program
 .PROGRAM93
 .PROGRAM Instruction158
 Program Linkage Instructions166
 Program Structure Definition Instructions157
Pseudo-instructions
 Pseudo-instructions for which an Expression
 Containing the Size Operator cannot
 beSpecified134
 Scope of Integer Constants Handled by Pseudo-
 instructions156
purge
 #purge Instruction.....235
 -pw
 -pw.....35

INDEX

- R**
 - Range
 - Range of Operand Values..... 130
 - Range of the Representable Floating-point Constants..... 126
 - REG
 - .REG..... 96
 - .REG Instruction 172
 - Register
 - Optimization that Prevents Interlocks Caused by Register Interference..... 77
 - Register Lists 136
 - reglst_check
 - reglst_check 65
 - Relational
 - Relational Operators 131
 - Relationship
 - Relationship with Start-time Options..... 84
 - Relationships with Startup Options..... 158, 186, 189, 193
 - Relative
 - Relative Values 129
 - Repeat
 - #repeat Instruction Rules..... 219
 - Repeat Expansion..... 218
 - Replacement
 - Concatenating the Characters to be Replaced by Macro Replacement (## Operator) 232
 - Formal Argument Replacement Rules 208
 - Local Symbol Replacement Rules..... 209
 - Macro Name Replacement 228
 - Macro Replacement Rules..... 228
 - Replacing
 - Replacing Formal Macro Arguments with Character Strings (# Operator)..... 231
 - RES
 - .RES 101
 - .RES Instruction..... 180
 - Reserved
 - Reserved Words..... 116
 - Restrictions
 - Other Restrictions..... 278
 - Restrictions on Option Files 278
 - Restrictions Related to Assembly Processing 278
 - Restrictions Related to Preprocessor Processing .. 278
 - ROM
 - Setting ROM Storage Sections..... 149
 - Rules
 - Rules for Startup Options..... 26
- S**
 - Scope
 - Scope of Integer Constants Handled by Pseudo-instructions..... 156
 - SDATA
 - .SDATA..... 102
 - .SDATA Instruction 182
 - SDATAB
 - .SDATAB 102
 - .SDATAB Instruction..... 183
 - Search
 - File Search for Format 1 236
 - File Search for Formats 2 and 3 236
 - SECTION
 - .SECTION 93
 - .SECTION Instruction 160
 - Section Allocation Patterns 141, 145
 - Section Description Format..... 140
 - Section Linkage Methods 146
 - Section List 107
 - Section Size Extraction (SIZEOF Operator) 133
 - Section Types 140, 142
 - Section Types and Attributes..... 144
 - Section Values 129
 - Section-location-format
 - Section-location-format 160
 - Section-type
 - Section-type 160
 - set
 - #set Instruction 233
 - Setting
 - Setting ROM Storage Sections 149
 - Shared
 - Shared Linkage..... 147
 - Single
 - Specification of Single or Double Precision 124
 - Single-precision
 - Data Format of Single-precision Floating-point Constants 125
 - Size
 - Size Operator..... 117
 - SIZEOF
 - Section Size Extraction (SIZEOF Operator) 133
 - SKIP
 - .SKIP 94
 - .SKIP Instruction 165
 - Source
 - Source List..... 89
 - SPACE
 - .SPACE 106
 - .SPACE Instruction 195
 - Specification
 - Specification of Single or Double Precision 124

| | | |
|------------------------------------------------------|--------------------|--|
| Specifying | | |
| Format for Specifying a File Name | 18, 19 | |
| Specifying a File | 17 | |
| Specifying a File Name with Components Omitted | 18, 20 | |
| Start-time | | |
| Relationship with Start-time Options | 84 | |
| Startup | | |
| Relationships with Startup Options | 158, 186, 189, 193 | |
| Rules for Startup Options | 26 | |
| Startup Options | 27 | |
| Statement | | |
| Statement Format | 112 | |
| Strings | | |
| Strings | 122 | |
| STRUCT | | |
| .STRUCT and .ENDS Instructions | 184 | |
| Structure | | |
| Structure Area Definition | 184 | |
| Symbol | | |
| Symbol Definition Instructions | 170 | |
| Symbol Field | 112 | |
| T | | |
| -tab | | |
| -tab | 39 | |
| Target-dependent | | |
| Target-dependent Options | 29, 47 | |
| Term | | |
| Term Types | 129 | |
| Termination Code | | |
| Termination Code | 24 | |
| TITLE | | |
| .TITLE | 105 | |
| .TITLE Instruction | 190 | |
| TMP | | |
| TMP | 10 | |
| Transfer | | |
| Transfer of Initial-value Data | 149 | |
| U | | |
| -U | | |
| -U | 43 | |
| -UDSW | | |
| -UDSW | 61 | |
| undef | | |
| #undef Instruction | 234 | |
| V | | |
| -V | | |
| -V | 57 | |
| W | | |
| -w | | |
| -w | 55 | |
| WORD | | |
| .WORD | 97 | |
| .WORD Instruction | 175 | |
| X | | |
| -Xcmsg | | |
| -Xcmsg | 58 | |
| -Xcwno | | |
| -Xcwno | 59 | |
| -Xdof | | |
| -Xdof | 53 | |
| -XFPU | | |
| FPU Information Options (-FPU and -XFPU) | 49 | |
| -XFPU | 49 | |
| -Xg | | |
| -Xg | 32 | |
| -Xl | | |
| -Xl | 34 | |
| -Xo | | |
| -Xo | 31 | |
| -XOVFW | | |
| -XOVFW | 63 | |
| -Xreglst_check | | |
| -Xreglst_check | 66 | |
| -XUDSW | | |
| -XUDSW | 61 | |
| -XV | | |
| -XV | 57 | |

CM71-00203-5E

FUJITSU MICROELECTRONICS • CONTROLLER MANUAL
FR FAMILY
SOFTUNE™ ASSEMBLER MANUAL
for V6

April 2008 the fifth edition

Published **FUJITSU MICROELECTRONICS LIMITED**
Edited Strategic Business Development Dept.
