Hitachi Microcomputer Development Environment System

H8S, H8/300 Series

Cross Assembler

Users' Manual

HITACHI

Notice

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

Preface

This manual describes the cross assembler for the H8S and H8/300 series of Hitachi microprocessors.

This manual consists of three parts: the Language Guide, which describes the programming methods supported, the Usage Guide, which describes the use of the cross assembler, and the Appendices.

Appendix D provides a complete listing of the differences between this version and the previous version, H8/300 Series Cross Assembler version 3. Users who have used H8/300 Series Cross Assembler version 3 should refer to this appendix.

When developing software for an H8S or H8/300 series microprocessor, the following manuals should be used as references.

For details on the instruction set:

- H8S/2600 Series and H8S/2000 Series Programming Manual
- H8/300H Series Programming Manual
- H8/300 Series Programming Manual
- H8/300L Series Programming Manual

For details on related software:

- H8S, H8/300 Series C Compiler User's Manual
- H Series Linkage Editor User's Manual
- H Series Librarian User's Manual
- H8S, H8/300 Series Simulator/Debugger User's Manual
- SYSROF File Converter User's Manual

Notes: 1. MS-DOS is an operating system administrated by Microsoft Corporation.
2. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Usage Guide

Language Guide

Section 1 Overview

1.1 Features

This cross assembler provides the following features.

1. Object CPUs
   This cross assembler generates code for the following CPUs.

   — H8S/2600 series
   — H8S/2000 series
   — H8/300H series
   — H8/300 series
   — H8/300L series
2. Preprocessor Functions
   The following preprocessor functions are provided. These functions enable source programs to be coded quickly and efficiently.

   — File inclusion
   — Conditional assembly
   — Macros
   — Structured assembly
3. Based on Industry Standards
   Instruction set and assembler directive naming (mnemonics) forms a unified system based on the naming standards prescribed in the IEEE 694 specifications.

1.2 Assembler Overview

The basic function of this assembler is to convert assembler language source programs into relocatable binary object modules. Output of assembly listings, which show the results of the assembly process, is also provided.

The following source programs are accepted by this assembler.

- Assembly language source programs produced using an editor
- Assembly language source programs produced by the H8S, H8/300 series C compiler

The object modules produced by this assembler are accepted as input by the following software.

- H Series Linkage Editor
- H Series Librarian
- H8S, H8/300 Series Simulator/Debugger
- SYSROF File Converter

Section 2 Rules for Source Statement Coding

2.1 Source Statements

A source program is a set of source statements.

A source statement is an ASCII character string of up to 255 characters forming a single line.

2.1.1 Source Statement Types

The following source statement types are recognized by the assembler.

1. Instruction statements
   Statements that code executable instructions, assembler directives, preprocessor directives, and macro instructions.
2. Label statements
   Statements that only consist of a label. They are used to indicate branch destinations and data locations.
3. Comment statements
   Statements that onnly consist of a comment, and are used to include notes about the source program.
4. Blank statementsn
   Statements with no notations. They are used to delimit sections of source programs so that they are easier to read.

2.1.2 Source Statement Structure

Source statements consist of the following fields.

1. Label field
   Field to code symbols which represent the location of that statement.
2. Operation field
   Field to code an executable instructions, an assembler directive, a preprocessor directive, or a macro instruction.
3. Operand filed
   Field to code the operands for the instruction in the operation field.
4. Comment field
   Field for comments.
   The position of these fields in a source statement is shown in the figure below.

Label	Operation	Operand	Comment

2.1.3 Source Statement Field Syntax

1. Label field

   There are three formats for specifying the label field.

   a. The label starts in column 1.
   b. The label starts in column 1 and ends with a colon(:).
   c. The LABEL starts in column 2 or greater and ends with a colon(:).

      When there is no need for a symbol, the label field may be left blank.

Examples LABEL 1 LABEL 2: LABEL 3:

2. Operation field

   There are two formats for specifying the operation field.

   a. When there is no entry in the label field:

      The operation field starts in column 2 or greater.

   b. When there is an entry in the label field:

      The operation field starts one or more spaces after the end of the label field.

      When there is no need for an instruction, the operation field may be left blank.

```
Examples
       ADD.B  R0L,R1L
LABEL1 ADD.B  R0L,R1L
LABEL2
```

3. Operand field

   There are three formats for specifying the operand field.

   a. When one or more operand are required for the operation field instruction:

      The operands start one or more spaces or tabs after the instruction.

   b. When no operand is required for the operation field instruction:

      The operand field is left blank.

   c. When there is no entry in the operation field:

      The operand field is left blank.

```
Examples
       ADD.B  R0L,R1L
       NOP
LABEL
```

4. Comment field

   There are two formats for the comment field.

   a. When a comment is to be entered:

      The comment field starts with a semicolon(;).

   b. When no comment is to be entered:

      The comment field is left blank.

```
Examples
   ADD.B  R0L,R1L   ;  R1L=R0L+R1L
   NOP              ;  NO OPERAND
LABEL               ;  LABEL STATEMENT
; This line is a comment statement.
```

2.1.4 Continuation Lines

Source statements can be written over multiple lines.

The operand field comma(,) is taken as a delimiter, and the following operand(s) is/are entered on the following line. A plus sign (+) must be entered in the first column of the continuation line.

```
Example
   ADD.B  R0L,   ;  SOURCE OPERAND
+         R1L    ;  DESTINATION OPERAND
```

2.2 Symbol

Symbols, which are labels that are used to express the position of the source statement in which they are defined, have the value of the location counter (i.e., the address) of their defining source statement.

Also, when symbols are assigned values by assembler directives, they hold the assigned value.

2.2.1 Symbol Syntax

1. Characters used in symbols

   The following characters can be used in symbols.

   — Upper and lower case letters (A to Z, a to z)
   — Numbers (0 to 9)
   — The underscore character(_)
   — The dollar sign character ($)

   Upper and lower case letters are distinguished.
2. Syntax

   Symbols must begin with a letter, an underscore, or a dollar sign.

   Symbols can have up to 32 characters.
3. Names that cannot be used as symbols

   The following names may not be used as symbols.

   — Register mnemonics (ER0-ER7, E0-E7, R0-R7, R0H-R7H, R0L-R7L, SP, CCR, EXR, MACH, MACL)
   — Operators (STARTOF, SIZEOF, HIGH, LOW, HWORD, LWORD)
   — The location counter symbol ($)
   — Internal symbols (_$$00000 to _$$FFFFF)

   Note that the same name may not be used as both a symbol and as a section name.

Note: Internal symbols are required for internal processing of the assembler. They are not output to assembly listings or object modules.

2.2.2 Symbol Types

Symbols are classified into the following types depending on the value held by that symbol.

1. Absolute symbols
   Symbols whose value is determined at assembly time. There are two types of absolute symbols.
   a. Constant symbols
      Symbols whose value is a constant.
   b. Absolute address symbols
      Symbols whose value is an absolute address.
2. Relative symbols
   Symbols whose value cannot be determined at assembly time. The value of these symbols is determined when the program is linked into an object module by the linkage editor.
   There are two types of relative symbols.
   a. Relative address symbols
      Symbols whose value is a relative address.
   b. External reference symbols
      Symbols whose value is an external reference.
   These values are discussed in section 2.3, Values.

2.2.3 Symbol Reference

Symbols can be referenced in the following three ways:

- Forward reference
- Backward reference
- External reference

Supplement: This manual defines forward, backward, and external as shown in figures 2-1 and 2-2.

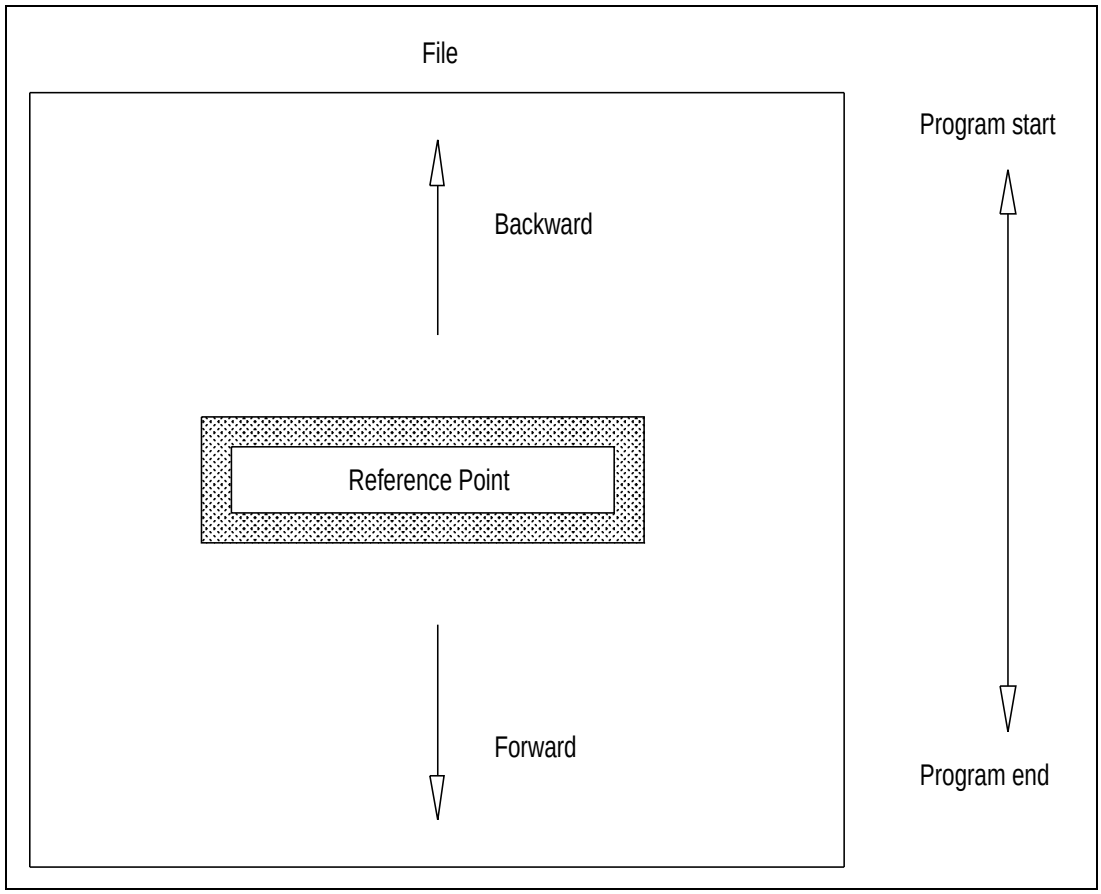


Figure 2-1 Meaning of Forward and Backward

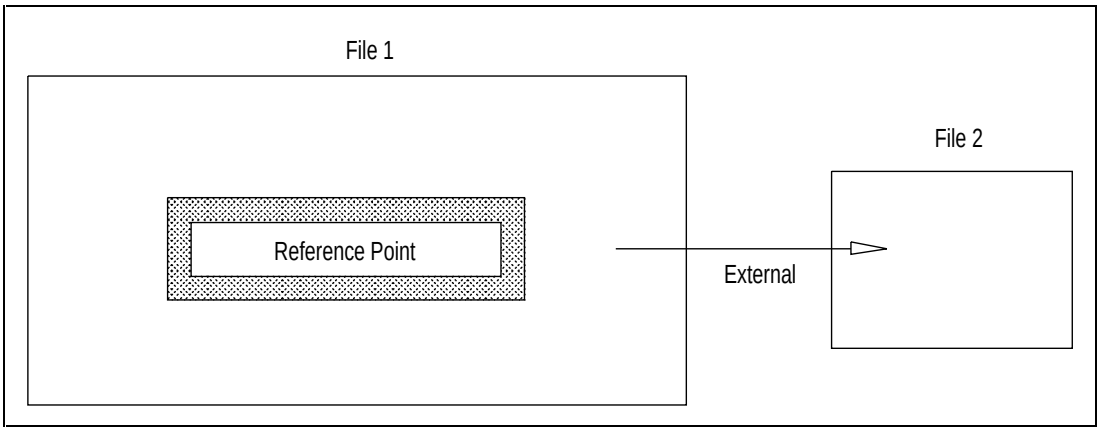


Figure 2-2 Meaning of External

- Forward reference

  Forward reference means to reference a symbol whose definition follows the line making the reference in the source program.

```
Example
          .
          .
   BRA    FORWARD   ;  Branch instruction BRA references symbol
                    ;  FORWARD located after this point.
          .
          .
FORWARD:
          .
          .
```

- Backward reference

  Backward reference means to reference a symbol whose definition precedes the line making the reference in the source program.

```
Example
          .
          .
BACK:
          .
          .
   BRA    BACK   ;  Branch instruction BRA references symbol BACK
                 ;  located before this point.
          .
          .
```

- External reference

  External reference means to reference a symbol defined in another source program when there are multiple source programs.

2.3 Values

2.3.1 Value Types

Values are classified into the following types.

1. Absolute values

   Values that can be determined by the assembler at assembly. Absolute values are further classified as follows:

   a. Constant values

      (I) Integer constant values

      (II) Character constant values

      (III) Constant symbol values

   b. Absolute address values

      (I) Location counter values (addresses) within an absolute addressing section

      (II) Location counter values (addresses) within a dummy section

      (III) Values of absolute address symbols

2. Relative values

   Values which cannot be determined at assembly. These values are determined when the program is linked into an object module by the linkage editor.

   Relative values are further classified as follows:

   a. Relative address values

      (I) Location counter values (Addresses) within a relative addressing section

      (II) Values of relative address symbols

   b. External references

      Values of external reference symbols

Symbol values are also classified according to the form of the reference to that symbol as follows:

1. Backward reference values

   Values of backward reference symbols.
2. Forward reference values

   Values of forward reference symbols.
3. External reference values

   Values of external reference symbols.

Note: The term "backward reference absolute value" used in this document refers to the following types of values.

- Integer constant values
- Character constant values
- Values of backward reference constant symbols
- A location counter value in an absolute addressing section ù A location counter value in a dummy section
- Values of backward reference absolute address symbols

2.3.2 Constants

Both integer and character constants are provided.

1. Integer constants

   Binary, octal, decimal, and hexadecimal are provided.

   Integer constants are expressed as a base and a value.

   — Binary numbers..... Expressed as the base "B'" and a binary number
   — Octal numbers..... Expressed as the base "Q'" and an octal number.
   — Decimal numbers..... Expressed as the base "D'" and a decimal number.
   — Hexadecimal numbers..... Expressed as the base "H'" and a hexadecimal number.

   Normally, decimal is used as the default base when the base is omitted. (See section 5.9.2, .RADIX.)

```
Examples
MOV.B  #B'00001010:8, R0L
MOV.B  #Q'012:8,R0L
MOV.B  #D'10:8,R0L
MOV.B  #H'0A:8,R0L
```

2. Character constants

   Character constants are coded by enclosing up to four characters in double quotation marks (").

   The characters which can be used in character constants are the ASCII code H'09 (tab) and ASCII codes from H'20 (space) to H'7E (tilde).

   To specify a double quotation mark in a character constant, enter two double quotation marks in succession.

```
Examples
.CPU   2600A
MOV.B  #"A":8,R0L   ;A
MOV.B  #"""":8,R1L  ;"
MOV.W  #"AB":16,R2  ;AB
```

2.3.3 Location Counter

The location counter is referenced by the symbol consisting of a single dollar sign ($).

The value of the location counter is the value (address) of the first location of the source statement in which it appears. When the location counter is referenced in an absolute addressing section, an absolute address value is acquired, and when it is referenced in an relative addressing section, a relative address value is acquired.

```
Examples
       .SECTION A,CODE,LOCATE=H'1000 ....    [1]
ABS    .EQU     $ ......................    [2]
       MOV.W    R0,R1
       .SECTION  B,CODE,ALIGN=2 ..........   [3]
REL    .EQU     $ ......................    [4]
       MOV.W    R0,R1
[1]Declares an absolute addressing section.
[2]The "$" symbol will be an absolute address value of H'1000 at this
point.
[3]Declares a relative addressing section.
[4]The "$" symbol will be a relative address value of H'0000 at this
point.
```

2.4 Expressions

2.4.1 Expression Elements

Expressions consist of terms, operators, and parentheses.

1. Terms
   A term is one of the following.
   — A constant
   — The location counter symbol ($)
   — A symbol
   — The result of a calculation specified by a combination of the above terms and operators.

2. Operators
   Table 2-1 shows the operators provided by this assembler.Table 2-1 Operator Table

Operator Class	Operator	Operation	Usage
Arithmetic operators	+	Unary plus	+<term>
	-	Unary minus	-<term>
	+	Addition	<term 1>+<term 2>
	-	Subtraction	<term 1>-<term 2>
	*	Multiplication	<term 1>*<term 2>
	/	Division	<term 1>/<term 2>
Logical operators	~	Unary negation	~<term>
	&	Logical and	<term 1>&<term 2>
	|	Logical or	<term 1>|<term 2>
	~	Exclusive or	<term 1>~<term 2>
Segment group operators	STARTOF	The starting address of a section group	STARTOF <section name>
	SIZEOF	he size of a section group	SIZEOF <section name>
Extruction operators	HIGH	Upper byte extraction	HIGH <term>
	LOW	Lower byte extraction	LOW <term>
	HWORD	Upper word extraction	HWORD <term>
	LWORD	Lower word extraction	LWORD <term>

Note: The LWORD and LWORD operators can only be used in programs for the H8S/2600, H8S/2000, and H8/300H in advanced or normal mode. They cannot be used in programs for the H8/300 and H8/300L series microprocessors.

3. Parentheses

   Parentheses are used to override the operator precedence order and control the order of evaluation.

2.4.2 Operator Precedence Order

Table 2-2 gives the operator precedence order.

Table 2-2 Operator Precedence

Operator Precedence	Operator	Association Rule
1	+ - ~ STARTOF SIZEOF HIGH LOW HWORD LWORD	←
2	* /	→
3	+ -	→
4	&	→
5	| ~	→

Notes: 1. The operator precedence 1 operators are the unary operators.
2. When operators of the same precedence are combined, the operations are performed in the direction indicated by the association rule. Note that "←" indicates from right to left in the expression, and that "→" indicates from left to right in the expression.

Examples	
10+20/5	Evaluates to 14.
(10+20)/5	Evaluates to 6.
H'0000FFFF | H'00FF00FF&H'0F0F0F0F	Evaluates to H'000FFFFF.
(H'0000FFFF | H'00FF00FF)&H'0F0F0F0F	Evaluates to H'000F0F0F.
HIGH H'12345678	Evaluates to H'00000056.
LOW HWORD H'12345678	Evaluates to H'00000034.
~H'000000C3	Evaluates to H'FFFFFF3C.
~H'000000C3&H'000000FF	Evaluates to H'0000003C.
Note: Decimal is the default base in the above examples.	

2.4.3 Operator Descriptions

1. The STARTOF operator

   Returns the starting address of a section group.

   Returns the starting address of the specified section linked by the linkage editor.

2. The SIZEOF operator

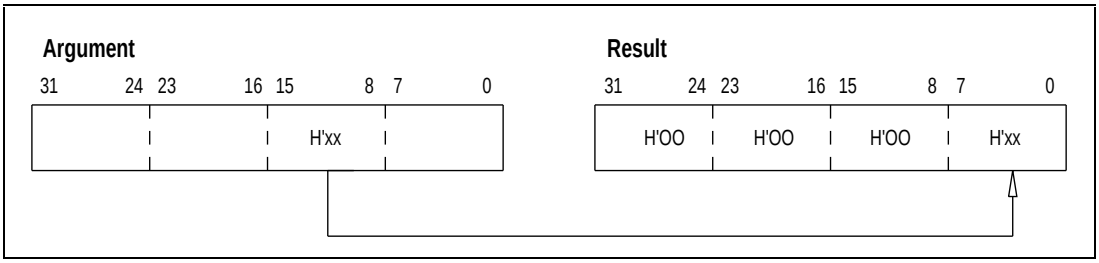
   Returns the size of a section group.

   Returns the size of the specified section linked by the linkage editor.

```
Examples
                .CPU 2600A
                .SECTION       INIT_RAM, DATA, ALIGN=2
                .RES           H'100
;
   .SECTION  INIT_DATA, DATA, ALIGN=2
INIT_BGN  .DATA.L   STARTOF INIT_RAM                          [1]
INIT_END  .DATA.L   STARTOF    INIT_RAM + SIZEOF INIT_RAM     [2]
;
;
                .SECTION MAIN, CODE, ALIGN=2
INITAL:   ]
                MOV.L  @INIT_BGN, ER1       ]
                MOV.L  @INIT_END, ER2       ]
                MOV.W  #0,R3                ]
LOOP:                                       ] The data area of section
                CMP.L  ER1, ER2             ] INIT-RAM is initialized
                BEQ    END                  ] with zero.
                MOV.W  R3, @ER1             ]
                ADDS.L #1, ER1              ]
                BRA    LOOP                 ]
END:                                        ]
                SLEEP
                .END
[1]Returns the starting address of section INIT_RAM.
[2]Returns the end address of section INIT_RAM.
```

3. The HIGH operator

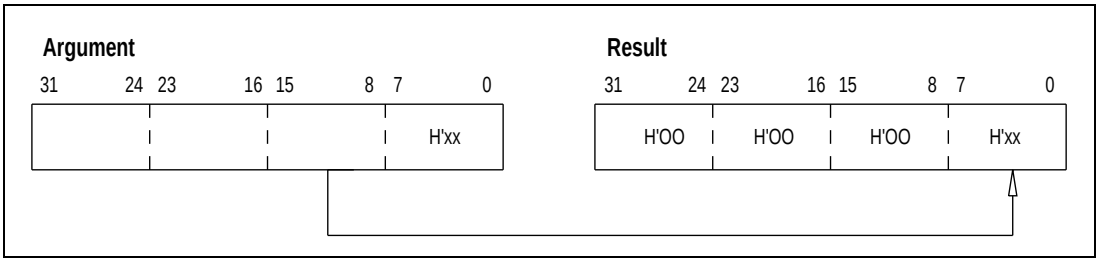
   Extracts the high-order byte of the low-order two bytes of a 4-byte value.




```
Examples
LABEL  .EQU   H'00007FFF
   MOV.W  #HIGH LABEL, R0      ;R0=H'007F
```

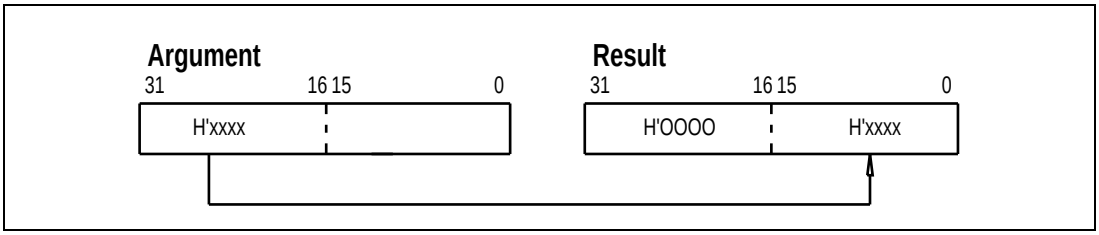
4. The LOW operator

   Extracts the lowest byte of a 4-byte value.



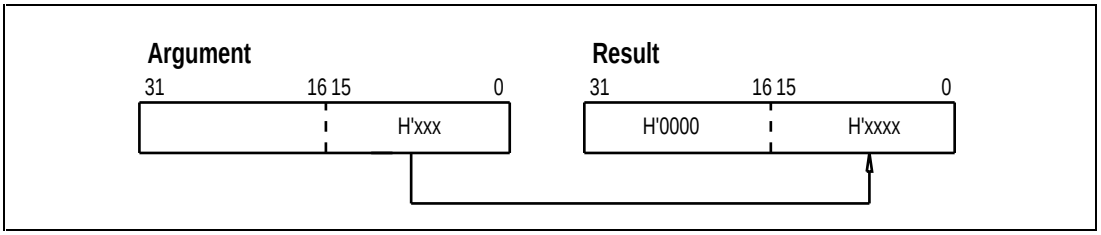
5. The HWORD operator

   Extracts the high-order two bytes of a 4-byte value.



6. The LWORD operator

   Extracts the low-order two bytes of a 4-byte value.



2.4.4 Notes on Operator Usage

1. Operation specifications

   All operations are signed 32-bit operations.

   Even when the operation arguments are byte (8 bit) or word (16 bit) values, the operation will be performed as a signed 32-bit operation. As a result, the following cases generate errors.

```
Examples
   MOV.B  #~H'80:8,R0L The expression "#~H'80:8" generates an error.
The expression "~H'80" is calculated as follows:
   ~  H'80
   ~H'00000080
   H'FFFFFF7F
However, the value H'FFFFFF7F exceeds the range of 8-bit values, and
thus is an error.
Use the following methods to avoid this error.
   MOV.B  #H'7F:8,R0L                Code the result of the operation
                                     directly
   MOV.B  #~H'80&H'FF:8,R0L          Use a logical AND to make
                                     only the lower 8 bits valid.
   MOV.B  #LOW ~H'80:8,R0L           Use the low byte extraction
                                     operation to make only the lower
                                     8 bits valid.
```

2. Arithmetic operations
   a. Relative values may not be specified as operands to the multiplication and division operators. Do not specify relative values as these operands.
   b. Zero (0) may not be specified as the divisor with the division operator. Do not use zero as a divisor.
3. Logical operations

   Relative values may not be specified as operands to logical operators. Do not specify relative values as these operands.
4. Section group operators

   Only section names may be specified as the operand of a section group operator. Do not specify any value other than a section name for these operands.

2.5 Character Strings

Character strings are coded by enclosing the desired characters in double quotation marks (").

The characters which can be used in character strings are the ASCII code H'09 (tab) and ASCII codes from H'20 (space) to H'7E (tilde).

To specify a double quotation mark in a character string, enter two double quotation marks in succession.

```
Examples
.SDATA "ABCDEF"  ; ABCDEF
.SDATA "ABC""DE" ; ABC"DE
```

Section 3 Programming Overview

This section describes methods for external reference and external definition, and the section concept required in programming the various H8S and H8/300 series microprocessors.

3.1 Section

The section is the organizational unit from which programs are constructed, and is a collection of instructions or data. Also, the section is the unit in which the linkage editor links the object modules or allocate them on memory.

When programming, the section attribute and addressing format classification are selected based on the use of that section and on its location in memory.

1. Section attribute

   A section has one of the following section attributes.

   a. Code section

      Code section hold executable instructions and data whose value will not be changed during execution.

      Code section are allocated to ROM.

   b. Data section

      Data section hold data areas both for data that will not be changed during program execution, as well as for data that will be changed during program execution.

      The former type of data section is allocated to ROM, and the later to RAM.

   c. Stack section

      Stack section are regions that contain no initial data, and are used as program stack areas and working areas.

      Stack section are allocated to RAM.

   d. Common section

      Common section hold data areas both for data that will not be changed during program execution, as well as for data that will be changed during program execution. Common section are used when data is shared between partitioned programs.

The former type of common data section is allocated to ROM, and the later to RAM.

e. Dummy section

Dummy section are section that hold no initial data, and hold variable structures.

Dummy section are not output to object modules.

2. Addressing format classification

There are two addressing format classifications as follows:

a. Absolute addressing format

In absolute addressing, the address to which the section is allocated is specified in advance in the source program.

The addresses in an absolute addressing section (section with an absolute addressing format) are not changed by the linkage editor. They are specified using LOCATE in the .SECTION assembler directive.

b. Relative addressing format

In relative addressing, the address to which the section is allocated is not specified in advance in the source program.

The addresses in a relative addressing section (section with a relative addressing format) are determined when the object modules are linked by the linkage editor.

The boundary alignment value which is used when the linkage editor links the object modules can be set for relative addressing sections.

The linkage editor determines addresses so that the section address is a multiple of the boundary alignment value.

The section boundary alignment value is useful when sections must be allocated on even memory addresses, such as the executable instructions in a code section.

A section's attribute and addressing classification are declared using the .SECTION assembler directive.

Examples

```
                .CPU            2600A
                .OUTPUT         DBG
SIZE            .EQU            8
;
                .SECTION A, CODE, ALIGN=2.....................[1]
START                                           ]
                MOV.L           #CONST:32,ER0   ]
                MOV.L           #DATA:32,ER1    ]
                MOV.L           #SIZE:32,ER2    ]
LOOP                                            ]
                CMP.L           #0:32,ER2       ]
                BEQ             EXIT            ]
                MOV.B           @ER0,R3L        ]
                MOV.B           R3L,@ER1        ] Relative addressing code section
                ADD.L           #1:32, ER0      ]
                ADD.L           #1:32, ER1      ]
                SUB.L           #1:32,ER2       ]
                BRA             LOOP            ]
EXIT                                            ]
                SLEEP                           ]
                BRA             START           ]
;
                .SECTION B, DATA, LOCATE=H'00001000.....[2]
CONST
                .DATA.B H'01,H'02,H'03,H'04     ]
                .DATA.B H'05,H'06,H'07,H'08     ] Absolute addressing data section
;
                .SECTION C, STACK, ALIGN=2.....................[3]
DATA                                            ]
        .RES.B  SIZE                            ] Relative addressing stack section
;
.END  START
```

[1] Declares a relative addressing code section
[2] Declares an absolute addressing data section
[3] Declares a relative addressing stack section

3.2 External Reference and External Definition

When programs are divided into separate programs, a given program will need to call subroutines and reference data in other programs. Since subroutines are called and data is referenced by referencing symbols in assembly language programming, symbols defined in other source programs and not in the current program need to be referenced. Referencing a symbol in another program is called external reference, and enabling other programs to reference one's own symbols is called external definition.

External reference symbols and external definition symbols are declared using the .IMPORT and .EXPORT assembler directives.

Examples

External reference symbol declaration

```
    .CPU        2600A
    .OUTPUT     DBG
;
    .IMPORT     SUBRTN.....[1]
    .IMPORT DATA1, DATA2...[1]
;
    .SECTION A, CODE, ALIGN=2
MAIN
    MOV.L       #STACK:32, SP
    MOV.L       @DATA1:32, ER0
    MOV.L       @DATA2:32, ER1
    MOV.L       #0:32, ER2
    JSR         @ SUBRTN:24
    SLEEP
    BRA         MAIN
;
    .SECTION B, STACK, ALIGN=2
    .RES.B      H'500
STACK
;
    END         MAIN
```

External definition symbol declaration

```
    .CPU        2600A
    .OUTPUT     DBG
;
    .EXPORT     SUBRTN.....[2]
    .EXPORT DATA1,      DATA2...[2]
;
    .SECTION A, CODE, ALIGN=2
SUBRTN
    MOV.L       ER0,ER2
    ADD.L       ER1,ER2
    RTS
;
    .SECTION C, DATA, ALIGN=1
DATA1
    .DATA.B H'01
DATA2
    .DATA.B H'02
;
    .END
```

[1] The symbols SUBRTN, DATA1, and DATA2 are external reference symbols.
[2] The symbols SUBRTN, DATA1, and DATA2 are external definition symbols.

Section 4 Executable Instructions

4.1 Coding Executable Instructions

Executable instructions are coded as follows:

Label	Operation	Operand
[Symbol]	Mnemonic[.s]	[Addressing format[, addressing format]]

s (size): {B⇊W⇊L}

1 Label field

A symbol is specified in the label field as necessary.

That symbol will have as its value the location counter (address) at that source statement.

2. Operation field

An executable instruction is specified in the operation field.

The operand size can be specified separated from the instruction mnemonic by a period (.).

The size specifiers are interpreted as follows:

B ..... Byte (1 byte)

W. .... Word (2 bytes)

L ..... Long word (4 bytes)

The sizes that can be specified differ according to the instruction used.

3 .Operand field

The operands of the executable instruction are specified in the operand field.

The number of operands and the operand addressing formats differ according to the instruction used.

4.2 Executable Instruction Operand Sizes and Addressing Formats

4.2.1 H8S/2600 Series

1. H8S/2600 advanced mode

Table 4-1 lists the H8S/2600 advanced mode instruction operand sizes.

Table 4-1 H8S/2600 Advanced Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DEC	B W L (B)	51	ROTL	B W L (B)
2	ADDS	L (L)	27	DIVXS	B W (B)	52	ROTR	B W L (B)
3	ADDX	B (B)	28	DIVXU	B W (B)	53	ROTXL	B W L (B)
4	AND	B W L (B)	29	EEPMOV	B W (B)	54	ROTXR	B W L (B)
5	ANDC	B (B)	30	EXTS	W L (W)	55	RTE	--
6	BAND	B (B)	31	EXTU	W L (W)	56	RTS	--
7	Bcc	--	32	INC	B W L (B)	57	SHAL	B W L (B)
8	BCLR	B (B)	33	JMP	--	58	SHAR	B W L (B)
9	BIAND	B (B)	34	JSR	--	59	SHLL	B W L (B)
10	BILD	B (B)	35	LDC	B W (B)	60	SHLR	B W L (B)
11	BIOR	B (B)	36	LDM	L (L)	61	SLEEP	--
12	BIST	B (B)	37	LDMAC	L (L)	62	STC	B W (B)
13	BIXOR	B (B)	38	MAC	--	63	STM	L (L)
14	BLD	B (B)	39	MOV	B W L (B)	64	STMAC	L (L)
15	BNOT	B (B)	40	MOVFPE	B (B)	65	SUB	B W L (B)
16	BOR	B (B)	41	MOVTPE	B (B)	66	SUBS	L (L)
17	BSET	B (B)	42	MULXS	B W (B)	67	SUBX	B (B)
18	BSR	--	43	MULXU	B W (B)	68	TAS	B (B)
19	BST	B (B)	44	NEG	B W L (B)	69	TRAPA	--
20	BTST	B (B)	45	NOP	--	70	XOR	BW L (B)
21	BXOR	B (B)	46	NOT	B W L (B)	71	XORC	B (B)
22	CLRMAC	--	47	OR	B W L (B)			
23	CMP	B W L (B)	48	ORC	B (B)			
24	DAA	B (B)	49	POP	W L (L)			
25	DAS	B (B)	50	PUSH	W L (L)			

Notes: () ..... Indicates the default size.

-- ..... Indicates that no size specification is allowed.

Table 4-2 lists the H8S/2600 advanced mode addressing modes.

Table 4-2 H8S/2600 Advanced Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{ERn | En | Rn | RnH | RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@(d[:{16 | 32}],ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[:{8 | 16 | 24 | 32}]
7	Immediate data addressing	#xx[: {8 | 16 | 32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[: {8 | 16}]
10	Control register	CCR, EXR, MACH, MACL

Notes: 1 n..... A register number (0 to 7)
d..... A displacement
aa..... An absolute address
xx..... Immediate data

2 The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

2. H8S/2600 normal mode

Table 4-3 lists the H8S/2600 normal mode instruction operand sizes.

Table 4-3 H8S/2600 Normal Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DEC	B W L (B)	51	ROTL	B W L (B)
2	ADDS	L (L)	27	DIVXS	B W (B)	52	ROTR	B W L (B)
3	ADDX	B (B)	28	DIVXU	B W (B)	53	ROTXL	B W L (B)
4	AND	B W L (B)	29	EEPMOV	B W (B)	54	ROTXR	B W L (B)
5	ANDC	B (B)	30	EXTS	W L (W)	55	RTE	--
6	BAND	B (B)	31	EXTU	W L (W)	56	RTS	--
7	Bcc	--	32	INC	B W L (B)	57	SHAL	B W L (B)
8	BCLR	B (B)	33	JMP	--	58	SHAR	B W L (B)
9	BIAND	B (B)	34	JSR	--	59	SHLL	B W L (B)
10	BILD	B (B)	35	LDC	B W (B)	60	SHLR	B W L (B)
11	BIOR	B (B)	36	LDM	L (L)	61	SLEEP	--
12	BIST	B (B)	37	LDMAC	L (L)	62	STC	B W (B)
13	BIXOR	B (B)	38	MAC	--	63	STM	L (L)
14	BLD	B (B)	39	MOV	B W L (B)	64	STMAC	L (L)
15	BNOT	B (B)	40	MOVFPE	B (B)	65	SUB	B W L (B)
16	BOR	B (B)	41	MOVTPE	B (B)	66	SUBS	L (L)
17	BSET	B (B)	42	MULXS	B W (B)	67	SUBX	B (B)
18	BSR	--	43	MULXU	B W (B)	68	TAS	B (B)
19	BST	B (B)	44	NEG	B W L (B)	69	TRAPA	--
20	BTST	B (B)	45	NOP	--	70	XOR	B W L (B)
21	BXOR	B (B)	46	NOT	B W L (B)	71	XORC	B (B)
22	CLRMAC	--	47	OR	B W L (B)			
23	CMP	B W L (B)	48	ORC	B (B)			
24	DAA	B (B)	49	POP	W L (W)			
25	DAS	B (B)	50	PUSH	W L (W)			

Notes: ()..... Indicates the default size.

--..... Indicates that no size specification is allowed.

Table 4-4 lists the H8S/2600 normal mode addressing modes.

Table 4-4 H8S/2600 Normal Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{ERn | En | Rn | RnH | RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@(d[: 16], ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[: {8 | 16}]
7	Immediate data addressing	#xx[: {8 | 16 | 32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[: {8 | 16}]
10	Control register	CCR,EXR,MACH,MACL

Notes: 1. n..... A register number (0 to 7)
d..... A displacement
aa..... An absolute address
xx..... Immediate data
2. The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The REG assembler directive must be used to use "FP" as a register.

4.2.2 H8S/2000 Series

1. H8S/2000 advanced mode

   Table 4-5 lists the H8S/2000 advanced mode instruction operand sizes.

Table 4-5 H8S/2000 Advanced Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DIVXS	B W (B)	51	ROTXR	B W L (B)
2	ADDS	L (L)	27	DIVXU	B W (B)	52	RTE	--
3	ADDX	B (B)	28	EEPMOV	B W (B)	53	RTS	--
4	AND	B W L (B)	29	EXTS	W L (W)	54	SHAL	B W L (B)
5	ANDC	B (B)	30	EXTU	W L (W)	55	SHAR	B W L (B)
6	BANDB	(B)	31	INC	B W L (B)	56	SHLL	B W L (B)
7	Bcc	--	32	JMP	--	57	SHLR	B W L (B)
8	BCLR	B (B)	33	JSR	--	58	SLEEP	--
9	BIAND	B (B)	34	LDC	B W (B)	59	STC	B W (B)
10	BILD	B (B)	35	LDM	L (L)	60	STM	L (L)
11	BIOR	B (B)	36	MOV	B W L (B)	61	SUB	B W L (B)
12	BIST	B (B)	37	MOVFPE	B (B)	62	SUBS	L (L)
13	BIXOR	B (B)	38	MOVTPE	B (B)	63	SUBX	B (B)
14	BLD	B (B)	39	MULXS	B W (B)	64	TAS	B (B)
15	BNOT	B (B)	40	MULXU	B W (B)	65	TRAPA	--
16	BOR	B (B)	41	NEG	B W L (B)	66	XOR	B W L (B)
17	BSET	B (B)	42	NOP	--	67	XORC	B (B)
18	BSR	--	43	NOT	B W L (B)			
19	BST	B (B)	44	OR	B W L (B)			
20	BTST	B (B)	45	ORC	B (B)			
21	BXOR	B (B)	46	POP	W L (L)			
22	CMP	B W L (B)	47	PUSH	W L (L)			
23	DAA	B (B)	48	ROTL	B W L (B)			
24	DAS	B (B)	49	ROTR	B W L (B)			
25	DEC	B W L (B)	50	ROTXL	B W L (B)			

Notes: ()..... Indicates the default size.

--..... Indicates that no size specification is allowed.

Table 4-6 lists the H8S/2000 advanced mode addressing modes.

Table 4-6 H8S/2000 Advanced Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{ERn | En | Rn | RnH | RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@(d[: {16 | 32}], ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[: {8 | 16 | 24 | 32}]
7	Immediate data addressing	#xx[: {8 | 16 | 32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[: {8 | 16}]
10	Control register	CCR, EXR

Notes: 1. n..... A register number (0 to 7)
d..... A displacement
aa..... An absolute address
xx..... Immediate data
2. The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

2. H8S/2000 normal mode

Table 4-7 lists the H8S/2000 normal mode instruction operand sizes.

Table 4-7 H8S/2000 Normal Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DIVXS	B W (B)	51	ROTXR	B W L (B)
2	ADDS	L (L)	27	DIVXU	B W (B)	52	RTE	--
3	ADDX	B (B)	28	EEPMOV	B W (B)	53	RTS	--
4	AND	B W L (B)	29	EXTS	W L (W)	54	SHAL	B W L (B)
5	ANDC	B (B)	30	EXTU	W L (W)	55	SHAR	B W L (B)
6	BAND	B (B)	31	INC	B W L (B)	56	SHLL	B W L (B)
7	Bcc	--	32	JMP	--	57	SHLR	B W L (B)
8	BCLR	B (B)	33	JSR	--	58	SLEEP	--
9	BIAND	B (B)	34	LDC	B W (B)	59	STC	B W (B)
10	BILD	B (B)	35	LDM	L (L)	60	STM	L (L)
11	BIOR	B (B)	36	MOV	B W L (B)	61	SUB	B W L (B)
12	BIST	B (B)	37	MOVFPE	B (B)	62	SUBS	L (L)
13	BIXOR	B (B)	38	MOVTPE	B (B)	63	SUBX	B (B)
14	BLD	B (B)	39	MULXS	B W (B)	64	TAS	B (B)
15	BNOT	B (B)	40	MULXU	B W (B)	65	TRAPA	--
16	BOR	B (B)	41	NEG	B W L (B)	66	XOR	B W L (B)
17	BSET	B (B)	42	NOP	--	67	XORC	B (B)
18	BSR	--	43	NOT	B W L (B)			
19	BST	B (B)	44	OR	B W L (B)			
20	BTST	B (B)	45	ORC	B (B)			
21	BXOR	B (B)	46	POP	W L (W)			
22	CMP	B W L (B)	47	PUSH	W L (W)			
23	DAA	B (B)	48	ROTL	B W L (B)			
24	DAS	B (B)	49	ROTR	B W L (B)			
25	DEC	B W L (B)	50	ROTXL	B W L (B)			

Notes: ()..... Indicates the default size.

--..... Indicates that no size specification is allowed.

Table 4-8 lists the H8S/2000 normal mode addressing modes.

Table 4-8 H8S/2000 Normal Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{ERn | En | Rn | RnH | RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@(d[: {16], ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[: {8 | 16}]
7	Immediate data addressing	#xx[: {8 | 16 | 32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[: {8 | 16}]
10	Control register	CCR, EXR

Notes: 1. n..... A register number (0 to 7)
d..... A displacement
aa..... An absolute address
xx..... Immediate data
2. The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

4.2.3 H8/300H Series

1. H8/300H advanced mode

Table 4-9 lists the H8/300H advanced mode instruction operand sizes.

Table 4-9 H8/300H Advanced Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DIVXS	B W (B)	51	RTE	--
2	ADDS	L (L)	27	DIVXU	B W (B)	52	RTS	--
3	ADDX	B (B)	28	EEPMOV	B W (B)	53	SHAL	B W L (B)
4	AND	B W L (B)	29	EXTS	W L (W)	54	SHAR	B W L (B)
5	ANDC	B (B)	30	EXTU	W L (W)	55	SHLL	B W L (B)
6	BAND	B (B)	31	INC	B W L (B)	56	SHLR	B W L (B)
7	Bcc	--	32	JMP	--	57	SLEEP	--
8	BCL	R B (B)	33	JSR	--	58	STC	B W (B)
9	BIAND	B (B)	34	LDC	B W (B)	59	SUB	B W L (B)
10	BILD	B (B)	35	MOV	B W L (B)	60	SUBS	L (L)
11	BIOR	B (B)	36	MOVFPE	B (B)	61	SUBX	B (B)
12	BIST	B (B)	37	MOVTPE	B (B)	62	TRAPA	--
13	BIXOR	B (B)	38	MULXS	B W (B)	63	XOR	B W L (B)
14	BLD	B (B)	39	MULXU	B W (B)	64	XORC	B (B)
15	BNOT	B (B)	40	NEG	B W L (B)			
16	BOR	B (B)	41	NOP	--			
17	BSET	B (B)	42	NOT	B W L (B)			
18	BSR	--	43	OR	B W L (B)			
19	BST	B (B)	44	ORC	B (B)			
20	BTST	B (B)	45	POP	W L (L)			
21	BXOR	B (B)	46	PUSH	W L (L)			
22	CMP	B W L (B)	47	ROTL	B W L (B)			
23	DAA	B (B)	48	ROTR	B W L (B)			
24	DAS	B (B)	49	ROTXL	B W L (B)			
25	DEC	B W L (B)	50	ROTXR	B W L (B)			

Notes: ()..... Indicates the default size.
--..... Indicates that no size specification is allowed.

Table 4-10 lists the H8/300H advanced mode addressing modes.

Table 4-10 H8/300H Advanced Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	[ERn| En | Rn| RnH| RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@d[:{16 | 24}], ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[:{8 | 16 | 24}]
7	Immediate data addressing	#xx[:{8 |16 |32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[:{8 | 16}]
10	Condition code register	CCR

Notes: 1. n A register number (0 to 7)
d A displacement
aa An absolute address
xx Immediate data
2. The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

2. H8/300H normal mode

Table 4-11 lists the H8/300H normal mode instruction operand sizes.

Table 4-11 H8/300H Normal Mode Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	size	Item No.	Instruction	Size
1	ADD	B W L (B)	26	DIVXS B W	(B) W	51	RTE -	
2	ADDS	L (L)	27	DIVXU B W	(B)	52	RTS	-
3	ADDX	B (B)	28	EEPMOV	B W (B)	53	SHAL	B W L (B)
4	AND	B W L (B)	29	EXTS	W L (W)	54	SHAR	B W L (B)
5	ANDC	B (B)	30	EXTU	W L (W)	55	SHLL	B W L (B)
6	BAND	B (B)	31	INC	B W L (B)	56	SHLR	B W L (B)
7	Bcc	-	32	JMP	-	57	SLEEP	--
8	BCLR	B (B)	33	JSR	-	58	STC	B W (B)
9	BIAND	B (B)	34	LDC	B W (B)	59	SUB	B W L (B)
10	BILD	B (B)	35	MOV	B W L (B)	60	SUBS	L (L)
11	BIOR	B (B)	36	MOVFPE	B (B)	61	SUBX	B (B)
12	BIST	B (B)	37	MOVTPE	B (B)	62	TRAPA	-
13	BIXOR	B (B)	38	MULXS	B W (B)	63	XOR	B W L (B)
14	BLD	B (B)	39	MULXU	B W (B)	64	XORC	B (B)
15	BNOT	B (B)	40	NEG	B W L (B)			
16	BOR	B (B)	41	NOP	-			
17	BSET	B (B)	42	NOT	B W L (B)			
18	BSR	-	43	OR	B W L (B)			
19	BST	B (B)	44	ORC	B (B)			
20	BTST	B (B)	45	POP	W L (W)			
21	BXOR	B (B)	46	PUSH	W L (W)			
22	CMP	B W L (B)	47	ROTL	B W L (B)			
23	DAA	B (B)	48	ROTR	B W L (B)			
24	DAS	B (B)	49	ROTXL	B W L (B)			
25	DEC	B W L (B)	50	ROTXR	B W L (B)			

Notes: () Indicate the default size.

- Indicates that no size specification is allowed.

Table 4-12 lists the H8/300H normal mode addressing modes.

Table 4-12 H8/300H Normal Mode Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{ERn | En | Rn | RnH | RnL}
2	Register indirect addressing	@ERn
3	Register indirect addressing with displacement	@(d[:{16}], ERn)
4	Post-increment register indirect addressing	@ERn+
5	Pre-decrement register indirect addressing	@-ERn
6	Absolute addressing	@aa[:{8 | 16}]
7	Immediate data addressing	#xx[: {8 | 16 | 32}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[:{8 | 16}]
10	Control register	CCR

Notes: 1. n A register number (0 to 7)
d A displacement
aa An absolute address
xx Immediate data
2. The SP notation (for the stack pointer) is identical to the ER7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

4.2.4 H8/300 and H8/300L Series

Table 4-13 lists the H8/300 and H8/300L instruction operand sizes.

Table 4-13 H8/300 and H8/300L Instruction Operand Sizes

Item No.	Instruction	Size	Item No.	Instruction	Size	Item No.	Instruction	Size
1	ADD	B W (B)	26	DIVXU	B (B)	51	SHLL	B (B)
2	ADDS	W (W)	27	EEPMOV	-	52	SHLR	B (B)
3	ADDX	B (B)	28	INC	B (B)	53	SLEEP	--
4	AND	B (B)	29	JMP	-	54	STC	B (B)
5	ANDC	B (B)	30	JSR	-	55	SUB	B W (B)
6	BAND	B (B)	31	LDC	B (B)	56	SUBS	W (W)
7	Bcc	-	32	MOV	B W (B)	57	SUBX	B (B)
8	BCLR	B (B)	33	MOVFPE B	(B)	58	XOR	B (B)
9	BIAND	B (B)	34	MOVTPE	B (B)	59	XORC	B (B)
10	BILD	B (B)	35	MULXU	B (B)			
11	BIOR	B (B)	36	NEG	B (B)			
12	BIST	B (B)	37	NOP -	-			
13	BIXOR	B (B)	38	NOT	B (B)			
14	BLD	B (B)	39	OR	B (B)			
15	BNOT	B (B)	40	ORC	B (B)			
16	BOR	B (B)	41	POP	W (W)			
17	BSET	B (B)	42	PUSH	W (W)			
18	BSR	-	43	ROTL	B (B)			
19	BST	B (B)	44	ROTR	B (B)			
20	BTST	B (B)	45	ROTXL	B (B)			
21	BXOR	B (B)	46	ROTXR	B (B)			
22	CMP	B W (B)	47	RTE	-			
23	DAA	B (B)	48	RTS -	-			
24	DAS	B (B)	49	SHAL	B (B)			
25	DEC	B (B)	50	SHAR	B (B)			

Notes: () Indicates the default size.

- Indicates that no size specification is allowed.

Caution: Instructions 33 (MOVFPE) and 34 (MOVTPE) cannot be used in the H8/300L series.

Table 4-14 lists the H8/300 and H8/300L addressing modes.

Table 4-14 H8/300 and H8/300L Addressing Modes

Item No.	Addressing Mode	Syntax
1	Register direct addressing	{Rn | RnH | RnL}
2	Register indirect addressing	@Rn
3	Register indirect addressing with displacement	@(d[:16], Rn)
4	Post-increment register indirect addressing	@Rn+
5	Pre-decrement register indirect addressing	@-Rn
6	Absolute addressing	@aa[:{8 | 16}]
7	Immediate data addressing	#xx[:8 | 16}]
8	Memory indirect addressing	@@aa[:8]
9	Program counter relative addressing (branch instruction displacement)	d[:[8]
10	Control register	CCR

Notes: 1. n A register number (0 to 7)
d A displacement
aa An absolute address
xx Immediate data
2. The SP notation (for the stack pointer) is identical to the R7 notation.

Caution: The "FP" notation is not a register mnemonic. The .REG assembler directive must be used to use "FP" as a register.

Section 5 Assembler Directive

5.1 Assembler Directive Types

Assembler directives (hereafter abbreviated as directives) are instructions that the assembler interprets and executes. In other words, the assembler assembles the source program according to the specifications of the directives included in the program.

Table 5-1 gives an overview of the assembler directives.

Table 5-1 Assembler Directive List

Item Number	Directive	Function	Manual Section
1	CPU Directives		
	.CPU	CPU specification	5.2.1
2	Section and location counter directives		
	.SECTION	Section declaration	5.3.1
	.ORG	Location counter value assignment	5.3.2
	.ALIGN	Location counter value boundary alignment	5.3.3
3	Symbol directives		
	.EQU	Symbol value assignment	5.4.1
	.ASSIGN	Symbol value assignment	5.4.2
	.REG	Register alias assignment	5.4.3
	.BEQU	Bit data name assignment	5.4.4
4	Data area allocation directives		
	.DATA	Integer data allocation	5.5.1
	.DATAB	Integer data block allocation	5.5.2
	.SDATA	Character string data allocation	5.5.3
	.SDATAB	Character string data block allocation	5.5.4
	.SDATAC	Counted character string data allocation	5.5.5
	.SDATAZ	Zero terminated character string data allocation	5.5.6
	.RES	Integer data area allocation	5.5.7
	.SRES	Character string data area allocation	5.5.8
	.SRESC	Counted character string data area allocation	5.5.9
	.SRESZ	Zero terminated character string data area allocation	5.5.10

Item Number	Directive	Function	Manual Section
5	External reference and external definition directives		
	.IMPORT	External reference symbol declaration	5.6.1
	.EXPORT	External definition symbol declaration	5.6.2
	.GLOBAL	External reference and external definition symbol declaration	5.6.3
6	Object module directive		
	.OUTPUT	Object module and debugging information output control	5.7.1
	.DEBUG	Debugging information output control	5.7.2
	.LINE	Changes the file name and the line number in debugging information	5.7.3
	.DISPSIZE	The displacement size specification	5.7.4
7	Assembly listing directives		
	.PRINT	Assembly listing output control	5.8.1
	.LIST	Source program listing partial output control	5.8.2
	.FORM	Listing size control	5.8.3
	.HEADING	Listing heading specification	5.8.4
	.PAGE	New page	5.8.5
	.SPACE	Blank line output	5.8.6
8	Other directives		
	.PROGRAM	Object module name specification	5.9.1
	.RADIX	Radix specification	5.9.2
	.END	Source program termination	5.9.3

5.2 CPU Directives

5.2.1 .CPU CPU Specification

Format

Label	Operation	Operand
X	.CPU	<CPU classification>
		[:<address space bit width>]

CPU Classification: {2600A | 2600N | 2000A | 200HA | 300HN | 300 | 300L

CPU Clarification	Address Space Bit Width	Default Size
2600A	20, 24 28, 32	32
2000A	20, 24, 28, 32	32
300HA	20, 24	24

Description

This directive specifies the object CPU for the source program.

The address space bit width can be selected only when the CPU for advanced mode is selected. The accessible space depends on the selected bit width.

The assembler code for the specified CPU.

The CPU classifications are interpreted as follows:

2600A[:32], 2600A:28,

2600A:24, 2600A:20 ..... H8S/2600 advanced mode

2600N ..... H8S/2600 normal mode

2000A[:32], 2000A:28,

2000A:24, 2000A:20 ..... H8S/2000 advanced mode

2600N ..... H8S/2600 normal mode

300HA [:24], 300HA:20 ..... H8/300H advanced mode

300HN ..... H8/300H normal mode

300 ..... H8/300

300L H8/300L

This directive should be used at the beginning of the source program. It is an error for any directive other than an assembly listing directive to precede the .CPU directive.

This directive is valid only once in a given program.

This directive is valid when there is no "CPU" command line option to the assembler.

If both this directive and the "CPU" command line option are not specified, the CPU classification specified by the H38CPU environmental variable is valid.

Refer to section 2.2, H38CPU Environmental Variable, in the Usage Guide, for details on using the H38CPU environmental variable.

Example

```
.CPU 300HA:20
.SECTION A, CODE, ALIGN=2
MOV.W R0, R1
MOV.W R0, R2
```

In this example, the source program will be assembled for the 1M mode in H8/300H advanced movd.

5.3 Section and Location Counter Directives

5.3.1 .SECTION Section Declaration

Format

Label	Operation	Operand
X	.SECTION	<section name> [,<section attribute>
		[,<format classification>]]
<section attribute>:	{CODE | DATA | STACK | COMMON | DUMMY}	
<format classification>:	{LOCATE=<starting address> | ALIGN=<boundary alignment value>}	

Description This directive declares the start of a section or restart to a section.

1. Section start

   A section is started, and the section name, attribute, and format classification are specified.

   a. Section name

      This operand specifies the section name. The syntax for section names is the same as the syntax for symbols. Note that upper and lower case letters are distinguished in section names.

   b. Section attribute

      This operand specifies the section attribute. One of the following section attributes may be specified.

      CODE..... Code section
      DATA..... Data section
      STACK..... Stack section
      COMMON..... Common section
      DUMMY..... Dummy section

CODE is used as the default when no section attribute is specified.

c. Format classification

This operand specifies the addressing format classification. The following format classifications may be specified.

LOCATE=<Starting address> ..... Absolute addressing format

ALIGN=<boundary alignment value>..... Relative addressing format

"ALIGN=2" is used as the default when no format is specified.

In absolute addressing, the starting address of the section is specified.

The starting address must be a backward reference absolute value.

The maximum value of the starting address for each CPU is listed below.

- H8S/2600 advanced mode
  2600A[:32] .....H'FFFFFFFF
  2600A:28 .....H'0FFFFFFF
  2600A:24 .....H'00FFFFFF
  2600A:20 .....H'000FFFFF
- H8S/2600 normal mode
  2600N .....H'0000FFFF
- H8S/2000 advanced mode
  2000A [:32] .....H'FFFFFFFF
  2000A:28 .....H'0FFFFFFF
  2000A:24.....H'00FFFFFF
  2000A:20 .....H'000FFFFF
- H8S/2000 normal mode
  2000N .....H'0000FFFF
- H8/300H advanced mode
  300HA [:24] .....H'00FFFFF
  300HA:20 .....H'000FFFFF

- H8/300H normal mode

  300HN .....H'0000FFFF
- H8/300

  H8/300 .....H'0000FFFF
- H8/300L

  H8/300L .....H'0000FFFF

  In relative addressing, the boundary alignment value is specified.

  The linkage editor adjusts the starting address of relative addressing section to be a multiple of the boundary alignment value.

  The boundary alignment value must be a backward reference absolute value.

  The boundary alignment value must be a power of 2(2n).

  If no section has been declared using this directive, the assembler operates with the following as the default section.

  .SECTION P, CODE, ALIGN=2

2. Section restart

   The assembler restarts a section that was previously declared.

   In section restart the section name of a previously declared section is specified. The originally declared section attribute and format classification are maintained in the restarted section.

   Examples

```
   .SECTION A, CODE, ALIGN=2 .....................[1]
   MOV.W  R0, R1
   .SECTION B, DATA, LOCATE=H'1000 ...............[2]
DATA1
   .DATA.W   H'0001
   .SECTION A                          ............[3]
   MOV.W  R0, R3
```

   [1] Starts section "A". The section name is "A", the section attribute is code section, and the format classification is relative addressing with a boundary alignment value of 2.

   [2] Starts section "B". The section name is "B", the section attribute is data section, and the format classification is absolute addressing with a starting address at location H'1000.

   [3] This directive restarts section "A". The section name is "A", the section attribute is code section, and the format classification is relative addressing with a boundary alignment value of 2.

5.3.2 .ORG Location Counter Value Assignment

Format

Label	Operation	Operands
X	.ORG	<location counter value>

Description

This directive sets the value of the location counter within the section to the specified value.

The <location counter value> must be a backward reference absolute value or backward reference to an address value in a relative address section.

The maximum value of the <location counter value> for each CPU is listed below.

- H8S/2600 advanced mode
  2600A [:32] .....H'FFFFFFFF
  2600A :28 .....H'0FFFFFFF
  2600A :24 .....H'00FFFFFF
  2600A :20 .....H'000FFFFF
- H8S/2600 normal mode
  2600N .....H'OOOOFFFF
- H8S/2000 advanced mode
  2000A[:32] .....H'FFFFFFFF
  2000A :28 .....H'OFFFFFFF
  2000A :24 .....H'OOFFFFFF
  2000A :20 .....H'OOOFFFFF
- H8S/2000 normal mode
  2000N .....H'OOOOFFFF
- H8/300H advanced mode
  300HA [:24] .....H'OOFFFFFF
  300HA :20 .....H'OOOFFFFF
- H8/300H normal mode
  300HN .....H'OOOOFFFF
- H8/300
  300 .....H'OOOOFFFF
- H8/300L

300L .....H'OOOOFFFF

When specified in an absolute addressing section, the specified location counter value must be a location following (i.e. greater than) the starting address of the section.

When this directive is used in an absolute addressing section, the specified location counter value is an absolute address, and when used in a relative addressing section, the specified location counter value is a relative address.

Examples

```
    .SECTION A, DATA, LOCATE = H'0100
DATA1
    .DATA.W H'0001
    .DATA.W H'0002
    .ORG H'0200     ........[1]
DATA2
    .DATA.W H'0003
    .DATA.W H'0004
```

[1] The value of the location counter is changed to absolute location H'0200.

```
    .SECTION B, DATA, ALIGN=2
DATA3
    .DATA.W H'0001
    .DATA.W H'0002
    .ORG H'0300     ........[2]
DATA4
    .DATA.W H'0003
    .DATA.W H'0004
```

[2] The value of the location counter is changed to relative location H'0300 in section A.

5.3.3 .ALIGN Location Counter Value Boundary Alignment

Format

Label	Operation	Operands
X	.ALIGN	<boundary alignment value>

Description

This directive adjusts the value of the location counter to be a multiple of the <boundary alignment value> parameter.

The boundary alignment value must be a backward reference absolute value.

The boundary alignment value must be a power of 2(2n).

The boundary alignment value is set based on the starting address of a relative addressing section.

The boundary alignment value depends on the selected address space.

Examples

```
    .CPU 2600A
    .SECTION A, DATA, ALIGN=2   [1]
DATA1
    .DATA.W H'0001
DATA2
    .DATA.B H'02 [2]
    .ALIGN 2  [3]
DATA3
    .DATA.W H'0003
    .END
```

[1] The starting address of the relative addressing section is adjusted to be a multiple of 2 (an even number) at object module linkage.

[2] The location counter value for the next data becomes an odd location since a single byte of data is allocated.

[3] The location counter value is adjusted to be a multiple of 2 (an even number).

5.4 Symbol Directives

5.4.1 .EQU Symbol Value Assignment

Format

Label	Operation	Operands
<symbol>	.EQU	<value>

Description

This directive assigns a value to a symbol.

The value must be either a backward reference absolute value or a backward reference address value.

The value of symbols defined with this directive cannot be changed.

Examples

```
SYM1   .EQU   1
SYM2   .EQU   2
       .SECTION A, CODE, ALIGN=2
       MOV.B #SYM1:8,ROL .............[1]
       MOV.B #SYM2:8,R1L .............[2]
```

[1] This is the same as "MOV.B#1:8, ROL."
[2] This is the same as "MOV.B#2:8, R1L."

The value 1 is assigned to the symbol SYMI, and the value 2 to the symbol SYM2.

5.4.2 .ASSIGN Symbol Value Assignment

Format

Label	Operation	Operands
<symbol>	.ASSIGN	<value>

Description

This directive assigns a value to a symbol.

The value must be either a backward reference absolute value or a backward reference address value.

This directive can be used to change the value of a symbol originally defined using this directive.

The values returned when referencing symbols defined with this directive behave as shown in the example below.

```
Example
    MOV.B  #SYM:8,ROL .............The value of SYM is 0.
SYM .ASSIGN 1
    MOV.B  #SYM:8, R1L .............The value of SYM is 1.
SYM .ASSIGN 2
    MOV.B  #SYM:8,R2L .............The value of SYM is 2.
```

Symbols defined with this directive cannot be defined to be external symbols.

Symbols defined with this directive cannot be referenced by the simulator/debugged.

Examples

```
        SYM1    .ASSIGN 1
        SYM2    .ASSIGN 2
                .SECTION A, CODE, ALIGN=2
                MOV.B  #SYM1:8,ROL     [1]
                MOV.B  #SYM2:8,R1L     [2]
SYM2    .       ASSIGN 3
                MOV.B  #SYM1:8,R2L     [3]
                MOV.B  #SYM2:8,R3L     [4]
```

[1] This is the same as "MOV.B#1:8, ROL."
[2] This is the same as "MOV.B#2:8, R1L."
[3] This is the same as "MOV.B#1:8, R2L."
[4] This is the same as "MOV.B#3:8, R3L."

The value 1 is assigned to SYM1 and the value 2 to SYM2. The value of SYM2 is then changed to 3.

5.4.3 .REG Register Alias Assignment

Format

Label	Operation	Operands
<symbol>	.REG	(<register>)

Description

This directive specifies an alias for one or more registers.

An alias is specified in the following ways:

- Single register

  One alias is specified for one register. The register alias can be used in any position where a register can be specified. A general register can be specified as the register.
- Multiple registers

  One alias is specified for two or more registers. This specification is allowed only for the H8S/2600 or H8S/2000 series of CPUs. Multiple registers can be specified as operands of this directive and the LDM and STM instructions. 32-bit general registers can be specified as multiple registers.

  The following table summarizes how to specify aliases for registers.

Specification	Description	Examples
Single register	One register is specified from among RO to R7, EO to E7, or ERO to ER7.	SINGLREG .REG (RO) The alias SINGLREG is specified for register RO.
Multiple registers	Multiple registers are specified as a range of registers expressed with a hyphen(-). If the register on the left side of the hyphen is numerically greater than that on the right side, an error occurs and this directive is ignored.	RNG1 .REG (ERO-ER3) The alias RNG1 is specified for registers ERO, ER1, ER2, and ER3. RNG2 .REG (ER3-ERO) The right register (ERO) is less than the left register (ER3); an error occurs. (See note.)
Register alias respecification	A defined register alias is specified as an operand.	ER00 .REG (ER0-ER3) ER01 .REG (ER00)
		The alias ER01 is specified for registers ER0 to ER3.

Note: Register combinations are the following:

ERO-ER1, ER2-ER3, ER4-ER5, ER6-ER7, ER0-ER2, ER4-ER6, ER0-ER3, and ER4-ER7.

Register aliases defined by this directive cannot be redefined.

A register alias defined by this directive is valid in all source statements following the defining directive statement.

Register aliases defined by this directive cannot be defined as external symbols.

Symbols defined by this directive cannot be referenced by the simulator/debugged.

Examples

```
        .CPU 2600A
RLST1: .REG (R0) .................[1]
RLST2: .REG (ER0-ER2) ............[2]
       MOV.W RLST1,@R6
       LDM.L @SP+, (RLST2)
       STM.L (RLST2), @-SP
```

[1] The alias RLST1 is specified for register R0.
[2] The alias RLST2 is specified for registers ERO, ER1, and ER2.

5.4.4 .BEQU Bit Data Name Assignment

Format

Label	Operation	Operands
<symbol>	.BEQU	<bit number>, <replacement symbol>

Description

This directive assigns a name to a single memory data bit which will then be used as the object of a bit manipulation instruction.

The bit data name can then be used as the operand for bit manipulation instructions. The specified bit data name is replaced with an address of the form #xx,@aa.

Example

```
BITO    .BEQU     0,ADDRESS .....Defines a bit data name.
        BSET.B    BITO:8    .....That bit data name is then specified
                                  as the operand of a bit operation
                                  instruction.
```

Code is generated for the "BSET.B BITO:8" instruction as though it were written "BSET.B #0,@ADDRESS:8."

The bit number must be a backward reference absolute value.

The bit number must have a value in the range 0 to 7.

When the CPU is H8S/2600 series or H8S/2000 series: A symbol which can be specified in the 8-bit absolute addressing format @aa:8), 16-bit absolute addressing format @aa:16), or 32-bit absolute addressing format @aa:32) is specified for the replacement symbol.

When the CPU is H8/300H series, H8/300 series: A symbol which can be specified in the 8-bit absolute addressing format @aa:8) is specified for the replacement symbol.

The following bit manipulation instructions can use a bit data name.

BSET, BCLR, BNOT, BTST, BAND, BIAND, BOR, BIOR, BXOR,

BIXOR, BLD, BILD, BST, BIST

Bit data names defined with this directive are valid in all source statements following the defining directive.

Bit data names defined with this directive cannot be defined as external symbols.

Bit data names defined with this directive cannot be referenced by the simulator/debugged.

Examples

```
        .CPU   2600A
AD1     .EQU   H'FFFFFF00
AD2     .EQU   H'FFFF8000
AD3     .EQU   H'FF000000
AD1B0   .BEQU  0,AD1
AD1B1   .BEQU  1,AD1
AD2B2   .BEQU  2,AD2
AD2B3   .BEQU  3,AD2
AD3B4   .BEQU  4,AD3
AD3B5   .BEQU  5,AD3
        .SECTION A, CODE,ALIGN=2
        BSET.B AD1B0 ..............  [1]
        BSET.B AD1B1 ..............  [2]
        BCLR.B AD2B2 ..............  [3]
        BCLR.B AD2B3 ..............  [4]
        BNOT.B AD3B4 ..............  [5]
        BNOT.B AD3B5 ..............  [6]
```

These bit data names refer to the following bits:

```
    AD1B0 .............. Bit 0 of location H'FFFFFF00.
    AD1B1 .............. Bit 1 of location H'FFFFFF00.
    AD2B2 .............. Bit 2 of location H'FFFF8000.
    AD2B3 .............. Bit 3 of location H'FFFF8000.
    AD3B4 .............. Bit 4 of location H'FF000000.
    AD3B5 .............. Bit 5 of location H'FF000000.
```

[1] This is the same as "BSET.B#0,@AD1:8".

[2] This is the same as "BSET.B#1,@AD1:8".

[3] This is the same as "BCLR.B#2,@AD2:16".

[4] This is the same as "BCLR.B#3,@AD2:16".

[5] This is the same as "BNOT.B#4,@AD3:32".

[6] This is the same as "BNOT.B#5,@AD3:32".

5.5 Data Area Allocation Directives

5.5.1 .DATA Integer Data Allocation

Format

Label	Operation	Operands
[<symbol>]	.DATA[.s]	<integer data>[,<integer data> ...]

s(size): {B | W | L}

Description

This directive allocates integer data according to the specified size.

The size specifies are interpreted as follows:

B ..... Byte (1 byte)

W ..... Word (2 bytes)

L ..... Long word (4 bytes)

Byte is taken as the default when a size specification is omitted.

The values which may be specified for the integer data vary with the size as shown below.

B ..... -128 to 255

W ..... -32,768 to 65,535

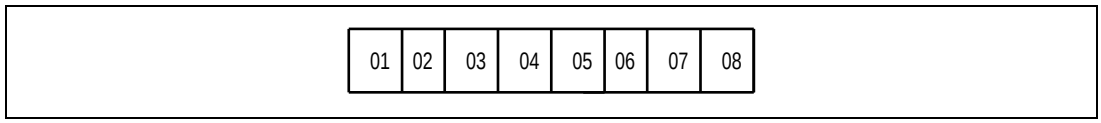
L ..... -2,147,483,648 to 4,294,967,295

Integer data depends on the selected address space.

Examples

```
.SECTION A, DATA,ALIGN=2
.DATA.W H'0102,H'0304
.DATA.B H'05,H'06,H'07,H'08
```

The following data is allocated.



5.5.2 .DATAB Integer Data Block Allocation

Format

Label	Operation	Operands
[<symbol>]	.DATAB[.s]	<block count>, <integer data>

s(size):{B | W | L}

Description

This directive allocates <block count> units of integer data in the specified size.

The size specifies are interpreted as follows:

B ..... Byte (1 byte)

W ..... Word (2 bytes)

L ..... Long word (4 bytes)

Byte is taken as the default when a size specification is omitted.

The block count must be a backward reference absolute value.

values of 1 or more can be specified as the block count.

The values which may be specified for the integer data vary with the size as shown below.

B ..... -128 to 255

W ..... -32, 768 to 65, 535

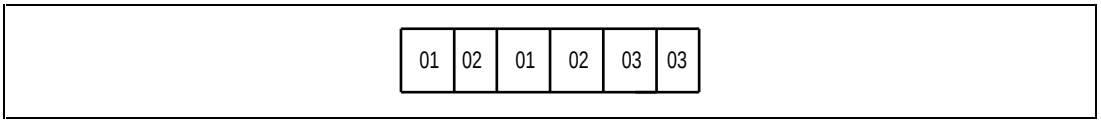
L ..... -2, 147, 483, 648 to 4, 294, 967, 295

The block count depends on the selected address space.

Examples

```
.SECTION A, DATA, ALIGN=2
.DATAB.W 2, H'0102
.DATAB.B 2, H'03
```

The following data is allocated.



5.5. 3 .SDATA Character String Data Allocation

Format

Label	Operation	Operands
[<symbol>]	.SDATA	"<character string>"[,
		"<character string>"...]

Description

This directive allocates character string data.

Character strings are specified by the character surrounded by double quotation marks.

To specify a double quotation mark in a character string, enter two double quotation marks in succession.

To specify control codes in character strings, insert the control codes surrounded by angle brackets (<>) immediately following the character string surrounded by double quotation marks.

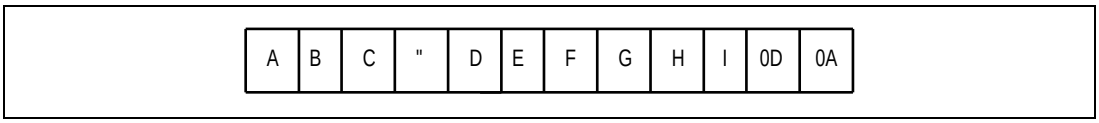
"character string"<control code>

Control codes are specified with backward reference absolute values.

Examples

```
.SECTION A,DATA,ALIGN=2
.SDATA "ABC" "DEF"
.SDATA "GHI" <H'OD><H'OA>
```

The following data is allocated.



5.5.4 .SDATAB Character String Data Block Allocation

Format

Label	Operation	Operands
[<symbol>]	.SDATAB	<block count>, "<character string>"

Description

This directive allocates <block count> units of character string data.

The block count must be a backward reference absolute value.

Values of 1 or more can be specified as the block count.

The block count depends on the selected address space.

Character strings are specified by the characters surrounded by double quotation marks(").

To specify a double quotation mark in a character string, enter two double quotation marks in succession(" ").

To specify control codes in character strings, insert the control codes surrounded by angle brackets (<>) immediately following the character string surrounded by double quotation marks.

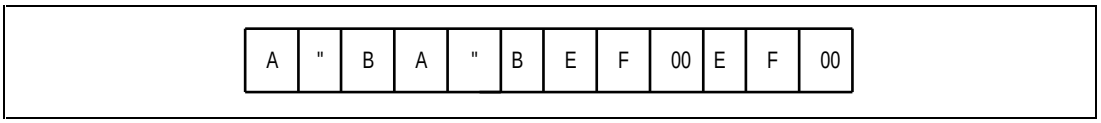
"character string"<control code>

Control codes are specified with backward reference absolute values.

Examples

```
.SECTION A,DATA,ALIGN=2
.SDATAB 2, "A" "B"
.SDATAB 2, "EF"<H'00>
```

The following data is allocated.



5.5.5 .SDATAC Counted Character String Data Allocation

Format

Label	Operation	Operands
[<symbol>]	.SDATAC	"<character string>"[,
		"<character string>"...]

Description

This directive allocates counted character string data.

When allocating character string data, this directive attaches the count (the number of bytes in the character string) at the start of the character string data. The count does not include the byte taken by the count itself.

Character strings are specified by characters surrounded by double quotation marks(").

To specify a double quotation mark in a character string, enter two double quotation marks in succession(" ").

To specify control codes in character strings, insert the control codes surrounded by angle brackets (<>) immediately following the character string surrounded by double quotation marks.

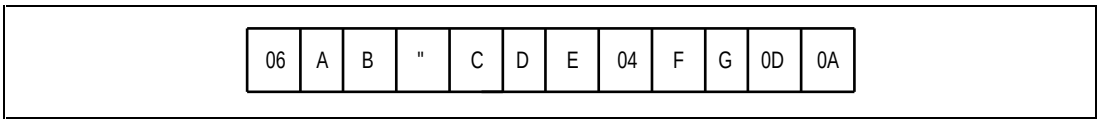
"character string"<control code>

Control codes are specified with backward reference absolute values.

Examples

```
.SECTION A,DATA,ALGIN=2
.SDATAC "AB" "CDE", "FG" <H'0D><H'0A>
```

The following data is allocated.



5.5.6 .SDATAZ Zero Terminated Character String Data Allocation

Format

Label	Operation	Operands
[<symbol>]	.SDATAZ	"<character string>"[,
		"<character string>"...]

Description

This directive allocates zero terminated character string data.

When allocating character string data, this directive attaches a single byte with the value 0 at the end of the allocated data.

Character strings are specified by characters surrounded by double quotation marks(").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

To specify control codes in character strings, insert the control codes surrounded by angle brackets (<>) immediately following the character string surrounded by double quotation marks.

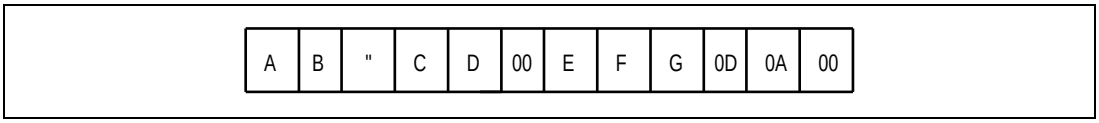
"character string"<control code>

Control codes are specified with backward reference absolute values.

Examples

```
.SECTION A,DATA,ALIGN=2
.SDATAZ "AB" "CD", "EFG" <H'OD><H'OA>
```

The following data is allocated.



5.5.7 .RES Integer Data Area Allocation

Format

Label	Operation	Operands
<symbol>]	.RES[.s]	<area count>

s (size): {B | W | L}

Description

This directive reserves (allocates) an integer data area.

An area of <area count> units of data of the specified size is allocated.

The size specifies are interpreted as follows:

B ..... Byte (1 byte)

W ..... Word (2 bytes)

L ..... Long word (4 bytes)

Byte is taken to be the default when the specification is omitted.

The area count must be a backward reference absolute value.

Values of 1 or more can be specified as the area count.

The area count depends on the selected address space.

Examples

```
.SECTION A, DATA, ALIGN=2
.RES.W 10
.RES.B 255
```

A20-byte area and a 255-byte area are allocated.

5.5.8 .SRES Character string Data Area Allocation

Format

Label	Operation	Operands
[<symbol>]	.SRES	<area size>[,<area size>...]

Description

This directive reserves (allocates) character string data areas.

Areas of the specified area size (byte count) are allocated.

The area size must be a backward reference absolute value.

Values of 1 or more can be specified as the area size.

The area size depends on the selected address space.

Examples

```
.SECTION A,DATA,ALIGN=2
.SRES 10,20,30
.SRES 255
```

Areas of 10, 20, and 30 bytes, and then an area of 255 bytes, are allocated.

5.5.9 .SRESC Counted Character string Data Area Allocation

Format

Label	Operation	Operands
[<symbol>]	.SRESC	<area size>[,<area size>...]

Description

This directive reserves (allocates) counted character string data areas.

Areas of the specified area size (byte count) plus 1 (for the count byte) are allocated.

The area size must be a backward reference absolute value.

Values from 0 to 255 can be specified as the area size.

Examples

```
.SECTION A,DATA,ALIGN=2
.SRESC 10,20,30
.SRESC 255
```

Areas of 11,21, and 31 bytes, and then an area of 256 bytes, are allocated.

5.5.10 .SRESZ Zero Terminated Character String Data Area Allocation

Format

Label	Operation	Operands
[<symbol>]	.SRESZ	<area size>[,<area size> ...]

Description

This directive reserves (allocates) zero terminated character string data areas.

Areas of the specified area size (byte count) plus 1 (for the termination byte) are allocated.

The area size must be a backward reference absolute value.

Values from 0 to 255 can be specified as the area size.

Examples

```
.SECTION A,DATA,ALIGN=2
.SRESZ 10,20,30
.SRESZ 255
```

Areas of 11,21, and 31 bytes, and then an area of 256 bytes, are allocated.

5.6 External Reference and External Definition Directives

5.6.1 .IMPORT External Reference Symbol Declaration

Format

Label	Operation	Operands
X	.IMPORT	<symbol>[,<symbol>...]

Description

This directive declares external reference symbols.

This directive is used when symbols defined in another program are referenced. The symbols defined in a source program cannot be declared as the external reference symbols within the program.

When symbols are declared to be external reference symbols and referenced externally, it is necessary for those symbols to be declared to be external definition symbols with the .EXPORT directive in the defining program.

Examples

```
    .CPU   2600A
    .IMPORT   PROG   [1]
;
    .SECTION A, CODE, ALIGN=2
START
    MOV.L  #STACK:32,SP
    MOV.L  #'12345678:32,ER0
    JSR@PROG:24  [2]
    SLEEP
    BRA START
;
    .SECTION B, STACK, ALIGN=2
    .RES.B H'500
STACK
;
    .END   START
```

[1] PROG is declared to be an external reference symbol.
[2] PROG is referenced.

The symbol PROG is referenced externally in this example.

5.6.2 .EXPORT External Definition Symbol Declaration

Format

Label	Operation	Operands
X	.EXPORT	<symbol>[,<symbol>...]

Description

This directive declares external definition symbols.

Symbols defined within a source program are declared to be external definition symbols with this directive when they are referenced in another program.

The following types of symbols can be declared to be external definition symbols.

Symbols which hold an absolute value

Symbols which hold an address value

Note that symbols defined with the .ASSIGN directive, and symbols which hold an address value in a dummy section cannot be specified to be external definition symbols.

To reference symbols declared to be external definition symbols with this directive, the referencing program must declare those symbols to be external reference symbols with the .IMPORT directive.

```
Examples  .CPU   2600A
    .EXPORT PROG ..............[1]
;
    .SECTION A, CODE, ALIGN=2
PROG ........................[2]
    MOV.L  ER0, ER1
    MOV .L ER0, ER3
    RTS
;
    .END
```

[1] PROG is declared to be an external definition symbol.
[2] PROG is defined.

The symbol PROG is defined externally in this example.

5.6.3 .GLOBAL External Reference and External Definition Symbol Declaration

Format

Label	Operation	Operands
X	.GLOBAL	<symbol>[,<symbol>...]

Description

This directive declares external reference and external definition symbols.

This declaration is used when symbols defined in other source programs are referenced, and also when symbols defined within the current source program are to be referenced from other source programs.

This directive declares symbols that are not defined in the current source program to be external reference symbols, and declares symbols defined in the source program to be external definition symbols.

The following types of symbols can be declared to be external definition symbols.

- Symbols which hold an absolute value.
- Symbols which hold an address value.

Note that symbols defined with the .ASSIGN directive, and symbols which hold an address value in a dummy segment cannot be specified.

Examples

```
    .CPU      2600A
    .GLOBAL   PROG2 ..........[1]
    .GLOBAL   PROG3
;
    .SECTION C,CODE, ALIGN=2
PROG2 ......................[2]
    MOV.L     ER0, ER1
    JSR       @PROG3:24 [3]
    MOV.L     ER1, ER2
    RTS
;
    .END
```

[1] PROG2 is declared to be an external reference symbol and PROG3 is declared to be an external definition symbol.

[2] PROG2 is defined.

[3] PROG3 is referenced.

In this program, the symbol PROG2 is taken to be an external definition symbol, and the symbol PROG3 is taken to be an external reference symbol.

5.7 Object Module Directives

5.7.1 .OUTPUT Object Module and Debugging Information Output Control

Format

Label	Operation	Operands
X	.OUTPUT	<output classifier>[, <output classifier>]

<output classifier>: {OBJ | NOOBJ | DBG | NODBG}

Description

This directive controls object module and debugging information output.

1. Object module output

   This directive can be used to specify the output of an object module or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   OBJ .....Output

   NOOBJ .....Output suppressed

   OBJ is used as the default if neither of these classifiers is specified.

   This specification is only valid once in a given source program.

   This specification is only valid when the [NO]OBJECT command line option has not been specified.

2. Debugging information output

   This directive can be used to specify the output of debugging information or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   DBG .....Output

   NODBG .....Output suppressed

   NODBG is used as the default if neither of these classifiers is specified.

This specification is only valid once in a given source program.

This specification is only valid when the [NO]DEBUG command line option has not been specified.

This specification is only valid when an object module is output.

Examples

```
.OUTPUT OBJ, DBG
```

This instruction specifies that both an object module and debugging information are output.

5.7.2 .DEBUG Debugging Symbol Information Output Control

Format

Label	Operation	Operands
X	.DEBUG	<output classifier>

<output classifier>: {ON | OFF}

Description

This directive specifies the output or suppression of symbol information in the debugging information.

This directive is used when only those symbols in the source program that are actually required for debugging are to be output.

The output classifiers for this function are interpreted as follows:

ON .....Output

OFF .....Output suppressed

The initial setting of this output classifier is ON.

This directive can be used as many times as desired in a source program, and the specified output classifier remains valid for all source statements until the next occurrence of the directive.

This specification is only valid when debugging information is output.

Examples

```
     .OUTPUT DBG
     .DEBUG OFF
     .SECTION A, CODE, ALIGN=2
START
    MOV .W @DAT:16, R1
     .SECTION B, DATA, ALIGN=2
     .DEBUG ON
DAT
     .DATA.W H'1234
```

The symbol DAT will be output to the debugging information.

5.7.3 .LINE Changes the File Name and Line Number of Debugging Information

Format

Label	Operation	Operands
X	.LINE	["<file name",] line number

Description

This directive changes the file name and line number of debugging information; it supports C source level debugging.

This directive is embedded by the C compiler into an assembly source program output by the C compiler. This enables C source level debugging even after the assembly source program output by the C compiler is assembled.

The assembler manages the file name and line number specified by this directive beginning with the line following this directive.

The file name and line number specified by this directive are valid only in the program containing this directive.

```
> ch38  -cpu=2600a -code=asmcode -debug aaa . c
        C source program (aaa . c)
```

```
int func()                                          /*1*/
 {                                                  /*2*/
    int i, j ;                                      /*3*/
    j = 0 ;                                         /*4*/
    for (i=1 ; i <=10 ; i++)  {                     /*5*/
         j += i ;                                   /*6*/
    return (j) ;                                    /*7*/
}                                                   /*8*/
```

Assembly source program

```
        .CPU      2600A
        .EXPORT   _func
        .SECTION  P, CODE, ALIGN=2
        .LINE     "/asm/code/aaa.c",1
_func:                ;function : func
        .LINE     2
        .LINE     4
        SUB.W     E0 , E0
        .LINE     5
        MOV.W     #1;16,R0
        .LINE     5
L5 :
        .LINE     6
        ADD:W     R0 , E0
        .LINE     5
        INC.W     #1,R0
        CMP.W     #10:16,R0
        BLE       L5:8
        .LINE     7
        MOV.W     E0,R0
        .LINE     8
        RTS
        .END
```

5.7.4 .DISPSIZE Displacement Size Specification

Format

Label	Operation	Operands
X	.DISPSIZE	<object item>=<bit count>[,<object item>=<bit count>...]

<object item>: {FBR | XBR | FRG | XRG | FWD | XTN | ALL}

Description

This directive sets the default size for branch instruction displacements, displacements for register indirect with displacement format addressing, and displacements for forward reference values and external reference values.

The objects of the settings specified by this directive are displacements which have no displacement size specification (:8,:16,:24 or:32).

The object items are interpreted as follows:

FBR .....Forward reference branch instructions

XBR .....External reference branch instructions

FRG .....Forward reference register indirect with displacement format addressing
XRG .....External reference register indirect with displacement format addressing

FWD .....Equivalent to simultaneous specification of FBR and FRG

XTN .....Equivalent to simultaneous specification of XBR and XRG

ALL .....Equivalent to simultaneous specification of FBR, FRG, XBR, and XRG

The following bit counts can be specified for the indicated object items.

H8S/2600 advanced mode .....FBR=8,16, XBR=8,16, FRG=16,32,
XRG=16,32, FWD=16, XTN=16, ALL=16

H8S/2600 normal mode .....FBR=8,16, XBR=8,16, FRG=16, XRG=16

H8S/2000 advanced mode .....FBR=8,16, XBR=8,16, FRG=16,32,
XRG=16,32, FWD=16, XTN=16, ALL=16

H8S/2000 normal mode .....FBR=8,16, XBR=8,16, FRG=16, XRG=16

H8/300H advanced mode .....FBR=8,16, XBR=8,16, FRG=16,24,

XRG=16,24, FWD=16, XTN=16, ALL=16

H8/300H normal mode .....FBR=8,16, XBR=8,16, FRG=16, XRG=16

H8/300 FBR=8, XBR=8, FRG=16, XRG=16

H8/300L .....FBR=8, XBR=8, FRG=16, XRG=16

The underlined is the default if nothing is specified.

The bit count must be a backward reference absolute value.

This directive can be used as many times as desired in a source program, and the specified bit count settings remain valid for all source statements until the next occurrence of the directive.

FBR is valid when neither the OPTIMIZE nor the BR RELATIVE command line option is specified.

This directive is only valid for the H8S/2600, H8S/2000, and H8/300H advanced mode and the H8S/2600, H8S/2000, and H8/300H normal mode. Since these settings are fixed at FBR=8, XBR=8, FRG=16, and XRG=16 for the H8/300 and H8/300L, this directive has no meaning for those processors.

Examples

```
    .CPU   2600A
    .SECTION A, CODE, ALIGN=2
    .DISPSIZE FBR=16 ........... [1]
    BRA    SYM ................. Equivalent to "BRA SYM:16."
    .DISPSIZE FBR=8 ............ [2]
    BRA    SYM ................. Equivalent to "BRA SYM:8."
SYM
    MOV.W  R0, R1
```

[1] Sets the displacement size for forward reference branch instructions to 16 bits.
[2] Sets the displacement size for forward reference branch instructions to 8 bits.

5.8 Assembly Listing Directives

5.8.1 .PRINT Assembly Listing Output Control

Format

Label	Operation	Operands
X	.PRINT	<output classifier>[,<output classifier>...]

<output classifier>: {LIST | NOLIST | SRC | NOSRC | CREF | NOCREF | SCT | NOSCT}

Description

This directive controls the output of an assembly listing, a source program listing, a cross-reference listing, and section information listing.

1. Assembly listing output control

   This directive can be used to specify the output of an assembly listing or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   LIST .....Output

   NOLIST .....Output suppressed

   NOLIST is used as the default when neither of the above is specified.

   This specification is only valid once in a given source program.

   This specification is valid when the [NO] LIST command line option has not been specified.

2. Source program listing output control

   This directive can be used to specify the output of a source program listing or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   SRC .....Output

   NOSRC .....Output suppressed

   SRC is used as the default when neither of the above is specified.

This specification is only valid once in a given source program.

This specification is valid when the [NO]SOURCE command line option has not been specified.

This specification is only valid when an assembly listing is output.

3. Cross-reference listing output control

   This directive can be used to control the output of a cross-reference listing or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   CREF.....Output

   NOCREF.....Output suppressed

   CREF is used as the default when neither of the above is specified.

   This specification is only valid once in a given source program.

   This specification is valid when the [NO]CROSS-REFERENCE command line option has not been specified.

   This specification is valid only when an assembly listing is output.

4. Section information listing output control

   This directive can be used to specify the output of section information listing or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   SCT .....Output

   NOSCT .....Output suppressed

   SCT is used as the default when neither of the above is specified.

This specification is only valid once in a given source program.

This specification is valid when the [NO]SECTION command line option has not been specified.

This specification is valid only when an assembly listing is output.

Examples

```
    .PRINT LIST, SRC, NOCREF, NOSCT
;
    .SECTION A, CODE, ALIGN=2
START
    MOV.W  R0,R1
    MOV.W  R0,R2
```

Only a source program listing is output in the assembly listing.

5.8.2 .LIST Source Program Partial Output Control

Format

Label	Operation	Operands
X	.LIST	<output classifier>[, <output classifier>...]

<output classifier>: {ON | OFF | COND | NOCOND | DEF | NODEF | CALL | NOCALL | EXP | NOEXP | STR | NOSTR | CODE | NOCODE}

Description

This directive controls partial output of source statements to the source program listing, partial output of preprocessor function source statements, and partial output of object code display lines.

1. Source statement partial output control

   This directive can be used to specify the partial output of program source statements or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   ON .....Output

   OFF .....Output suppressed

   ON is the initial setting of this output classifier.

2. Preprocessor function source statement partial output control

   This directive can be used to specify the partial output of preprocessor function source statements or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   COND .....Failed conditions are output

   NOCOND .....Failed conditions are not output

   DEF .....Definitions are output

   NODEF .....Definitions are not output

   CALL .....Calls are output

   NOCALL.....Calls are not output

   EXP.....Expansions are output

NOEXP.....Expansions are not output

STR .....Structured expansions are output

NOSTR .....Structured expansions are not output

This specification is only valid when source statement partial output is specified.

COND, DEF, CALL, EXP, and STR are the initial settings of these output classifiers.

These preprocessor functions have the following meanings.

Failed conditions .....The branches of .AIF statements that were not satisfied.

Definitions.....Macro definition bodies, .AREPEAT and .AWHILE definition bodies, .INCLUDE directives, .ASSIGNA and .ASSIGNC directives.

Calls .....Macro call statements, structured assembly directives, .AIF and .AENDI directives.

Expansions..... Macro expansion results, .AREPEAT and .AWHILE expansion results.

Structured expansions ..... Structured assembly expansion results.

This specification is valid when the [NO]SHOW command line option has not been specified.

3. Object code display line partial output control

   This directive can be used to specify the partial output of sections where the object code display for directives exceeds the number of source statement lines, or the suppression of that output.

   The output classifiers for this function are interpreted as follows:

   CODE .....Output

   NOCODE .....Output suppressed

   CODE is the initial setting of this output classifier.

This specification is valid when the [NO]SHOW command line option has not been specified.

This directive is not output to the assembly listing.

This directive can be used as many times as desired in a source program, and the specified output specified settings remain valid for all source statements until the next occurrence of the directive.

This specification is valid only when a source program listing is output.

Examples

```
    .PRINT LIST
;
    .LIST  OFF ........... [1]
    .INCLUDE "bbb.h"
    .LIST  ON ............ [2]
;
    .SECTION A, CODE, ALIGN=2
START
    MOV.W  R0, R1
`   MOV.W  R0, R2
```

Output of source statements is suppressed partially in this example. The source statements between [1] and [2] are not output to the source program listing.

```
    .PRINT LIST
    .LIST  NOSTR ......... [3]
;
    .SECTION A, CODE, ALIGN=2
START
    .IF.B (ROL<EQ>R1L) ... [4]
       MOV.W  R0, R2
    .ELSE ................ [4]
       MOV.W  R0, R3
    .ENDI ................ [4]
```

Suppression of structured expansions is specified at line [3]. As a result, the source statements expanded at the lines marked [4] will not be output to the source program listing.

5.8.3 .FORM Listing Size Control

Format

Label	Operation	Operand
X	.FORM	<listing size>[,<listing size>]

<listing size>: {LIN=<number of lines>1 COL=<number of columns>}

Description

This directive specifies the size of the assembly listing.

The listing size specifications used with this function are interpreted as follows:

LIN=<number of lines> .....The number of lines on a page.

COL=<number of columns>.....The number of columns in a line.

The initial listing size settings are LIN=60 and COL=132.

1. Number of lines on a page
   This setting specifies the number of lines on a page.
   The number of lines is specified with a backward reference absolute value.
   The range for the number of lines setting is from 20 to 255. If a value less than 20 is specified, the value 20 is used, and if a value greater than 255 is specified, 255 is used. No error is issued in these cases.
   This specification is valid when the LINES command line option has not been specified.
2. Number of columns per line
   This setting specifies the number of columns per line.
   The number of columns is specified with a backward reference absolute value.
   The range for the number of columns setting is from 79 to 255. If a value less than 79 is specified, the value 79 is used, and if a value greater than 255 is specified, 255 is used. No error is issued in these cases.

This specification is valid when the COLUMNS command line option has not been specified. The specified listing size is valid for pages following the occurrence of this directive.

Examples

```
.FORM  LIN=60, COL=80
```

This directive sets the assembly listing size to 60 lines per page and 80 columns per line.

5.8.4 .HEADING Listing Heading Specification

Format

Label	Operation	Operands
X	.HEADING	"<character string>"

Description

This directive specifies the title to be displayed in the source program listing page header.

Character strings are specified by enclosing the desired characters in double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

Up to 60 characters may be specified for the character string. However, no error is displayed if more than 60 characters are specified.

The page header will be displayed from the page where this directive appears if this directive appears on the first line, otherwise it will be displayed starting with the following page.

Examples

```
     .HEADING "MODULE NAME = ""MAIN"""
     .PRINT LIST
 ;
     .SECTION A, CODE, ALIGN=2
 MAIN
    MOV.W  R0, R1
    MOV.W  R0, R2
```

The string MODULE NAME="MAIN" will be displayed as the page header.

5.8.5 .PAGE New Page

Format

Label	Operation	Operands
X	.PAGE	X

Description

This directive starts a new page in the source program listing.

A new page is not started if this directive appears on the first line of a page.

This directive is not output to the listing.

This specification is valid only when a source program listing is output.

Examples

```
    .PRINT LIST
;
.INCLUDE "bbb.h"
;
.PAGE
.SECTION A, CODE, ALIGN=2 ..... [1]
START
MOV.W  R0, R1
MOV.W  R0, R2
;
.PAGE
.SECTION B, DATA, ALIGN=2 ..... [2]
DAT
.DATA.W H'0001
.DATA.W H'0002
```

New pages will be started so that both lines [1] and [2] appear on the first line of the page.

5.8.6 .SPACE Blank Line Output

Format

Label	Operation	Operands
X	.SPACE	[<number of lines>]

Description

This directive outputs the specified number of blank lines to the source program listing.

The number of lines must be a backward reference absolute value.

The range for the number of lines setting is from 1 to 50. If a value less than 1 is specified, 1 is used, and if a value greater than 50 is specified, 50 is used. No error is issued in these cases.

A single blank line is output if the line count specification is omitted.

The section of the specified blank lines that exceeds the current page is not output.

No markings whatsoever, including line numbers, are displayed on the blank lines specified by this directive.

This directive is not output to the listing.

This specification is valid only when a source program listing is output.

Examples

```
     .PRINT LIST
  ;
     .SECTION A, CODE, ALIGN=2
  START
     MOV.W  R0, R1
     MOV.W  R0, R2 ............ [1]
     .SPACE 5
     MOV.W  R0, R3 ............ [2]
```

Five blank lines will be inserted in the source program listing between the lines marked [1] and [2] in the above example.

5.9 Other Directives

5.9.1 .PROGRAM Object Module Name Specification

Format

Label	Operation	Operands
X	.PROGRAM	<object module name>

Description

This directive assigns a name to the object module.

The object module name is used by the H series linkage editor, the H series librarian, and other support software.

The syntax used for the object module name is the same as that for symbols.

If this directive is omitted, the main file name of the object file will be used as the object module name.

This directive is valid only once in a given source program.

Examples

```
    .PROGRAM main
;
.SECTION A, CODE, ALIGN=2
START
MOV.W  R0, R1
MOV.W  R0, R2
```

In this example, "main" is specified as the object module name.

5.9.2 .RADIX Radix Specification

Format

Label	Operation	Operands
X	.RADIX	<radix classifier>

<radix classifier>: {B | Q | D | H}

Description

This directive sets the default radix for integer constants.

The radix classifiers are interpreted as follows:

B	Binary
Q	Octal
D	Decimal
H	Hexadecimal

The initial value of the default radix is decimal.

Integer constant expressions that do not include a radix specification are the object of this directive's specification.

However, this specification does not apply to operands of conditional assembly function directives. The radix of constants in conditional assembly function directives is taken to be decimal if the specification is omitted.

When specifying hexadecimal constants without a radix specification, the first digit must be between 0 and 9. Notations whose first character is A to F, are always interpreted to be symbols, and are not affected by this directive. Attach a leading zero to use such constants.

This directive can be used as many times as desired in a source program, and the specified radix remains valid for all source statements until the next occurrence of the directive.

Examples

```
.SECTION A, DATA, ALIGN=2
.DATA.W -32768 ........This is the same as -D'32768.
.DATA.W 65535 ..........This is the same as D'65535.
.RADIX H
.DATA.W 8000 ...........This is the same as H'8000.
.DATA.W 0FFFF ..........This is the same as H'FFFF.
```

5.9.3 .END Source Program Termination

Format

Label	Operation	Operands
X	.END	[<execution start address>]

Description

This directive terminates the source program.

The assembler terminates assembly at the point this directive appears.

The <execution start address> specification is used to specify the simulation execution start address for the simulator/debugged.

The <execution start address> is set to a code section address.

The <execution start address> must be either an absolute value or an address value.

The <execution start address> depends on the selected address space.

Examples

```
    .CPU   2600A
    .OUTPUT DBG
;
    .SECTION A, CODE, ALIGN=2
START
    MOV.L  #0:32, ER0
    MOV.L  #1:32, ER1
    MOV.L  #2:32, ER2
    BRA START
;
    .END   START
```

The simulator/debugged will start simulation from the address of the START symbol.

Section 6 File Inclusion Function

6.1 Overview of the File Inclusion Function

The file inclusion function allows other files to be inserted into a file at assembly time.

6.1.1 File Inclusion

In file inclusion, the include file, which is specified by an .INCLUDE directive, is inserted directly following that .INCLUDE directive by the assembler during the assembly process.

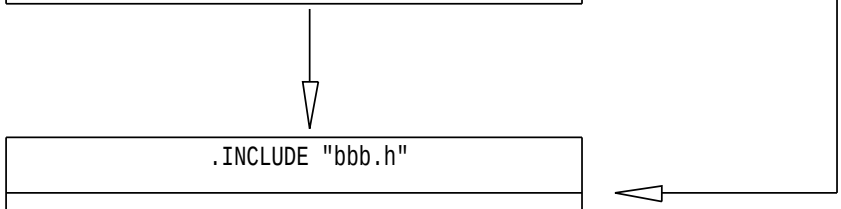
Examples

aaa.mar

```
                .INCLUDE "bbb.h"
;
                .SECTION A,CODE,ALIGN=2
                MOV.W   #ON:16,R0
```

bbb.h

```
;  INCLUDE FILE
ON      .EQU            1
OFF     .EQU            0
```



```
                .INCLUDE "bbb.h"
;  INCLUDE  FILE
ON              .EQU        1
OFF             .EQU        0
;
                .SECTION A,CODE,ALIGN=2
                MOV.W  #ON:16,R0
```

When the file aaa.mar is assembled, it is assembled with the file bbb.h inserted directly following the .INCLUDE directive

6.2 File Inclusion Preprocessor Directives

6.2.1 .INCLUDE File Inclusion

Format

Label	Operation	Operands
X	.INCLUDE	"<file name>"

Description

This directive includes the specified file.

The file name is specified surrounded by double quotation marks (").

The syntax for the file name conforms to the naming rule of the host machine.

If no extension is included in the file name, a file with no extension is used.

It is possible to include other files within an include file. However the nesting depth for file inclusion is limited to 30 levels.

The directory name specified with this directive can be changed using the INCLUDE command line option.

Examples

```
        .CPU   2600A
    ;
        .INCLUDE "import.h"
    ;
        .SECTION A, CODE, ALIGN=2
    START
        MOV.L  #STACK:32,SP
        JSR@PROG1:24
        JSR@PROG2:24
    ;
        .END
```

The file import.h is included.

Section 7 Conditional Assembly Function

7.1 Overview of the Conditional Assembly Function

The conditional assembly function allows the specification of whether a certain part of a source program should be assembled.

7.1.1 Preprocessor Variables

Preprocessor variables are variables that are mainly used by the conditional assembly function. Preprocessor variables are either integer or character type variables.

1. Integer preprocessor variables

   Integer preprocessor variables are assigned integer values by the .ASSIGNA directive or the ASSIGNA command line option. Also, they can be reassigned new integer values by the .ASSIGNA directive.

   A backlash and ampersand (\&) are inserted in front of integer preprocessor variables when they are referenced.

```
Example
PRE    ASSIGNA 1 This integer value 1 is assigned to the variable PRE.
          .AIF\&PRE EQ 1 This is the same as: .AIF 1 EQ 1.
       MOV.W  R0, R1
       .AENDI
```

2. Character preprocessor variables

   Character preprocessor variables are assigned character string values by the .ASSIGNC directive or the ASSIGNC command line option. Also, they can be reassigned new character values by the.ASSIGNC directive.

   A backlash and an ampersand (\&) are inserted at the front of integer preprocessor variables when they are referenced.

```
Example
PRE .ASSIGNC "ON".          The character string "ON" is assigned to the variable PRE.
.AIF "\&PRE" EQ "ON"        This is the same as: .AIF "ON" EQ "ON"
 MOV.W          R0, R1
.AENDI
```

7.1.2 Conditional Assembly

The conditional assembly function assembles the statements between .AIF, .AELSE, and .AENDI directive statements (note that the .AELSE statement may be omitted) according to the condition specified by the .AIF statement.

1. When the .AIF condition holds:

   If the .AIF condition is true, the statements between the .AIF and .AELSE statements are assembled, and the statements between the .AELSE and .AENDI statements are not assembled.

   When the .AELSE statement is omitted, the statements between the .AIF and .AENDI statements are assembled.
2. When the .AIF condition is false:

   If the .AIF condition is false, the statements between the .AELSE and .AENDI statements are assembled, and the statements between the .AIF and .AELSE statements are not assembled.

   When the .AELSE statement is omitted, the statements between the .AIF and .AENDI statements are not assembled.

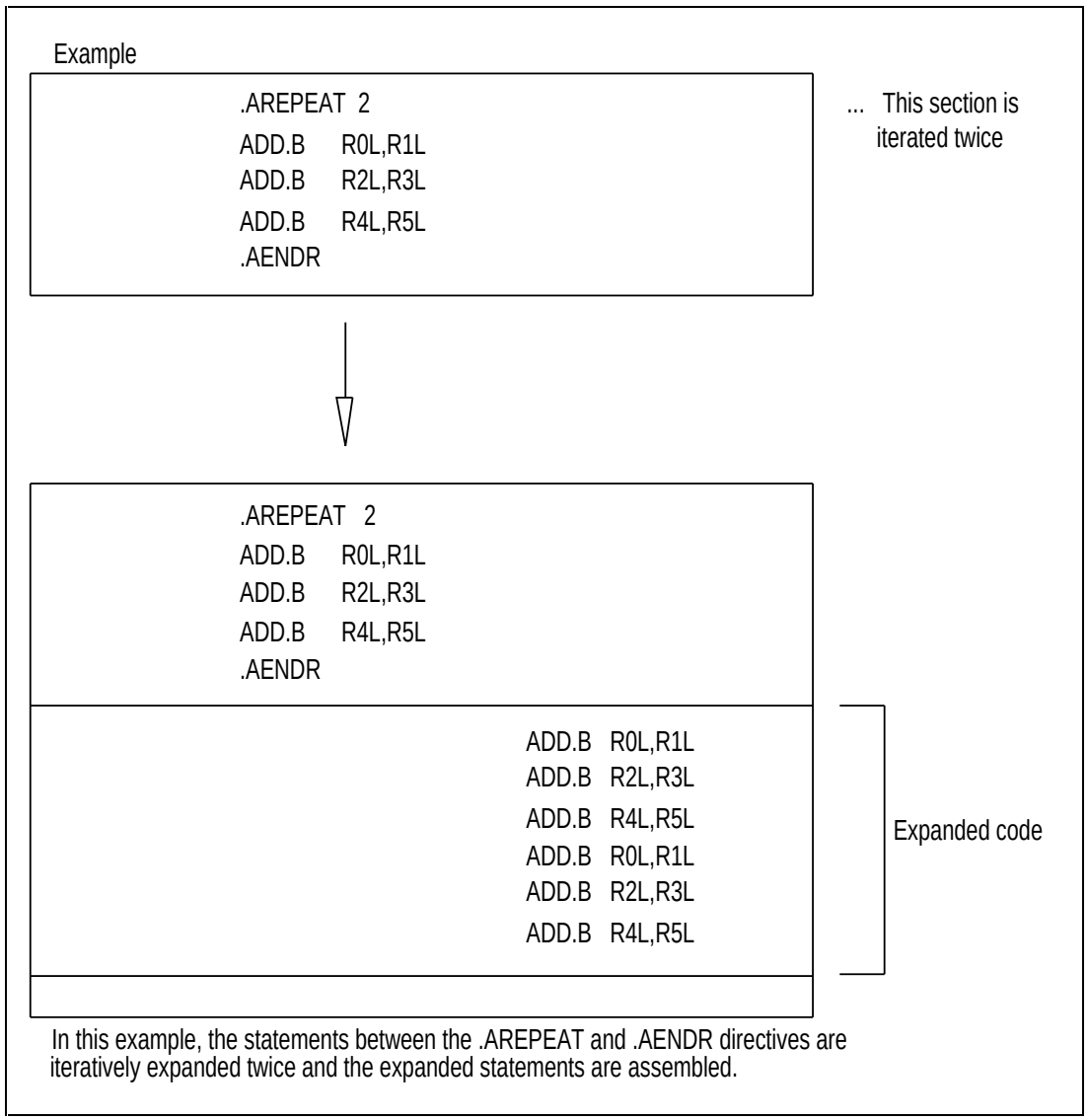
Example

```
FLAG    .ASSIGNA 1.....FLAG is set to 1.
                .AIF \&FLAG EQ 1.....The condition is FLAG=1.
                MOV.WR0, R1                 ]
                MOV.WR2, R3                 ] This section will be assembled.
                MOV.WR4, R5                 ]
                .AELSE
                MOV.WR1, R0                 ]
                MOV.WR3, R2                 ] This section will not be assembled.
                MOV.WR5, R4                 ]
        .AENDI
```

Since FLAG is 1, the source statements between the .AIF and .AELSE statements are assembled.

7.1.3 Iterated Expansion

In iterated expansion, source statements between an .AREPEAT directive and its matching .AENDR directive are iteratively expanded the number of times specified in the .AREPEAT directive. The expanded statements are then assembled.



7.1.4 Conditional Iterated Expansion

In conditional iterated expansion, source statements between an .AWHILE directive and its matching .AENDW directive are iteratively expanded while the condition specified in the .AWHILE directive is true. The expanded statements are then assembled.

Example

```
COUNT      .ASSIGNA  2

           .AWHILE \&COUNT NE 0
           ADD.B   R0L,R1L
           ADD.B   R0L,R2L
           INC.B   R0L
COUNT      .ASSIGNA \&COUNT-1
           .AENDW
```

.... COUNT is set to 2.

... The condition is COUNT ≠ 0

.... One is subtracted from the value of COUNT.

```
COUNT      .ASSIGNA  2

           .AWHILE  \&COUNT NE 0
           ADD.B     R0L,R1L
           ADD.B     R0L,R2L
           INC.B     R0L
COUNT      .ASSIGNA   \&COUNT-1
           .AENDW
```

```
                      ADD.B   R0L,R1L
                      ADD.B   R0L,R2L
                      INC.B   R0L
                COUNT .ASSIGNA \&COUNT-1
                      ADD.B   R0L,R1L
                      ADD.B   R0L,R2L
                      INC.B   R0L
                COUNT .ASSIGNA \&COUNT-1
```

Expanded statements

In this example, the statements between the .AWHILE and .AENDW directives are iteratively expanded while COUNT ≠ 0, and the expanded statements are assembled.

7.2 Conditional Assembly Preprocessor Directives

7.2.1 .ASSIGNA Integer Preprocessor Variable Assignment

Format

Label	Operation	Operands
<preprocessor variable>	.ASSIGNA	<value>

Description

This directive assigns a value to an integer preprocessor variable.

1. Integer preprocessor variable definition
   The syntax of integer preprocessor variables is the same as that for symbols.
   The value must be one of the following:
   — Constant
   — Preprocessor variable defined before
   — Expression consisting of the above values
   The value of preprocessor variables defined with this directive can be changed by another instance of this directive.
   Defined preprocessor variables are valid in all source statements following the defining directive.
   This specification is valid when no ASSIGNA command line option was specified.
2. Preprocessor variable reference
   Preprocessor variables are referenced as follows:
   \&<preprocessor variable>[']
   The apostrophe (') is included when it is desirable to clearly distinguish the preprocessor variable from the rest of the source statement.
   Referenced integer preprocessor variables are replaced with their assigned values as decimal integers with the radix specification omitted.

Preprocessor variables can be referenced in the following locations.

- .ASSIGNA and .ASSIGNC directives
- .AIF, .AREPEAT, .AWHILE, and .ALIMIT directives
- Macro bodies (source statements between a .MACRO directive and its matching .ENDM directive.)

Examples

```
FLAG   .ASSIGNA 1..............FLAG is set to 1.
;
    .SECTION A, CODE, ALIGN=2
START
    .AIF \&FLAG EQ 1............This is the same as ".AIF 1 EQ 1."
    MOV .W RO, R2
    .AENDI
    .AIF \&FLAG EQ 2............This is the same as ".AIF 1 EQ 2."
    MOV .W R1, R2
    .AENDI
;
FLAG   .ASSIGNA 2..............The value of FLAG is changed to 2.
;
    .AIF \&FLAG EQ 1............This is the same as ".AIF 2 EQ 1."
    MOV .W R3,R5
    .AENDI
    .AIF \&FLAG EQ 2............This is the same as ".AIF 2 EQ 2."
    MOV .W R4,R5
    .AENDI
```

In this example, the integer preprocessor variable FLAG is referenced in .AIF directives.

7.2.2 .ASSIGNC Character Preprocessor Variable Assignment

Format

Label	Operation	Operands
<preprocessor variable>	.ASSIGNC	"<character string>"

Description

This directive assigns a value to a character preprocessor variable.

1. Character preprocessor variable definition

   The syntax of character preprocessor variables is the same as that for symbols.

   Character strings are specified by enclosing the desired characters in double quotation marks (").

   To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

   The value of preprocessor variables defined with this directive can be changed by another instance of this directive.

   Defined preprocessor variables are valid in all source statements following the defining directive.

   This specification is only valid when no ASSIGNC command line option is specified.

2. Preprocessor variable reference

   Preprocessor variables are referenced as follows:

   \&<preprocessor variable>[']

   The apostrophe (') is included when it is desirable to clearly distinguish the preprocessor variable from the rest of the source statement.

   Referenced character preprocessor variables are replaced with their assigned character string excluding the surrounding double quotation marks (").

Preprocessor variables can be referenced in the following locations.

- .ASSIGNA and .ASSIGNC directives
- .AIF, .AREPEAT, and .AWHILE directives
- Macro bodies (source statements between a .MACRO directive and its matching .ENDM directive.)

Examples

```
FLAG   .ASSIGNC "ON".....................[1]
;
       .SECTION A, CODE, ALIGN=2
START
       .AIF "\&FLAG" EQ "ON"..............[2]
       MOV .W RO, R2
       .AENDI
       .AIF "\&FLAG" EQ "OFF".............[3]
       MOV .W R1, R2
       .AENDI
;
FLAG   .ASSIGNC "OFF"....................[4]
;
       .AIF "\&FLAG" EQ "ON"..............[5]
       MOV .W R3, R5
       .AENDI
       .AIF "\&FLAG" EQ "OFF".............[6]
       MOV .W R4, R5
       .AENDI
```

[1] FLAG is set to the character string ON.
[2] This is the same as .AIF "ON" EQ "ON".
[3] This is the same as .AIF "ON" EQ "OFF".
[4] The value of FLAG is changed to OFF.
[5] This is the same as .AIF "OFF" EQ "ON".
[6] This is the same as .AIF "OFF" EQ "OFF".

In this example, the character preprocessor variable FLAG is referenced in .AIF directives.

7.2.3 .AIF Conditional Assembly

Format

Label	Operation	Operands
X	.AIF	<term 1><relational perator><term 2>
[X	.AELSE	X
X	.AENDI	X
<relational operator>:	{EQ | NE | GT | LT | GE | LE}	

Description

This directive specifies conditional assembly.

The section of the statements between .AIF, .AELSE, and .AENDI directives for which the condition is satisfied are assembled.

The .AELSE directive may be omitted.

The terms are specified with numeric values or character strings. However, numeric values and character strings cannot be compared. When a numeric value and a character string are compared, the condition always fails.

Numeric values are specified with backward reference absolute values or preprocessor variables.

Character strings are specified by preprocessor variables or characters surrounded by double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

The relational operators are interpreted as follows.

EQ ..... <term 1>=<term 2>

NE ..... <term 1>≠<term 2>

GT ..... <term 1>><term 2>

LT ..... <term 1><<term 2>

GE ..... <term 1>≥<term 2>

LE ..... <term 1>≤<term 2>

Examples

```
FLAG1    .ASSIGNA 1
      .AIF \&FLAG1 EQ 1.............The condition here is FLAG1 = 1.
      MOV .W R0, R1         ]
      MOV .W R2, R3         ]   [1]
      MOV .W R4, R5         ]
      .AELSE                 ]
      MOV .W R1, R0         ]
      MOV .W R3, R2         ]   [2]
      MOV .W R5, R4         ]
      .AENDI
```

Since FLAG1 is 1, the statements [1] are assembled. The statements [2] will not be assembled.

```
FLAG2     .ASSIGNA 0
      .AIF \&FLAG2 EQ 1.............The condition here is FLAG2=1.
      .DATA .W H'0001        ]
      .DATA .W H'0002        ]
      .DATA .W H'0003        ]   [3]
      .DATA .W H'0004        ]
      .DATA .W H'0005        ]
      .AENDI
```

Since FLAG2 is not 1, the statements [3] will not be assembled.

7.2.4 .AREPEAT Iterated Expansion

Format

Label	Operation	Operands
X	.AREPEAT	<count>
X	.AENDR	X

Description

This directive iterates a part of source program statements.

The statements between the .AREPEAT directive and its matching. AENDR directive are iterated the specified number of times.

The count must be a backward reference absolute value or a preprocessor variable.

Values of 1 or larger can be specified as the count. No expansion occurs if a value of 0 or smaller is specified.

Examples

```
.AREPEAT 2
SHAL .B R0L
SHAL .B R1L
SHAL .B R2L
.AENDR
```

These source statements will be repeated twice.

```
.AREPEAT 3
.DATA.W H'0001, H'0002
.DATA.W H'0003, H'0004
.AENDR
```

These source statements will be repeated three times.

7.2.5 .AWHILE Conditional Iterated Expansion

Format

Label	Operation	Operands
X	.AWHILE	<term 1><relational operator><term 2>
[X	.AENDW	X
<relational operator>:	{EQ | NE | GT | LT | GE | LE}	

Description

This directive specifies conditional iterated assembly.

The statements between an .AWHILE directive and its matching .AENDW directive are expanded iteratively while the condition is satisfied.

The terms are specified with numeric values or character strings. However, numeric values and character strings cannot be compared. When a numeric value and a character string are compared, the condition always fails.

Numeric values are specified with backward reference absolute values or preprocessor variables.

Character strings are specified by preprocessor variables or characters surrounded by double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

The relational operators are interpreted as follows.

EQ ..... <term 1>=<term 2>

NE ..... <term 1>≠<term 2>

GT ..... <term 1>><term 2>

LT ..... <term 1><<term 2>

GE ..... <term 1>≥<term 2>

LE ..... <term 1>≤<term 2>

Conditional iterated expansion terminates when the condition finally fails.

When the condition does not fail, expansion is iterated a maximum of 65,535 times. The maximum iteration count can be specified using the .ALIMIT directive.

Examples

```
 COUNT .ASSIGNA 2 ................COUNT is set to 2.
       .AWHILE \&COUNT NE 0.........The condition is COUNT≠0.
       ADD .B R0L, R1L
       ADD .B R0L, R2L
       INC .B R0L
COUNT  .ASSIGNA \&COUNT-1       ..........Subtracts 1 from COUNT.
       .AENDW
```

The source statements will be iteratively expanded while COUNT is not 0.

```
STOP   .ASSIGNA 0 ................STOP is set to 0.
       .AWHILE \&STOP LE 10.........The condition is STOP≤10.
       ADD.B  R0L, R1L
       ADD.B  R0L, R2L
       INC.B  R0L
STOP   .ASSIGNA \&STOP+3 ...........Adds 3 to STOP.
.AENDW
```

The source statements will be iteratively expanded while STOP is 10 or less.

7.2.6 .ALIMIT Conditional Iterated Expansion Limit

Format

Label	Operation	Operands
X	.ALIMIT	<maximum number of times>

Description

This directive sets the maximum number of times that conditional iterated expansion is to be performed.

The limit value must be one of the following:

- Constant
- Preprocessor variable defined previously
- Expression consisting of the above values

The conditional iterated expansion limit specified by this directive can be changed by respecifying this directive. It is valid in all source statements following the defining directive statement.

When this directive is not specified, the default value is 65,535.

Example

```
COUNT   .ASSIGNA 3
        .ALIMIT 10 ..............The conditional iterated
        .AWHILE \&COUNT NE 4    expansion limit is set to 10.
        ADD.B R0L, R1L       ]
        ADD.B R0L, R1L       ]
        INC.B R0L         ]   [1]
COUNT   .ASSIGNA \&COUNT-1     ]
        .AENDW           ]
```

[1] is iteratively expanded while COUNT is not 4.
After 10 iterations, warning message 854 is output and iterated expansion is terminated.

7.2.7 EXITM Expansion Exit

Format

Label	Operation	Operands
X	EXITM	X

Description

This directive exits from (i.e., terminates) an iterative expansion specified by an .AREPEAT or .AWHILE directive.

This directive can be specified between an .AREPEAT directive and its matching AENDR directive or between an .AWHILE directive and its matching .AENDW directive. The expansion will terminate at the point that this directive appears.

This directive is also used to exit from macro expansions. The positioning of this directive requires care when iteration and macro expansion are combined.

Examples

```
COUNT  .ASSIGNA 0
       WHILE 1 EQ 1            [1]
       D.B R0L,R1L
       D.B R2L,R3L
COUNT  .ASSIGNA \&COUNT+1
       IF \&COUNT EQ 2         [2]
       XITM
       ENDI
       ENDW
```

[1] This specifies an infinite loop expansion.
[2] The condition is COUNT = 2.

When COUNT is updated and the .AIF condition is satisfied, the .EXITM directive will be assembled. At the point that the .EXITM directive is assembled, the .AWHILE expansion will be terminated.

Section 8 Macros

8.1 Overview of the Macro Function

The macro function allows commonly used sequences of instructions to be named and then recalled simply by specifying that name.

8.1.1 Macro Definitions and Macro Calls

A macro definition defines the statements between a. MACRO directive and its matching. ENDM statement (the macro body) as a macro instruction, and associates that code with a macro name.

In a macro call, the macro name is specified in an instruction operation field, and the macro body is inserted in the program at that location.

Example

```
.MACRO   MC0
MOV.W    R0,R1
MOV.W    R2,R3
MOV.W    R4,R5
. ENDM

MC0
```

Macro definition

... Macro call

```
.MACRO   MC0
MOV.W    R0,R1
MOV.W    R2,R3
MOV.W    R4,R5
.ENDM

MC0
```

```
                         MOV.W    R0,R1
                         MOV.W    R2,R3
                         MOV.W    R4,R5
```

Macro expansion

The macro body is expanded and assembled immediately after the macro call in the source program.

An important aspect of the macro function is the ability to specify parameters to a macro call. This allows part of the macro body to be replaced by arbitrary text that differs between macro calls.

The procedure for using macro parameters is as follows:

1. Declare the formal parameters in the operands of the. MACRO directive.
2. Use the formal parameters in the macro body. Formal parameters must be identified in the macro body by attaching a back-slash (\) at the front.
3. Specify the macro parameters corresponding to the formal parameters in the macro calls.

Example

```
.MACRO   MC0 P1,P2                    ... [1]
MOV.W     \P1,R0                      [2]
MOV.W     R1,\P2
.ENDM
MC0       R2,R3                       ... [3]
MC0       #H'1234,R4                  ... [4]
```

```
.MACRO   MC0 P1,P2
MOV.W     \P1,R0
MOV.W     R1,\P2
.ENDM
MC0       R2,R3

                    MOV.W     R2,R0     [5]
                    MOV.W     R1,R3

MC0         #H'1234,R4

                    MOV.W     #H'1234,R0  [6]
                    MOV.W     R1,R4
```

[1] Declares the formal parameters P1 and P2.

[2] The formal parameters are referenced in the macro body statements.

[3] The macro parameters R2 and R3 are specified in this macro call.

[4] The macro parameters #H'1234 and R4 are specified in this macro call.

[5] In this macro expansion \P1 is replaced by R2 and \P2 is replaced by R3.

[6] In this macro expansion \P1 is replaced by #H'1234 and \P2 is replaced by R4.

8.2 Macro Function Related Preprocessor Directives

8.2.1 .MACRO Macro Definition

Format Label Operation Operands

X .MACRO <macro name> [Δ <formal parameter>
[,<formal parameter>...]]

X .ENDM X

<formal parameter>: <formal parameter name>[=<default value>]

Description This directive defines a macro instruction.

The source statements (the macro body) between a .MACRO directive and its matching .ENDM statement are defined as a macro instruction corresponding to the macro name.

The defined macro instruction is recalled by a macro call.

1. Macro name

   The macro name is the name associated with the macro instruction, and is used in calls to the macro.

   The syntax for macro names is identical to that for symbols. However, the cross assembler pre-defined mnemonics for machine instructions, directives (excluding the period), and preprocessor directives (excluding the period) may not be used. Upper and lower case letters are not distinguished in macro names.
2. Formal parameters

   Formal parameters are specified when it is desired to replace parts of the macro body text at expansion time. The replacement text is specified by the macro parameters in the macro call.

a. Formal parameter syntax

   The syntax for formal parameter names is identical to that for symbols. Note that upper and lower case letters are distinguished in formal parameter names.

b. Formal parameter reference

   Formal parameters are used (referenced) in the body of the macro to specify the location of the replacement text.

   The syntax of formal parameter reference in macro bodies is as follows:

   \<formal parameter name>[']

   The apostrophe (') is included when it is desirable to clearly distinguish the formal parameter from the rest of the source statement.

3. Formal parameter defaults

   Default values for formal parameters can be specified in macro definitions. The default value specifies the replacement character string for the formal parameter when the corresponding parameter is omitted in a macro call.

   The default value must be surrounded by double quotation marks (") or angle (<>) brackets if any of the following characters are included in the default value.

   — Space
   — Tab
   — Comma(,)
   — Semicolon (;)
   — Double quotation marks(")
   — Angle brackets (<>)

   Default values are inserted with the surrounding double quotation marks or angle brackets removed in macro expansion.

The following limitations apply to macro definitions.

1. Macros cannot be defined in the following locations.Macro bodies (between
   — MACRO and .ENDM directives)
   — Between .AREPEAT and .AENDR directives
   — Between A.WHILE and .AENDW directives
2. The .END directive cannot appear in a macro.
3. No symbol may be inserted in the label field of the .ENDM directive. The .ENDM directive is ignored if its label field is not blank. No error is generated in this case.

Examples

```
.MACRO MC0 P1, P2
MOV.W  \P1, P2
MOV.W  \P2, R3   [1]
MOV.W  R4, R5
.ENDM
MC0 R0, R2       [2]
```

[1] This defines the macro instruction MCO, and declares the formal parameters P1 and P2.

[2] This is a macro call. R0 is specified for the P1 parameter, and R2 for the P2 parameter.

In the macro expansion, \P1 will be replaced by R0, and \P2 by R2.

```
.MACRO MCO P1 = R0, P2 = R2     [3]
MOV.W  \P1, R1
MOV.W  \P2, R3
MOV.W  R4, R5
.ENDM
MC0,R6                          [4]
```

[3] In this macro definition R0 is specified as the default for P1, and R2 as the default for P2.

[4] This is a macro call. In this call the macro parameter for the parameter P1 is omitted. R6 is specified as the parameter corresponding to the P2 parameter.

In the macro expansion, \P1 will be replaced by R0, and \P2 by R6.

8.2.2 Macro Body

Description

The following describes the functions which can be used in a macro body, i.e. in the source statements between the .MACRO and .ENDM directives.

1. Formal parameter reference

   Formal parameters names are used to specify the parts to be replaced by macro parameters in macro expansion.

   The syntax of formal parameter reference in macro bodies is as follows:

   \<formal parameter name>[']

   The apostrophe (') is included when it is desirable to clearly distinguish the formal parameter from the rest of the source statement.

   Example

```
        .MACRO      MC0 P1
        MOV.W       \P1, R1
        .ENDM
        MCO         R0
```

2. Preprocessor variable (.ASSIGNA and .ASSIGNC) reference

   Preprocessor variables can be referenced inside macro bodies.

   The syntax for preprocessor variable reference is as follows:

   \&<preprocessor variable name>[']

   The apostrophe (') is included when it is desirable to clearly distinguish the preprocessor variable from the rest of the source statement.

   Example

```
        .MACRO      MC0
        MOV.W       \&PRE, R1
        .ENDM
PRE     .ASSIGNC "R0"
        MC0
```

3. Macro generation number

   The macro generation number marker is expanded as a 5-digit decimal number (between 00000 and 99999) unique to the macro expansion.

   The syntax for specifying the macro generation number marker is as follows:

   \@

   The macro generation number facility is used to avoid the problem that symbols used within a macro body will be multiply defined if the macro is expanded multiple times. To avoid this problem, specify the macro generation number marker as part of any symbol used locally to a macro. This will result in symbols that are unique to each macro call.

Example

```
        .MACRO MC0
        BEQ     SYM\@:8
        SYM\@
        .ENDM
        MC0
        MC0
```

4. Macro replacement processing exclusion

   When the backlash character (\) appears in a macro body, it specifies macro replacement processing. Therefore a means for excluding this macro processing is required for cases where it is necessary to use the backlash character as an ordinary character.

   The syntax for macro replacement processing exclusion is as follows:

   \(<macro replacement processing excluded character string>)

   The backlash and the parentheses will be removed in macro processing.

   Note that the character string manipulation functions are also excluded from replacement processing in macro processing exclusion.

```
Example
.MACRO MC0
\(MOV.B        #"\", ROL)
.ENDM
MCO
```

5. Comments in macros

   Comments in macro bodies can be coded as normal comments or as macro internal comments. When the comments in the macro body are not required in the macro expansion code, those comments can be coded as macro internal comments to suppress their expansion.

   Macro internal comments are coded as follows:

   \;<comment>

```
Example
            .MACRO          MCO     P1
            MOV.W\P1, R1    \;      \P1:Rn
            .ENDM
            MCO     RO
```

6. Character string manipulation functions

   Character string manipulation functions can be used in the body of a macro.

   The following character string manipulation functions are provided.

.LEN	Character string length (See section 8.2.5, .LEN.)
.INSTR	Character string search (See section 8.2.6,. INSTR.)
.SUBSTR	Character string substring (See section 8.2.7,. SUBSTR.)

8.2.3 Macro Call

Format Label Operation Operand

[<symbol>] <macro name> [<macro parameter>][,[<macro parameter>]...]
<macro parameter>: [<formal parameter>=]<character string>

Description This form invokes a macro expansion.

Macros must be defined with the .MACRO directive prior to a macro call.

The macro parameters specify strings to be substituted into the macro body during macro expansion. It is necessary to declare these parameters in advance in the macro definition's. MACRO statement.

1. Macro parameter specification
   Macro parameters can be specified by either positional specification or keyword specification.
   a. Positional specification
      In positional specification, the macro parameters are specified in the same order as that of the formal parameters declared in the macro definition.
   b. Keyword specification
      In keyword specification, each macro parameter is specified following its corresponding formal parameter, separated by an equal sign (=).

2. Macro parameter syntax

   If a macro parameter includes any of the following characters, enclose the parameter in double quotation marks (") or angle brackets (<>).

   — Space
   — Tab
   — Comma (,)
   — Semicolon (;)
   — Double quotation mark (")
   — Angle brackets (<>)

   In macro expansion, the parameters will be substituted with the surrounding double quotation marks or angle brackets removed.

Examples .MACRO MCO P1, P2

```
        MOV.W       \P1, R1
        MOV.W       \P2, R3  [1]
        MOV.W       R4, R5
        .ENDM
        MCO RO, R2   [2]
        MCO P1=RO, P2=R2     [3]
```

[1] Defines the macro "MCO". The formal parameters P1 and P2 are declared.

[2] The macro parameters are specified using positional specification.

[3] The macro parameters are specified using keyword specification.

In both macro expansions (lines [2] and [3] the formal parameter \P1 is replaced by RO, and \P2 by R2.

8.2.4 .EXITM Expansion Termination

Format Label Operation Operand

X EXITM X

Description This directive terminates macro expansion.

This directive is specified in the body of a macro, i.e., between a .MACRO directive and its matching .ENDM directive. When this directive appears during macro expansion, this macro expansion is terminated.

This directive is also used to terminate. AREPEAT and .AWHILE iterative expansions. The positioning of this directive requires care when iteration and macro expansion are combined.

Examples

```
.MACRO MCO P1
MOV.W  RO, R1
MOV.W  R2, R3    [1]
MOV.W  R4, R5
\P1
MOV.W  R6, R7
.ENDM
MCO .EXITM
```

The .EXITM will be expanded, and the macro expansion will terminate at that point. Only the statements labelled [1] will be expanded.

8.2.5 .LEN Character String Manipulation

Format .LEN [Δ] ("<character string>")

Description

This function counts the length of the character string given as its argument, and is replaced by that value as a decimal number with no radix specification.

Character string are specified by enclosing the desired characters in double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

Macro formal parameters and preprocessor variables can be specified in the character string as shown below.

.LEN ("\<formal parameter>")
.LEN("\&<preprocessor variable>")

This function can only be used inside a macro body, i.e., between a .MACRO directive and its matching .ENDM directive.

This directive is not replaced when it contains a syntax error. No error is displayed in this case.

Examples

```
.MACRO      MCO   P1
    .SRES .LEN ("\P1")
    .ENDM
    MC0   ABCDEF      [1]
    MC0   GHI     [2]
```

[1] This expands to ".SRES 6".
[2] This expands to ".SRES 3".

8.2.6 .INSTR Character String Manipulation

Format .INSTR[Δ]("<character string 1>","<character string 2>"[,<search start position>])

Description

This function searches in character string 1 for character string 2, and is replaced by the numerical value of the position (with 0 indicating the start of the string 1.) as a decimal number with no radix specification.

The function call is replaced by--1 if character string 2 does not appear in character string 1.

Character strings are specified by enclosing the desired characters in double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

The <search start position> parameter specifies starting the search at the character position indicated, with 0 indicating the start of the string 1. Zero (0) is used as the default when omitted.

The <search start position> parameter must be a backward reference absolute value, with a value of 0 or greater.

Macro formal parameters and preprocessor variables can be specified in the character strings and as the search start position as shown below.

.INSTR("\<formal parameter>","\<formal parameter>",\<formal parameter>)
.INSTR("\&<preprocessor variable>","\&<preprocessor variable>",\&<preprocessor variable>)

This function can only be used inside a macro body, i.e., between a .MACRO directive and its matching. ENDM directive.

This directive is not replaced when it contains a syntax error. No error is displayed in this case.

Examples .MACRO MC0 P1

```
.DATA.W .INSTR("ABCDEFG","\P1",0)
.ENDM
MC0 CDE    [1]
MC0 H      [2]
```

[1] This expands to ".DATA.W 2".

[2] This expands to ".DATA.W -1".

8.2.7 .SUBSTR Character String Manipulation

Format .SUBSTR[Δ]("<character string>",<fetch start>,<fetch length>)

Description

This functions fetches from the specified character string a substring starting at the specified fetch start position of the specified fetch length. The function call is replaced by the fetched character string surrounded in double quotation marks (").

Character strings are specified by enclosing the desired characters in double quotation marks (").

To specify a double quotation mark in a character string, enter two double quotation marks in succession (" ").

The <fetch start> and <fetch length> parameters must be backward reference absolute values.

The value of the fetch start position must be 0 or greater.

The value of the fetch length must be 1 or greater.

If illegal or inappropriate values for the <fetch start> and <fetch length> parameters are specified, the function call is replaced by the null string, i.e. two double quotation marks.

Macro formal parameters and preprocessor variables can be specified in the character string, fetch start, and fetch length parameter positions as shown below.

.SUBSTR("\<formal parameter>",\<formal parameter>,\<formal parameter>)
.SUBSTR("\&<preprocessor variable>",\&<preprocessor variable>,\&<preprocessor variable>)

This function can only be used inside a macro body, i.e., between a .MACRO directive and its matching .ENDM directive.

This directive is not replaced when it contains a syntax error. No error is displayed in this case.

Examples

```
.MACRO MC0P1
.SDATA .SUBSTR("ABCDEFG",0,\P1)
.ENDM
MC02  [1]
MC05  [2]
```

[1] This expands to .SDATA "AB".

[2] This expands to .SDATA "ABCDE".

Section 9 Structured Assembly

9.1 Overview of Structured Assembly

The structured assembly functions provided by this assembler expand instructions which perform testing and iteration.

9.1.1 Conditional Execution (.IF)

The conditional execution statements .IF, .ELSE, and .ENDI expand into a source code that selects execution of the source statements between the .IF statement and its matching .ELSE and .ENDI statements according to the condition specified in the .IF statement. (Note that the .ELSE statement is optional.)

When the instructions expanded for an .IF statement are executed, the following occurs.

1. The condition specified in the .IF statement is tested.
2. If that condition is true, then the source statements between the .IF and .ELSE statements are executed. If there is no .ELSE statement, the source statements between the .IF and .ENDI statements are executed.
3. If the .IF condition is false, then the source statements between the .ELSE and .ENDI statements are executed. If there is no .ELSE statement, then the statements between the .IF and .ENDI statements are not executed.

Example

```
        .IF.B (R0L<EQ>R1L)
                MOV.W   # 0:16,R2
                MOV.W   # 0:16,R3
                MOV.W   # 0:16,R4
                MOV.W   # 0:16,R5
        .ELSE
                ADD.W    R2,R3
                ADD.W    R2,R4
                ADD.W    R2,R5
        .ENDI
```

... The condition is that R0L be equal to R1L.

```
        .IF.B (R0L<EQ>R1L)
                              CMP   R0L,R1L      *
                              BNE   _$I00000     *
                MOV.W   # 0:16,R2
                MOV.W   # 0:16,R3
                MOV.W   # 0:16,R4
                MOV.W   # 0:16,R5
        .ELSE
                               BRA   _$I00001    *
                _$I00000:  .EQU  $               *
                ADD.W    R2,R3
                ADD.W    R2,R4
                ADD.W    R2,R5
        .ENDI
                _$I00001:  .EQU  $               *
```

Note: * Generated code.

In this example, the code will be generated so that when R0L is equal to R1L, the source statements between the .IF and .ELSE statements are executed, and when R0L is not equal to R1L, the source statements between the .ELSE and .ENDI statements are executed.

9.1.2 Execution Selection (.SWITCH)

The processing selection statements .SWITCH, .CASE, .OTHERS, and .ENDS generate a code that selects and executes the source statements between .CASE and .BREAK statements or between .OTHERS and .ENDS statements according to the conditions specified in the .SWITCH and .CASE statements.

When the instructions expanded for a .SWITCH statement are executed, the following occurs.

1. The condition specified in the .SWITCH and .CASE statements is tested. The .CASE conditions are tested in the order specified.
2. When the condition specified by a .SWITCH and .CASE combination is satisfied, the source statements between that .CASE and the corresponding .BREAK statements are executed.
3. If none of the conditions specified by the .SWITCH and .CASE statements are satisfied, the source statements between the .OTHERS and .ENDS statements are executed.

Example

```
.SWITCH.B   (R0L)
.CASE # 0
        MOV.W   R1,R4
.BREAK
.CASE # 1
        MOV.W   R2,R4
.BREAK
.OTHERS
        MOV.W   R3,R4
.ENDS
```

... The condition is that R0L be 0.

... The condition is that R0L be 1.

```
.SWITCH.B    (R0L)
.CASE # 0
                        CMP   #0,R0L      *
                        BNE   _$S00000    *
        MOV.W   R1,R4
.BREAK
                        BRA   _$S00001    *
.CASE # 1
        _$S00000:  .EQU  $
                        CMP   = 1,R0L     *
                        BNE   _$S00002    *
        MOV.W   R2,R4
.BREAK
                        BRA   _$S00001    *
.OTHERS
        _$S00002:  .EQU  $                *
        MOV.W   R3,R4
.ENDS                                     *
        _$S00001:  .EQU  $
```

Note: * Generated code.

In this example, the code will be generated so that when R0L is 0, the source statements between the .CASE #0 and .BREAK statements are executed, and when R0L is 1, the source statements between the .CASE #1 and .BREAK statements are executed. In all other cases, the source statements between the .OTHERS and .ENDS statements are executed.

9.1.3 Iteration (.FOR[U])

The processing iteration statements .FOR[U] and .ENDF generate a code that iteratively executes the source statements between the .FOR[U] and .ENDF statements while the condition specified in the .FOR[U] statement holds.

There are two forms of the .FOR[U] statement: the .FOR statement, which iterates using a signed range test, and the .FORU statement, which iterates using an unsigned range test.

The loop counter and the loop counter initial, final, and increment values are specified in the .FOR[U] statement.

When the instructions expanded for a .FOR[U] statement are executed, the following occurs.

1. The loop counter is initialized.
2. The loop counter continuation condition is tested.
3. If the condition holds, the source statements between the .FOR[U] and .ENDF statements are executed, and the loop counter value is updated by increment value.
4. If the condition fails, the source statements between the .FOR[U] and .ENDF statements are not executed, and the loop is terminated.

Example

```
        .FOR.B (R0L= #1, #5, #1)
                ADD.W    R1,R2
                ADD.W    R1,R3
                ADD.W    R1,R4
                ADD.W    R1,R5
                ADD.W    R1,R6
        .ENDF
```

... This statement specifies that R0L is used as the loop counter with an initial value of 1, a final value of 5, and an increment of 1. The continuation condition is that R0L be less than or equal to 5.

```
        .FOR.B (R0L= #1, #5, #1)
                          MOV    #1,R0L      *
                          BRA    _$F00002    *
                _$F00000 : .EQU  $           *
                ADD.W    R1,R2
                ADD.W    R1,R3
                ADD.W    R1,R4
                ADD.W    R1,R5
                ADD.W    R1,R6
        .ENDF
                _$F00001:  .EQU  $           *
                           ADD   #1,R0L      *
                _$F00002:  .EQU  $           *
                           CMP   #5,R0L      *
                           BLE   _$F00000    *
                _$F00003:  .EQU  $           *
```

Note: * Generated code.

In this case, the code will be generated so that while R0L is less than or equal to 5, the source statements between the .FOR and .ENDF statements are interatively executed.

9.1.4 Iteration (.WHILE)

The processing iteration statements .WHILE and .ENDW generate a code that iteratively executes the source statements between the .WHILE and .ENDW statements while the condition specified in the .WHILE statement holds.

When the instructions expanded for a .WHILE statement are executed, the following occurs.

1. The condition is tested.
2. If the condition holds, the source statements between the .WHILE and .ENDW statements are executed.
3. If the condition fails, the source statements between the .WHILE and .ENDW statements are not executed, and the loop terminates.

Example

```
.WHILE.B (# 5 <GT>R0L)
        ADD. W     R1,R2
        INC. B       R0L
.ENDW
```

... The condition is that 5 be greater than R0L.

```
.WHILE.B (# 5<GT>R0L)
        _$W00000: .EQU      $
                  CMP       # 5, R0L
                  BGE       _$W00001
        ADD.W     R1 , R2
        INC. B    R0L
.ENDW
                  BRA       _$W00000
        _$W00001:  .EQU $
```

* *

Note: *Generated code.
In this case, the code will be generated so that while 5 is greater than R0L, the source statements between the .WHILE and .ENDW statements are interatively executed.

9.1.5 Iteration (.REPEAT)

The processing iteration statements .REPEAT and .UNTIL generate a code that iteratively executes the source statements between the .REPEAT and .UNTIL statements until the condition specified in the .UNTIL statement holds.

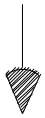
When the instructions expanded for a .REPEAT statement are executed, the following occurs.

1. The source statements between the .REPEAT and .UNTIL statements are executed.
2. The condition is tested.
3. If the condition fails, the source statements between the .REPEAT and .UNTIL statements are executed.
4. If the condition holds, the loop terminates.

Example

```
                    .REPEAT
                              ADD. W  R1,R2
                              INC. B  R0L
                    .UNTIL. B  (#5<LT>R0L)
```

... The condition is that 5 be less than R0L.



```
                    .REPEAT
                              _$R00000: .EQU  $
                              ADD. W  R1,R2
                              INC. B  R0L
                    .UNTIL. B  (#5<LT>R0L)
                              _$R00000: .EQU  $
                                       CMP    #5,R0L
                                       BLE    _$R00000
                              _$R00000: .EQU  $
```

*

*

Note: *Generated code.
In this case, the code is generated so that until 5 is less than R0L, the source statements between the.REPAET and UNTIL statements are iteratively executed.

9.1.6 Notes on Structured Assembly

1. The structured assembly function expands the structured assembly directives into predetermined instructions and symbols, and performs no optimizations whatsoever. Thus the values that can be specified as arguments to these directives are limited by the specifications of the instructions that are generated. Furthermore, there are cases where inefficient code and/or unnecessary symbols are generated.

Example

```
        .IF .B (R0L<LT>#10)     The generated instruction will cause an error.
        MOV .W      R1, R2
        .ENDI
```

The .IF directive is expanded into a CMP instruction, and since this statement is expanded into the statement CMP R0L, #10, an error occurs. To avoid this problem, code the above statement as follows:

```
        IF .B (#10<GT>R0L)      This is expanded into CMP#10, R0L.
        MOV .W      R1, R2
        ENDI
```

2. The structured assembly statements generate symbols of the forms shown below.

```
.IF         -$I00000 to -$I99999
.SWITCH     -$S00000 to -$S99999
.FOR[U]     -$F00000 to -$F99999
.WHILE      -$W00000 to -$W99999
.REPEAT     -$R00000 to -$R99999
```

Accordingly, symbols of these forms cannot be used as user symbols.

9.2 Structured Assembly Function Directive

The condition codes used in structured assembly are listed in table 9-1. Refer to this table in conjunction with the descriptions of the structured assembly function directives that follow.

Table 9-1 Condition Codes

Item	Condition Codes	Comparison Type	Condition Code Specification Type
1	<EQ>	<term 1>=<term 2>	Z=1
2	<NE>	<term 1>=<term 2>	Z=0
3	<GT>	<term 1>=<term 2> (signed comparison)	Z∨ (N⊕V)=0
4	<LT>	<term 1>=<term 2> (signed comparison)	N⊕V=1
5	<GE>	<term 1>=<term 2> (signed comparison)	N⊕V=0
6	<LE>	<term 1>=<term 2> (signed comparison)	Z∨ (N⊕V)=1
7	<HI>	<term 1>=<term 2> (unsigned comparison)	C∨Z=0
8	<LO><CS>	<term 1>=<term 2> (unsigned comparison)	C=1
9	<HS><CC>	<term 1>=<term 2> (unsigned comparison)	C=0
10	<LS>	<term 1>=<term 2> (unsigned comparison)	CvZ=1
11	<VC>		V=0
12	<VS>		V=1
13	<PL>		N=0
14	<MI>		N=1
15	<T>		Always true
16	<F>		Always false

```
Notes: N  The CCR (condition code register) N (negative) flag
Z  The CCR Z (zero) flag
V  The CCR V (overflow) flag
C  The CCR C (carry) flag
v  Logical or
⊕  Logical exclusive or
```

9.2.1 .IF Conditional Execution

```
Format Label  Operation Operands

    X  .IF[.s] [: d]   {(<term 1> <cc> <term 2>) | (<cc>)}

    [X .ELSE[ :d]   X]

    X  .ENDI  X

    s (size): {B | W | L}

    d (branch size):{8 | 16}

    cc (condition code):
    {EQ|NE|GT|LT|GE|LE|HI|LO|HS|LS|CC|CS|VC|VS|PL| MI|T|F}
```

Description

Source statements are selected and executed based on the result of testing the condition specified in the .IF statement.

When the condition holds, the source statements between the .IF and the .ELSE statements are executed, and when the condition fails, the source statements between the .ELSE and the .ENDI statements are executed.

The .ELSE statement may be omitted. When omitted, the source statements between the .IF and the .ENDI statements are executed if the condition holds.

There are two types of conditions as follows:

1. Comparison type

   In the comparison type, a decision is made based on a condition code based comparison of two terms.

   The terms must have addressing modes that can be used with the CMP instruction.
2. Condition code specification type

   In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

The size specifies the size of the terms compared in a comparison type condition. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

B	Byte (1 byte)
W	Word (2 bytes)
L	Long word (4 bytes)

Byte is taken as the default when the size specifier is omitted.

The branch size can be specified on both the .IF and the .ELSE statements.

The .IF branch size specifies the branch size from the .IF statement to the .ELSE or .ENDI statement.

The .ELSE branch size specifies the branch size from the .ELSE statement to the .ENDI statement.

The following branch sizes can be specified.

8	8 bits
16	16 bits

Refer to section 5.7.4, DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

Refer to table 9-1, Condition Codes, for details on the condition code conditions.

Limitations

1. "L" cannot be specified as the size with the H8/300 and H8/300L processors.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.
3. The size of the code generated by the source statements between an .IF statement and an .ELSE statement, between an .ELSE statement and an .ENDI statement, or between an .IF statement and an .ENDI statement (when the .ELSE statement is omitted) cannot exceed the range corresponding to the specified branch size.

The maximum source code size for the different branch sizes are as follows:

8	About 100 bytes
16	About 32,700 bytes

4. When this directive is used, symbols from _$I00000 to _$I99999 may be generated. Thus these symbols should not be used in programs which use the .IF directive.

   Examples

```
.IF .B (R0L <EQ> R1L)
   ADD.B  #1:8, R0L
   MOV.W  R2, R3
   MOV.W  R4, R5    [1]
.ELSE
   ADD.B  #1:8, R1L
   MOV.W  R3, R2
   MOV.W  R5, R4    [2]
.ENDI
```

   This is an example of the comparison type condition.

   When ROL is equal to R1L, statements [1] will be executed, and when R0L is not equal to R1L, statements [2] will be executed.

```
.IF .B (#H'10 <LT> R0L)
   MOV.W  R1, R2
   MOV.W  R1, R3
   MOV.W  R1, R4    [3]
.ENDI
```

   This is an example of the comparison type condition.

   Statements [3] will be executed when H'10 is less than R0L (under a signed comparison).

```
.IF (<NE>)
   ADD.B  #5:8, R0L    [4]
.ELSE
   MOV.B  R0L, R1L     [5]
.ENDI
```

This is an example of the condition code specification type condition.

When the CCR (condition code register) Z (zero) flag is 0, statement [4] will be executed, and when 1, statement [5] will be executed.

```
.IF (<VS>)
   MOV.B  #0:8, R0L
   MOV.B  #0:8, R1L
   MOV.B  #0:8, R2L    [6]
.ENDI
```

This is an example of the condition code specification type condition.

When the CCR (condition code register) V (overflow) flag is 1, statements [6] will be executed.

```
.IF.B  (#0 <LE>R0L)
   .IF.B  (#50<GE>R0L)
   MOV.B  R0L, R1L
   MOV.B  R0L, R2L
   MOV.B  R0L, R3L     [7]
   .ENDI
.ENDI
```

This is an example of a nested .IF construction.

If the condition 0≤R0L≤50 holds under signed comparison, then statements [7] will be executed.

9.2.2 .SWITCH Execution Selection

```
Format Label  Operation Operands
   X  .SWITCH[.s]  {(<register>)|(CCR)}
   X  .CASE[:d] {<term>|<cc>}
   [X .BREAK[:d]   X]
   X  .CASE[:d} {<term>|<cc>}
   [X .BREAK[:d]   X]
   ⋮
   [X .OTHERS   X]
   X  .ENDS  X
s (size): {B | W | L}
d (branch size):{8 | 16}
cc (condition code):   {EQ | NE | GT | LT | GE | LE | HI | LO | HS |
LS | CC | CS | VC | VS | PL | MI | T | F}
```

Description

Source statements are selected and executed based on the result of testing the conditions specified in the .SWITCH and .CASE statements.

When the condition specified by a .SWITCH statement and a corresponding .CASE statement are satisfied, the source statements between that .CASE statement and its corresponding .BREAK statement are executed. When no .CASE condition is satisfied, the source statements between the .OTHERS and the .ENDS statements are executed.

The .SWITCH and .CASE conditions are tested in order.

When a .BREAK statement is omitted, execution continues to the statements between the next .CASE and .BREAK, or to the following statements between .OTHERS and .ENDS.

There are two types of conditions as follows:

1. Comparison type

   In the comparison type, a register and a term are tested for equality.

   The register is specified in the .SWITCH statement.

   The term is specified in the .CASE statement using an addressing mode that can be used as the source operand in the CMP instruction.

2. Condition code specification type

   In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

   CCR is specified in the .SWITCH statement.

   The condition code(s) are specified in the .CASE statement(s).

   The size specifies the size of the terms compared in a comparison type condition. It has no meaning with condition code specification type conditions.

   The size specifiers are interpreted as follows:

   B Byte (1 byte)

   W Word (2 bytes)

   L Long word (4 bytes)

   Byte is taken as the default when the size specifier is omitted.

   The branch size can be specified on both the .CASE and the .BREAK statements.

   The .CASE branch size specifies the branch size from the .CASE statement to the next .CASE, .OTHERS, or .ENDS statement.

   The .BREAK branch size specifies the branch size from the .BREAK statement to the .ENDS statement.

The following branch sizes can be specified.

8 8 bits

16 16 bits

Refer to section 5.7.4, .DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

Refer to table 9-1, Condition Codes, for details on the condition code conditions.

Limitations

1. "L" cannot be specified as the size with the H8/300 and H8/300L processors.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.
3. The size of the code generated by the source statements corresponding to each .CASE statement and the size of the code between a .BREAK statement and corresponding .ENDS statement cannot exceed the range corresponding to the specified branch size.

   The maximum source code size for the different branch sizes are as follows:

   8 About 100 bytes

   16 About 32,700 bytes
4. When this directive is used, symbols from _$S00000 to _$S99999 may be generated. Thus these symbols should not be used in programs which use the .SWITCH directive.

Examples

```
.SWITCH.B (R0L)
.CASE #0
   MOV.W  R1, R4    [1]
.BREAK
.CASE #1
   MOV.W  R2, R4    [2]
.BREAK
.OTHERS
   MOV.W  R3, R4    [3]
.ENDS
```

This is an example of the comparison type condition.

When R0L is equal to 0, statement [1] will be executed, and when R0L is equal to 1, statement [2] will be executed, and in all other cases, statement [3] will be executed.

```
.SWITCH (CCR)
.CASE <CS>
      MOV.W  R0, R3    [4]
.BREAK
.CASE <MI>
      MOV.W  R1, R3    [5]
.ENDS
```

This is an example of the condition code type condition.

When the CCR (condition code register) C (carry) flag is 1, statement [4] will be executed, and when the N (negative) flag is 1, statement [5] will be executed.

```
.SWITCH.B (R0L)
.CASE #0
.CASE #1
.CASE #2
      MOV.W  R1, R3    [6]
.BREAK
.CASE #3
      MOV.W  R2, R3    [7]
.ENDS
```

This is an example of omitting the .BREAK statement.

When R0L is equal to 0, 1, or 2, statement [6] will be executed, and when R0L is 3, statement [7] will be executed.

9.2.3 .FOR[U] Iteration

```
Format Label  Operation Operands
   X  .FOR [U] [.s] [:d]      (<loop counter>=<initial value>,<final
value>[,[{+ | - }]<increment value>]
   X  .ENDF  X
   s (size): {B | W | L}
   d (branch size): {8 | 16}
```

Description

The condition specified by the loop counter and final value is tested, and the source statements between the .FOR[U] and .ENDF statements are iterated while that condition is true.

There are two forms of the .FOR[U] statement: the .FOR statement, which iterates using a signed range test, and the .FORU statement, which iterates using an unsigned range test.

The size specification specifies the size of the loop counter, initial value, final value and increment value.

The size specifiers are interpreted as follows:

- B Byte (1 byte)
- W Word (2 bytes)
- L Long word (4 bytes)

Byte is taken as the default when the size specifier is omitted.

The branch size specifies the branch size from the .FOR[U] statement to the .ENDF statement.

The following branch sizes can be specified.

- 8 8 bits
- 16 16 bits

Refer to section 5.7.4, .DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

The operands are interpreted as follows:

1. <loop counter>=<initial value>

   This specifies the loop counter's initial value.

   The loop counter must be a register.

   The initial value must have an addressing mode that can be specified as the source operand of the MOV instruction.

2. <final value>

   The final value is the value which is compared with the loop counter.

   There are two types of iteration conditions as follows:

Positive increment direction	<loop counter>≤<final value>
Negative increment direction	<loop counter>≥<final value>

   The final value must have an addressing mode that can be specified as the source operand of the CMP instruction.

(3) <increment value>

   The increment value is the amount the loop counter is incremented or decremented on each loop iteration.

   The increment direction is specified by a plus (+) to indicate a positive increment direction and a minus (-) to indicate a negative decrement direction.

   Plus (+) is taken as the default when no increment direction is specified.

   The increment value must have an addressing mode that can be specified as the source operand for the ADD and SUB instructions.

   The value +#1 is used as the default when no increment value is specified.

The following table indicates the possible ranges of the loop counter value. Pay careful attention to the loop counter range, since infinite loops can result from inappropriate values.

Directive	Increment Direction	Size	Loop Counter Range (initial value to final value)
.FOR	+	B	-128 to 126
		W	-32,768 to 32,766
		L	-2,147,483,648 to 2,147,483,646
	-	B	127 to -127
		W	32,767 to -32,767
		L	2,147,483,647 to -2,147,483,647
.FORU	+	B	0 to 254
		W	0 to 4,294,967,294
	-	B	255 to 1
		W	65,535 to 1
		L	4,294,967,295 to 1

Limitations

1. "L" cannot be specified as the size with the H8/300 and H8/300L processors.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.
3. The size of the code generated by the source statements between a .FOR[U] statement and its corresponding .ENDF statement cannot exceed the range corresponding to the specified branch size.

   The maximum source code size for the different branch sizes are as follows:

   8.....About 100 bytes

   16.....About 32,700 bytes

4. When this directive is used, symbols from _$F00000 to _$F99999 may be generated. Thus these symbols should not be used in programs which use the .FOR[U] directive.

Examples

```
.FOR.B (R0L = #1, #10, +#1)
   ADD.B  R0L, R1L    [1]
.ENDF
```

This is an example of a .FOR loop.

The loop counter is R0L, the initial value is #1, the final value is #10, and the increment value is +#1.

Statement [1] will be iterated while R0L is less than or equal to 10 under a signed comparison.

```
.FOR.W (R0=R1, R2, -R3)
   ADD.B  #1:8, R5L   [2]
.ENDF
```

This is an example of a .FOR loop.

The loop counter is R0, the initial value is R1, the final value is R2, and the increment value is -R3.

Statement [2] will be iterated while R0 is greater than or equal to R2 under a signed comparison.

```
.FORU.B   (R0L= #1, #200, +#1)
   ADD.W  R1, R2
   ADD.W  R3, R4
   ADD.W  R5, R6    [3]
.ENDF
```

This is an example of a .FORU loop.

The loop counter is R0L, the initial value is #1, the final value is #200, and the increment value is +#1.

Statement [3] will be iterated while R0L is less than or equal to 200 under an unsigned comparison.

```
.FORU.L   (ER0= #H'00000100, #H'000001FC, +#4)
   MOV.L  @ER0, ER2
   MOV.L  ER2, @(H'00001100:32, ER1)
   ADDS.L #4, ER1     [4]
.ENDF
```

This is an example of a .FORU loop.

The loop counter is ER0, the initial value is #H'00000100, the final value is #H'000001FC, and the increment value is +#4.

Statement [4] will be iterated while ER0 is less than or equal to #H'000001FC under an unsigned comparison.

```
.FOR.B (R0L = #1, #10, +#1)        [5]
   .FOR.B (R1L = #1, #10, +#1)     [6]
      MOV.B  R0L, R2L
      ADD.B  R1L, R2L
      MOV.B  R2L, @ER3
      ADDS.L #1, ER3
   .ENDF         [6]
.ENDF         [5]
```

This is an example of nested .FOR loops.

.FOR loop [6] is iterated by .FOR loop [5].

9.2.4 .WHILE Iteration

```
Format Label  Operation Operands
   X  .WHILE[.s] [:d] {(<term 1><term 2>) | (<cc>)}
   X  .ENDW  X
```

s(size): {B | W | L}

d (branch size): {8 | 16}

cc (condition code): {EQ | NE | GT | LT | GE |HI | LE | HI | LO | HS | LS | CC | CS | VC | VS | PL | MI | T | F}

Description

The condition specified in the .WHILE statement is tested, and the source statements between the .WHILE and .ENDW statements are iterated while that condition is true.

There are two types of conditions as follows:

1. Comparison type

   In the comparison type, a decision is made based on a condition code based comparison of two terms.

   The terms must have addressing modes that can be used with the CMP instruction.
2. Condition code specification type

   In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.

The size specifies the size of the terms compared in a comparison type condition. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

B Byte (1 byte)
W Word (2 bytes)
L Long word (4 bytes)

Byte is taken as the default when the size specifier is omitted.

The branch size specifies the branch size from the .WHILE statement to the .ENDW statement.

The following branch sizes can be specified.

8 8 bits
16 16 bits

Refer to section 5.7.4 .DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

Refer to table 9-1, Condition Codes, for details on the condition code conditions.

Limitations

1. "L" cannot be specified as the size with the H8/300 and H8/300L processors.
2. The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.
3. The size of the code generated by the source statements between a .WHILE statement and its corresponding .ENDW statement cannot exceed the range corresponding to the specified branch size.

   The maximum source code size for the different branch sizes are as follows:

   8 About 100 bytes
   16 About 32,700 bytes
4. When this directive is used, symbols from_$W00000 to _$W99999 may be generated. Thus these symbols should not be used in programs which use the .WHILE directive.

Examples

```
.WHILE.B (#50 <GT>R0L)
   ADD.W R1, R2
   SUB.W R1, R3
   ADD.B #1:8, R0L   [1]
.ENDW
```

This is an example of the comparison type condition.

Statements [1] will be iterated while 50 is greater than R0L under signed comparison.

```
.WHILE .W (R0<LS>R1)
   SUB.B R2L, R3L
   SUB.B R2L, R4L
   SUB.W R5, R1    [2]
.ENDW
```

This is an example of the comparison type condition.

Statements [2] will be iterated while R0 is less than or equal to R1 under unsigned comparison.

```
.WHILE (<NE>)
   MOV.L @ER2, ER4
   MOV.L ER4, @ER3
   ADDS.L #4, ER2
   ADDS.L #4, ER3
   SUB.B R1L, R0L    [3]
.ENDW
```

This is an example of the condition code specification type condition.

Statements [3] will be iterated while the CCR (condition code register) Z (zero) flag is 0.

```
.WHILE (<PL>)
   MOV.L ER2, @ER1
   ADDS.L #4, ER1
   MOV.L ER3, @ER1
   ADDS.L #4, ER1
   ADD.W #-1, R0     [4]
.ENDW
```

This is an example of the condition code specification type condition.

Statements [4] will be iterated while the CCR (condition code register) N (negative) flag is 0.

```
.WHILE.L (#H'000200<HI>ER0)        [5]
   MOV.B #0:8, R2L
   MOV.B #H'FF:8, R3L
   .WHILE.B (#H'00<NE>R3L          [6]
      MOV.B @ER0, R3L
      ADDS.L #1, ER0
      ADD.B #1:8, R2L
   .ENDW         [6]
   MOV.B R2L, @ER1
   ADDS.L #1, ER1
.ENDW             [5]
```

This is an example of nested .WHILE loops.

.WHILE loop [6] will be iterated by .WHILE loop [5].

9.2.5 .REPEAT Iteration

```
Format Label  Operation Operands
   X   .REPEAT   X
   X   .UNTIL[.s]   {(<Term 1><cc><term 2>) | (<cc>)}
```

s(size): {B | W | L}

cc (condition code): {EQ | NE | GT | LT | GE | LE | HI | LO | HS | LS | CC | CS | VC | VS | PL | MI | T | F}

Description

The source statements between the .REPEAT and .UNTIL statements are iterated until the condition specified in the .UNTIL statement is satisfied.

The source statements between the .REPEAT and the .UNTIL statements are executed at least once so that the .UNTIL condition can be tested.

There are two types of conditions as follows:

1. Comparison type
   In the comparison type, a decision is made based on a condition code based comparison of two terms.
   The terms must have addressing modes that can be used with the CMP instruction.
2. Condition code specification type
   In the condition code specification type, a decision is made based on the specified CCR (condition code register) state.
   The size specifies the size of the terms compared in a comparison type condition. It has no meaning with condition code specification type conditions.

The size specifiers are interpreted as follows:

B.....Byte (1 byte)
W.....Word (2 bytes)
L.....Long word (4 bytes)

Byte is taken as the default when the size specifier is omitted.

Refer to table 9-1, Condition Codes, for details on the condition code conditions.

Limitations

1. "L" cannot be specified as the size with the H8/300 and H8/300L processors.
2. The size of the code generated by the source statements between the .REPEAT statement and its corresponding .UNTIL statement cannot exceed the following limitations.
   H8S/2600 advanced mode.....About 32,700 bytes
   H8S/2600 normal mode.....About 32,700 bytes
   H8S/2000 advanced mode.....About 32,700 bytes
   H8S/2000 normal mode.....About 32,700 bytes
   H8/300H advanced mode.....About 32,700 bytes
   H8/300H normal mode.....About 32,700 bytes
   H8/300.....About 100 bytes
   H8/300L.....About 100 bytes
3. When this directive is used, symbols from_$R00000 to _$R99999 may be generated. Thus these symbols should not be used in programs which use the .REPEAT directive.

Examples

```
.REPEAT
   MOV.L @ER0, ER2
   MOV.L ER2, @ER1
   ADDS.L #4, ER0
   ADDS.L #4, ER1 [1]
.UNTIL.L (#H'001000<LS>ER0)
```

This is an example of the comparison type condition.

Statements [1] will be iterated until H'001000 is less than or equal to ER0 under unsigned comparison.

```
.REPEAT
   ADD.W R2, R3
   ADD.W R2, R4
   SUB.B R1L, R0L [2]
.UNTIL (<EQ>)
```

This is an example of the condition code specification type condition.

Statements [2] will be iterated until the CCR (condition code register) Z (zero) flag is 1.

9.2.6 .BREAK Iteration Termination

```
Format Label  Operation Operands
   X   .BREAK[:d]    X
   d (branch size): {8 | 16}
```

Description

The .BREAK statement terminates .FOR[U], .WHILE, and .REPEAT loops, exiting the loop without executing the source statements following the .BREAK statement. More specifically, the .BREAK statement executes an unconditional jump to the .ENDF, .ENDW, or .UNTIL statement that closes the corresponding .FOR[U], .WHILE, or .REPEAT loop, thus terminating the processing.

The branch size specifies the branch size from the .BREAK statement to the corresponding .ENDF, .ENDW, or .UNTIL statement.

The following branch sizes can be specified.

8.....8 bits

16.....16 bits

Refer to section 5.7.4, .DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

This directive can also be used with the .SWITCH directive.

Refer to section 9.2.2, .SWITCH, for details on use of the .SWITCH directive.

Limitation The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.

Example

```
.WHILE (  )
.IF.B (#10 <LE>R0L)
   .BREAK
.ENDI
ADD.W R1, R2
INC.B R0L
.ENDW
```

The iteration will terminate when 10 is less than or equal to R0L.

9.2.7 .CONTINUE Loop Interrupt and Restart

```
Format Label  Operation Operands
   X  .CONTINUE[:d]    X
   d (branch size): {8 | 16}
```

Description

The .CONTINUE statement restarts loop processing without executing the remaining source statements in the .FOR[U], .WHILE, and .REPEAT loops. More specifically, the .CONTINUE statement branches unconditionally to the loop test point in a .FOR[U], .WHILE, or .REPEAT loop.

The branch size specifies the branch size from the .CONTINUE statement to the corresponding .ENDF, .WHILE, or .UNTIL statement.

The following branch sizes can be specified.

8.....8 bits

16.....16 bits

Refer to section 5.7.4, .DISPSIZE in the Language Guide, and section 3.3.3, BR_RELATIVE and section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for the setting used when the branch size specification is omitted.

Limitation The value 16 cannot be specified as the branch size with the H8/300 and H8/300L processors.

Example

```
.WHILE.B (#10<GT>R0L)
   INC.B R0L
   INC.B R1L
   .IF.B (#10<LT>R1L)
      .CONTINUE
   .ENDI
   ADD.W R2, R3 [1]
.ENDW
```

Statement [1] will not be executed when 10 is less than R1L.

Usage Guide

Section 1 Overview

1.1 Assembler Positioning

Figure 1-1 shows the position of the assembler in the program development process.

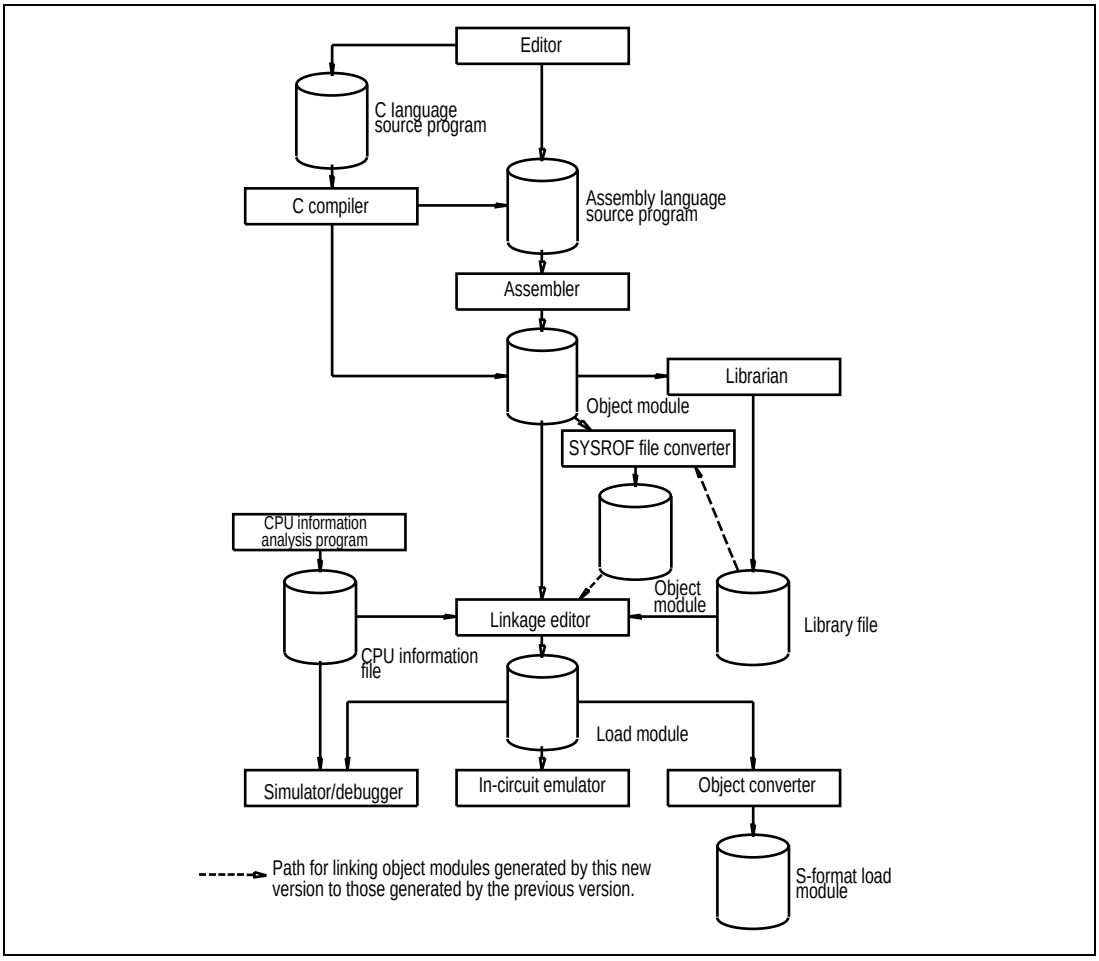


Figure 1-1 Assembler Positioning

1.2 I/O Organization

Figure 1-2 shows the assembler I/O organization.

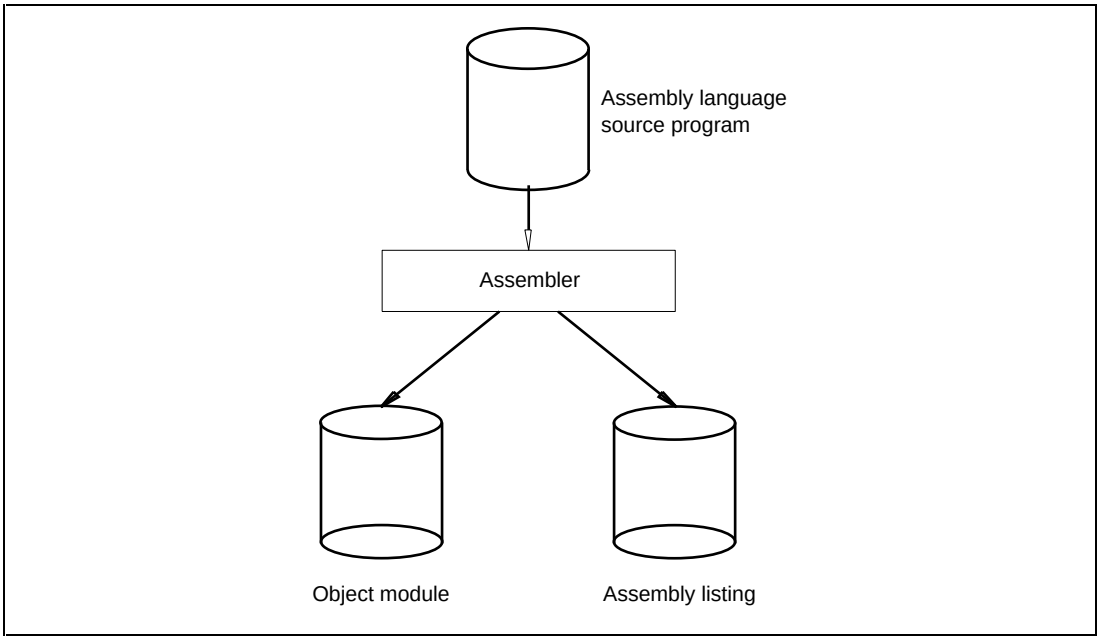


Figure 1-2 I/O Organization

Section 2 Invoking the Assembler

2.1 Command Format

The assembler is invoked from the command line, with the input file, output file, and other parameters specified on the same command line. The assembly method can also be specified using command line options.

The format of the command line that invokes the assembler is shown below.

>asm38 Δ<input file>[,<input file>...] [[Δ]-<command line option>[[Δ]-<command line option>...]]

[1] [2] [3]

[1] The assembler invocation command

[2] The specification of the source file(s) to be assembled. When two or more input files are specified, they are divided with a comma (,). They are joined together in the order of input and then assembled as one source file.

[3] Command line option specifications

Example > asm38 aaa.mar-cpu=2600a -list -debug This command line assembles the source file aaa.mar.

Note: When using MS-DOS, command line options can be divided with hyphens (-) or slashes (/).

2.2 H38CPU Environmental Variable

The assembler assembles source files for the CPU classification specified by the H38CPU environmental variable.

The CPU classification is specified as follows:

- UNIX
  - — C Shell
    setenv H38CPU* CPU classification [:Bit width of address space]
  - — Bourne/Korn Shell
    H38CPU* = CPU classification [:Bit width of address space] export H38CPU*
- MS-DOS
  SET H38CPU* = CPU classification [:Bit width of address space]

Note: This environmental variable must be specified in capital letters.

The CPU classifications that can be specified are 2600A, 2600N, 2000A, 2000N, 300HA, 300HN, 300, and 300L. If the CPU classification is an advanced mode, the bit width of the address space can be specified.

The bit width of address spaces are listed in table 2-1.

Table 2-1 Bit Width of Address Spaces

CPU Classification	Bit Width of Address Space	Default Size
2600A	32,28,24,20	32
2000A	32,28,24,20	32
300HA	24,20	24

The CPU classification specified by the H38CPU environmental variable is valid only when neither the CPU command line option nor the .CPU directive has been specified.

2.3 Files Used by the Assembler

The assembler uses the following three types of file.

1. Source files
   These are the files that holds source programs.
2. Object files
   These are object module output files.
3. Assembly listing files
   These are files that hold the generated assembly listings.

The following defaults are used when the extension is omitted in file specifications.

- Source files..... xxx.mar or xxx.src
- Object files..... xxx.obj
- Assembly listing files..... xxx.lis

Note: When using MS-DOS, the file extension is specified in capital letters.

2.4 Value Returned to OS

The assembler returns assemble results to the OS (Operating System) as a return value.

Values that can be returned to the OS are listed in table 2-2.

Table 2-2 Values Returned to OS

	Return Value	
Assemble Result	**MS-DOS**	**UNIX**
Normal completion	0	0
Warning	0	0
Error	2	1
Fatal error	4	1

The values returned to the OS can be changed by specifying an ABORT command line option. Refer to section 3.3.6, ABORT, in the Usage Guide, for details on specifying the ABORT command line options.

Section 3 Command Line Options

3.1 Types of Command Line Option

The assembly method can be specified using command line options when the assembler is invoked.

Table 3-1 lists the command line options

Table 3-1 Command Line Options

Item	Command Line Option	Function	Reference Section
1	Option concerning the CPU		
	CPU	CPU specification	3.2.1
2	Options concerning object modules		
	[NO]OBJECT	Object module output	3.3.1
	[NO]DEBUG	Debugging information output	3.3.2
	BR_RELATIVE	Displacement size specification	3.3.3
	[NO]OPTIMIZE	Optimization specification	3.3.4
	[NO]EXCLUDE	Unreferenced external reference symbols information output	3.3.5
	ABORT	Change of values returned to OS and object output control at error generation	3.3.6
3	Options concerning the assembly listing		
	[NO]LIST	Assembly listing output	3.4.1
	[NO]SOURCE	Source program listing output	3.4.2
	[NO]CROSS_REFERENCE	Cross reference listing output	3.4.3
	[NO]SECTION	Section information listing output	3.4.4
	[NO]SHOW	Source program listing partial output	3.4.5
	LINES	Listing lines/page specification	3.4.6
	COLUMNS	Listing columns/line specification	3.4.7
4	Option concerning file inclusion		
	INCLUDE	Include file directory specification	3.5.1
5	Options concerning conditional assembly		
	ASSIGNA	Integer preprocessor variable definition	3.6.1
	ASSIGNA	Character preprocessor variable definition	3.6.2
6	Option concerning command line specification		
	SUBCOMMAND	Subcommand file specification	3.7.1

3.2 Command Line Option Concerning the CPU

3.2.1 CPU CPU Specification

Format

CPU=<CPU type>[:<address space bit width>]

<CPU type>: {2600A | 2600N | 2000A | 300HA | 300HN³300 | 300L}

CPU Classification	Address Space Bit Widths	Default Size
2600A	32,28,24,20	32
2000A	32,28,24,20	32
300HA	24,20	24

The underlined part is the abbreviated form of this option.

Description

The CPU command line option specifies the object CPU for the source program to be assembled.

The address space bit width can be selected only when the CPU for advanced mode is selected. The accessible space depends on the selected bit width.

CPU=2600A [:32}..... H8S/2600 advanced mode

CPU=2600A :28..... H8S/2600 advanced mode

CPU=2600A :24..... H8S/2600 advanced mode

CPU=2600A :20..... H8S/2600 advanced mode

CPU=2600N..... H8S/2600 normal mode

CPU=2000A [:32]..... H8S/2000 advanced mode

CPU=2000A :28..... H8S/2000 advanced mode

CPU=2000A :20..... H8S/2000 advanced mode

CPU=2000N..... H8S/2000 normal mode

CPU=300HA [:24]..... H8S/300H advanced mode

CPU=300HA :20..... H8S/300H advanced mode

CPU=300HN..... H8S/300H normal mode

CPU=300..... H8/300

CPU=300L..... H8/300L

The assembler assembles the program for the specified CPU.

Relationship with Assembler Directives

```
CPU=<CPU type>[:<address space bit width>]....The CPU type specified
by
                                          CPU command line option
Specification omitted --- .CPU<CPU type>......The .CPU type specified
                                          by .CPU directive
Specification omitted  H38CPU environmental variable.......The CPU
type
                                                        of the H38
                                                        CPU
                                                        environmental
                                                        variable
Specification omitted                     Error message 933
```

Examples

```
> asm38 aaa.mar-cpu=2600a:32
```

The source program will be assembled for the 4G mode in H8S/2600 advanced mode.

```
> asm38 aaa.mar-cpu=2600a:28
```

The source program will be assembled for the 256M mode in H8S/2600 advanced mode.

```
> asm38 aaa.mar-cpu=2600a:24
```

The source program will be assembled for the 16M mode in H8S/2600 advanced mode.

```
> asm38 aaa.mar-cpu=2600a:20
```

The source program will be assembled for the 1M mode in H8S/2600 advanced mode.

3.3 Command Line Options Concerning Object Modules

3.3.1 [NO]OBJECT Object Module Output Control

Format

OBJECT[=<file name>]

NOOBJECT

The underlined part is the abbreviated form of this option.

Description

This command line option specifies either the output of an object module or the suppression of that output.

OBJECT..... Output

NOOBJECT..... Output suppressed

The file name specifies the name of the file to which the object module is written.

When omitted, the object module is written to a file with the same name as the source module, but with the extension ".obj."

If the extension is omitted, the extension ".obj" is used.

Do not use the same file name and extension as the source program for the object module.

Relationship with Assembler Directives

OBJECT..... Output

NOOBJECT..... Output suppressed

Specification omitted-.OUTPUT OBJ..... Output

.OUTPUT NOOBJ..... Output suppressed

Specification omitted ..... Output

Examples

```
> asm38 aaa.mar-object
```

An object module will be output. Its file name will be aaa.obj.

```
> asm38 aaa.mar-noobject
```

No object module will be output.

3.3.2 [NO]DEBUG Debugging Information Output Control

Format

DEBUG

NODEBUG

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of debugging information or the suppression of that output.

DEBUG ..... Output

NODEBUG ..... Output suppressed

This command line option is valid when an object module is output.

Relationship with Assembler Directives

DEBUG ..... Output

NODEBUG ..... Output suppressed

Specification omitted-.OUTPUT DBG ..... Output

.OUTPUT NODBG ..... Output suppressed

.Specification omitted ..... Output suppressed

```
Examples
> asm38 aaa.mar-debug
```

Debugging information will be output.

```
> asm38 aaa.mar-nodebug
```

No debugging information will be output.

3.3.3 BR_RELATIVE Displacement Size Specification

Format

BR_RELATIVE=<bit count>

<bit count>: {8╨ 16}

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the default displacement size used when the branch displacement is a forward reference value.

BR_RELATIVE=8 ..... 8 bits

BR_RELATIVE=16 ..... 16 bits

When the OPTIMIZE command line option is not specified, displacements with no displacement size (:8 or :16) are the objects of this command line option.

This command line option is valid when the object CPU is either the H8S/2600, H8S/2000, and H8/300H advanced mode or the H8S/2600, H8S/2000, and H8/300H normal mode.

The displacement size is set to 8 bits in the H8/300 and H8/300L.

Relationship with Assembler Directives

OPTIMIZE command line option

NOOPTIMIZE command line option -BR_RELATIVE=<bit count> ... The BR_RELATIVE bit count

Specification omitted .DISPSIZE FBR=<bit count> ..... The .DISPSIZE bit count

Specification omitted ..... The defaults are 16 bits for the H8S/2600, H8S/2000, and H8/300H advanced mode, and 8 bits for the H8S/2600, H8S/2000,

and H8/300H normal mode.

The underlined option is the default setting.

Refer to section 3.3.4, [NO]OPTIMIZE in the Usage Guide, for details on displacement size optimization.

Examples

```
> asm38 aaa.mar-br_relative=16
```

In this example, the forward reference branch instruction default displacement size is set to be 16 bits.

```
> asm38 aaa.mar-br_relative=8
```

In this example, the forward reference branch instruction default displacement size is set to be 8 bits.

3.3.4 [NO]OPTIMIZE Optimization Specification

Format

OPTIMIZE

NOOPTIMIZE

The underlined part is the abbreviated form of this option.

Description

This command line option specifies whether or not to optimize the displacement size in the register-indirect-with-displacement addressing mode and the absolute address size in the absolute addressing mode.

This command line option is valid for executable instructions that do not specify displacement size (:8 or :16) nor absolute address storage size (:8, :16, :24, or :32).

OPTIMIZE ..... The sizes are optimized.

NOOPTIMIZE ..... The sizes are not optimized.

The displacement size is determined as follows, depending on the PC relative displacement value type.

When the CPU classification is H8S/2600 advanced mode and OPTIMIZE is not specified:

Displacement value-Absolute value --32,768 to 32,767 .... 16 bits*

Relative value ..... 16 bits

External reference value ..... 16 bits

When the CPU classification is H8S/2600 advanced mode and OPTIMIZE is specified:

Displacement value--Absolute value-128 to 127 ..... 8 bits

-32,768 to-129, 128 to 32,767 ..... 16 bits

Relative value ..... 16 bits

External reference value ..... 16 bits

* This is valid when the referenced absolute symbol is defined after the referencing instruction.

Relationship with Assembler Directives

OPTIMIZE ..... Optimized bit count

NOOPTIMIZE-BR_RELATIVE <bit count>= .... The BR_RELATIVE bit count

Specification omitted-.DISPSIZE FBR=<bit count> ..... The .DISPSIZE bit count

Specification omitted ..... 8 bits

The underlined option is the default setting.

The OPTIMIZE command line option takes priority over command line option BR_RELATIVE and directive .DISPSIZE concerning object module output.

Examples

```
> asm38 aaa.mar-optimize
```

The object module will be optimized.

```
> asm38 aaa.mar-nooptimize
```

The object module will not be optimized.

3.3.5 [NO]EXCLUDE Information Output Control for Unreferenced External Reference Symbols

Format

EXCLUDE

NOEXCLUDE

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of debugging information on unreferenced external reference symbols, or the suppression of that output.

EXCLUDE ..... Output suppressed

NOEXCLUDE ..... Output

The underlined option is the default setting.

The object module size can be reduced by suppressing output of information on unreferenced external reference symbols.

Examples

```
> asm38 aaa.mar-exclude
```

Information on unreferenced external reference symbols will not be output.

```
> asm38 aaa.mar-noexclude
```

Information on unreferenced external reference symbols will be output.

3.3.6 ABORT Change of Values Returned to OS and Object Output Control at Error Generation

Format

ABORT=<error level>

<error level>: {WARNING I ERROR}

The underlined part is the abbreviated form of this option.

Description

This command line option changes the values returned to the OS according to the error level specification and assemble results.

The values returned to the OS are shown in the following table:

			Value returned to OS when ABORT is Specified			
Number of Errors			ABORT=WARNING		ABORT=ERROR	
Warning	Error	Fatal Error	MS-DOS	UNIX	MS-DOS	UNIX
0	0	0	0	0	0	0
1 or more	0	0	2	1	0	0
-	1 or more	0	2	1	2	1
-	-	1 or more	4	1	4	1

The return values for the underlined error level are the default settings. This command line option suppresses object module output when the return value to the OS is 1 or more.

This command line option is valid only when an object module has been specified to be output by the .OUTPUT directive or [NO]OBJECT command line option.

Examples

```
> asm38 aaa.mar -abort=warning
```

If an error issuing a warning occurs:

- MS-DOS
  Because the return value to the OS is 2, an object module will not be output.
- UNIX
  Because the return value to the OS is 1, an object module will not be output.

```
> asm38 aaa.mar -abort=error
```

If no error or fatal error occurs:

- MS-DOS
  Because the return value to the OS is 0, an object module will be output.
- UNIX
  Because the return value to the OS is 0, an object module will be output.

3.4 Command Line Options Concerning the Assembly Listing

3.4.1 [NO]LIST Assembly Listing Output Control

Format

LIST[=<file name>]

NOLIST[=<file name>]

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of an assembly listing, or the suppression of that output.

LIST ..... Output

NOLIST ..... Output suppressed

The file name specifies the file to which the assembly listing is output.

When the file name is omitted, the following defaults are used:

LIST ..... A file with the same name as the source file, but with the extension ".lis.

"NOLIST ..... Lines which generated errors are displayed on the screen.

If the extension is omitted from the file name specification, the extension ".lis" is used.

If a file name is specified with the NOLIST command line option, only lines which generate an error will be output to that file as an assembly listing.

Do not use the same file name and extension as the source program for the assembly listing.

Relationship with Assembler Directives

LIST ..... Output

NOLIST ..... Output suppressed

Specification omitted -- .PRINT LIST ... Output

.PRINT NOLIST ..... Output suppressed

Specification omitted ..... Output suppressed

Examples

```
> asm38 aaa.mar -list
```

An assembly listing will be output. The file name will be "aaa.lis".

```
> asm38 aaa.mar -nolist
```

No assembly listing will be output.

3.4.2 [NO] SOURCE Source Program Listing Output Control

Format

SOURCE

NOSOURCE

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of a source program listing to the assembly listing, or the suppression of that output.

SOURCE ..... Output

NOSOURCE ..... Output suppressed

This command line option is valid when an assembly listing is output.

Relationship with Assembler Directives

SOURCE ..... Output

NOSOURCE ..... Output suppressed

Specification omitted -- .PRINT SRC ... Output

.PRINT NOSRC ..... Output suppressed

Specification omitted ..... Output

Examples

```
> asm38 aaa.mar -list -source
```

A source program listing will be output with the assembly listing.

```
> asm38 aaa.mar -list -nosource
```

No source program listing will be output with the assembly listing.

3.4.3 [NO] CROSS_REFERENCE Cross Reference Listing Output Control

Format

CROSS_REFERENCE

NOCROSS_REFERENCE

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of a cross reference listing to the assembly listing, or the suppression of that output.

CROSS_REFERENCE ..... Output

NOCROSS_REFERENCE ..... Output suppressed

This command line option is valid when an assembly listing is output.

Relationship with Assembler Directives

CROSS_REFERENCE ..... Output

NOCROSS_REFERENCE ..... Output suppressed

Specification omitted -- .PRINT CREF .. Output

.PRINT NOCREF ..... Output suppressed

Specification omitted ..... Output

Examples

```
> asm38 aaa.mar -list -cross_reference
```

A cross reference listing will be output with the assembly listing.

```
> asm38 aaa.mar -list -nocross_reference
```

No cross reference listing will be output with the assembly listing.

3.4.4 [NO] SECTION Section Information Listing Output Control

Format

SECTION

NOSECTION

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the output of a section information listing to the assembly listing, or the suppression of that output.

SECTION ..... Output

NOSECTION ..... Output suppressed

This command line option is valid when an assembly listing is output.

Relationship with Assembler Directives

SECTION ..... Output

NOSECTION ..... Output suppressed

Specification omitted -- .PRINT SCT .... Output

.PRINT NOSCT ..... Output suppressed

Specification omitted ..... Output

Examples

```
> asm38 aaa.mar -list -section
```

A section information listing will be output with the assembly listing.

```
> asm38 aaa.mar -list -nosection
```

No section information listing will be output with the assembly listing.

3.4.5 [NO]SHOW Source Program Listing Partial Output Control

Format

SHOW[=<output classifier>[,<output classifier>...]]

NOSHOW[=<output classifier>[,<output classifier>...]]

<output classifier>: {CONDITIONALS ⇓ DEFINITIONS ⇓ CALLS ⇓

EXPANSIONS ⇓ STRUCTURED ⇓ CODE}

The underlined part is the abbreviated form of this option.

When using the MS-DOS, multiple output classifiers are enclosed within parentheses.

Description

This command line option specifies the partial output of preprocessor function source statements to the source program listing, or the suppression of that output. It is also used to specify the partial output of object code display lines to the source program listing, or the suppression of that output.

SHOW .....Output

NOSHOW .....Output suppressed

The output classifiers are interpreted as follows:

CONDITIONALS .....Failed conditional expansions

DEFINITIONS .....Definitions

CALLS .....Calls

EXPANSIONS .....Expansions

STRUCTURED .....Structured assembly function expansions

CODE .....Object code display lines

More specifically these classifiers specify the following output:

Failed conditional expansions .. .AIF clauses that were not expanded.

Definitions ..... Macro definitions, .AREPEAT and .A WHILE definitions,

.INCLUDE directives, and

.ASSIGNA and .ASSIGNC directives.

Calls ..... Macro call statements,

structured assembly directives,

.AIF and .AENDI directives.

Expansions ..... Code resulting from macro expansions Code resulting from

AREPEAT and .A WHILE expansions.

Structured assembly function expansions ... Code resulting from structured assembly

expansions.

Object code display lines ..... Code sections where the bject code display for

directives exceeds the number of source

..... statement lines.

These command line options result in the inclusion (or the suppression) of the output for the output classifiers specified. When no output classifiers are specified, SHOW results in the inclusion of output for all items, and NOSHOW results in the suppression of output for all items.

This command line option is valid when a source program listing is output.

Relationship with Assembler Directives

SHOW=<output classifier> ..... Output

NOSHOW=<output classifier> ..... Output suppressed

Specification omitted -.LIST <output classifier> (output) ..... Output

.LIST <output classifier> (output suppression) ..... Output suppressed

Specification omitted ..... Output

Examples

```
>asm38 aaa.mar - list - show
```

In this example, all items will be output to the assembly listing.

```
> asm38 aaa.mar - list - noshow=expansions, structured
```

In this example, expansions and structured assembly expansions will bot be output to the assembly listing.

```
> asm38 aaa.mar - list - noshow=code
```

In this example, object code display lines exceeding the number of source statement lines will not be output to the assembly listing.

3.4.6 LINES Listing Lines/Page Specification

Format

LINES=<line count>

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the number of lines per page in the assembly listing.

A line count of between 20 and 255 lines can be specified.

This command line option is valid when an assembly listing is output.

Relationship with Assembler Directives

LINES=<line count> ..... The line count from the LINES command line option

Specification omitted .FORM LIN= <line count> .... The line count from the .FORM directive

Specification omitted ..... 60 lines

Example

```
> asm38 aaa.mar - list - lines=40
```

In this example, the assembly listing will have 40 lines per page.

3.4.7 COLUMNS Listing Columns/Line Specification

Format

COLUMNS=<column count>

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the number of columns (characters) on a line in the assembly listing.

A column count of between 79 and 255 columns can be specified.

This command line option is valid when an assembly listing is output.

Relationship with Assembler Directives

COLUMNS=<column count> ..... The column count from the

COLUMNS command line option

Specification omitted .FORM COL= <column count> ..... The column count from

the omitted .FORM directive

Specification omitted ..... 132 columns

Example

```
> asm38 aaa.mar - list - columns=80
```

In this example, the assembly listing will have 80 columns per line.

3.5 Command Line Option Concerning File Inclusion

3.5.1 INCLUDE Include File Directory Specification

Format

INCLUDE=<directory name>[,<directory name> ...]

The underlined part is the abbreviated form of this option.

Description

This command line option specifies the directory name of the file to be included.

The syntax of the directory name must conform to the standard file name syntax of the host machine.

Any number of directory names can be specified, but they must be specified within one command line.

The assembler searches the directory for the include file in the order in which they are specified in the command line. When the include file is found in a directory, the directories specified afterwards are not searched.

Relationship with Assembler Directives

INCLUDE = <directory name> ..... 1) <directory name>

specified by

.INCLUDE directive

2) <directory name>*

specified by

INCLUDE option

Specification omitted ..... <directory name>

specified by

.INCLUDE directive

Note: The directory specified with the INCLUDE option is added before the directory specified with the .INCLUDE directive.

Examples

```
> asm38 aaa.mar - include=/usr/tmp,/tmp (UNIX)
> asm38 aaa.mar - include=\usr\tmp,\tmp (MS-DOS)
```

(.INCLUDE "file.h" is specified in aaa.mar.)

In this example, file.h will be searched for in the current directory, in/usr/tmp, and in/tmp in this order.

3.6 Command Line Options Concerning Conditional Assembly

3.6.1 ASSIGNA Integer Preprocessor Variable Definition

Format

ASSIGNA=<preprocessor variable name>=<integer constant> [,<preprocessor variable name>=<integer constant>...]

The underlined part is the abbreviated form of this option.

Description

This command line option assigns an integer constant to a preprocessor variable.

The syntax of integer preprocessor variables is the same as that of symbols. However, when using UNIX, a backslash (\) must be added in front of a dollar mark ($) in a preprocessor variable name.

Integer constants are usually prefixed with a radix (B', Q', D', or H'). When omitted, the radix is assumed to be decimal. When using UNIX, a backslash (\) must be added in front of the radix apostrophe (').

Values from - 2,147,483,648 to 4,294,967,295 can be assigned for integer constants. However, negative values must not be specified in decimal (D').

Relationship with Assembler Directives

ASSIGNA=<preprocessor variable name> -.ASSIGNA*=<preprocessor variable name> =<integer constant> ..... <integer constant> specified by ASSIGNA option

Specification omitted ..... <integer constant> specified by ASSIGNA option

Specification omitted --.ASSIGNA= <preprocessor variable name>

=<integer constant> ..... <integer constant> specified by

.ASSIGNA directive

Note:

When a value is assigned to a preprocessor variable with this ASSIGNA command line option, a value assigned to the preprocessor variable with the .ASSIGNA directive becomes invalid.

Examples

```
> asm38 aaa.mar - assigna=x=0
```

In this example, 0 is assigned to preprocessor variable x. All references (\&x) to preprocessor variable x become 0 in the source program.

```
> asm38 aaa.mar - assigna=_\$=H\'FF (UNIX)
> asm38 aaa.mar - assigna=_$=H'FF (MS-DOS)
```

In this example, H'FF is assigned to preprocessor variable_$. All references (&_$) to preprocessor variable_$ become H'FF in the source program.

3.6.2 ASSIGNC Character Preprocessor Variable Definition

Format

ASSIGNC=<preprocessor variable name>="<character string>"...]

[,<preprocessor variable name>="<character string>"...]

The underlined part is the abbreviated form of this option.

Description

This command line option assigns a character string to a preprocessor variable.

The syntax of character preprocessor variables is the same as that of symbols. However, when using UNIX, a backslash (\) must be added in front of a dollar mark ($) in a preprocessor variable name.

Character strings are specified by enclosing them in double quotation marks (").

When using UNIX, a backslash (\) must be added in front of each of the following symbols in character strings. In addition, if any one of the symbols is preceded or followed by a character string, that character string must be enclosed with double quotation marks (").

- Exclamation (!)
- Double quotation (")
- Dollar ($)
- Back quotation(`)

Up to 255 characters (bytes) can be specified for a character string.

Relationship with Assembler Directives

ASSIGNC=<preprocessor variable name> ="<character string>"-- .ASSIGNC*<preprocessor variable name><character string> ..... ="<character string>" speciefed by

ASSIGNC option

Specification omitted .....<character string>specified by

ASSIGNC option

Specification omitted -- .ASSIGNC= <preprocessor. variable name>.="<character string>"..... <character string> specified by

.ASSIGNC directive

Note:

When a character string is assigned to a preprocessor variable with this ASSIGNC command line option, a character string assigned to the preprocessor variable with the .ASSIGNC directive becomes invalid.

Examples

```
> asm38 aaa.mar -assignc=X="OK"
```

In this example, character string OK is assigned to preprocessor variable x.All references (\&x) to preprocessor variable x become OK in the source program.

```
> asm38   aaa.mar -aasignc=_\$="ON"\!"OFF" (UNIX)
> asm38 aaa.mar -assignc=_$="ON!OFF" (MS-DOS)
```

In this example, character string ON!OFF is assigned to preprocessor variable _$.All references (\&_$) to preprocessor variable _$ become ON!OFF in the source program.

3.7 Command Line Options Concerning Command Line Specification

3.7.1 SUBCOMMAND Subcommand File Specification

Format

SUBCOMMAND=<subcommand file>

The underlined part is the abbreviated form of this option.

Description

This command line option specifies a subcommand file that includes an input file name and a command line option to be expanded in the command line.

The subcommand file format is as follows:

- An input file name and command line options in the same order as specified on a normal command line.
- One input file or command line option for each line.

Note:

This command line option must be specified at the end of a command line and cannot be specified in a subcommand file.

Examples

```
> asm38 aaa.mar -subcommand=aaa.sub
```

The subcommand file contents are expanded in a command line and assembled.

Contents of

```
aaa.sub
bbb.mar
-list
-noobj
```

The expanded example becomes as follows:

```
> asm38 aaa.mar, bbb.mar -list -noobj
```

Appendices

Appendix A Limitations

Table A-1 lists the limitations on source programs.

Table A-1 Source Program Limitations

Number	Item	Limitation	
1	Character set	ASCII characters only	
2	Number of characters per line	Up to 255 characters	
3	Lines assembled	Up to 65,535 lines (including lines generated by expansions)	
4	Character constants	Up to 4 characters	
5	Symbol length	Up to 32 characters	
6	Number of symbols	Up to 65,535 symbols	
7	Number of external reference symbols	Up to 65,535 symbols	
8	Number of external definition symbols	Up to 65,535 symbols	
9	Maximum section size*	H8S/2600 advanced mode	H'FFFFFFFF bytes
		H8S/2600 normal mode	H'00010000 bytes
		H8S/2600 advanced mode	H'FFFFFFFF bytes
		H8S/2000 normal mode	H'00010000 bytes
		H8/300H advanced mode	H'01000000 bytes
		H8/300H normal mode	H'00010000 bytes
		H8/300	H'0000FFFF bytes
		H8/300L	H'0000FFFF bytes
10	Number of sections	Up to 65,535 sections	
11	File inclusion	Up to 30 nesting levels	

*: Maximum section size depends on the specified address space.

Appendix B Assembly Listing

B.1 Structure of an Assembly Listing

An assembly listing has the following structure.

1. Source program listing
   Displays information related to the source program.

2. Cross reference listing
   Displays information related to the symbols in the source program.

3 Section information listing
   Displays information related to the section in the source program.

B.2 Source Program Listing

The source program listing includes the following information.

(1) The listing line number

(2) The location counter value

The location counter value is displayed as an absolute address in absolute addressing sections, and as a relative address in relative addressing sections.

(3) The object code

(4) The source line number

This is the line number of the source statement in the source program file. These line numbers are not displayed for source statements generated by the assembler.

(5) The expansion type

This specifies the type of preprocessor function source statement.

The expansion types are interpreted as follows:

I ..... File include

C ..... Expansions due to true conditional assembly conditions, iteration expansions, and conditional iteration expansions

M ..... Macro expansions

S ..... Structured assembly expansions

The inclusion nesting level is also displayed for expansion type I.

(6) The source statement

```
*** H8S, H8/300 ASSEMBLER Ver. 1.0 *** 06/10/94 17:15:00 PAGE    1
PROGRAM NAME =
1                                   1               .CPU    2600A
2                                   2       ;
3   00000000                        3               .SECTION AAA, CODE, ALIGN=2
4   00000000                        4       START
5   00000000    7A0700000000        5               MOV.L   #STACK:32,SP
6   00000006    F800                6               MOV.B   #0:8,R0L
7   00000008    6AA800000000        7               MOV.B   R0L,@ANS:32
8   0000000E    7A0200001000        8               MOV.L   #DATA:32,ER2
9                                   9               .FOR.B(R1L=#1,#8,+#1)
10  00000014    F901                        S                       MOV     #1,R1L
11  00000016    5800000A                    S                       BRA
    _$F00002
12      0000001A                        S               _$F00000:       .EQU $
13  0000001A    6828                10                  MOV.B   @ER2,R0L
14  0000001C    0B02                11                  ADDS.L  #1,ER2
15  0000001E    5E000000            12                  JSR     @CHANGE:24
16                                  13              .ENDF
17      00000022                        S               _$F00001:       .EQU $
18  00000022    8901                        S                       ADD     #1,R1L
19      00000024                        S               _$F00002:       EQU $
20  00000024    A908                        S                       CMP     #8,R1L
21  00000026    4FF2                        S                       BLE
    _$F00000
22      00000028                        S               _$F00003:       .EQU $
23  00000028    0180                14              SLEEP
24  0000002A    40D4                15              BRA     START
25                                  16      ;
26  0000002C                        17      CHANGE
27  0000002C    6A2900000000        18              MOV.B   @ANS:32, R1L
28                                  19              .IF.B   (R1L<LT>R0L)
29  00000032    1C98                        S                       CMP  R1L, R0L
30  00000034    58F00006                    S                       BLE  _$I00000
31  00000038    6AA800000000        20                  MOV.B   R0L, @ANS:32
32                                  21              .ENDI
33      0000003E                        S               _$I00000:       .EQU $
34      0000003E                        S               _$I00001:       .EQU $
```

```
35  0000003E   5470              22            RTS
36                               23     ;
37  00001000                     24     .SECTION BBB, DATA, LOCATE=H'00001000
38  00001000                     25     DATA
39  00001000   03020405          26            .DATA.B H'03, H'02, H'04, H'05
40  00001004   01080607          27            .DATA.B H'01, H'08, H'06, H'07
41                               28     ;
42  00000000                     29            .SECTION CCC, DATA, ALIGN=2
43  00000000                     30     ANS
44  00000000   00000001          31            .RES.B  1
45                               32     ;
46  00000000                     33            .SECTION DDD, STACK, ALIGN=2
47  00000000   00000500          34            .RES.B  H'500
48  00000500                     35     STACK
49                               36     ;
50             00000000          37            .END    START
(1) (2)        (3)          (4)          (5)           (6)
*****TOTAL ERRORS         0
*****TOTAL WARNINGS       0
```

Figure B-1 Source Program Listing Example

B.3 Cross Reference Listing

The cross reference listing contains the following information.

(1) The symbol name

(2) The section name

This column specifies the name of a section consisting of various symbols. Up to 8 characters can be displayed.

(3) The symbol attribute

This column specifies the symbol attribute. The entries are interpreted as follows.

No entry ..... Symbol defined as a label

EQU ..... Symbol defined with the .EQU directive

ASGN ..... Symbol defined with the .ASSIGN directive

IMPT ..... External definition symbol

EXPT ..... External definition symbol

SCT ..... Section name

REG ..... Symbol defined with the .REG directive

MDEF ..... Multiply defined symbol

UDEF ..... Undefined symbol

(4) The symbol's value

This entry indicates the symbol's value, displayed as an 8-digit hexadecimal number.

(5) ymbol definition and reference line numbers

This entry displays the listing line numbers where the symbol is defined and referenced. The definition line is marked with an asterisk (*).

```
*** H8S, H8/300 ASSEMBLER VER. 1.0 ***     06\10\94 17:15:00 PAGE    2
*** CROSS REFERENCE LIST
NAME               SECTION   ATTR    VALUE              SEQUENCE
AAA                    AAA   SCT     00000000       3*
ANS                    CCC           00000000       7       27      31
                                                     43*
BBB                    BBB   SCT     00001000       37*
CCC                    CCC   SCT     00000000       42*
CHANGE                 AAA           0000002C       15      26*
DATA                   BBB           00001000       8       38*
DDD                    DDD   SCT     00000000       46*
STACK                  DDD           00000500       5       48*
START                  AAA           00000000       4*      24      50
_$F00000               AAA   EQU     0000001A       12*     21
_$F00001               AAA   EQU     00000022       17*
_$F00002               AAA   EQU     00000024       11      19*
_$F00003               AAA   EQU     00000028       22*
_$I00000               AAA   EQU     0000003E       30      33*
_$I00001               AAA   EQU     0000003E       34*
   (1)                 (2)   (3)     (4)            (5)
```

Figure B-2 Cross Reference Listing Example

B.4 Section Information Listing

The section information listing displays the following information.

(1) The section name

(2) The section attribute

This entry indicates the section attribute. The format classifier and section attribute are displayed.

a. Format classifier

ABS ..... Absolute addressing format

REL ..... Relative addressing format

b. Section attribute

CODE.....Code section

DATA.....Data section

STACK.....Stack section

COMMON.....Common section

DUMMY.....Dummy section

(3) Section size

This entry indicates the section size as a hexadecimal number.

(4) Section starting address

This entry displays the starting address for an absolute addressing section. Nothing is displayed for relative addressing section.

```
*** H8S, H8/300 ASSEMBLER Ver. 1.0 *** 06/10/94 17:15:00 PAGE 3
*** SECTION DATA LIST
SECTION   ATTRIBUTE SIZE START
AAA    REL-CODE  0000040
BBB    ABS-DATA  0000008 001000
CCC    REL-DATA  0000001
DDD    REL-STACK 000500
(1)    (2)       (3)     (4)
```

Figure B-3 Section information Listing Example

Appendix C Error Messages

C.1 Error Message Types

There are three types of error messages.

1. Invocation errors

   These are errors on the command line that was used to invoke the assembler.

   These errors include file specification errors and command line option errors.

   The source program is not assembled when a command line error occurs.

   The display format is as follows:

   — MS-DOS

   <file name> <line number> <message>

   — UNIX

   "<file name>", line <line number> : <message>

   Invocation error numbers range from 0 to 99.

2. Source program error

   These are syntax errors in the source program, and are displayed immediately following the source statement that caused the error.The display format is as follows:

- MS-DOS

  Screen display ..... <file name> <line number> <message>

  Assembly listing ..... *****ERROR <error number> (<file name>)

  *****WARNING <error number> (<file name>)

  — UNIX

  Screen display ..... "<file name>", line <line number> : <message>

  Assembly ..... *****ERROR <error number> (<file name>)

  *****WARNING <error number> (<file name>)

  Source program error numbers are classified as follows.

  1 xx errors ..... General source program syntax errors

  2 xx errors ..... Symbol errors

  3 xx errors ..... Operation and operand errors

  4 xx errors ..... Expression errors

  5 xx errors ..... Assembler directive errors

  6 xx errors ..... Preprocessor directive errors

  8 xx errors ..... General source program warnings

3. Fatal errors

   These are errors related to the assembler operating environment, and include such situations as insufficient memory for the source program size and insufficient disk capacity.

Source program assembly is terminated on the occurrence of a fatal error.
The display format is as follows:
— MS-DOS
<file name><line number><message>
— UNIX
"<file name>", line <line number> : <message>
Fatal error numbers are in the nine hundreds.

C.2 Invocation Errors

Table C-1 lists invocation errors and recommended corrective actions.

Table C-1 Invocation Errors

10	Message: Meaning: Corrective action:	NO INPUT FILE SPECIFIED No input file specification was provided. Specify an input file.
20	Message: Meaning: Corrective action:	CANNOT OPEN FILE <file name> The assembler could not open the file as specified. Reexamine the file name and directory path.
30	Message: Meaning: Corrective action:	INVALID COMMAND PARAMETER There was an error in a command line option or subcommand file contents. Reexamine the command line options and subcommand file contents.
40	Message: Meaning: Corrective action:	CANNOT ALLOCATE MEMORY The assembler ran out of memory while processing the input file. This error should not occur in normal operation. Please contact your Hitachi sales or technical representative.
50	Message: Meaning: Corrective action: Supplement:	COMPLETED FILE NAME TOO LONG <file name> The file name including the directory was too long. Simplify the directory structure. This message is handled as a warning, and assembly will continue. It is possible that the simulator debugger may not operate correctly following the appearance of this message.

C.3 Source Program Errors

Table C-2 lists source program errors and recommended corrective actions.

Table C-2 Source Program Errors (1)

100	Message: Meaning: Corrective action:	OPERATION TOO COMPLEX An expression was too complex. Simplify the expression.
101	Message: Meaning: Corrective action:	SYNTAX ERROR IN SOURCE STATEMENT Source statement syntax error. Reexamine the whole source statement.
102	Message: Meaning: Corrective action:	SYNTAX ERROR IN DIRECTIVE Source statement syntax error. Reexamine the whole source statement.
103	Message: Meaning: Corrective action:	.END NOT FOUND No. END statement. Add an .END statement to the program.
104	Message: Meaning: Corrective action:	LOCATION COUNTER OVERFLOW The location counter value (address) exceeded the maximum value. Reduce the size of the program.
105	Message: Meaning: Corrective action:	ILLEGAL INSTRUCTION IN STACK SECTION An instruction that cannot be used in a stack section was specified. Remove the instruction.
106	Message: Meaning: Corrective action:	TOO MANY ERRORS Number of errors is too large: message display terminated. Reexamine all source statements.
108	Message: Meaning: Corrective action:	ILLEGAL CONTINUATION LINE Illegal continuation line is specified. Correct the syntax of the continuation line.
109	Message: Meaning: Corrective action:	LINE NUMBER OVERFLOW The number of statements in the program exceeded 65,535. Divide the program into smaller sections.
200	Message: Meaning: Corrective action:	UNDEFINED SYMBOL REFERENCE An undefined symbol was referenced. Define the symbol.
201	Message: Meaning: Corrective action:	ILLEGAL SYMBOL OR SECTION NAME A register mnemonic, operator, or the location counter symbol was used as a symbol or as a section name. Correct the symbol or section name.

Table C-2 Source Program Errors(2)

202	Message: Meaning: Corrective action:	ILLEGAL SYMBOL OR SECTION NAME Error in a symbol or a section name. Correct the symbol or section name.
300	Message: Meaning: Corrective action:	ILLEGAL MNEMONIC Error in an instruction mnemonic. Correct the instruction mnemonic.
301	Message: Meaning: Corrective action:	TOO MANY OPERANDS OR ILLEGAL COMMENT Too many operands to an instruction, or comment syntax error. Correct the operands or the comment.
304	Message: Meaning: Corrective action:	LACKING OPERANDS Too few operands specified for an instruction. Correct the operands.
306	Message: Meaning: Corrective action:	SYNTAX ERROR IN REGISTER LIST Error specifying multiple registers. Specify the register list correctly.
307	Message: Meaning: Corrective action:	ILLEGAL ADDRESSING MODE An addressing mode not allowed for the instruction operand was specified. Correct the operand's addressing mode.
308	Message: Meaning: Corrective action:	SYNTAX ERROR IN OPERAND Instruction operand syntax error. Correct the operand.
400	Message: Meaning: Corrective action:	CHARACTER CONSTANT TOO LONG More than four characters were specified in a character constant. Correct the character constant.
402	Message: Meaning: Corrective action: Supplement: of range portion.	ILLEGAL VALUE IN OPERAND Out of range value specified in an instruction operand. Correct the value. Object code will be generated for the operand except for the out
403	Message: Meaning: Corrective action:	ILLEGAL OPERATION FOR RELATIVE VALUE Multiplication or logical operation applied to relative value. Correct the expression.

Table C-2 Source Errors(3)

404 .	Message: Meaning: Corrective action:	ILLEGAL IMMEDIATE DATA Relative value specified where immediate data #1, #2, #4, #0 to 3, or #0 to 7 is required Relative values cannot be specified. Correct the value.
407	Message: Meaning: Corrective action:	MEMORY OVERFLOW Memory overflow during expression evaluation. Simplify the expression.
408	Message: Meaning: Corrective action:	DIVISION BY ZERO Divide by 0. Correct the expression.
409	Message: Meaning: Corrective action:	REGISTER IN EXPRESSION Register was specified in an expression. Correct the expression.
411	Message: Meaning: Corrective action:	INVALID START OF/SIZE OF OPERAND START OF or SIZE OF operator was applied to an item other than a section name Correct the expression.
500	Message: Meaning: Corrective action:	SYMBOL NOT FOUND No symbol specified in label field of directive requiring a symbol. Specify in the label field.
501	Message: Meaning: Corrective action:	ILLEGAL ADDRESS VALUE IN OPERAND Out of range value was specified in. SECTION or, .ORG statement. Correct the value.
502	Message: Meaning: Corrective action:	ILLEGAL SYMBOL IN OPERAND Relative or forward reference value specified where a backward reference absolute value is required. Correct the value.
503	Message: Meaning: Corrective action:	UNDEFINED EXPORT SYMBOL Symbol defined externally with .EXPORT not defined within program. Remove the symbol from the .EXPORT statement.
504	Message: Meaning: Corrective action:	INVALID RELATIVE SYMBOL IN OPERAND External reference value or forward reference value specified in .EQU or .ASSIGN statement, or relative value which cannot be expressed as the offset from a section starting address specified. Correct the value.

Table C-2 Source Program Errors(4)

505	Message:	ILLEGAL OPERAND
	Meaning:	Error in directive operand.
	Corrective action:	Correct the operand.
506	Message:	ILLEGAL OPERAND
	Meaning:	Error in directive operand.
	Corrective action:	Correct the operand.
508	Message:	ILLEGAL VALUE IN OPERAND
	Meaning:	Out of range value in directive operand.
	Corrective action:	Correct the value.
510	Message:	ILLEGAL BOUNDARY VALUE
	Meaning:	Error in boundary adjustment value in .SECTION or .ALIGN statement.
	Corrective action:	Correct the boundary alignment value.
511	Message:	ILLEGAL DISPLACEMENT SIZE
	Meaning:	Error in .DISPSIZE bit count.
	Corrective action:	Correct the bit count.
512	Message:	ILLEGAL EXECUTION START ADDRESS
	Meaning:	Error in .END execution start address.
	Corrective action:	Correct the execution start address.
513	Message:	ILLEGAL REGISTER NAME
	Meaning:	Error in .REG register name.
	Corrective action:	Correct the register name.
514	Message:	INVALID EXPORT SYMBOL
	Meaning:	Symbol that cannot be defined externally specified in .EXPORT or GLOBAL statement.
	Corrective action:	Remove the symbol from the .EXPORT or .GLOBAL statement.
516	Message:	EXCLUSIVE DIRECTIVES
	Meaning:	Inconsistency in directive specification.
	Corrective action:	Reexamine the related directives.
517	Message:	INVALID VALUE IN OPERAND
	Meaning:	External reference value or forward reference value specified in .ORG directive, or relative address value from different section specified.
	Corrective action:	Correct the value.

Table C-2 Source Program Errors (5)

518	Message:	INVALID IMPORT SYMBOL
	Meaning:	Symbol defined within the program specified in .IMPORT statement.
	Corrective action:	Remove the symbol from the external reference specification.
520	Message:	ILLEGAL .CPU DIRECTIVE POSITION
	Meaning:	.CPU directive specified at point other than start of program, or multiple. CPU directives.
	Corrective action:	Move the .CPU directive to the start of the program and confirm that there is only one .CPU directive in the program.
521	Message:	ILLEGAL SYMBOL IN OPERAND
	Meaning:	A location counter value or a symbol whose value is an address specified as a constant value operand while in the OPTIMIZE option was specified.
	Corrective action:	Correct the value.
	Supplement:	Specify the NOOPTIMIZE option when using a location counter value or a symbol whose value is an address.
523	Message:	ILLEGAL OPERAND
	Meaning:	An Illegal operand in directive
	Corrective action:	Correct the illegal operand in the directive
524	Message:	ILLEGAL ADDRESSING SPACE SIZE
	Meaning:	An illegal bit width for address space was specified in the .CPU directive operand.
	Corrective action:	Correct the bit width for address space.
525	Message:	ILLEGAL .LINE DIRECTIVE POSITION
	Meaning:	.LINE directive was specified in macro expansion or conditional iterated expansion
	Corrective action:	Remove the .LINE directive
601	Message:	INVALID DELIMITER
	Meaning:	Delimiter character error.
	Corrective action:	Reexamine the whole source statement.
602	Message:	INVALID CHARACTER STRING FORMAT
	Meaning:	Character string error.
	Corrective action:	Reexamine the whole source statement.
603	Message:	SYNTAX ERROR IN SOURCE STATEMENT
	Meaning:	Source statement syntax error.
	Corrective action:	Reexamine the whole source statement.

Table C-2 Source Errors (6)

604	Message:	ILLEGAL SYMBOL IN OPERAND
	Meaning:	Undefined symbol, relative value, or forward reference value specified for operand requiring a backward reference absolute value.
	Corrective action:	Correct the value.
610	Message:	MULTIPLE MACRO NAMES
	Meaning:	Macro name reused in macro definition (.MACRO directive).
	Corrective action:	Correct the macro name.
611	Message:	MACRO NAME NOT FOUND
	Meaning:	Macro name not specified (.MACRO directive).
	Corrective action:	Specify a macro name in the name field of the .MACRO directive.
612	Message:	ILLEGAL MACRO NAME
	Meaning:	Error in macro name (.MACRO directive).
	Corrective action:	Correct the macro name.
	Supplement:	Executable instruction mnemonics, assembler directives (with period removed), and preprocessor directives (with period removed) cannot be used as macro names.
613	Message:	ILLEGAL .MACRO DIRECTIVE POSITION
	Meaning:	.MACRO directive appears in macro body (between .MACRO and .ENDM directives), between .AREPEAT and .AENDR directives, or between .AWHILE and .AENDW directives.
	Corrective action:	Remove the .MACRO directive.
614	Message:	MULTIPLE MACRO PARAMETERS
	Meaning:	Formal parameter repeated in macro definition (.MACRO directives) formal parameter definitions.
	Corrective action:	Correct the formal parameters.
615	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive appears in macro body (between .MACRO and .ENDM directives).
	Corrective action:	Remove the .END directive.
616	Message:	MACRO DIRECTIVES MISMATCH
	Meaning:	An .ENDM directive appeared without preceding .MACRO directive, or an .EXITM directive appeared outside of a macro body (between .MACRO and .ENDM directives), outside of .AREPEAT and .AENDR directives), or outside of .AWHILE and .AENDW directives.
	Corrective action:	Remove the .ENDM or .EXITM directives.

Table C-2 Source Program Error (7)

618	Message:	MACRO EXPANSION TOO LONG
	Meaning:	Macro expansion generated line with over 255 characters.
	Corrective action:	Correct the definition or call so that the line is less than 255 characters.
619	Message:	ILLEGAL MACRO PARAMETER
	Meaning:	Macro parameter formal parameter name error in macro call, or error in formal parameter in a macro body (between .MACRO and .ENDM directives).
	Corrective action:	Correct the formal parameter.
	Supplement:	When there is an error in a formal parameter in a macro body, the error will be discovered and flagged during macro expansion.
620	Message:	UNDEFINED PREPROCESSOR VARIABLE
	Meaning:	Reference to an undefined preprocessor variable.
	Corrective action:	Define the preprocessor variable.
621	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive in macro expansion.
	Corrective action:	Remove the .END directive.
622	Message:	`)' NOT FOUND
	Meaning:	Matching parenthesis missing in macro processing exclusion.
	Corrective action:	Supply the missing macro processing exclusion parenthesis.
623	Message:	SYNTAX ERROR IN STRING FUNCTION
	Meaning:	Syntax error in character string manipulation function.
	Corrective action:	Correct the character string manipulation function.
624	Message:	MACRO PARAMETERS MISMATCH
	Meaning:	Too many positional specification macro parameters in macro call.
	Corrective action:	Correct the number of macro parameters.
630	Message:	SYNTAX ERROR IN OPERAND
	Meaning:	Syntax error in structured assembly directive operand.
	Corrective action:	Reexamine the directive.
631	Message:	END DIRECTIVE MISMATCH
	Meaning:	Terminating preprocessor directive does not agree with matching directive.
	Corrective action:	Reexamine the preprocessor directives.
632	Message:	SYNTAX ERROR IN OPERAND
	Meaning:	Error in structured assembly directive operand condition code.
	Corrective action:	Correct the condition code.

Table C-2 Source Program Errors (8)

633	Message:	ILLEGAL .BREAK OR .CONTINUE DIRECTIVE POSITION
	Meaning:	.BREAK or .CONTINUE directive outside .FOR[U] and .ENDF, .WHILE and .ENDW, or .REPEAT and .UNTIL directive ranges.
	Corrective action:	Remove the .BREAK or .CONTINUE directive.
634	Message:	EXPANSION TOO LONG
	Meaning:	Line over 255 characters generated by structured assembly expansion.
	Corrective action:	Correct the structured assembly expansion to generate lines of less than 255 characters.
640	Message:	SYNTAX ERROR IN OPERAND
	Meaning:	Syntax error in conditional assembly directive operand.
	Corrective action:	Reexamine the whole source statement.
641	Message:	INVALID RELATIONAL OPERATOR
	Meaning:	Error in conditional assembly directive relational operator.
	Corrective action:	Correct the relational operator.
642	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive between .AREPEAT and .AENDR directives or between .AWHILE and .AENDW directives.
	Corrective action:	Remove the .END directive.
643	Message:	DIRECTIVE MISMATCH
	Meaning:	.AENDR or .AENDW directive does not form proper pair with .AREPEAT or .AWHILE directive.
	Corrective action:	Reexamine the preprocessor directives.
644	Message:	ILLEGAL .AENDW OR .AENDR DIRECTIVE POSITION
	Meaning:	.AENDW or .AENDR directive appears between .AIF and .AENDI directives.
	Corrective action:	Remove the .AENDW or .AENDR directive.
645	Message:	EXPANSION TOO LONG
	Meaning:	Line over 255 characters generated by .AREPEAT or .AWHILE expansion.
	Corrective action:	Correct the .AREPEAT or .AWHILE to generate lines of less than 255 characters.
650	Message:	INVALID INCLUDE FILE
	Meaning:	Error in .INCLUDE file name.
	Corrective action:	Correct the file name.
651	Message:	CANNOT OPEN INCLUDE FILE
	Meaning:	Could not open .INCLUDE file.
	Corrective action:	Correct the file name.

Table C-2 Source Program Errors (9)

652	Message:	INCLUDE NEST TOO DEEP
	Meaning:	File inclusion nesting exceeded 30 levels.
	Corrective action:	Limit the nesting to 30 or fewer levels.
653	Message:	SYNTAX ERROR IN OPERAND
	Meaning:	Syntax error in .INCLUDE operand.
	Corrective action:	Correct the operand.
660	Message:	.ENDM NOT FOUND
	Meaning:	Missing .ENDM directive following .MACRO.
	Corrective action:	Insert an .ENDM directive.
661	Message:	END DIRECTIVE NOT FOUND
	Meaning:	Terminating directives insufficient for specified structured assembly directives.
	Corrective action:	Insert the required terminating directives.
662	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive appears between .AIF and .AENDI directives.
	Corrective action:	Remove the .END directive.
663	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive appears in .INCLUDE file.
	Corrective action:	Remove the .END directive.
664	Message:	ILLEGAL .END DIRECTIVE POSITION
	Meaning:	.END directive appears between .AIF and .AENDI directives.
	Corrective action:	Remove the .END directive.
665	Message:	ILLEGAL SYMBOL IN OPERAND
	Meaning:	A symbol which is not a preprocessor variable is specified in a preprocessor directive while the OPTIMIZE option is specified.
	Corrective action:	Correct the specification.
	Supplement:	Use the NOOPTIMIZE option when specifying a symbol which is not a preprocessor variable.
800	Message:	SYMBOL NAME TOO LONG
	Meaning:	Symbol exceeds 32 characters.
	Corrective action:	Correct the symbol.
	Supplement:	Only the first 32 characters are valid in a symbol.
801	Message:	MULTIPLE SYMBOLS
	Meaning:	Repeated symbol in symbol definition position.
	Corrective action:	Correct the symbol.

Table C-2 Source Program Errors (10)

805	Message:	ILLEGAL OPERATION SIZE
	Meaning:	Error in branch size specification (:8 or :16) in structured assembly statement.
	Corrective action:	Correct the branch size.
807	Message:	ILLEGAL OPERATION SIZE
	Meaning:	Error in instruction size specification.
	Corrective action:	Correct the size specification.
808	Message:	ILLEGAL CONSTANT SIZE
	Meaning:	Error in integer constant interpretation size (.xx).
	Corrective action:	Correct the interpretation size.
	Supplement:	The interpretation size can be either byte (.B) or word (.W). The value is interpreted as a 1 or 2 byte signed quantity, respectively.
810	Message:	TOO MANY OPERANDS
	Meaning:	Too many instruction operands.
	Corrective action:	Correct the operands.
811	Message:	ILLEGAL SYMBOL DEFINITION
	Meaning:	Symbol specified in label field.
	Corrective action:	Remove the symbol.
812	Message:	SECTION OR MODULE NAME TOO LONG
	Meaning:	Section or module name exceeds 32 characters.
	Corrective action:	Correct the section or module name.
	Supplement:	Only the first 32 characters are valid.
813	Message:	SECTION ATTRIBUTE MISMATCH
	Meaning:	Different section attribute or format classifier specified on section re-entry.
	Corrective action:	Correct the section attribute or format classifier.
	Supplement:	The "Locate=<starting address>" specification is invalid for reentry to an absolute section.
814	Message:	ILLEGAL OBJECT CODE SIZE
	Meaning:	Error in allocation size (:8, :16, :24 or :32).
	Corrective action:	Correct the allocation size.
	Supplement:	Although notations of the form "#xx:2" and "#xx;3" are used in the manuals, these notations cannot be used with the assembler.
815	Message:	MULTIPLE MODULE NAMES
	Meaning:	Multiple module name specifications (.PROGRAM).
	Corrective action:	Specify the module name exactly once.

Table C-2 Source Program Errors (11)

816	Message:	START ODD ADDRESS
	Meaning:	Even-sized byte data or data region allocated from an odd address.
	Corrective action:	Correct the address specification to be an even number.
817	Message:	OPERATION SIZE MISMATCH
	Meaning:	Either an @-SP or an @SP + addressing mode specified for byte size instruction (.B).
	Corrective action:	This instruction form cannot be used, since the stack pointer would take on an odd value.
	Supplement:	Object code will be generated ignoring the @-SP or @SP+ addressing mode specification.
825	Message:	ILLEGAL INSTRUCTION IN DUMMY SECTION
	Meaning:	Instruction illegal in dummy section specified in dummy section.
	Corrective action:	Remove the instruction.
830	Message:	OPERATION SIZE MISMATCH
	Meaning:	ERn or Rn specified in byte size (.B) instruction, or ERn specified in word size (.W) instruction.
	Corrective action:	Correct the register specification.
	Supplement:	Code will be generated with RnL for the byte size case, and Rn for the word size case.
832	Message:	MULTIPLE `P' DEFINITIONS
	Meaning:	P was defined as a symbol before the default section P was referenced.
	Corrective action:	Do not use P as a symbol in programs that use P for the default section.
835	Message;	ILLEGAL VALUE IN OPERAND
	Meaning:	Out of range value in instruction operand.
	Corrective action:	Correct the value.
	Supplement:	Object code will be generated with the out of range part of the value ignored.
836	Message:	CONSTANT SIZE OVERFLOW
	Meaning:	Out of range value for specified interpretation size (.B or .W) in integer constant.
	Corrective action:	Correct the value.
	Supplement:	The interpretation size can be either byte (.B) or word (.W). The value is interpreted as a 1-or 2-byte signed quantity, respectively.
837	Message:	SOURCE STATEMENT TOO LONG
	Meaning:	Source statement too long (over 255 characters).
	Corrective action:	Rewrite the source statement to within 255 characters.

Table C-2 Source Program Errors (12)

850	Message:	ILLEGAL SYMBOL DEFINITION
	Meaning:	Symbol specified in preprocessor directive label field.
	Corrective action:	Remove the symbol.
851	Message:	MACRO SERIAL NUMBER OVERFLOW
	Meaning:	Macro generation counter exceeded 99999.
	Corrective action:	Reduce the number of macro calls.
852	Message:	UNNECESSARY CHARACTER
	Meaning:	Characters appear after the end of the operands in a preprocessor directive.
	Corrective action:	Correct the operand(s).
853	Message:	NEGATIVE IMMEDIATE VALUE
	Meaning:	Negative immediate value (#-xx) specified as .FOR[U] directive increment value.
	Corrective action:	Correct the specification to the -#xx format.
	Supplement:	The .FOR[U] loop will be expanded as such.
854	Message:	.AWHILE ABORTED BY .ALIMIT
	Meaning:	Expansion was terminated because the maximum number of iterated expansion specified by the .ALIMIT directive was exceeded.
	Corrective action:	Check the iterated expansion condition.

C.4 Fatal Errors

Table C-3 lists the fatal errors and recommended corrective actions.

Table C-3 Fatal Errors (1)

900	Message:	FILE NAME TOO LONG
	Meaning:	The file name including the directory was too long.
	Corrective action:	Simplify the directory structure.
901	Message:	SOURCE FILE INPUT ERROR
	Meaning:	Input error reading source file.
	Corrective action:	Check the amount of free disk space.
902	Message:	MEMORY OVERFLOW
	Meaning:	Memory overflow.
	Corrective action:	Divide the program into smaller modules.
903	Message:	LISTING FILE OUTPUT ERROR
	Meaning:	Output error writing listing file.
	Corrective action:	Check the amount of free disk space.
904	Message:	OBJECT FILE OUTPUT ERROR
	Meaning:	Output error writing object file.
	Corrective action:	Check the amount of free disk space.
905	Message:	MEMORY OVERFLOW
	Meaning:	Memory overflow.
	Corrective action:	Divide the program into smaller modules.
906	Message:	MEMORY OVERFLOW
	Meaning:	Memory overflow.
	Corrective action:	Divide the program into smaller modules.
907	Message:	MEMORY OVERFLOW
	Meaning:	Memory overflow.
	Corrective action:	Divide the program into smaller modules.
908	Message:	SECTION OVERFLOW
	Meaning:	The number of section exceeded 65,535.
	Corrective action:	Divide the program into smaller modules.
909	Message:	SYMBOL OVERFLOW
	Meaning:	The number of symbols exceeded 65,535.
	Corrective action:	Divide the program into smaller modules.

Table C-3 Fatal Errors (2)

910	Message:	SOURCE LINE NUMBER OVERFLOW
	Meaning:	The number of lines in the assembly listing exceeded 65,535.
	Corrective action:	Divide the program into smaller modules.
911	Message:	IMPORT SYMBOL OVERFLOW
	Meaning:	The number of external reference symbols exceeds 65,535.
	Corrective action:	Reduce the number of external reference symbols.
912	Message:	EXPORT SYMBOL OVERFLOW
	Meaning:	The number of external reference symbols exceeds 65,535.
	Corrective action:	Reduce the number of external definition symbols.
933	Message:	LACKING CPU SPECIFICATION
	Meaning:	CPU classification not specified.
	Corrective action:	Specify the CPU classification with the CPU command line option, .CPU directive, or H38CPU environmental variable.
935	Message:	SUBCOMMAND FILE INPUT ERROR
	Meaning:	Input error in subcommand file.
	Corrective action:	Check if disk has sufficient capacity.
954	Message:	MEMORY OVERFLOW
	Meaning:	Not enough memory space.
	Corrective action:	Divide the source program.

Please contact your Hitachi sales or technical representative if a fatal error cannot be avoided with the recommended measures.

Appendix D Differences with the Previous Version

This chapter describes the differences between the new version of this cross assembler (H8S, H8/300 Series Cross Assembler Ver. 1) and the previous version (H8/300 Series Cross Assembler Ver.3).

D.1 CPU

In addition to the function of assembling source programs for the H8/300H series, H8/300 series, and H8/300L series CPUs, the new version of this assembler also assembles programs for the H8S/2600 series and H8S/2000 series CPUs.

The CPU classification is specified with the CPU command line option or the .CPU directive in the previous version. In the new version, it can also be specified with the H38CPU environmental variable.

Because the H38CPU environmental variable in the assembler is the same as that in the C compiler and simulator/debugger, it can be shared by various development tools.

The priority taken when different CPU values are specified with the CPU command line option, .CPU directive, and H38CPU environmental variable, is as follows:

1 CPU command line option

2. .CPU directive

3. H38CPU environmental variable

Note: In the previous version, source program assembly defaults to the H8/300H advanced mode CPU when the CPU specification is omitted. However, in the new version, an error occurs; the CPU classification must be specified with the CPU command line option, CPU directive, or H38CPU environmental variable.

D.2 Address Space Support

This new version supports assemble functions for the 1M mode and 16M mode in the H8/300H advanced mode, as well as for the 1M mode, 16M mode, 256M mode, and 4G mode in the H8S/2600 and H8S/2000 advanced modes of the new CPU classifications.

The mode is specified as follows:

- For 1M mode, specify: 20 for address space bit width at CPU classification specification.
- For 16M mode, specify :24 for address space bit width at CPU classification specification.
- For 256M mode, specify :28 for address space bit width at CPU classification specification.
- For 4G mode, specify :32 for address space bit width at CPU classification specification.

In the new version, the accessible address space differs according to the specified bit width of the address space.

Accessible address spaces are listed in table D-1.

Table D-1 Accessible Address Spaces (1)

CPU Valid in Previous Version	Address Space	Absolute Addressing Mode Range @aa:8)	Absolute Addressing Mode Range @aa:16)
300	H'0-H'FFFF	H'FF00-H'FFFF	H'0-H'FFFF
300L	H'0-H'FFFF	H'FF00-H'FFFF	H'0-H'FFFF
300HN	H'0-H'FFFF	H'FF00-H'FFFF	H'0-H'FFFF
300HA*	H'0-H'FFFFFF	H'FFFF00-H'FFFFFF	H'0-H'7FFF, H'FF8000-H'FFFFFF

Note: The default setting of address space bit width is the same as that of 300HA:24.

Table D-1 Accessible Address Spaces (2)

CPU Valid in		Absolute Addressing Mode	Absolute Addressing
New Version	**Address Space**	**Range (@aa:8)**	**Mode Range (@aa:16)**
300HA:20	H'0-H'FFFFF	H'FFF00-H'FFFFF	H'0-H'7FFF, H'F8000-H'FFFFF
300HA:24	H'0-H'FFFFFF	H'FFFF00-H'FFFFFF	H'0-H'7FFF, H'FF8000-H'FFFFFF
2600N,	H'0-H'FFFF	H'FF00-H'FFFF	H'0-H'FFFF
2000N			
2600A:20,	H'0-H'FFFFF	H'FFF00-H'FFFFF	H'0-H'7FFF,
2000A;20			H'F8000-H'FFFFF
2600A:24,	H'0-H'FFFFF	H'FFF00-H'FFFFF	H'O-H'7FF,
2000A:24			H'F8000-FFFFF
2600a:28,	H'0-H'FFFFFF	H'FFFF00-H'FFFFFFF	H'0-H'7FFF,
2000A:28			H'FFF8000-H'FFFFFFFF
2600A:32,	H'0-H'FFFFF	H'FFF00-H'FFFFF	H'O-H'7FF,
2000A32			H'F8000-FFFFF
2600A,	H'0-H'FFFFFFFF	H'FFFFFF00-H'FFFFFFFF	H'0-H'7FFF,
2000A*			H'FFFF8000-H'FFFFFFFF

Note: The default setting of address space bit width is the same as that of 2600A:32 and 2000A:32.

D.3 Source File Extension

While the previous version assumes a source file extension of .mar, the new version uses .mar or .src as the default. In other words, the new version of the assembler uses .mar or .src as the extension when the extension is omitted from the source file specification, in the priority of .mar first and .src second.

Example

```
> asm 38 aaa
        When file aaa.mar exists, file aaa.mar is assembled.
        When file aaa.mar does not exist, file aaa.src is assembled.
```

D.4 Added Assembler and Preprocesser Directives

The new assembler and assembly directives in the new version are listed in table D-2.

Table D-2 Added Assembler and Preprocesser Directives

Assembler or Preprocesser Directive	Function
.LINE	Changes the file name and line number of debugging information
.ALIMIT	Sets conditional iterated expansion limit

D.5 Added Command Line Options

The new command line options in the new version are listed in table D-3.

Table D-3 Added Command Line Options

Command Line Option	Function
INCLUDE	Include file directory specification
ASSIGNA	Integer preprocessor variable definition
ASSIGNC	Character preprocessor variable definition
[NO]OPTIMIZE	Optimization specification
[NO]EXCLUDE	Output control of information on unreferenced external reference symbols
ABORT	Change of values returned to OS and object output control at error generation
SUBCOMMAND	Subcommand file specification

D.6 Compatibility of Object Modules

Object modules generated in the new version are different from those in the previous version. When linking object modules generated by the new version with those generated by the previous version, the output format of object modules generated by either the new or previous version must be converted by the SYSROF file converter included in the assembler system.

Figure D-1 shows how compatibility can be achieved between object modules.

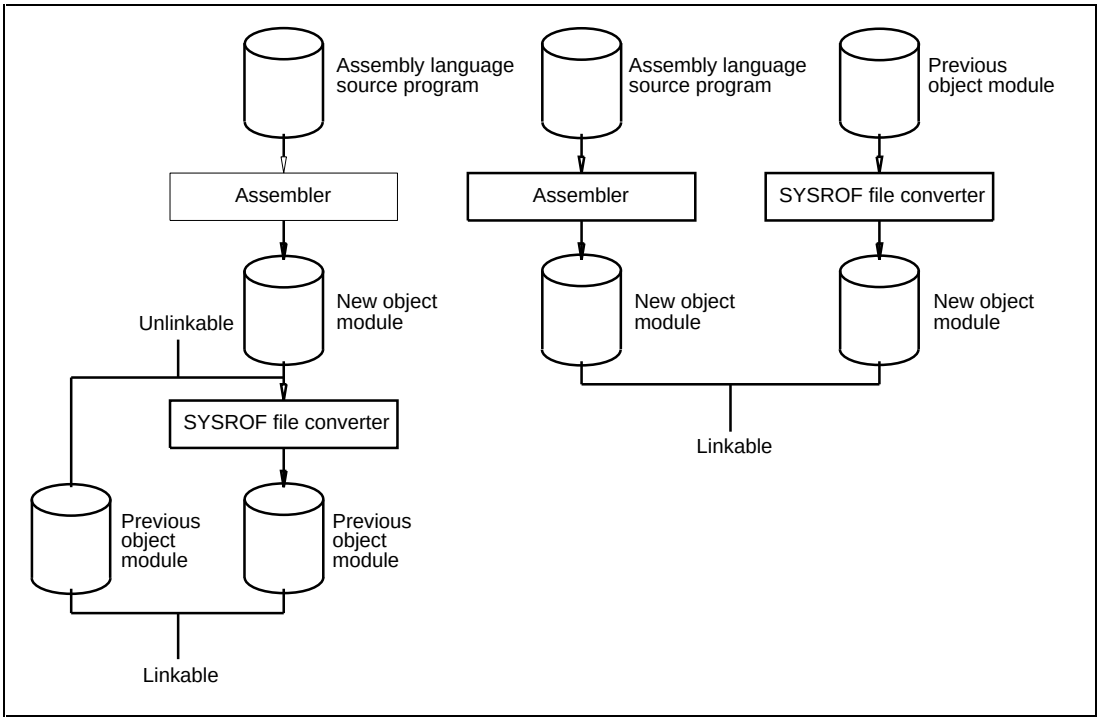


Figure D-1 Object Module Compatibility

D.7 Object Module Optimization

The new version has a function whereby the assembler optimizes the size of instructions in the object code when the size specification is omitted for the displacement in the PC relative addressing mode or the register-indirect-with-displacement addressing mode, or in the absolute address storage in the absolute addressing mode.

In case of H8/300H, because the previous version assumes 8 bits as the default size for forward-reference absolute values, the user needed to specify the size as 16 bits in the source program wherever an error occurred. However, the new version automatically selects the most optimal size from 8 bits or 16 bits for forward-reference absolute values.

Table D-4 shows the differences in displacement size.

Table D-4 Displacement Size Difference

		Displacement Size		
Reference Method	**Value Type**	**Previous Version**	**New Version with Non-optimization**	**New Version with Optimization**
Backward reference	Absolute value	8/16 bits	8/16 bits	8/16 bits
Backward reference	Relative value	16 bits	16 bits	16 bits
Backward reference	External reference value	16 bits	16 bits	16 bits
Forward reference	Absolute value	8 bits	8 bits	8/16 bits
Forward reference	Relative value	16 bits	16 bits	16 bits
Forward reference	External reference value	16 bits	16 bits	16 bits

Example

```
        .CPU 300HA
        .SECTION A, CODE, LOCATE=H'001000
        CMP     #10, R0L
        BEQ     LAB1            .....[1]
        BLT     LAB2            .....[2]
        MOV     R1L, R2L
    LAB1:
        MOV     R3L, R4L
        .ORG    H'002000
    LAB2:
        MOV     R5L, R6L
        .END
```

[1] Both the previous and new versions generate an object code with an 8-bit displacement size.

[2] The previous version generates an error, while the new version with optimization generates an object code with a 16-bit displacement size.

D.8 Change of Include Base Directory

When searching for include files, the previous version sets the search-start directory to the directory in which the assembler was initiated in, whereas the new version sets it to the read-destination directory.

Figure D-2 shows the base of the relative directory.

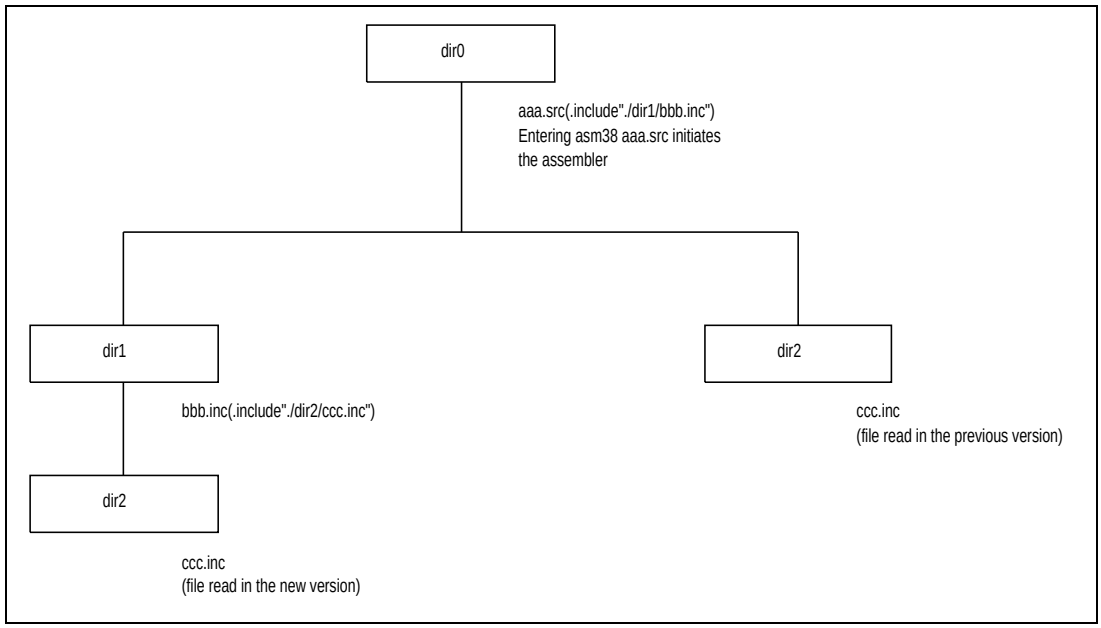


Figure D-2 Base of Relative Directory

In figure D-2, the assembler is initiated by entering asm 38 aaa.src while in directory dir0. Then include file 1 (./dir1/bbb.inc) specified by the relative directory is read into file aaa.src by the .INCLUDE directive. Include file 2 (./dir2/ccc.inc) specified by the relative directory is read into include file 1 (bbb.inc) by the .INCLUDE directive.

In this case, include file 2 (./dir2/ccc.inc), whose base directory is the assembler initiating directory (dir0), is read in the previous version, while in the new version, include file 2 (./dir2/ccc.inc), whose base directory is the directory (dir 1) containing include file 1 (bbb.inc), is read.

Note: MS-DOS directories are divided with a backlash (\).

D.9 Tag File Output

The previous version outputs a tag file when detecting at assembly an error for which an error message or warning message will be issued, but the new version does not.

To create a tag file, send the output to a file using redirection.

Examples
> asm 38 aaa.mar > & aaa.tag (UNIX)
> asm 38 aaa.mar > aaa.tag (MS-DOS)
Error and warning information are output to file aaa.tag.

Appendix E ASCII Code Table

Table E-1 ASCII Codes

Upper 4 bits / Lower 4 bits	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	,	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	ü	O	_	o	DEL

Characters within regions surrounded by double lines can be used in source programs.