Hitachi

SH Series Cross Assembler User's Manual

PN: H009-A

February 18, 1994

SH Series Cross Assembler

User's Manual

HS0700ASCU1SE

Preface

This manual describes the SH Series Cross Assembler.

This manual is organized as follows:

Overview:	Gives an overview of the functions of the assembler.
Programmer's Guide:	Describes the assembler language syntax and programming techniques.
User's Guide:	Describes the use (invocation) of the assembler program itself.
Appendix:	Describes assembler limitations and error messages.

The related manuals are listed below.

For information concerning the SH microprocessor hardware:

"SH7000 Series Hardware Manual"

For information concerning the SH microprocessor executable instructions:

"SH Series Programming Manual"

For information concerning software development support tools:

"SH Series C Compiler User's Manual"
"H Series Linkage Editor User's Manual"
"H Series Librarian User's Manual"
"SH Series Simulator/Debugger User's Manual"

Notes:

- The following symbols have special meanings in this manual.

<item>:	<specification item>
Δ:	Blank space(s) or tab(s)
>:	The OS prompt (indicates the input waiting state)
(RET):	Press the Return (Enter) key.
... :	The preceding item can be repeated.
[ ]:	The enclosed item is optional (i.e., can be omitted.)

- Numbers are written as follows in this manual.

Binary:	A prefix of "B'" is used.
Octal:	A prefix of "Q'" is used.
Decimal:	A prefix of "D'" is used.
Hexadecimal:	A prefix of "H'" is used.

However, when there is no specification, the number without a prefix is decimal.

UNIX is an operating system administrated by the UNIX System Laboratories (United States).

MS-DOS is an operating system administrated by the Microsoft Corporation (United States).

Contents

Appendix

Figures

Tables

Overview

(This page intentionally left blank.)

Section 1 Overview

The "SH Series Cross Assembler" (referred to below as the (or this) assembler) converts source programs written in assembly-language into a format that can be handled by SH microprocessors, and outputs the result as an object module. Also, the results of the assembly processing are output as an assemble listing.

This assembler provides the following functions to support efficient program development:

- Assembler directives
  Give the assembler various instructions.
- File inclusion function
  Includes files into a source file.
- Conditional assembly function
  Selects source statements to be assembled or repeats assembly according to a specified condition.
- Macro function
  Gives a ame to a sequence of statements and defines it as one instruction.
- Automatic literal pool generation function
  Interpretes data transfer instructions MOV.W #imm, MOV.L #imm, and MOVA #imm that are no provided by the SH microprocessor as extended instructions and expands them into SH microprocessor executable instructions and constant data (literals).

Figure 1-1 shows the function of the assembler.

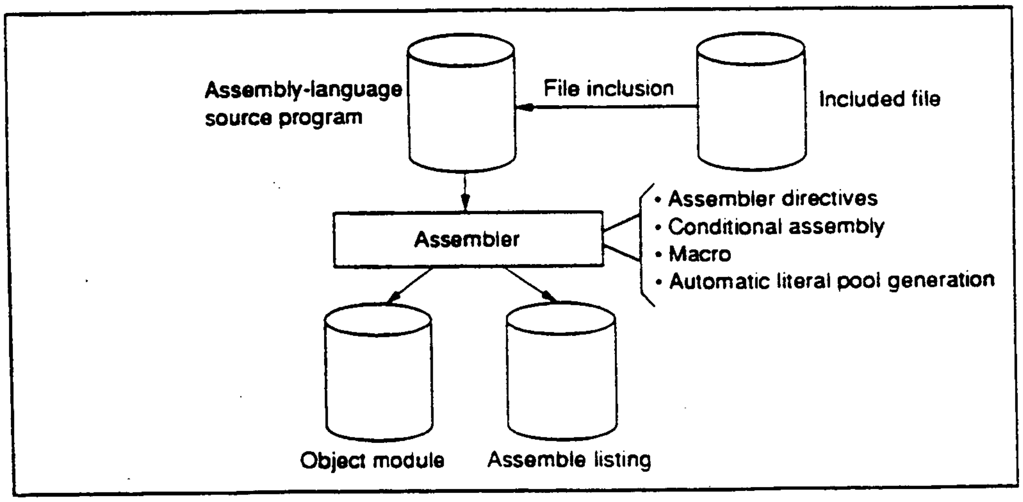


Figure 1-1 Function of the Assembler

(This page intentionally left blank.)

Section 2 Relationships between the Software Development Support Tools

The following software development support tools are available for the SH microprocessors.

- SH Series C Compiler (Referred to below as the C compiler.)
- H Series Linkage Editor (Referred to below as the linkage editor.)
- H Series Librarian (Referred to below as the librarian.)
- H Series Object Converter (Referred to below as the object converter.)
- SH Series Simulator/Debugger (Referred to below as the simulator/debugger.)

Note: The linkage editor refers to version 5.0 or later.

These tools assist in the efficient development of application software.

Figure 2-1 shows the relationships between the software development support tools.

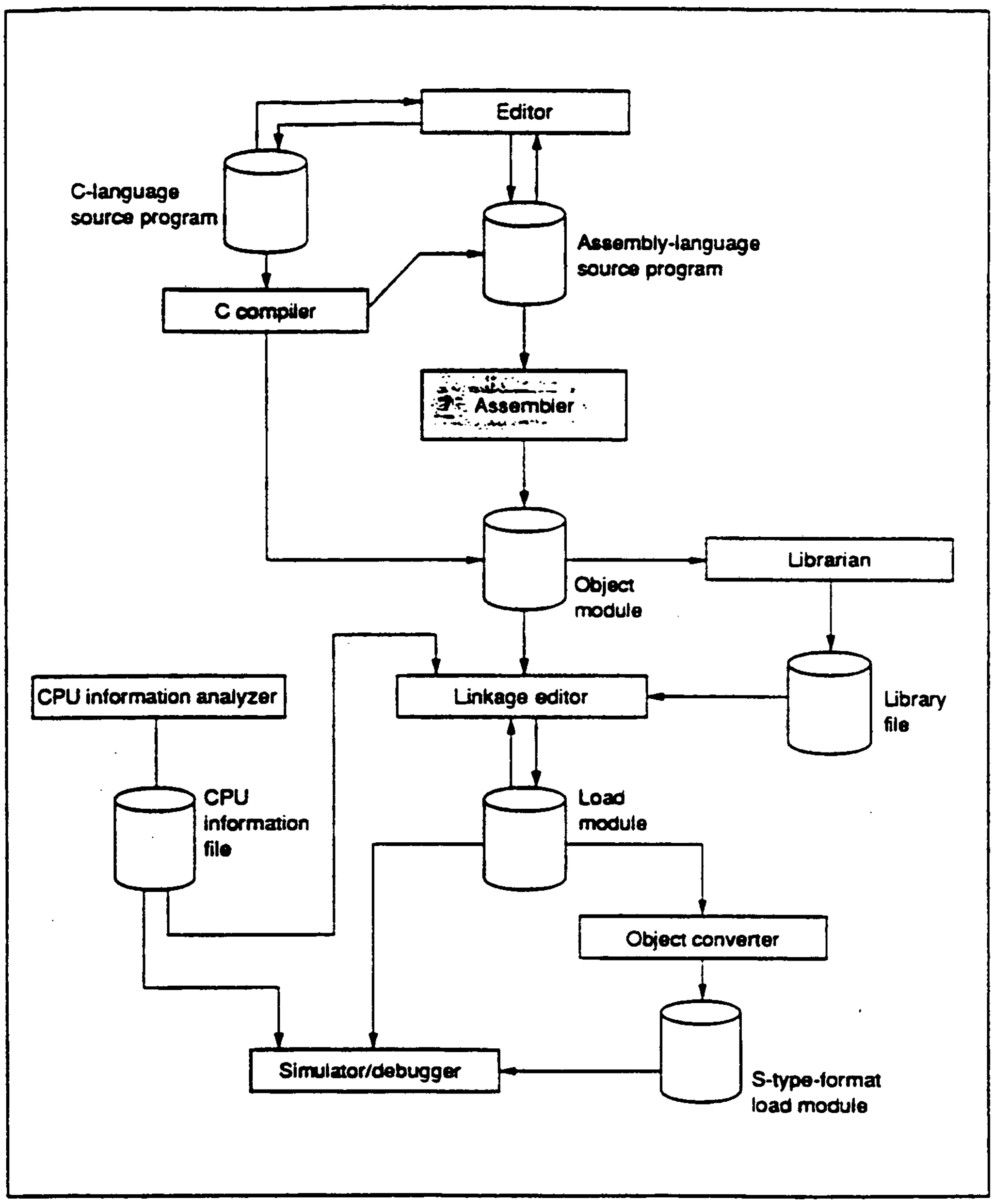


Figure 2-1 Relationships between the Software Development Support Tools

Supplement:

Use a general purpose editor (a text editor) to edit source programs.

The C compiler converts programs written in the C-language into either object modules or assembly-language source programs.

The librarian converts object modules and relocatable load modules into library files. We recommend handling processing that is common to multiple programs as a library file. (This has several advantages, including allowing modules to be easily managed.)

The linkage editor links together object modules and library files to produce load modules. (Load modules are programs in a format that a computer can execute.)

The object converter converts load modules into the S-type format. (The S-type format is a standard load module format.)

The simulator/debugger assists debugging microprocessor software.

Load modules created by this development support system can be input to several types of emulator. (Emulators are systems for debugging microprocessor system hardware and software.) Also, S-type-format load modules can be input into most EPROM writers.

(This page intentionally left blank.)

Programmer's Guide

(This page intentionally left blank.)

Section 1 Program Elements

If source programs are compared to natural language writing, a source statement will correspond to "a sentence." The "words" that make up a source statement are reserved words and symbols. This section describes these basic program elements.

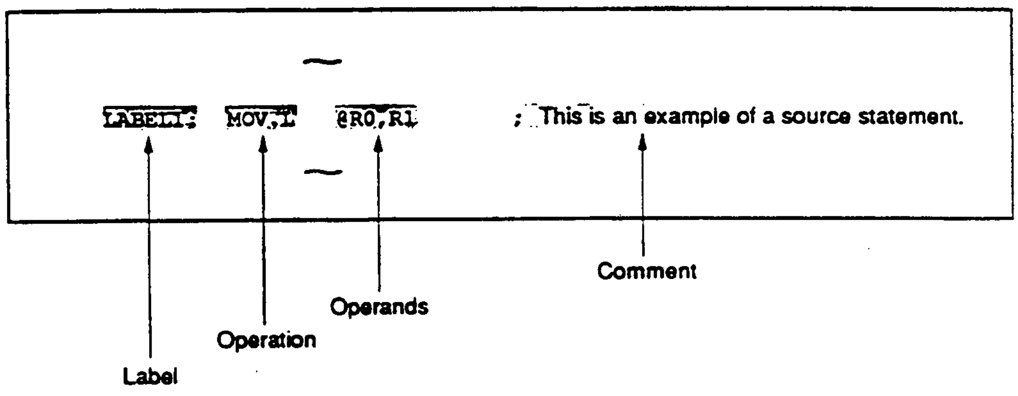
1.1 Source Statements

1.1.1 Source Statement Structure

The figure below shows the structure of a source statement.

```
[<label>] [Δ<operation>[Δ<operand(s)>]] [<comment>]
```

Example:



```
LABEL1: MOV.L @R0,R1          ; This is an example of a source statement.
```

(1) Label

A symbol that is a tag attached to a source statement is written as a label.

A symbol is a name defined by the programmer.

(2) Operation

The mnemonic of an executable instruction, an extended instruction, an assembler directive, or a directive statement is written as the operation.

Executable instructions must be SH microprocessor instructions.

Extended instructions are instructions that are expanded into executable instructions and constant data (literals). For details, refer to Programmer's Guide, 8, "Automatic Literal Pool Generation Function".

Assembler directives are instructions that give directions to the assembler.

Directive statements are used for file inclusion, conditional assembly, and macro functions. For details on each of these functions, refer to Programmer's Guide, 5, "File Inclusion Function", 6, "Conditional Assembly Function", or 7, "Macro Function".

(3) Operand

The object(s) of the operation's execution are written as the operand.

The number of operands and their types are determined by the operation. There are also operations which do not require any operands.

(4) Comment

Notes or explanations that make the program easier to understand are written as the comment.

1.1.2 Coding of Source Statements

Source statements are written using ASCII characters.

In principle, a single statement must be written on a single line. The maximum length of a line is 255 bytes.

(1) Coding of Label

The label is written as follows:

- Written starting in the first column,
  Or:
- Written with a colon (:) appended to the end of the label.

Examples:

```
LABEL1                          ; This label is written starting in the first column.
  LABEL2:                       ; This label is terminated with a colon.

 ----------------------------------------------------------------------------

      LABEL3                    ; This label is regarded as an error by the assembler,
                                ; since it is neither written starting in the first column
                                ; nor terminated with a colon.
```

(2) Coding of Operation

The operation is written as follows:

- When there is no label:
  Written starting in the second or later column.

- When there is a label:
  Written after the label, separated by one or more spaces or tabs.

Examples:

```
         ADD     R0,R1          ; An example with no label.
LABEL1:  ADD     R1,R2          ; An example with a label.
```

CAUTION!

Since white spaces and tabs are ASCII characters, each space or tab requires a byte of storage.

(3) Coding of Operand

The operand is written following the operation field, separated by one or more spaces or tabs.

Examples:

```
        ADD     R0,R1           ; The ADD instruction takes 2 arguments.
        SHAL    R1              ; The SHAL instruction takes 1 argument.
```

(4) Coding of Comment

The comment is written following a semicolon (;).

The assembler regards all characters from the semicolon to the end of the line as the comment.

Examples:

```
        ADD     R0,R1           ; Adds R0 to R1.
```

1.1.3 Coding of Source Statements across Multiple Lines

We recommend writing a single source statement across several lines in the following situations:

- When the source statement is too long as a single statement.
- When it is desirable to attach a comment to each operand.

Write source statements across multiple lines using the following procedure.

(a) Insert a new line writing a comma that separates operands as the point to break the line.
(b) Insert a plus sign (+) in the first column of the next line.
(c) Continue writing the source statement following the plus sign.

Spaces and tabs can be inserted following the plus sign.

Examples:

```
          .DATA.L   H'FFFF0000,
+                   H'FF00FF00,
+                   H'FFFFFFFF

; In this example, a single source statement is written across three lines.
```

A comment can be attached at the end of each line.

Examples:

```
          .DATA.L   H'FFFF0000,    ; Initial value 1.
+                   H'FF00FF00,    ; Initial value 2.
+                   H'FFFFFFFF     ; Initial value 3.

; This is an example of attaching a comment to each operand.
```

1.2 Reserved Words

Reserved words are names that the assembler reserves as symbols with special meanings.

This assembler uses the following reserved words.

- Register names

R0	R1	R2	R3	R4	R5
R6	R7	R8	R9	R10	R11
R12	R13	R14	R15	SP*	
SR	GBR	VBR	MACH	MACL	PR
PC					

Note: * R15 and SP indicate the same register.

- Operators (STARTOF, SIZEOF)
- The location counter symbol, the dollar sign (a single character symbol)

Reserved words cannot be used as user-defined symbols.

Reference:

Operators	→ Programmer's Guide, 1.6.1, "Expression Elements"
Location counter	→ Programmer's Guide, 1.5, "Location Counter"
Symbols	→ Programmer's Guide, 1.3, "Symbols"

1.3 Symbols

1.3.1 Functions of Symbols

Symbols are names defined by the programmer, and perform the following functions.

- Address symbols .... Express data storage and destination addresses.
- Constant symbols .... Express constants.
- Aliases of register name ..... Express general registers.
- Section names .... Express section names. *

Note: * A section is a part of program, and the linkage editor regards it as a unit of processing.

The following show examples of symbol usages.

Examples:

```
           ~
        BRA   SUB1             ; BRA is a branch instruction.
                               ; SUB1 is the address of the destination.
           ~

SUB1:

           ~
------------------------------------------------------------------------

           ~

MAX:    .EQU   100             ; .EQU is an assembler directive that sets a value to a
                               ; symbol.
        MOV   #MAX,R0          ; MAX expresses the constant value 100.

           ~
------------------------------------------------------------------------

           ~

MIN:    .REG   (R0)            ; .REG is an assembler directive that defines a register
                               ; name.
        MOV   #100,MIN         ; Here, MIN is a name for R0.

           ~
------------------------------------------------------------------------

           ~

        .SECTION CD,CODE,ALIGN=4

                               ; .SECTION is an assembler directive that declares a section.
                               ; CD is the name of the current section.
           ~
```

1.3.2 Coding of Symbols

(1) Available Characters

The following members of the ASCII character can be used.

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

The assembler distinguishes upper-case and lower-case in symbols.

(2) First Character in a Symbol

The first character in a symbol must be one of the following.

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

(3) Maximum Length of a Symbol

A symbol may contain up to 32 characters.

The assembler ignores any characters after the first 32.

(4) Names that Cannot Be Used as Symbols

Reserved words cannot be used as symbols. The following names must not be used because they are used as internal symbols by the assembler.

_$$nnnnn (n is a number from 0 to 9.)

Note: Internal symbols are necessary for assembler internal processing. Internal symbols are not output to assemble listings or object modules.

CAUTION!

The dollar sign character used alone is a reserved word that expresses the location counter.

References:

Reserved words → Programmer's Guide, 1.2, "Reserved Words"

1.4 Constants

1.4.1 Integer Constants

Integer constants are expressed with a prefix that indicates the radix.

The radix indicator prefix is a notation that indicates the base of the constant.

• Binary numbers	The radix indicator "B'" plus a binary constant.
• Octal numbers	The radix indicator "Q'" plus an octal constant.
• Decimal numbers	The radix indicator "D'" plus a decimal constant.
• Hexadecimal numbers	The radix indicator "H'" plus a hexadecimal constant.

The assembler does not distinguish upper-case and lower-case letters in the radix indicator.

The radix indicator and the constant value must be written with no intervening space.

Examples:

```
        .DATA.B  B'10001000   ;
        .DATA.B  Q'210        ; These source statements all the same
        .DATA.B  D'136        ; numerical value.
        .DATA.B  H'88         ;
```

The radix indicator can be omitted. Integer constants with no radix indicator are normally decimal constants, although the radix for such constants can be changed with the .RADIX assembler directive.

References:

Interpretation of integer constants without a radix specified
→ Programmer's Guide, 4.2.7, "Other Assembler Directives", .RADIX

Supplement:

"Q" is used instead of "O" to avoid confusion with the digit 0.

1.4.2 Character Constants

Character constants are considered to be constants that represent ASCII codes.

Character constants are written by enclosing up to 4 ASCII characters in double quotation marks.

The following ASCII characters can be used in character constants.

ASCII codes { H'09 (tab)
H'20 (space) to H'7E (tilde)

Examples:

```
.DATA.L  "ABC"        ; This is the same as .DATA.L H'00414243.
.DATA.W  "AB"         ; This is the same as .DATA.W H'4142.
.DATA.B  "A"          ; This is the same as .DATA.B H'41.

                            ; The ASCII code for A is:  H'41
                            ; The ASCII code for B is:  H'42
                            ; The ASCII code for C is:  H'43
```

Use two double quotation marks in succession to indicate a single double quotation mark in a character constant.

Example:

```
.DATA.B  """"         ; This is a character constant consisting of a single
                        double quotation mark.
```

1.5 Location Counter

The location counter expresses the address (location) in memory where the corresponding object code (the result of converting executable instructions and data into codes the microprocessor can regard) is stored.

The value of the location counter is automatically adjusted according to the object code output.

The value of the location counter can be changed intentionally using assembler directives.

Examples:

```
~

.ORG      H'00001000      ; This assembler directive sets the location counter to
                          ; H'00001000.

.DATA.W   H'FF            ; The object code generated by this assembler directive has
                          ; a length of 2 bytes.
                          ; The location counter changes to H'00001002.

.DATA.W   H'F0            ; The object code generated by this assembler directive has
                          ; a length of 2 bytes.
                          ; The location counter changes to H'00001004.

.DATA.W   H'10            ; The object code generated by this assembler directive has
                          ; a length of 2 bytes.
                          ; The location counter changes to H'00001006.

.ALIGN    4               ; The value of the location counter is corrected to be a multiple
                          ; of 4.
                          ; The location counter changes to H'00001008.

.DATA.L   H'FFFFFFFF      ; The object code generated by this assembler directive has
                          ; a length of 4 bytes.
                          ; The location counter changes to H'0000100C.

    ;       .ORG is an assembler directive that sets the value of the location counter.
    ;       .ALIGN is an assembler directive that adjusts the value of the location counter.

    ;       .DATA is an assembler directive that reserves data in memory.
    ;       .W is a specifier that indicates that data is handled in word (2 byte) size.
    ;       .L is a specifier that indicates that data is handled in long word (4 byte) size.
~
```

References:

Setting the value of the location counter

→ Programmer's Guide, 4.2.1, "Section and Location Counter Assembler Directives" .ORG

Correcting the value of the location counter

→ Programmer's Guide, 4.2.1, "Section and Location Counter Assembler Directives" .ALIGN

The location counter is referenced using the dollar sign symbol.

Examples:

```
LABEL1:   .EQU     $              ; This assembler directive sets the value of the
                                  ; location counter to the symbol LABEL1.

;         .EQU is an assembler directive that sets the value to a symbol.
```

1.6 Expressions

Expressions are combinations of constants, symbols, and operators that derive a value, and are used as the operands of executable instructions and assembler directives.

1.6.1 Elements of Expression

An expression consists of terms, operators, and parentheses.

(1) Terms

The terms are the followings:

- A constant
- The location counter reference ($)
- A symbol (excluding aliases of the register name)
- The result of a calculation specified by a combination of the above terms and an operator.

An independent term is also a type of expression.

(2) Operators

Table 1-1 shows the operators supported by the assembler.

Table 1-1 Operators

Operator Type	Operator	Operation	Coding
Arithmetic operations	+	Unary plus	+ <term>
	–	Unary minus	– <term>
	+	Addition	<term1> + <term2>
	–	Subtraction	<term1> – <term2>
	*	Multiplication	<term1> * <term2>
	/	Division	<term1> / <term2>
Logic operations	~	Unary negation	~ <term>
	&	Logical AND	<term1> & <term2>
	|	Logical OR	<term1> | <term2>
	~	Exclusive OR	<term1> ~ <term2>
Section set operations*	STARTOF	Derives the starting address of a section set.	STARTOF <section name>
	SIZEOF	Derives the size in bytes of a section set.	SIZEOF <section name>

Note: * See the supplement on the following page.

(3) **Parentheses**

Parentheses modify the operation priority.

See the next section, section 1.6.2, "Operation Order", for a description of the use of parentheses.

Supplement:

In this assembly-language, programs are divided into units called section. Sections are the units in which linkage processing is performed.

When there are multiple sections of the same type and same name within a given program, the linkage editor links them into a single "section set".

STARTOF is an operator that determines the starting address of the section set.

SIZEOF is an operator that determines the size of the section set in byte units.

References:

Sections → Programmer's Guide, 2.1, "Sections"

1.6.2 Operation Priority

When multiple operations appear in a single expression, the order in which the processing is performed is determined by the operator priority and by the use of parentheses. The assembler processes operations according to the following rules.

<Rule 1>
Processing starts from operations enclosed in parentheses.
When there are multiple parentheses, processing starts with the operations surrounded by the innermost parentheses.

<Rule 2>
Processing starts with the operator with the highest priority.

<Rule 3>
Processing proceeds in the direction of the operator association rule when operators have the same priority.

Table 1-2 shows the operator priority and the association rule.

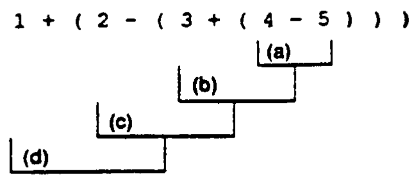
Table 1-2 Operator Priority and Association Rules

Priority	Operator	Association Rule
1 (high)	+ – ~ STARTOF SIZEOF*	Operators are processed from right to left.
2	* /	Operators are processed from left to right.
3	+ –	Operators are processed from left to right.
4	&	Operators are processed from left to right.
5 (low)	| ~	Operators are processed from left to right.

Note: * Unary operators have the first priority.

The figures below show examples of expressions.

Example 1:



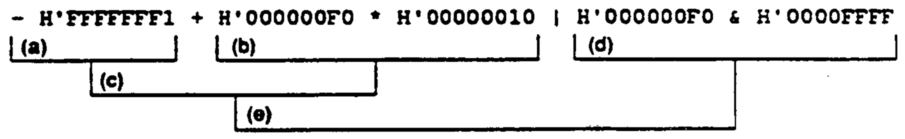
```
1 + ( 2 - ( 3 + ( 4 - 5 ) ) )
```

The assembler calculates this expression in the order (a) to (d).

The result of (a) is –1
The result of (b) is 2
The result of (c) is 0
The result of (d) is 1

The final result of this calculation is 1.

Example 2:



```
- H'FFFFFFF1 + H'000000F0 * H'00000010 | H'000000F0 & H'0000FFFF
```

The assembler calculates this expression in the order (a) to (e).

The result of (a) is H'0000000F
The result of (b) is H'00000F00
The result of (c) is H'00000F0F
The result of (d) is H'000000F0
The result of (e) is H'00000FFF

The final result of this calculation is H'00000FFF.

Example 3:

$$\underbrace{-\underbrace{\sim\underbrace{-\underbrace{\sim\texttt{H'0000000F}}_{(a)}}_{(b)}}_{(c)}}_{(d)}$$

The assembler calculates this expression in the order (a) to (d).

The result of (a) is H'FFFFFFF0
The result of (b) is H'00000010
The result of (c) is H'FFFFFFEF
The result of (d) is H'00000011

The final result of this calculation is H'00000011.

1.6.3 Notes on Expressions

(1) Internal Processing

The assembler regards expression values as 32-bit signed values.

Example:

```
~H'F0

    The assembler regards H'F0 as H'000000F0.
    Therefore, the value of ~H'F0 is H'FFFFFF0F. (Note that this is not H'0000000F.)
```

(2) Arithmetic Operators

The multiplication and division operators cannot take terms that contain relative values (values which are not determined until the end of the linkage process) as their operands.

Also, a divisor of 0 cannot be used with the division operator.

(3) Logic Operators

The logic operators cannot take terms that contain relative values as their operands.

References:

Relative values → Programmer's Guide, 2.2, "Absolute and Relative Values".

1.7 Character Strings

Character strings are sequences of character data.

The following ASCII characters can be used in character strings.

ASCII codes { H'09 (tab)
H'20 (space) to H'7E (tilde)

A single character in a character string has as its value the ASCII code for that character and is represented as a byte sized data object.

Character strings are written enclosed in double quotation marks.

Use two double quotation marks in succession to indicate a single double quotation mark in a character string.

Examples:

```
        .SDATA    "Hello!"          ; This statement reserves the character string data
                                    ; Hello!
        .SDATA    """Hello!"""      ; This statement reserves the character string data
                                    ; "Hello!"

;       .SDATA is an assembler directive that reserves character string data in memory.
```

Supplement:

The difference between character constants and character strings is as follows.

Character constants are numeric values. They have a data size of either 1 byte, 2 bytes, or 4 bytes.

Character strings cannot be handled as numeric values. A character string has a data size of between 1 byte and 255 bytes.

(This page intentionally left blank.)

Section 2 Basic Programming Knowledge

This section presents the basic knowledge required for programming in assembly-language.

2.1 Sections

If source programs are compared to natural language writing, a section will correspond to a "chapter." The section is the processing unit used when the linkage editor links an object module.

2.1.1 Section Types by Usage

Sections are classified by usage into the following types.

- Code section
- Data section
- Common section
- Stack section
- Dummy section

(1) Code Section

The following can be written in a code section:

- Executable instructions
- Extended instructions
- Assembler directives that reserve initialized data.

Examples:

```
         .SECTION   CD,CODE,ALIGN=4     ; This assembler directive declares a
                                        ; code section with the name CD.

          MOV.L     X,R1                ; This is an executable instruction.
          MOV       R1,R2

            ~

         .ALIGN     4
X:       .DATA.L    H'FFFFFFFF          ; This assembler directive reserves
                                        ; initialized data.
            ~
```

(2) Data Section

The following can be written in a data section:

- Assembler directives that reserve initialized data.
- Assembler directives that reserve uninitialized data.

Examples:

```
        .SECTION    DT1,DATA,ALIGN=4      ; This assembler directive declares
                                          ; a data section with the name DT1.

        .DATA.W     H'FF00                ; These assembler directives reserve
        .DATA.B     H'FF                  ; initialized data.

             ~
```

```
        .SECTION    DT2,DATA,ALIGN=4      ; This assembler directive declares
                                          ; a data section with the name DT2.

        .RES.W      10                    ; These assembler directives reserve
        .RES.B      10                    ; data areas that do not have initial
                                          ; values.
             ~
```

(3) **Common Section**

A common section is used as a section to hold data that is shared between files when a source program consists of multiple source files.

The following can be written in a common section:

- Assembler directives that reserve initialized data.
- Assembler directives that reserve uninitialized data.

Supplement:

The linkage editor reserves common sections with the same name to the same area in memory. In the example shown in figure 2-1, the common section CM declared in file A and the common section CM declared in file B are reserved to the same area in memory.

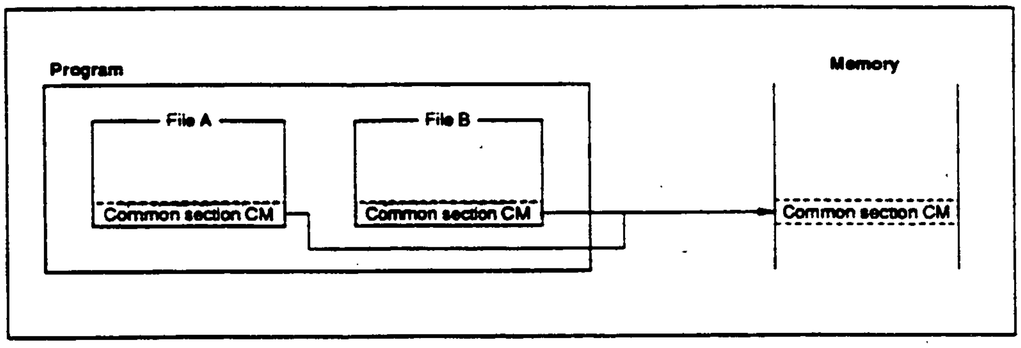


Figure 2-1 Memory Reservation of Common Section

(4) Stack Section

The section that SH microprocessors use as a stack area (an area for temporary data storage) is called the stack section.

The following can be written in the stack section:

- Assembler directives that reserve uninitialized data.

Examples:

```
          .SECTION ST,STACK,ALIGN=4         ; This assembler directive declares a stack
                                            ; section with the name ST.

          .RES.B   1024                     ; This assembler directive reserves a stack
                                            ; area of 1024 bytes.
```

(5) Dummy Section

A dummy section is a hypothetical section for representing data structures. The assembler does not output dummy sections to the object module.

The following can be written in a dummy section:

- Assembler directives that reserve uninitialized data.

Examples:

```
          .SECTION DM,DUMMY                 ; This assembler directive declares
                                            ; a dummy section with the name DM.
          .RES.B  1                         ; The assembler does not output the
A:        .RES.B  1                         ; section DM to the object module.
B:        .RES.B  2

              ~
```

Specific methods for specifying data structures are described in the supplement on the next page.

Supplement:

As shown in figure 2-2, it is possible to access areas in memory by using address symbols from a dummy section.

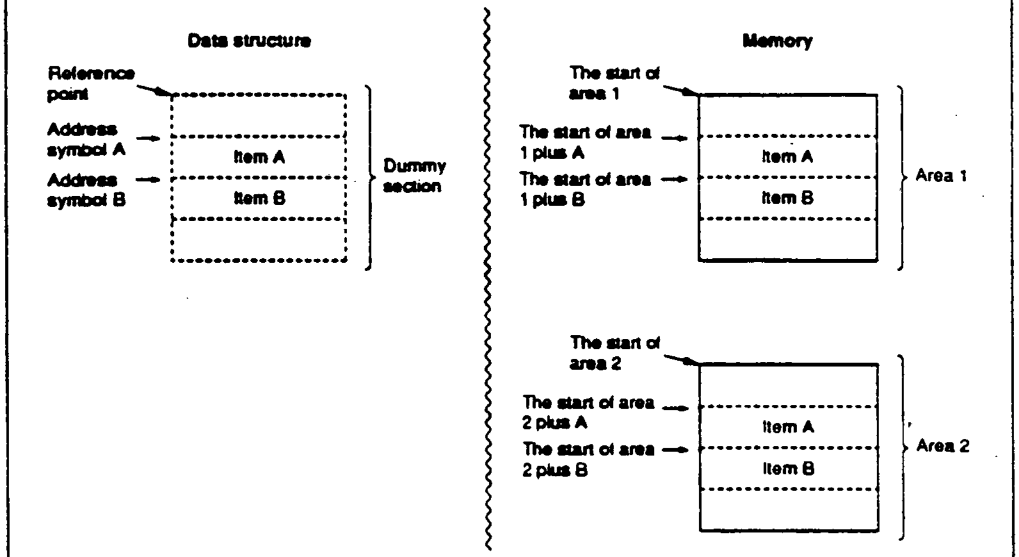


Figure 2-2 Data Structure Example Using Dummy Section

Coding Example:

```
        ~

; In the example above,
; assume that R1 holds the starting address of area 1 and R2 holds the starting address of
; area 2.

        MOV.L    @(B,R1),R0        ; Moves the contents of item B in area 1 to R0.
        MOV.L    R0,@(B,R2)        ; Moves the contents of R0 to item B in area 2.

        ~
```

CAUTION!

1. The following cannot be used in stack and dummy sections:

    - Executable instructions
    - Extended instructions
    - Assembler directives that reserve initialized data
    (.DATA, .DATAB, .SDATA, .SDATAB, .SDATAC, and .SDATAZ)

2. When using a data or common section, be sure to keep in mind whether that section is reserved to ROM or RAM.

2.1.2 Absolute Address Sections and Relative Address Sections

A section can be classified as either an absolute address section or as a relative address section depending on whether absolute start addresses are given to the sections at assembly.

(1) Absolute Address Sections

The memory location of absolute address sections is specified in the source program, and cannot be changed by the linkage editor. In this assembly language, locations in an absolute address section are expressed as absolute addresses, which are addresses that express the position in memory itself.

Examples:

```
.SECTION ABS,DATA,LOCATE=H'0000F000     ; ABS is an absolute-address section.
                                        ; The starting address of section ABS is
                                        ; the absolute address H'0000F000.

.DATA.W  H'1111                         ; The constant H'1111 is reserved at
                                        ; the absolute address H'0000F000.

.DATA.W  H'2222                         ; The constant H'2222 is reserved at
                                        ; the absolute address H'0000F002.
```

(2) Relative Address Section

The location in memory of relative sections is not specified in the source program, but rather is determined when the sections are linked by the linkage editor. In this assembly-language, locations in a relative address section are expressed as relative addresses, which are addresses that express the position relative to the start of the section itself.

Examples:

```
.SECTION  REL,DATA,ALIGN=4       ; REL is a relative address section.
                                 ; The starting address of section REL is
                                 ; determined after linkage.

.DATA.W   H'1111                 ; The constant H'1111 is reserved at the
                                 ; relative address H'00000000.

.DATA.W   H'2222                 ; The constant H'2222 is reserved at the
                                 ; relative address H'00000002.
```

Supplement:

Dummy sections correspond neither to relative nor to absolute sections.

2.2 Absolute and Relative Values

Absolute values are determined when assembly completes. Relative values are not determined until the linkage editor completes.

2.2.1 Absolute Values

The following are the absolute values handled by the assembler.

(1) Constants

- Integer constants
- Character constants
- Symbols that have a value that is one of the above (referred to below as constant symbols).

(2) Absolute Address Values

- The location counter referenced in an absolute address section
- The location counter referenced in a dummy section
- Symbols that have a value that is one of the above (referred to below as absolute address symbols).

(3) Other Absolute Values

Expressions whose value is determined when assembly completes.

2.2.2 Relative Values

The following are the relative values handled by the assembler.

(1) Relative Address Values

- The location counter referenced in a relative address segment
- Symbols that have the above as a value. (Such symbols are referred to as relative address symbols.)

(2) External Reference Values

Symbols that reference another file (referred to below as import symbols).

(3) Other Relative Values

Expressions whose value is not determined until the linkage editor completes.

2.3 Symbol Definition and Reference

2.3.1 Symbol Definition

(1) Normal Definition

The normal method for defining a symbol consists of writing that symbol in the label field of a source statement. The value of that symbol will then be the value of the location counter at that point in the program.

Examples:

```
        .SECTION DT1,DATA,LOCATE=H'0000F000      ; This statement declares an
                                                 ; absolute address section.
X1:     .DATA.W  H'1111                   ; The value of X1 becomes H'0000F000.
X2:     .DATA.W  H'2222                   ; The value of X2 becomes H'0000F002.
           ~
```

```
      .SECTION DT2,DATA,ALIGN=4           ; This statement declares a relative address
                                          ; section.
Y1:   .DATA.W  H'1111                     ; The value of Y1 is determined when the
                                          ; linkage editor completes, and its value is
                                          ;  the start address of the section.
Y2:   .DATA.W  H'2222                     ; The value of Y2 is determined when the
                                          ; linkage editor completes, and its value is
                                            the start address of the section plus 2.
```

(2) Definition by Assembler Directive

Symbols can be defined by using assembler directives to set an arbitrary value or a special meaning.

Examples:

```
         .SECTION DT1,DATA,ALIGN=4       ; DT1 is the section name.
                                         ; A section name is also a type of symbol,
                                         ; a symbol that expresses the start
                                         ; address of a section.
                                         ; However, the syntactic handling of address
                                         ; symbols and section names is different.

X:       .EQU     100                    ; The value of X is 100.
                                         ; X cannot be redefined.

Y:       .ASSIGN  10                     ; The value of Y is 10.
                                         ; Y can be redefined.

Z:       .REG     (R1)                   ; Z becomes an alias of the general
                                         ; register R1.
                                         ; Z cannot be redefined.
```

2.3.2 Symbol Reference

There are three forms of symbol reference as follows:

- Forward reference
- Backward reference
- External reference

Supplement:

Figure 2-3 shows the meaning of the terms forward and backward as used in this manual.

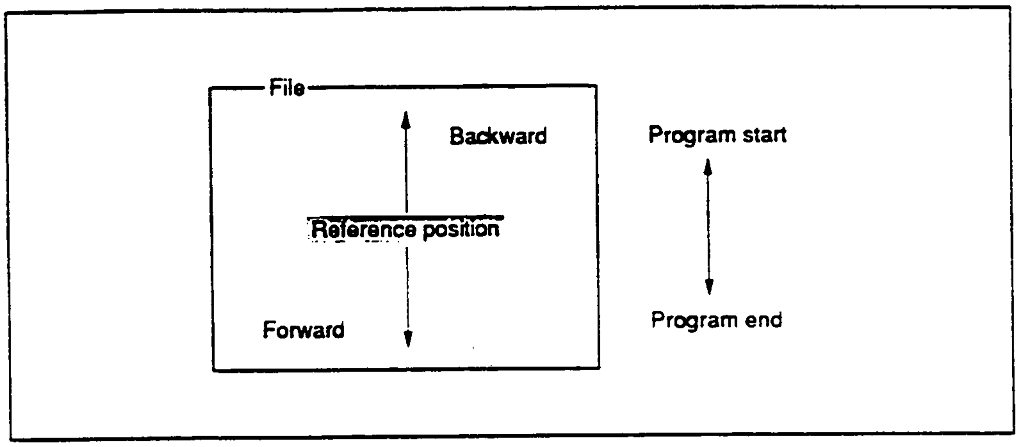


Figure 2-3 Meaning of the Terms Forward and Backward

Figure 2-4 shows the meaning of the term external as used in this manual.

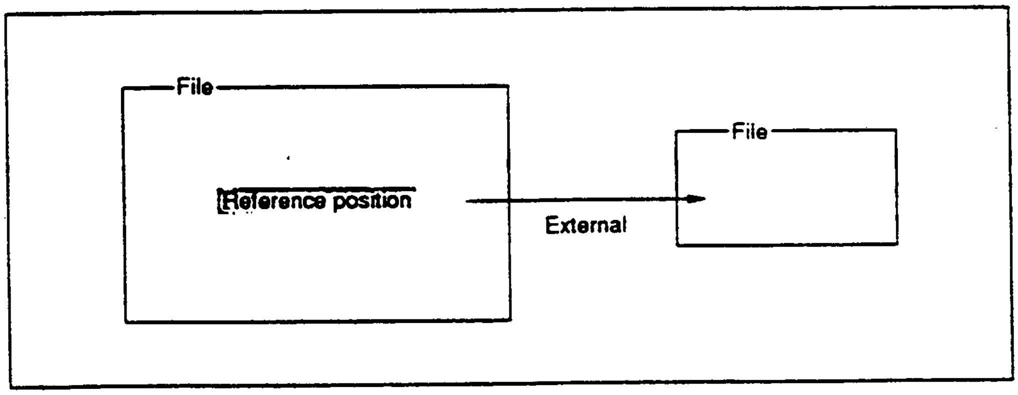


Figure 2-4 Meaning of the Term External

(1) Forward Reference

Forward reference means referencing a symbol that is defined forward from the point of reference. i.e., is defined later in the program.

Examples:

```
                ~
            BRA    FORWARD        ; BRA is a branch instruction.
                                  ; This is a forward reference to the symbol FORWARD.
                ~
FORWARD:
                ~
```

(2) Backward Reference

Backward reference means referring to a symbol that is defined backward from the point of reference, i.e., is defined earlier in the program.

Examples:

```
                ~
BACK:
                ~
            BRA    BACK           ; BRA is a branch instruction.
                                  ; This is a backward reference to the symbol BACK.
                ~
```

(3) External Reference

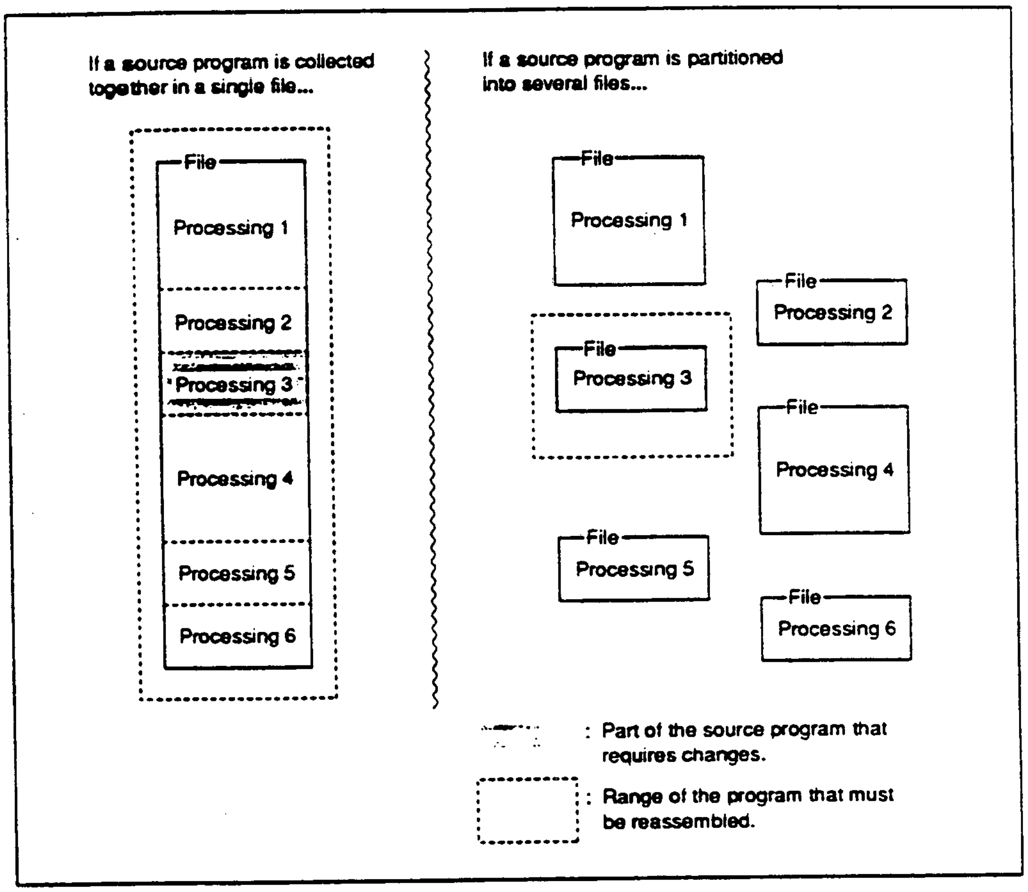
When a source program consists of multiple source files, a reference to a symbol defined in another file is called an external reference. External reference is described in the next section, 2.4, "Separate Assembly".

2.4 Separate Assembly

2.4.1 Separate Assembly

Separate assembly refers to the technique of creating a source program in multiple separate source files, and finally creating a single load module by linking together those source files' object modules using the linkage editor.

The process of developing software often consists of repeatedly correcting and reassembling the program. In such cases, if the source program is partitioned, it will be only necessary to reassemble the source file that was changed. As a result, the time required to construct the complete program will be significantly reduced.



If a source program is collected together in a single file...
File
Processing 1
Processing 2
Processing 3
Processing 4
Processing 5
Processing 6
If a source program is partitioned into several files...
File
Processing 1
File
Processing 2
File
Processing 3
File
Processing 4
File
Processing 5
File
Processing 6
: Part of the source program that requires changes.
: Range of the program that must be reassembled.

Figure 2-5 Relationship between the Changed Range of the Program and the Range of the Program that must be Reassembled

The procedure involved in separate assembly consists of steps (a) to (d).

(a) Investigate methods for partitioning the program.

Normally, programs are partitioned by function.

Note that the memory reservation of the section must also be considered at this point.

(b) Divide the source program into separate files and edit those files accordingly.

(c) Assemble the individual files.

(d) Link the individual object modules into a single load module.

2.4.2 Declaration of Export Symbols and Import Symbols

When a source program consists of multiple files, referencing a symbol defined in one file from another file is called "external reference" or "import." When referencing a symbol externally (this declaration is called "external definition" or "export"), it is necessary to declare to the assembler that "this symbol is shared between multiple files."

(1) Export Symbol Declaration

This declaration is used to declare that the definition of the symbol is valid in other files. . EXPORT or .GLOBAL assembler directive is used to make this declaration.

(2) Import Symbol Declaration

This declaration is used to declare that a symbol is defined in another file. .IMPORT or . GLOBAL assembler directive is used to make this declaration.

Examples:

In this example the symbol MAX is defined in file A and referenced in file B.

File A:

```
          ~
        .EXPORT  MAX          ; Declares MAX to be an export symbol.
MAX:    .EQU     100          ; Defines MAX.
          ~
```

File B:

```
          ~
        .IMPORT  MAX          ; Declares MAX to be an import symbol.
         MOV     #MAX,R0      ; References MAX.
          ~
```

References:

Symbol Export and Import
→ Programmer's Guide, 4.2.4, "Export and Import Assembler Directives", .EXPORT, .IMPORT, .GLOBAL

(This page intentionally left blank.)

Section 3 Executable Instructions

This section describes the points that must be kept in mind when using executable instructions in this assembler.

3.1 Overview of Executable Instructions

The executable instructions are the instructions of SH microprocessor. SH microprocessor interprets and executes the executable instructions in the object code stored in memory.

An executable instruction source statement has the following basic form.

```
[<symbol>:] Δ<mnemonic>[.<operation size>] [Δ<addressing mode>[,<addressing mode>]] [;<comment>]
└────────┘ └─────────────────────────────┘ └──────────────────────────────────────┘ └──────────┘
   Label              Operation                            Operand                     Comment
```

This section describes the mnemonic, operation size, and addressing mode. The other elements are described in detail in section 1, "Program Elements", in the Programmer's Guide.

(1) Mnemonic

The mnemonic expresses the type of executable instruction. Abbreviations that indicate the type of processing are provided as mnemonics for SH microprocessor instructions.

The assembler does not distinguish upper-case and lower-case letters in mnemonics.

(2) Operation Size

The operation size is the unit for processing data. The operation sizes vary with the executable instruction. The assembler does not distinguish upper-case and lower-case letters in the operation size.

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

(3) Addressing Mode

The addressing mode specifies the data area accessed, and the destination address. The addressing modes vary with the executable instruction. Table 3-1 shows the addressing mode.

Table 3-1 Addressing Modes

Addressing Mode	Name	Description
Rn	Register direct	The contents of the general register Rn.
SR	SR direct	The contents of the status register (SR).
GBR	GBR direct	The contents of the global base register (GBR).
VBR	VBR direct	The contents of the vector base register (VBR).
MACH	MACH direct	The contents of the accumulator register (MACH).
MACL	MACL direct	The contents of the accumulator register (MACL).
PR	PR direct	The contents of the procedure register (PR).
@Rn	Register indirect	A memory location. The value in Rn gives the start address of the memory accessed.
@Rn+	Register indirect with post-increment	A memory location. The value in Rn (before being incremented[*1]) gives the start address of the memory accessed. SH microprocessor first uses the value in Rn for the memory reference, and increments Rn afterwards.
@–Rn	Register indirect with pre-decrement	A memory location. The value in Rn (after being decremented[*2]) gives the start address of the memory accessed. SH microprocessor first decrements Rn, and then uses that value for the memory reference.
@(disp,Rn)	Register indirect with displacement[*3]	A memory location. The start address of the memory access is given by: the value of Rn plus the displacement (disp). The value of Rn is not changed.
@(R0,Rn)	Register indirect with index	A memory location. The start address of the memory access is given by: the value of R0 plus the value of Rn. The values of R0 and Rn are not changed.
@(disp,GBR)	GBR indirect with displacement	A memory location. The start address of the memory access is given by: the value of GBR plus the displacement (disp). The value of GBR is not changed.

Notes 1 to 3: See next page.

Table 3-1 Addressing Modes (cont)

Addressing Mode	Name	Description
@(R0,GBR)	GBR indirect with index	A memory location. The start address of the memory access is given by: the value of GBR plus the value of R0. The values of GBR and R0 are not changed.
@(disp,PC)	PC relative with displacement	A memory location. The start address of the memory access is given by: the value of the PC plus the displacement (disp).
symbol	PC relative specified with symbol	[When used as the operand of a branch instruction] The symbol directly indicates the destination address. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol − PC.
		[When used as the operand of a data move instruction] A memory location. The symbol expresses the starting address of the memory accessed. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol − PC.
#imm	Immediate	Expresses a constant.

Note:

Rn	A general register (R0 to R15)*4
SR	The status register
VBR	The vector base register
PR	The procedure register
R0	The general register R0 (when only R0 can be specified)
GBR	The global base register
MACH, MACL	The accumulator register
PC	The program counter

Notes:

1. Increment
   The amount of the increment is 1 when the operation size is a byte, 2 when the operation size is a word, and 4 when the operation size is a long word.
2. Decrement
   The amount of the decrement is 1 when the operation size is a byte, 2 when the operation size is a word, and 4 when the operation size is a long word.
3. Displacement
   A displacement is the distance between 2 points. In this assembly-language, the unit of displacement values is in bytes.
4. Concerning R15
   R15 can also be specified by the symbol SP. SP is an abbreviation for stack pointer, a pointer to the stack area.

The values that can be used for the displacement vary with the addressing mode and the operation size.

Table 3-2 Allowed Displacement Values

Addressing Mode	Displacement*
@(disp,Rn)	When the operation size is byte (B): H'00000000 to H'0000000F (0 to 15) When the operation size is word (W): H'00000000 to H'0000001F (0 to 31) When the operation size is long word (L): H'00000000 to H'0000003F (0 to 63)
@(disp,GBR)	When the operation size is byte (B): H'00000000 to H'000000FF (0 to 255) When the operation size is word (W): H'00000000 to H'000001FF (0 to 511) When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)
@(disp,PC)	When the operation size is word (W): H'00000000 to H'000001FF (0 to 511) When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)
symbol	[When used as a branch instruction operand] When used as an operand for a conditional branch instruction (BT or BF): H'00000000 to H'000000FF (0 to 255) H'FFFFFF00 to H'FFFFFFFF (–256 to -1) When used as an operand for an unconditional branch instruction (BRA, BSR, JMP, JSR, or RTS) H'00000000 to H'00000FFF (0 to 4095) H'FFFFF000 to H'FFFFFFFF (–4096 to -1) [When used as the operand of a data move instruction] When the operation size is word (W): H'00000000 to H'000001FF (0 to 511) When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)

Note: * Units are bytes, numbers in parentheses are decimal.

The values that can be used for immediate values vary with the executable instruction.

Table 3-3 Allowed Immediate Values

Executable Instruction	Immediate Value	
TST, AND, OR, XOR	H'00000000 to H'000000FF	(0 to 255)
MOV	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (–128 to -1) *
ADD, CMP/EQ	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (–128 to -1) *
TRAPA	H'00000000 to H"000000FF	(0 to 255)

Note: * Values in the range H'FFFFFF80 to H'FFFFFFFF can be written as positive decimal values.

CAUTION!

The assembler corrects the value of displacements under certain conditions.

Condition		Type of Correction
When the operation size is a word and the displacement is not a multiple of 2	→	The lower bit of the displacement is discarded, resulting in the value being a multiple of 2.
When the operation size is a long word and the displacement is not a multiple of 4	→	The lower 2 bits of the displacement are discarded, resulting in the value being a multiple of 4.
When a displacement of the branch instruction is not a multiple of 2	→	The lower bit of the displacement is discarded, resulting in the value being a multiple of 2.

Be sure to take this correction into consideration when using operands of the mode @(disp,Rn), @(disp,GBR), and @(disp,PC).

Example: `MOV.L @(63,R0)`

The assembler corrects the 63 to be 60, and generates object code identical to that for the statement MOV.L @(60,R0), and warning number 870 occurs.

3.2 Notes on Executable Instructions

3.2.1 Notes on the Operation Size

The operation sizes that can be specified vary with the mnemonic and the addressing mode combination. Table 3-4 shows the allowable executable instruction and operation size combinations.

Table 3-4 Executable Instruction and Operation Size Combinations (part 1)

1. Data Move Instructions		**Operation Sizes**				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**	
MOV	#imm,Rn	O	Δ	Δ	B	
MOV	@(disp,PC),Rn	×	O	O	L	
MOV	symbol,Rn	×	O	O	L	
MOV	Rn,Rm	×	×	O	L	
MOV	Rn,@Rm	O	O	O	L	
MOV	@Rn,Rm	O	O	O	L	
MOV	Rn,@–Rm	O	O	O	L	
MOV	@Rn+,Rm	O	O	O	L	
MOV	R0,@(disp,Rn)	O	O	O	L	
MOV	Rn,@(disp,Rm)	×	×	O	L	*1
MOV	@(disp,Rn),R0	O	O	O	L	
MOV	@(disp,Rn),Rm	×	×	O	L	*2
MOV	Rn,@(R0,Rm)	O	O	O	L	
MOV	@(R0,Rn),Rm	O	O	O	L	
MOV	R0,@(disp,GBR)	O	O	O	L	
MOV	@(disp,GBR),R0	O	O	O	L	
MOVA	@(disp,PC),R0	×	×	O	L	
MOVA	symbol,R0	×	×	O	L	
MOVT	Rn	×	×	O	L	
SWAP	Rn,Rm	O	O	×	W	
XTRCT	Rn,Rm	×	×	O	L	

Notes: 1. In this case Rn must be one of R1 to R15.
2. In this case Rm must be one of R1 to R15.

Table 3-4 Executable Instruction and Operation Size Combinations (part 1) (cont)

Symbol meanings:

Rn, Rm	A general register (R0 to R15)
SR	The status register
VBR	The vector base register
PR	The procedure register
R0	The general register R0 (when only R0 can be specified)
GBR	The global base register
MACH, MACL	The accumulator register
PC	The program counter
imm	An immediate value
symbol	A symbol
disp	A displacement value
B	Byte
L	Long word (4 bytes)
W	Word (2 bytes)
O	Valid specification
×	Invalid specification: The assembler regards instructions with this combination as the specification being omitted.
Δ	The assembler regards them as extended instructions.

References:

Extended Instructions

→ Programmer's Guide, 8.2, "Extended Instructions Related to Automatic Literal Pool Generation"

Table 3-4 **Executable Instruction and Operation Size Combinations (part 2)**

2. Arithmetic Operation Instructions **Operation Sizes**

Mnemonic	Addressing Mode	B	W	L	Default when Omitted
ADD	Rn,Rm	×	×	○	L
ADD	#imm,Rn	×	×	○	L
ADDC	Rn,Rm	×	×	○	L
ADDV	Rn,Rm	×	×	○	L
CMP/EQ	#imm,R0	×	×	○	L
CMP/EQ	Rn,Rm	×	×	○	L
CMP/HS	Rn,Rm	×	×	○	L
CMP/GE	Rn,Rm	×	×	○	L
CMP/HI	Rn,Rm	×	×	○	L
CMP/GT	Rn,Rm	×	×	○	L
CMP/PZ	Rn	×	×	○	L
CMP/PL	Rn	×	×	○	L
CMP/STR	Rn,Rm	×	×	○	L
DIV1	Rn,Rm	×	×	○	L
DIVOS	Rn,Rm	×	×	○	L
DIVOU	(no operands)	×	×	×	—
EXTS	Rn,Rm	○	○	×	W
EXTU	Rn,Rm	○	○	×	W
MAC	@Rn+,@Rm+	×	○	×	W
MULS	Rn,Rm	×	×	○	L
MULU	Rn,Rm	×	×	○	L
NEG	Rn,Rm	×	×	○	L
NEGC	Rn,Rm	×	×	○	L
SUB	Rn,Rm	×	×	○	L
SUBC	Rn,Rm	×	×	○	L
SUBV	Rn,Rm	×	×	○	L

Table 3-4 Executable Instruction and Operation Size Combinations (part 3)

3. Logic Operation Instructions — **Operation Sizes**

Mnemonic	Addressing Mode	B	W	L	Default when Omitted
AND	Rn,Rm	×	×	○	L
AND	#imm,R0	×	×	○	L
AND	#imm,@(R0,GBR)	○	×	×	B
NOT	Rn,Rm	×	×	○	L
OR	Rn,Rm	×	×	○	L
OR	#imm,R0	×	×	○	L
OR	#imm,@(R0,GBR)	○	×	×	B
TAS	@Rn	○	×	×	B
TST	Rn,Rm	×	×	○	L
TST	#imm,R0	×	×	○	L
TST	#imm,@(R0,GBR)	○	×	×	B
XOR	Rn,Rm	×	×	○	L
XOR	#imm,R0	×	×	○	L
XOR	#imm,@(R0,GBR)	○	×	×	B

Table 3-4 Executable Instruction and Operation Size Combinations (part 4)

4. Shift Instructions **Operation Sizes**

Mnemonic	Addressing Mode	B	W	L	Default when Omitted
ROTL	Rn	×	×	○	L
ROTR	Rn	×	×	○	L
ROTCL	Rn	×	×	○	L
ROTCR	Rn	×	×	○	L
SHAL	Rn	×	×	○	L
SHAR	Rn	×	×	○	L
SHLL	Rn	×	×	○	L
SHLR	Rn	×	×	○	L
SHLL2	Rn	×	×	○	L
SHLR2	Rn	×	×	○	L
SHLL8	Rn	×	×	○	L
SHLR8	Rn	×	×	○	L
SHLL16	Rn	×	×	○	L
SHLR16	Rn	×	×	○	L

Table 3-4 Executable Instruction and Operation Size Combinations (part 5)

5. Branch Instructions **Operation Sizes**

Mnemonic	Addressing Mode	B	W	L	Default when Omitted
BF	symbol	×	×	×	—
BT	symbol	×	×	×	—
BRA	symbol	×	×	×	—
BSR	symbol	×	×	×	—
JMP	@Rn	×	×	×	—
JSR	@Rn	×	×	×	—
RTS	(no operands)	×	×	×	—

Table 3-4 Executable Instruction and Operation Size Combinations (part 6)

6. System Control Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
CLRT	(no operands)	×	×	×	—
CLRMAC	(no operands)	×	×	×	—
LDC	Rn,SR	×	×	○	L
LDC	Rn,GBR	×	×	○	L
LDC	Rn,VBR	×	×	○	L
LDC	@Rn+,SR	×	×	○	L
LDC	@Rn+,GBR	×	×	○	L
LDC	@Rn+,VBR	×	×	○	L
LDS	Rn,MACH	×	×	○	L
LDS	Rn,MACL	×	×	○	L
LDS	Rn,PR	×	×	○	L
LDS	@Rn+,MACH	×	×	○	L
LDS	@Rn+,MACL	×	×	○	L
LDS	@Rn+,PR	×	×	○	L
NOP	(no operands)	×	×	×	—
RTE	(no operands)	×	×	×	—
SETT	(no operands)	×	×	×	—
SLEEP	(no operands)	×	×	×	—
STC	SR,Rn	×	×	○	L
STC	GBR,Rn	×	×	○	L
STC	VBR,Rn	×	×	○	L
STC	SR,@–Rn	×	×	○	L
STC	GBR,@–Rn	×	×	○	L
STC	VBR,@–Rn	×	×	○	L
STS	MACH,Rn	×	×	○	L
STS	MACL,Rn	×	×	○	L
STS	PR,Rn	×	×	○	L
STS	MACH,@–Rn	×	×	○	L
STS	MACL,@–Rn	×	×	○	L
STS	PR,@–Rn	×	×	○	L
TRAPA	#imm	×	×	○	L

3.2.2 Notes on Delayed Branch Instructions

The unconditional branch instructions (BRA, BSR, JMP, JSR, RTS, and RTE) are delayed branch instructions. SH microprocessors execute the delay slot instruction (the instruction directly following a branch instruction in memory) before executing the delayed branch instruction.

If an instruction inappropriate for a delay slot is specified, the assembler issues error number 150.

Table 3-5 shows the relationship between the delayed branch instruction and the delay slot instructions.

Table 3-5 Relationship between Delayed Branch Instruction and Delay Slot Instructions

		Delayed Branch					
Delay Slot		**BRA**	**BSR**	**JMP**	**JSR**	**RTS**	**RTE**
BF		×	×	×	×	×	×
BT		×	×	×	×	×	×
BRA		×	×	×	×	×	×
BSR		×	×	×	×	×	×
JMP		×	×	×	×	×	×
JSR		×	×	×	×	×	×
RTS		×	×	×	×	×	×
RTE		×	×	×	×	×	×
TRAPA		×	×	×	×	×	×
MOV	@(disp,PC),Rn	Δ	Δ	Δ	Δ	Δ	Δ
	symbol,Rn	Δ	Δ	×	×	×	×
MOVA	@(disp,PC),R0	Δ	Δ	Δ	Δ	Δ	Δ
	symbol,R0	Δ	Δ	×	×	×	×
Extended	MOV.L #imm,Rn	×	×	×	×	×	×
instructions	MOV.W #imm,Rn	×	×	×	×	×	×
	MOVA #imm,R0	×	×	×	×	×	×
Any other instruction		○	○	○	○	○	○

Symbol meanings:

○ .... Normal, i.e., the assembler generates the specified object code.

Δ .... Warning 871
Note on the value of PC: PC = <destination address for the delayed branch instruction> + 2
The assembler generates the specified object code.

× ..... Error 150 or 151
The instruction specified is inappropriate as a delay slot instruction.
The assembler generates object code with a NOP instruction (H'0009) in the object code.

CAUTION!

If the delayed branch instruction and the following instruction are coded in different sections, the assembler does not check the delay slot instruction.

References:

Extended Instructions

→ Programmer's Guide, 8.2, "Extended Instructions Related to Automatic Literal Pool Generation"

3.2.3 Notes on Address Calculations

When the operand addressing mode is PC relative with displacement, i.e., @(disp,PC), the value of PC must be taken into account in coding. The value of PC can vary depending on certain conditions.

(1) Normal Case

The value of PC is the first address in the currently executing instruction plus 4 bytes.

Examples:
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

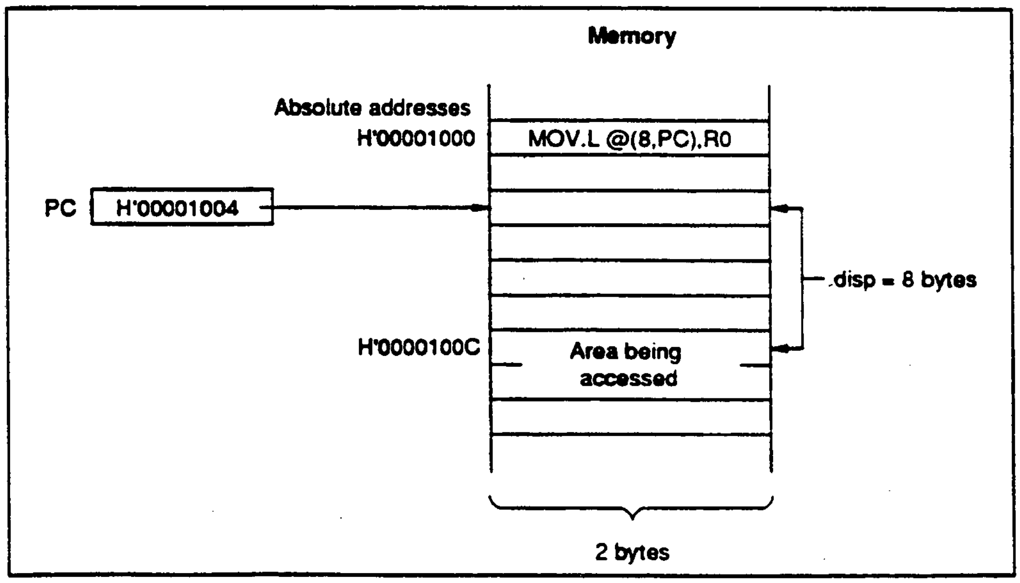


Figure 3-1 Address Calculation Example (normal case)

(2) During the Delay Slot Instruction

The value of PC is destination address for the delayed branch instruction plus 2 bytes.

Examples:
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

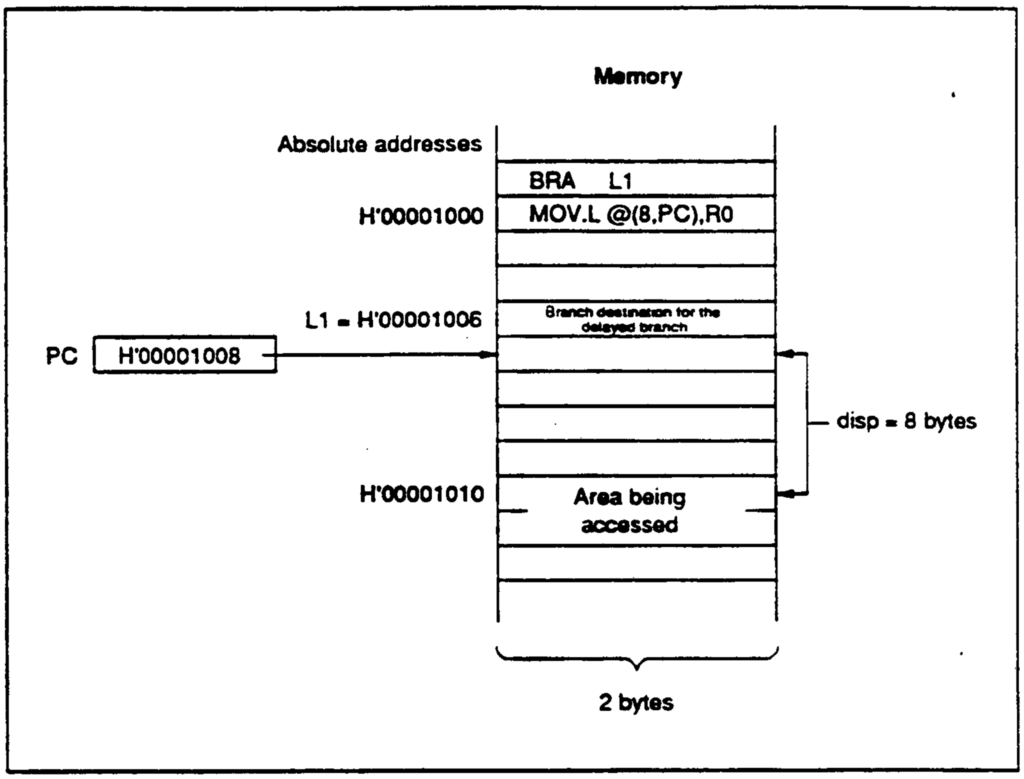


Figure 3-2 Address Calculation Example (when the value of PC differs due to a branch)

Supplement:

When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

(3) During the Execution of Either a MOV.L @(disp,PC),Rn or a MOVA @(disp,PC),R0

When the value of PC is not a multiple of 4 SH microprocessors correct the value by discarding the lower 2 bits when calculating addresses.

Examples: 1. When SH microprocessor corrects the value of PC.
(Consider the state when a MOV instruction is being executed at absolute address H'00001002.)

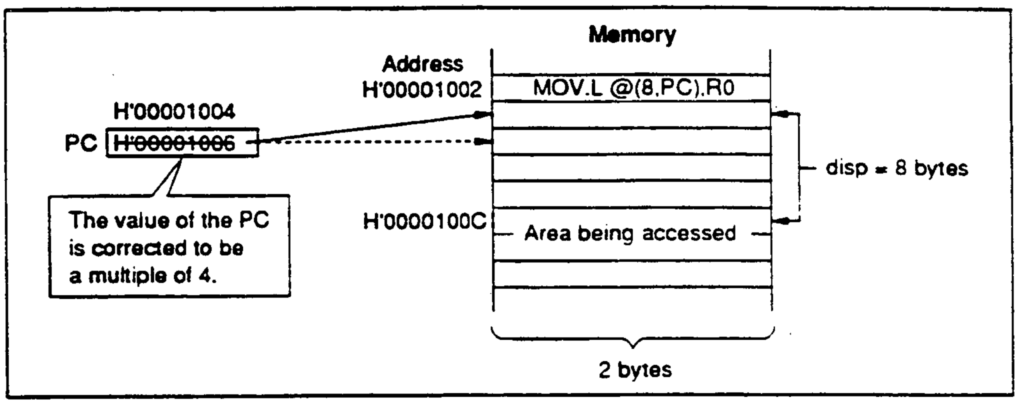


Figure 3-3 Address Calculation Example (when SH microprocessor corrects the value of PC)

2. When SH microprocessor does not correct the value of PC.
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

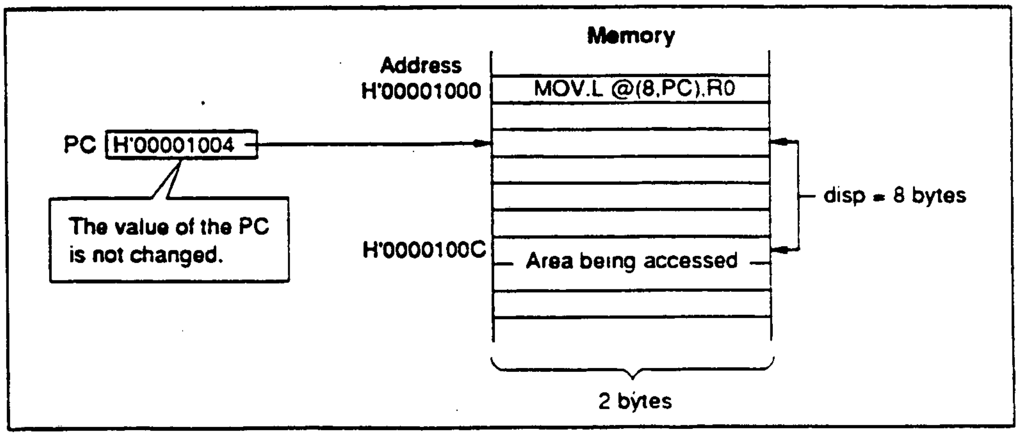


Figure 3-4 Address Calculation Example (when SH microprocessor does not correct the value of PC)

Supplement:

When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

Section 4 Assembler Directives

4.1 Overview of the Assembler Directives

The assembler directives are instructions that the assembler interprets and executes. Table 4-1 lists the assembler directives provided by this assembler.

Table 4-1 Assembler Directives

Type	Mnemonic	Function
Section and the location counter	.SECTION	Declares a section.
	.ORG	Sets the value of the location counter.
	.ALIGN	Corrects the value of the location counter.
Symbols	.EQU	Sets a symbol value (reset not allowed).
	.ASSIGN	Sets a symbol value (reset allowed).
	.REG	Defines the alias of a register name.
Data and data area reservation	.DATA	Reserves integer data.
	.DATAB	Reserves integer data blocks.
	.SDATA	Reserves character string data.
	.SDATAB	Reserves character string data blocks.
	.SDATAC	Reserves character string data (with length).
	.SDATAZ	Reserves character string data (with zero terminator).
	.RES	Reserves data area.
	.SRES	Reserves character string data area.
	.SRESC	Reserves character string data area (with length).
	.SRESZ	Reserves character string data area (with zero terminator).
Export and import symbol	.EXPORT	Declares export symbols.
	.IMPORT	Declares import symbols.
	.GLOBAL	Declares export and import symbols.
Object modules	.OUTPUT	Controls object module output.
	.DEBUG	Controls the output of symbolic debug information.

Table 4-1 Assembler Directives (cont)

Type	Mnemonic	Function
Assemble listing	.PRINT	Controls assemble listing output.
	.LIST	Controls the output of the source program listing.
	.FORM	Sets the number of lines and columns in the assemble listing.
	.HEADING	Sets the header for the source program listing.
	.PAGE	Inserts a new page in the source program listing.
	.SPACE	Outputs blank lines to the source program listing.
Other directives	.PROGRAM	Sets the name of the object module.
	.RADIX	Sets the radix in which integer constants with no radix specifier are interpreted.
	.END	Declares the end of the source program.

4.2 Assembler Directive Reference

4.2.1 Section and Location Counter Assembler Directives

This assembler provides the following assembler directives concerned with sections and the location counter.

`.SECTION`

Declares a section.

`.ORG`

Sets the value of the location counter.

`.ALIGN`

Adjusts the value of the location counter to a multiple of the boundary alignment value.

.SECTION

Section Declaration

Syntax

```
.SECTIONΔ<section name> [,<section attribute> [,{LOCATE=
<start address>,ALIGN=<boundary alignment value>}]]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .SECTION mnemonic in the operation field.

3. Operands

   a. First operand: the section name

      The rules for section names are the same as the rules for symbols.

      References: Naming sections
      → Programmer's Guide, 1.3.2, "Coding of Symbols"

   b. Second operand: the section attribute

Attribute	Section Type
CODE	Code section
DATA	Data section
STACK	Stack section
COMMON	Common section
DUMMY	Dummy section

The shaded section indicates the default value when the specifier is omitted.

The section usage type is determined by the attribute specification.
When the specification is omitted, the section will be a code section.

c. Third operand: start address or boundary alignment value

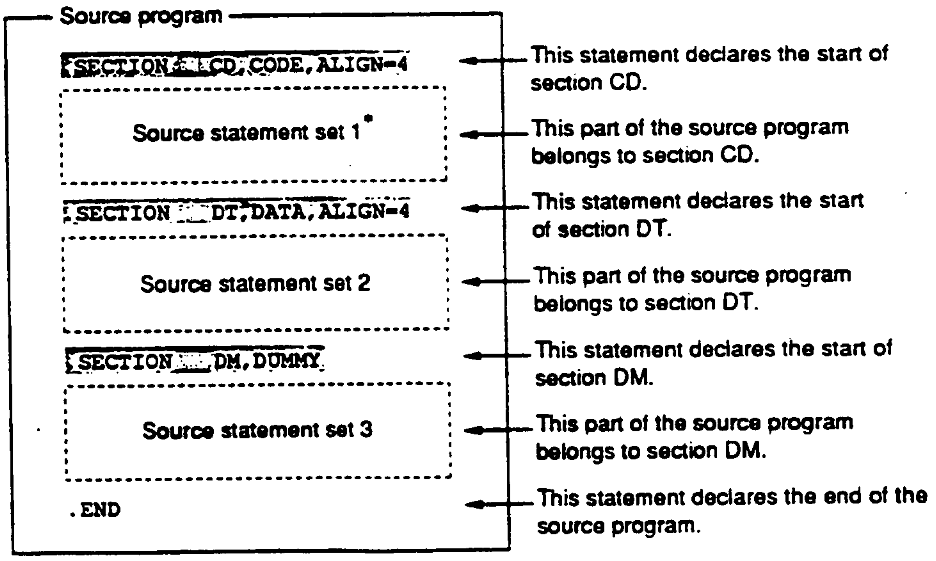
Specification	Section Type
LOCATE = <start address>	Absolute address section
ALIGN = <boundary alignment value>	Relative address section

The specification determines whether the section type will be an absolute address section or a relative address section. When the specification is omitted, the section will be a relative address section with boundary alignment value of 4.

Description

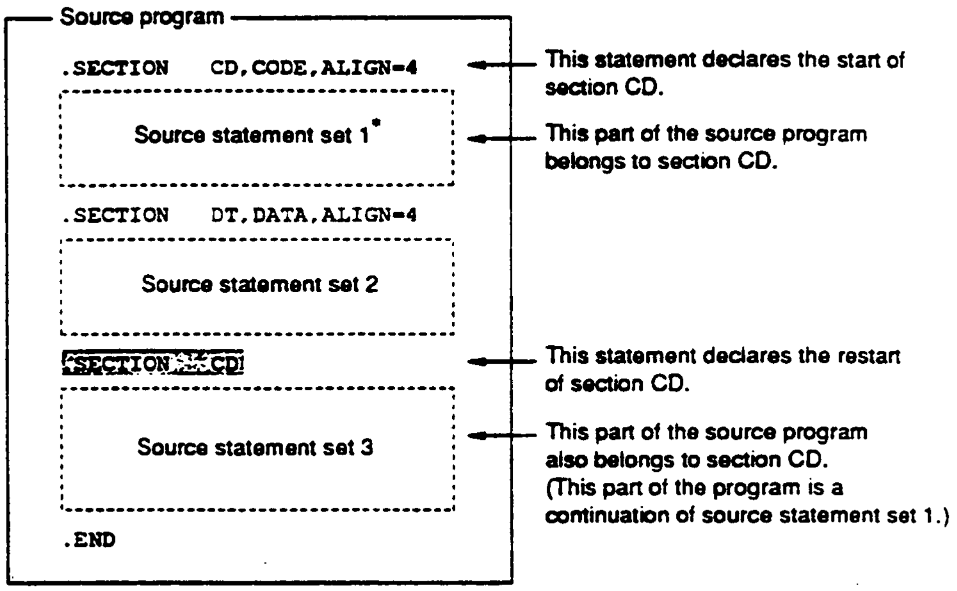
1. .SECTION is the section declaration assembler directive.

   A section is a part of a program, and the linkage editor regards it as a unit of processing. The following describes section declaration using the simple examples shown below.


Source program
.SECTION CD,CODE,ALIGN=4
Source statement set 1*
.SECTION DT,DATA,ALIGN=4
Source statement set 2
.SECTION DM,DUMMY
Source statement set 3
.END
This statement declares the start of section CD.
This part of the source program belongs to section CD.
This statement declares the start of section DT.
This part of the source program belongs to section DT.
This statement declares the start of section DM.
This part of the source program belongs to section DM.
This statement declares the end of the source program.

Note: * This example assumes that the .SECTION assembler directive does not appear in any of the source statement sets 1 to 3.

2. It is possible to redeclare (and thus restart, i.e., re-enter) a section that was previously declared in the same file. The following is a simple example of section restart.



```
.SECTION    CD,CODE,ALIGN=4
      Source statement set 1*
.SECTION    DT,DATA,ALIGN=4
      Source statement set 2
.SECTION    CD
      Source statement set 3
.END
```

Source program
This statement declares the start of section CD.
This part of the source program belongs to section CD.
This statement declares the restart of section CD.
This part of the source program also belongs to section CD. (This part of the program is a continuation of source statement set 1.)

Note: * This example assumes that the .SECTION assembler directive does not appear in any of the source statement sets 1 to 3.

CAUTION!

When using the .SECTION assembler directive to restart a section, the second and third operands must be omitted. (The original specifications when first declaring the section remain valid.)

3. Use LOCATE = <start address> as the third operand when starting an absolute address section. The start address is the absolute address of the start of that section.

   The start address must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

The values allowed for the start address are from H'00000000 to H'FFFFFFFF. (From −2,147,483,648 to 4,294,967,295 in decimal.)

4. Use ALIGN = <boundary alignment value> to start a relative address section. The linkage editor will adjust the start address of the section to be a multiple of the boundary alignment value.

   The boundary alignment value must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the boundary alignment value are powers of 2, e.g. 2^0, 2^1, 2^2, ..., 2^{31}. For code sections, the values must be 4 or larger powers of 2, e.g. 2^2, 2^3, 2^4, ..., 2^{31}.

5. The assembler provides a default section for the following cases.

   - The use of executable instructions when no section has been declared.
   - The use of data reservation assembler directives when no section has been declared.
   - The use of the .ALIGN assembler directive when no section has been declared.
   - The use of the .ORG assembler directive when no section has been declared.
   - Reference to the location counter when no section has been declared.
   - The use of statements consisting of only the label field when no section has been declared.

   The default section is the following section.

   - Section name: P
   - Section type: Code section
     Relative address section (with a boundary alignment value of 4)

Coding Example

```
 ┌──────────────────────────────────────┐
 │         .ALIGN      4                │  ; This section of the program belongs to the default section P.
 │         .DATA.L     H'11111111       │  ; The default section P is a code section, and is a relative
 │           ~                          │  ; address section with a boundary alignment value of 4.
 └──────────────────────────────────────┘
  .SECTION CD,CODE,ALIGN=4
 ┌──────────────────────────────────────┐
 │         MOV         R0,R1            │  ; This section of the program belongs to the section CD.
 │         MOV         R0,R2            │  ; The section CD is a code section, and is a relative address
 │           ~                          │  ; section with a boundary alignment value of 4.
 └──────────────────────────────────────┘
  .SECTION DT,DATA,LOCATE=H'00001000
 ┌──────────────────────────────────────┐
 │ X1:     .DATA.L     H'22222222       │  ; This section of the program belongs to the section DT.
 │         .DATA.L     H'33333333       │  ; The section DT is a data section, and is an absolute address
 │           ~                          │  ; section with a start address of H'00001000.
 └──────────────────────────────────────┘
         .END
```

Note: This example assumes the .SECTION assembler directive does not appear in the parts indicated by "~".

Location-Counter-Value Setting

Syntax

```
.ORGΔ<location-counter-value>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .ORG mnemonic in the operation field.

3. Operands

   Enter the new value for the location counter in the operand field.

Description

1. .ORG is an assembler directive that sets the value of the location counter. The .ORG assembler directive is used to place executable instructions or data at a specific address.

2. The location-counter-value must be specified as follows:

   - The specification must be an absolute value or an address within the section, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the location-counter-value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

   When the location-counter-value is specified with an absolute value, the following condition must hold:

   <location-counter-value> ≥ <section start address> (when compared as unsigned values)

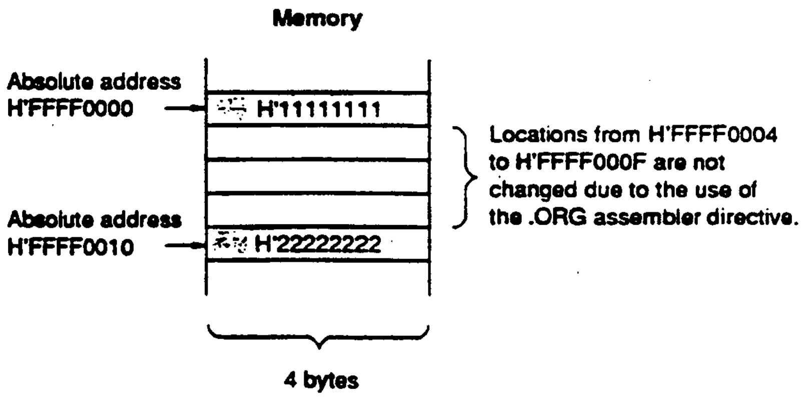
3. The assembler handles the value of the location counter as follows.

   - The value is regarded as an absolute address value within an absolute address section.
   - The value is regarded as a relative address value within a relative address section.

Coding Example

```
        .SECTION    DT,DATA,LOCATE=H'FFFF0000
        .DATA.L     H'11111111

        .ORG        H'FFFF0010       ; This statement sets the value of the location
                                       counter.
        .DATA.L     H'22222222       ; The integer data H'22222222 is stored at
                                     ; absolute address H'FFFF0010.
           ~
```

Explanatory Figure for the Coding Example


Memory
Absolute address H'FFFF0000
H'11111111
Absolute address H'FFFF0010
H'22222222
Locations from H'FFFF0004 to H'FFFF000F are not changed due to the use of the .ORG assembler directive.
4 bytes

Location-Counter-Value Correction

Syntax

```
.ALIGNΔ<boundary alignment value>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .ALIGN mnemonic in the operation field.

3. Operands

   Enter the boundary alignment value (the basis for adjusting the location-counter-value) in the operand field.

Description

1. .ALIGN is an assembler directive that corrects the location-counter-value to be a multiple of the boundary alignment value. Executable instructions and data can be allocated on specific boundary values (address multiples) by using the .ALIGN assembler directive.

2. The boundary alignment value must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the boundary alignment value are powers of 2, e.g. $2^0, 2^1, 2^2, ..., 2^{31}$.

   The boundary alignment value specified by .ALIGN directive must be less than or equal to the boundary alignment value specified by .SECTION directive.

3. When .ALIGN is used in a code section, data section, or dummy section, the assembler inserts NOP instructions in the object code* to adjust the value of the location counter. Odd byte size areas are filled with H'09.

When .ALIGN is used in a dummy or stack section, the assembler only adjusts the value of the location counter, and does not fill in any object code in memory.

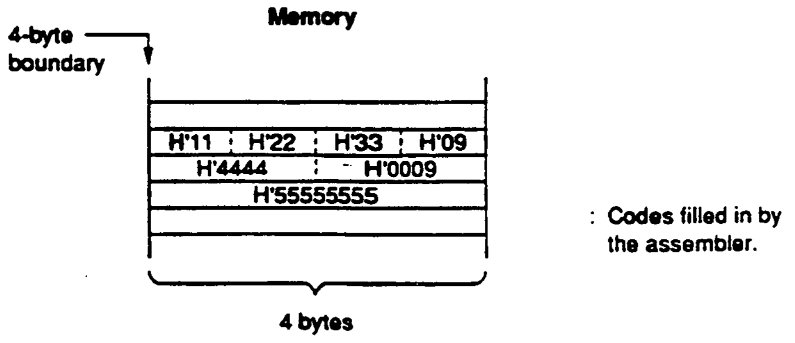
Note: * This object code is not displayed in the assemble listing.

Coding Example

```
       ~

       .DATA.B    H'11
       .DATA.B    H'22
       .DATA.B    H'33

       .ALIGN     2             ; This statement adjusts the value of the location
       .DATA.W    H'4444        ; counter to be a multiple of 2.

       .ALIGN     4             ; This statement adjusts the value of the location
       .DATA.L    H'55555555    ; counter to be a multiple of 4.

       ~
```

Explanatory Figure for the Coding Example

This example assumes that the byte sized integer data H'11 is originally located at the 4-byte boundary address. The assembler will insert the filler data as shown in the figure below.



4.2.2 Symbol Handling Assembler Directives

This assembler provides the following assembler directives concerned with symbols.

`.EQU`

Sets a symbol value.

`.ASSIGN`

Sets and resets a symbol value.

`.REG`

Defines the alias of a register name.

.EQU

Symbol Value Setting (resetting not allowed)

Syntax

```
<symbol>[:]Δ.EQUΔ<symbol value>
```

Statement Elements

1. Label

   Enter the symbol that is to be set a value in the label field.

2. Operation

   Enter the .EQU mnemonic in the operation field.

3. Operands

   Enter the value to be set to the symbol in the operand field.

Description

1. .EQU is an assembler directive that sets a value to a symbol.

   Symbols defined with the .EQU directive cannot be redefined.

2. The symbol value must be specified as follows:

   - The specification must be an absolute value or an address value, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the symbol value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

Coding Example

```
          ~

X1:     .EQU      10          ; The value 10 is set to X1.
X2:     .EQU      20          ; The value 20 is set to X2.

        CMP/EQ    #X1,R0      ; This is the same as CMP/EQ #10,R0.
        BT        LABEL1
        CMP/EQ    #X2,R0      ; This is the same as CMP/EQ #20,R0.
        BT        LABEL2

          ~
```

.ASSIGN

Symbol Value Setting (resetting allowed)

Syntax

```
<symbol>[:]Δ.ASSIGNΔ<symbol value>
```

Statement Elements

1. Label

   Enter the symbol that is to be set a value in the label field.

2. Operation

   Enter the .ASSIGN mnemonic in the operation field.

3. Operands

   Enter the value to be set to the symbol in the operand field.

Description

1. .ASSIGN is an assembler directive that sets a value to a symbol.

   Symbols defined with the .ASSIGN directive can be redefined with the .ASSIGN directive.

2. The symbol value must be specified as follows:

   - The specification must be an absolute value or an address value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the symbol value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

3. Definitions with the .ASSIGN directive are valid from the point of the definition forward in the program.

4. Symbols defined with .ASSIGN have the following limitations:

   - They cannot be used as export or import symbols.
   - They cannot be referenced from the simulator/debugger.

Coding Example

```
            ~

X1:     .ASSIGN   1
X2:     .ASSIGN   2
         CMP/EQ   #X1,R0       ; This is the same as CMP/EQ #1,R0.
         BT       LABEL1
         CMP/EQ   #X2,R0       ; This is the same as CMP/EQ #2,R0.
         BT       LABEL2

            ~

X1:     .ASSIGN   3
X2:     .ASSIGN   4
         CMP/EQ   #X1,R0       ; This is the same as CMP/EQ #3,R0.
         BT       LABEL3
         CMP/EQ   #X2,R0       ; This is the same as CMP/EQ #4,R0.
         BF       LABEL4

            ~
```

.REG

Alias of a Register Name Definition

Syntax

```
<symbol>[:]Δ.REGΔ(<register name>)
```

Statement Elements

1. Label

   Enter the symbol to be defined as the alias of a register name in the label field.

2. Operation

   Enter the .REG mnemonic in the operation field.

3. Operands

   Enter the register name for which the alias of a register name is being defined in the operand field.

Description

1. .REG is the assembler directive that defines the alias of a register name.

   The alias of a register name defined with .REG can be used in exactly the same manner as the original register name.

   The alias of a register name defined with .REG cannot be redefined.

2. The alias of a register name can only be defined for the general registers (R0 to R15, and SP).

3. Definitions with the .REG directive are valid from the point of the definition forward in the program.

4. Symbols defined with .REG have the following limitations:

   - They cannot be used as export or import symbols.
   - They cannot be referenced from the simulator/debugger.

Coding Example

```
          ~

MIN:    .REG        (R10)
MAX:    .REG        (R11)

        MOV         #0,MIN      ; This is the same as MOV #0,R10.
        MOV         #99,MAX     ; This is the same as MOV #99,R11.

        CMP/HS      MIN,R1
        BF          LABEL
        CMP/HS      R1,MAX
        BF          LABEL

          ~
```

4.2.3 Data and Data Area Reservation Assembler Directives

This assembler provides the following assembler directives that are concerned with data and data area reservation.

`.DATA`	Reserves integer data.
`.DATAB`	Reserves integer data blocks.
`.SDATA`	Reserves character string data.
`.SDATAB`	Reserves character string data blocks.
`.SDATAC`	Reserves character string data (with length).
`.SDATAZ`	Reserves character string data (with zero terminator).
`.RES`	Reserves data area.
`.SRES`	Reserves character string data area.
`.SRESC`	Reserves character string data area (with length).
`.SRESZ`	Reserves character string data area (with zero terminator).

Integer Data Reservation

Syntax

```
[<symbol>[:]]Δ.DATA[.<operation size>]Δ<integer data>[,<integer data>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .DATA mnemonic in the operation field.

   b. Operation size

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

The shaded section indicates the default value when the specifier is omitted.

The specifier determines the size of the reserved data.
The long word size is used when the specifier is omitted.

3. Operands

   Enter the values to be reserved as data in the operand field.

Description

1. .DATA is the assembler directive that reserves integer data in memory.
2. Arbitrary values, including relative values and forward reference symbols, can be used to specify the integer data.

3. The range of values that can be specified as integer data varies with the operation size.

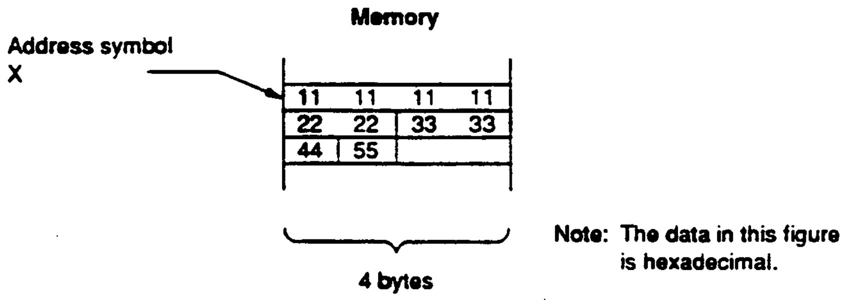
Operation Size	Integer Data Range*	
B	H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to -1)
W	H'00000000 to H'0000FFFF	(0 to 65,535)
	H'FFFF8000 to H'FFFFFFFF	(-32,768 to -1)
L	H'00000000 to H'7FFFFFFF	(0 to 4,294,967,295)
	H'80000000 to H'FFFFFFFF	(–2,147,483,648 to -1)

Note: * Numbers in parentheses are decimal.

Coding Example

```
          ~
        .ALIGN     4                 ; (This statement adjusts the value of the
                                       location counter.)
X:      .DATA.L    H'11111111        ;
        .DATA.W    H'2222,H'3333     ; These statements reserve integer data.
        .DATA.B    H'44,H'55         ;
          ~
```

Explanatory Figure for the Coding Example



Integer Data Block Reservation

Syntax

```
[<symbol>[:]]Δ.DATAB[.<operation size>]Δ<block count>,<integer data>
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .DATAB mnemonic in the operation field.

   b. Operation size

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L ▨	Long word (4 bytes)

*The shaded section indicates the default value when the specifier is omitted.

   The specifier determines the size of the reserved data.
   The long word size is used when the specifier is omitted.

3. Operands

   a. First operand: block count
   Enter the number of times the data value is repeated as the first operand.

   b. Second operand: integer data
   Enter the value to be reserved as the second operand.

Description

1. .DATAB is the assembler directive that reserves the specified number of integer data consecutively in memory.

2. The block count must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   Arbitrary values, including relative values and forward reference symbols, can be used to specify the integer data.

3. The range of values that can be specified as the block size and as the integer data varies with the operation size.

Operation Size	Block Size Range*	
B	H'00000001 to H'FFFFFFFF	(1 to 4,294,967,295)
W	H'00000001 to H'7FFFFFFF	(1 to 2,147,483,647)
L	H'00000001 to H'3FFFFFFF	(1 to 1,073,741,823)

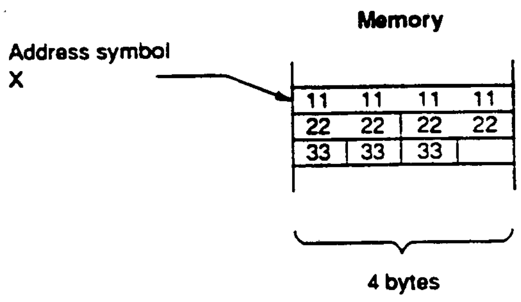
Operation Size	Integer Data Range*	
B	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (−128 to -1)
W	H'00000000 to H'0000FFFF H'FFFF8000 to H'FFFFFFFF	(0 to 65,535) (−32,768 to -1)
L	H'00000000 to H'7FFFFFFF H'80000000 to H'FFFFFFFF	(0 to 4,294,967,295) (−2,147,483,648 to -1)

Note: * Numbers in parentheses are decimal.

Coding Example

```
            ~
        .ALIGN      4               ; (This statement adjusts the value of the
                                    ; location counter.)
X:      .DATAB.L    1,H'11111111    ;
        .DATAB.W    2,H'2222        ; This statement reserves two blocks of integer
        .DATAB.B    3,H'33          ; data.
            ~
```

Explanatory Figure for the Coding Example



Note: The data in this figure is hexadecimal.

```
.SDATA
```

Character String Data Reservation

Syntax

```
[<symbol>[:]]Δ.SDATAΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATA mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATA is the assembler directive that reserves character string data in memory.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

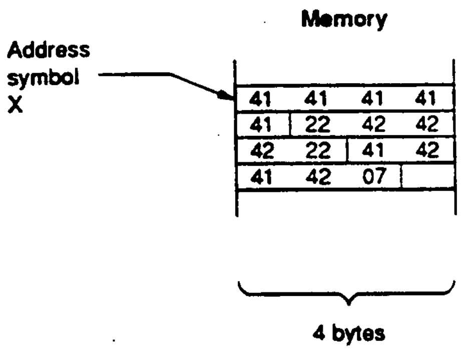
   The ASCII code for a control character must be specified as follows.

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

Coding Example

```
        ~
        .ALIGN     4              ; (This statement adjusts the value of
                                  ; the location counter.)
X:      .SDATA     "AAAAA"        ; This statement reserves character string data.
        .SDATA     """BBB"""      ; The character string in this example includes
                                  ; double quotation marks.
        .SDATA     "ABAB"<H'07>   ; The character string in this example has
                                  ; a control character appended.
        ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for """ is: H'22.

`.SDATAB`

Character String Data Blocks Reservation

Syntax

```
[<symbol>[:]]Δ.SDATABΔ<block count>,"<character string>"
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAB mnemonic in the operation field.

3. Operands

   a. First operand: <block count>
   Enter the number of character strings as the first operand.

   b. Second operand: <character string>
   Enter the character string to be reserved as the second operand.

Description

1. .SDATAB is the assembler directive that reserves the specified number of character strings consecutively in memory.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The <block count> must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   A value of 1 or larger must be specified as the block count.

   The maximum value of the block count depends on the length of the character string data.

(The length of the character string data multiplied by the block count must be less than or equal to HFFFFFFFF (4,294,967,295) bytes.)

3. A control character can be appended to a character string.

   The syntax for this notation is as follows.

```
"<character string>"<<ASCII code for a control character>>
```

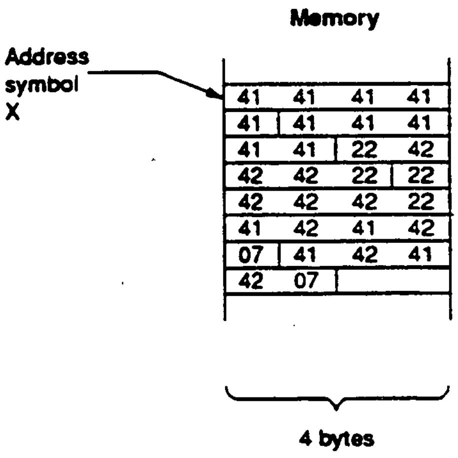
   The ASCII code for a control character must be specified as follows.

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

Coding Example

```
        ~
        .ALIGN     4               ; (This statement adjusts the value of the
                                   ; location counter.)
X:      .SDATAB    2,"AAAAA"       ; This statement reserves two character string
                                   ; data blocks.
        .SDATAB    2,"""BBB"""     ; The character string in this example includes
                                   ; double quotation marks.
        .SDATAB    2,"ABAB"<H'07>  ; The character string in this example has
                                   ; a control character appended.
        ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for """ is: H'22.

Character String Data Reservation (with length)

Syntax

```
[<symbol>[:]]Δ.SDATACΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAC mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATAC is the assembler directive that reserves character string data (with length) in memory.

   A character string with length is a character string with an inserted leading byte that indicates the length of the string.

   The length indicates the size of the character string (not including the length) in bytes.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings".

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

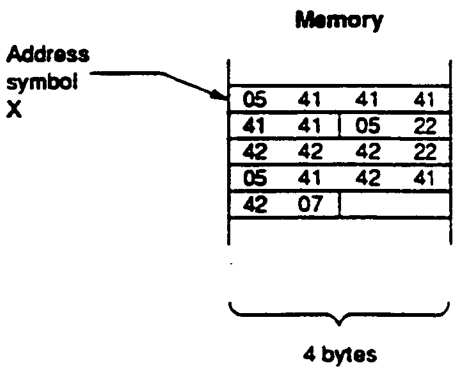
   The ASCII code for a control character must be specified as follows.

- The specification must be an absolute value,
  and,
- Forward reference symbols must not appear in the specification.

Coding Example

```
        ~
        .ALIGN      4               ; (This statement adjusts the value of the
                                    ; location counter.)
X:      .SDATAC     "AAAAA"         ; This statement reserves character string data
                                    ; (with length).
        .SDATAC     """"BBB""""     ; The character string in this example includes
                                    ; double quotation marks.
        .SDATAC     "ABAB"<H'07>    ; The character string in this example has
                                    ; a control character appended.
        ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for """ is: H'22.

Character String Data Reservation (with zero terminator)

Syntax

```
[<symbol>[:]]Δ.SDATAZΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAZ mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATAZ is the assembler directive that reserves character string data (with zero terminator) in memory.

   A character string with zero terminator is a character string with an appended trailing byte (with the value H'00) that indicates the end of the string.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

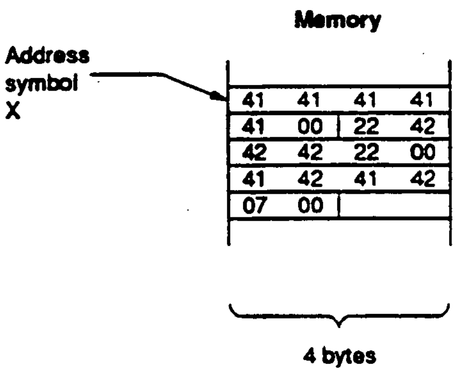
   The ASCII code for a control character must be specified as follows.

- The specification must be an absolute value,
  and,
- Forward reference symbols must not appear in the specification.

Coding Example

```
              ~

        .ALIGN      4             ; (This statement adjusts the value of the
                                  ; location counter.)
X:      .SDATAZ     "AAAAA"       ; This statement reserves character string
                                  ; data (with zero terminator).
        .SDATAZ     """BBB"""     ; The character string in this example includes
                                  ; double quotation marks.
        .SDATAZ     "ABAB"<H'07>  ; The character string in this example has
                                  ; a control character appended.
              ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
   The ASCII code for "B" is: H'42.
   The ASCII code for """ is: H'22.

Data Area Reservation

Syntax

```
[<symbol>[:]]Δ.RES[.<operation size>]Δ<area count>
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .RES mnemonic in the operation field.

   b. Operation size

Specifier	**Data Size**
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

   The shaded section indicates the default value when the specifier is omitted.

   The specifier determines the size of one area.
   The long word size is used when the specifier is omitted.

3. Operands

   Enter the number of areas to be reserved in the operand field.

.RES

Description

1. .RES is the assembler directive that reserves data areas in memory.

2. The area count must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

3. The range of values that can be specified as the area count varies with the operation size.

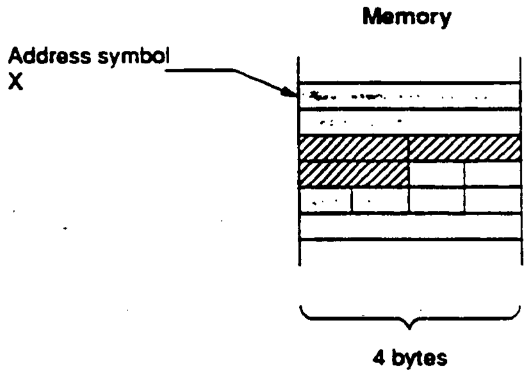
Operation Size	Area Count Range*	
B	H'00000001 to H'FFFFFFFF	(1 to 4,294,967,295)
W	H'00000001 to H'7FFFFFFF	(1 to 2,147,483,647)
L	H'00000001 to H'3FFFFFFF	(1 to 1,073,741,823)

Note: * Numbers in parentheses are decimal.

Coding Example

```
          ~
        .ALIGN      4           ; (This statement adjusts the value of the location
                                ; counter.)
X:      .RES.L      2           ; This statement reserves 2 long word size areas.
        .RES.W      3           ; This statement reserves 3 word size areas.
        .RES.B      5           ; This statement reserves 5 byte size areas.
          ~
```

Explanatory Figure for the Coding Example



```
.SRES
```

Character String Data Area Reservation

Syntax

```
[<symbol>[:]]Δ.SRESΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRES mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas to be reserved in the operand field.

Description

1. .SRES is the assembler directive that reserves character string data areas.

2. The character string area size must be specified as follows:

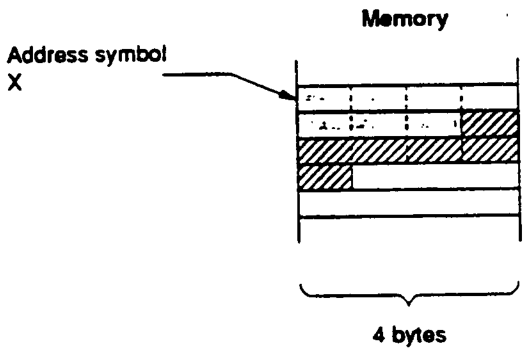
   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values that are allowed for the character string area size are from H'00000001 to H'FFFFFFFF (from 1 to 4,294,967,295 in decimal).

Coding Example

```
           ~
          .ALIGN    4        ; (This statement adjusts the value of the location
                             ; counter.)
X:        .SRES     7        ; This statement reserves a 7-byte area.
          .SRES     6        ; This statement reserves a 6-byte area.
           ~
```

Explanatory Figure for the Coding Example



.SRESC

Character String Data Area Reservation (with length)

Syntax

```
[<symbol>[:]]Δ.SRESCΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRESC mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas (not including the length) to be reserved in the operand field.

Description

1. .SRESC is the assembler directive that reserves character string data areas (with length) in memory.

   A character string with length is a character string with an inserted leading byte that indicates the length of the string.

   The length indicates the size of the character string (not including the length) in bytes.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The character string area size must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

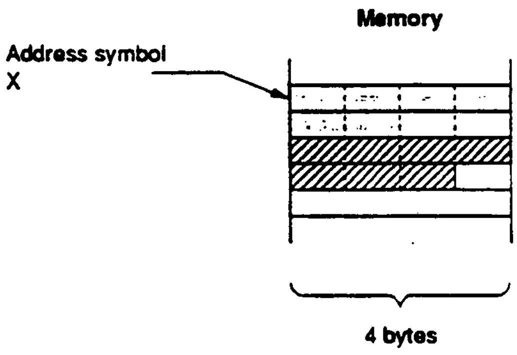
   The values that are allowed for the character string area size are from H'00000000 to H'000000FF (in decimal, from 0 to 255).

3. The size of the area reserved in memory is the size of the character string area itself plus 1 byte for the count.

Coding Example

```
          ~
          .ALIGN    4          ; (This statement adjusts the value of the location
                               ; counter.)
X:        .SRESC    7          ; This statement reserves 7 bytes plus 1 byte for
                               ; the count.
          .SRESC    6          ; This statement reserves 6 bytes plus 1 byte for
                               ; the count.
          ~
```

Explanatory Figure for the Coding Example



```
.SRESZ
```

Character String Data Area Reservation (with zero terminator)

Syntax

```
[<symbol>[:]]Δ.SRESZΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRESZ mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas (not including the terminating zero) to be reserved in the operand field.

Description

1. .SRESZ is the assembler directive that allocates character string data areas (with zero termination).

   A character string with length is a character string with an appended trailing byte (with the value H'00) that indicates the end of the string.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The character string area size must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

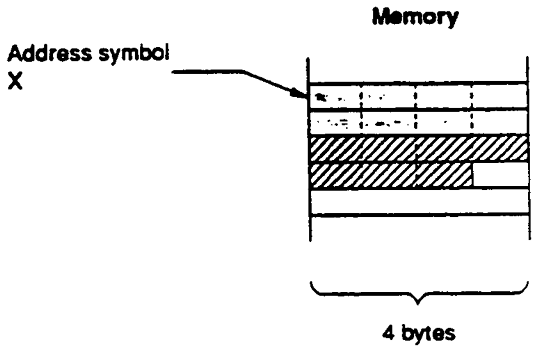
   The values that are allowed for the character string area size are from H'00000000 to H'000000FF (in decimal, from 0 to 255).

3. The size of the area reserved in memory is the size of the character string area itself plus 1 byte for the terminating zero.

Coding Example

```
          ~
        .ALIGN     4        ; (This statement adjusts the value of the location counter.)
X:      .SRESZ     7        ; This statement reserves 7 bytes plus 1 byte for
                            ; the terminating byte.
        .SRESZ     6        ; This statement reserves 6 bytes plus 1 byte for
                            ; the terminating byte.
          ~
```

Explanatory Figure for the Coding Example



4.2.4 Export and Import Assembler Directives

This assembler provides the following assembler directives concerned with export and import.

```
.EXPORT
```

Declares export symbols.

This declaration allows symbols defined in the current file to be referenced in other files.

```
.IMPORT
```

Declares import symbols.

This declaration allows symbols defined in other files to be referenced in the current file.

```
.GLOBAL
```

Declares export and import symbols.

This declaration allows symbols defined in the current file to be referenced in other files, and allows symbols defined in other files to be referenced in the current file.

Export Symbols Declaration

Syntax

```
.EXPORTΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXPORT mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as export symbols in the operand field.

Description

1. .EXPORT is the assembler directive that declares export symbols.

   An export symbol declaration is required to reference symbols defined in the current file from other files.

2. The following can be declared to be export symbols.

   - Constant symbols (other than those defined with the .ASSIGN assembler directive)
   - Absolute address symbols (other than address symbols in a dummy section)
   - Relative address symbols

3. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Import symbols are declared in the file in which they are referenced using either the . IMPORT or the .GLOBAL assembler directive.

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
        .EXPORT   MAX             ; This statement declares X to be an export
                                  ; symbol.
           ~
X:      .EQU      H'10000000      ; This statement defines X.
           ~
```

File B:

```
        .IMPORT   X               ; This statement declares X to be an import
                                  ; symbol.
           ~
        .ALIGN    4
        .DATA.L   X               ; This statement references X.
           ~
```

Import Symbols Declaration

Syntax

```
.IMPORTΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .IMPORT mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as import symbols in the operand field.

Description

1. .IMPORT is the assembler directive that declares import symbols.

   An import symbol declaration is required to reference symbols defined in another file.

2. Symbols defined in the current file cannot be declared to be import symbols.

3. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Export symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL assembler directive.

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
        .EXPORT   X                  ; This statement declares X to be an export
                                     ; symbol.
          ~

X:      .EQU      H'10000000         ; This statement defines X.

          ~
```

File B:

```
        .IMPORT   X                  ; This statement declares X to be an import
                                     ; symbol.
          ~

        .ALIGN    4
        .DATA.L   X                  ; This statement references X.

          ~
```

Export and Import Symbols Declaration

Syntax

```
.GLOBALΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .GLOBAL mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as export symbols or as import symbols in the operand field.

Description

1. .GLOBAL is the assembler directive that declares symbols to be either export symbols or import symbols.

   An export symbol declaration is required to reference symbols defined in the current file from other files. An import symbol declaration is required to reference symbols defined in another file.

2. A symbol defined within the current file is declared to be an export symbol by a .GLOBAL declaration.

   A symbol that is not defined within the current file is declared to be an import symbol by a . GLOBAL declaration.

3. The following can be declared to be export symbols.

   - Constant symbols (other than those defined with the .ASSIGN assembler directive)

- Absolute address symbols (other than address symbols in a dummy section)
- Relative address symbols

4. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Export symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL assembler directive.

   Import symbols are declared in the file in which they are referenced using either the . IMPORT or the .GLOBAL assembler directive.

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
          .GLOBAL   X                 ; This statement declares X to be an export
                                      ; symbol.
            ~

X:        .EQU      H'10000000        ; This statement defines X.

            ~
```

File B:

```
          .GLOBAL   X                 ; This statement declares X to be an import
                                      ; symbol.
            ~

          .ALIGN    4
          .DATA.L   X                 ; This statement references X.

            ~
```

4.2.5 Object Module Assembler Directives

This assembler provides the following assembler directives concerned with object modules.

$\boxed{\texttt{.OUTPUT}}$

Controls object module and debug information output.

$\boxed{\texttt{.DEBUG}}$

Controls the output of symbolic debug information.

```
.OUTPUT
```

Object Module Output Control

Syntax

```
.OUTPUTΔ<output specifier>[,<output specifier>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .OUTPUT mnemonic in the operation field.

3. Operands: <output specifier>

Output Specifier	Output Control
OBJ	An object module is output.
NOOBJ	No object module is output.
DBG	Debug information is output in the object module.
NODBG	No debug information is output in the object module.

The shaded section indicates the default value when the specifier is omitted.

The output specifiers control object module and debug information output.

Description

1. .OUTPUT is the assembler directive that controls object module and debug information output.

2. If the .OUTPUT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

Example:

```
    ~
 .OUTPUT  OBJ
 .OUTPUT  NODBG     ← OK
    ~
```

```
    ~
 .OUTPUT  OBJ
 .OUTPUT  NOOBJ     ← Error
    ~
```

3. Specifications concerning debug information output are only valid when an object module is output.

4. The assembler gives priority to command line option specifications concerning object module and debug information output.

   References: Object module output
   → User's Guide, 2.2.1, "Object Module Command Line Options" OBJECT NOOBJECT

   Debug information output
   → User's Guide, 2.2.1, "Object Module Command Line Options" DEBUG NODEBUG

Coding Example

Note: This example and its description assume that no command line options concerning object module or debug information output were specified.

```
        .OUTPUT    OBJ                ; An object module is output.
                                      ; No debug information is output.
           ~
-----------------------------------------------------------------------
        .OUTPUT    OBJ,DBG            ; Both an object module and debug information
                                      ; is output.
           ~
-----------------------------------------------------------------------
        .OUTPUT    OBJ,NODBG          ; An object module is output.
                                      ; No debug information is output.
           ~
```

Supplement:

Debug information is required when debugging a program using the simulator/debugger, and is part of the object module.

Debug information includes information about source statements and information about symbols.

Symbolic Debug Information Output Control

Syntax

```
.DEBUGΔ<output specifier>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .DEBUG mnemonic in the operation field.

3. Operands: output specifier

Output Specifier	Output Control
ON ▨	Symbolic debug information is output starting with the next source statement.
OFF	Symbolic debug information is not output starting with the next source statement.

The shaded section indicates the default value when the specifier is omitted.

The output specifier controls symbolic debug information output.

Description

1. .DEBUG is the assembler directive that controls the output of symbolic debug information.

   This directive allows assembly time to be reduced by restricting the output of symbolic debug information to only those symbols required in debugging.

2. The specification of the .DEBUG directive is only valid when both an object module and debug information are output.

References: Object module output

→ Programmer's Guide, 4.2.5, "Object Module Assembler Directives", .OUTPUT

→ User's Guide, 2.2.1 "Object Module Command Line Options" OBJECT NOOBJECT

Debug information output

→ Programmer's Guide 4.2.5, "Object Module Assembler Directives", .OUTPUT

→ User's Guide, 2.2.1, "Object Module Command Line Options" DEBUG NODEBUG

Coding Example

```
          ~

        .DEBUG    OFF              ; Starting with the next statement, the assembler
                                   ; does not output symbolic debug information.
          ~

        .DEBUG    ON               ; Starting with the next statement, the assembler
                                   ; outputs symbolic debug information.
          ~
```

Supplement:

The term "symbolic debug information" refers to the parts of debug information concerned with symbols.

4.2.6 Assemble Listing Assembler Directives

This assembler provides the following assembler directives for controlling the assemble listing.

`.PRINT`	Controls assemble listing output.
`.LIST`	Controls the output of the source program listing.
`.FORM`	Sets the number of lines and columns in the assemble listing.
`.HEADING`	Sets the header for the source program listing.
`.PAGE`	Inserts a new page in the source program listing.
`.SPACE`	Outputs blank lines to the source program listing.

Supplement:

The assemble listing is a listing to which the results of the assembly are output, and includes a source program listing, a cross-reference listing, and a section information listing.

References: For a detailed description of the assemble listing, see appendix C, "Assemble Listing Output Example".

.PRINT

Assemble Listing Output Control

Syntax

```
.PRINTΔ<output specifier>[,<output specifier>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PRINT mnemonic in the operation field.

3. Operands: output specifier

Output Specifier	Assembler Action
LIST	An assemble listing is output.
NOLIST	No assemble listing is output.
SRC	A source program listing is output in the assemble listing.
NOSRC	No source program listing is output in the assemble listing.
CREF	A cross-reference listing is output in the assemble listing.
NOCREF	No cross-reference listing is output in the assemble listing.
SCT	A section information listing is output in the assemble listing.
NOSCT	No section information listing is output in the assemble listing.

The shaded sections indicate the default settings when the specifier is omitted.

The output specifier controls assemble listing output.

Description

1. .PRINT is the assembler directive that controls assemble listing output.

2. If the .PRINT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

   Example:

```
  ~
.PRINT  LIST
.PRINT  NOSRC    ← OK
  ~
```

```
  ~
.PRINT  LIST
.PRINT  NOLIST   ← Error
  ~
```

3. The output specifiers concerned with the source program listing, the cross-reference listing, and the section information listing are only valid when an assemble listing is output.

4. The assembler gives priority to command line option specifications concerning assemble listing output.

   References: Assemble listing output
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options"
   LIST NOLIST
   SOURCE NOSOURCE
   CROSS_REFERENCE NOCROSS_REFERENCE
   SECTION NOSECTION

Coding Example

Note: This example and its description assume that no command line options concerning assemble listing output are specified.

```
        .PRINT    LIST                      ; All types of assemble listing are output.
          ~
----------------------------------------------------------------
        .PRINT    LIST,NOSRC,NOCREF
                                            ; Only a section information listing is output.
          ~
```

```
.LIST
```

Source Program Listing Output Control

Syntax

```
.LISTΔ<output specifier>[,<output specifier>...]

 Output type: {ON|OFF|COND|NOCOND|DEF|NODEF|CALL|NOCALL|EXP|
              NOEXP|CODE|NOCODE}
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .LIST mnemonic in the operation field.

3. Operands

   Enter the output specifiers in the operand filed.

Description

1. .LIST is the assembler directive that controls output of the source program listing in the following three ways:

   I Selects whether or not to output source statements.
   II Selects whether or not to output source statements related to the conditional assembly and macro functions.
   III Selects whether or not to output object code lines.

2. Output is controlled by output specifiers as follows:

Type	Output Specifier: Output	Output Specifier: Not output	Object	Description
I	ON	OFF	Source statements	The source statements following this directive
II	COND	NOCOND	Failed condition	Condition-failed .AIF directive statements
	DEF	NODEF	Definition	Macro definition statements .AREPEAT and .AWHILE definition statements .INCLUDE directive statements .ASSIGNA and .ASSSIGNC directive statements
	CALL	NOCALL	Call	Macro call statements, .AIF and .AENDI directive statements
	EXP	NOEXP	Expansion	Macro expansion statements .AREPEAT and .AWHILE expansion statements
III	CODE	NOCODE	Object code lines	The object code lines exceeding the source statement lines

The shaded sections indicate the default settings when the specifier is omitted.

3. The specification of the .LIST directive is only valid when an assemble listing is output.

References: Source program listing output
→ Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST SOURCE NOSOURCE

3. The assembler gives priority to command line option specifications concerning source program listing output.

References: Output on the source program listing
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options" SHOW NOSHOW

4. .LIST directive statements themselves are not output on the source program listing.

Coding Example

```
        - .LIST NOCOND,NODEF              -------- This statement controls source program
          .MACRO  SHLRN    COUNT,Rd       ------┐  listing output.
SHIFT     .ASSIGNA \COUNT                       ¦
          .AIF \&SHIFT GE 16                    ¦
          SHLR16  \Rd                           ¦
SHIFT     .ASSIGNA \&SHIFT-16                   ¦
          .AENDI                                ¦
          .AIF \&SHIFT GE 8                     ¦
          SHLR8    \Rd                          ¦
SHIFT     .ASSIGNA \&SHIFT-8                    ¦
          .AENDI                                ¦
          .AIF \&SHIFT GE 4                     ¦  These statements define a general-
          SHLR2    \Rd                          ¦  purpose multiple-bit shift proedure as a
          SHLR2    \Rd                          ¦  macro instruction.
SHIFT     .ASSIGNA \&SHIFT-4                    ¦
          .AENDI                                ¦
          .AIF \&SHIFT GE 2                     ¦
          SHLR2    \Rd                          ¦
SHIFT     .ASSIGNA \&SHIFT-2                    ¦
          .AENDI                                ¦
          .AIF \&SHIFT GE 1                     ¦
          SHLR     \Rd                          ¦
          .AENDI                                ¦
          .ENDM                           ------┘
          SHLRN     23,R0                 -------- Macro call
          .END
```

Note: This example and its description assume that no command line options concerning source program listing output are specified.

Source Listing Output of Coding Example

The .LIST assembler directive suppresses the output of the macro definition, .ASSIGNA directive statement, and .AIF condition-failed statements.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***     07/09/93 16:33:49
                                               PAGE     1
PROGRAM NAME =
   31                          31
   32                          32           SHLRN    23,R0
   33                            M
   35                            M
   36                            M          .AIF 23 GE 16
   37 00000000 4029              C          SHLR16   R0
   39                            M          .AENDI
   40                            M
   41                            M          .AIF 7 GE 8
   45                            M
   46                            M          .AIF 7 GE 4
   47 00000002 4009              C          SHLR2    R0
   48 00000004 4009              C          SHLR2    R0
   50                            M          .AENDI
   51                            M
   52                            M          .AIF 3 GE 2
   53 00000006 4009              C          SHLR2    R0
   55                            M          .AENDI
   56                            M
   57                            M          .AIF 1 GE 1
   58 00000008 4001              C          SHLR     R0
   59                            M          .AENDI
   60                          33           .END
  *****TOTAL ERRORS        0
  *****TOTAL WARNINGS      0
```

```
.FORM
```

Assemble Listing Line Count and Column Count Setting

Syntax

```
.FORMΔ<size specifier>[,<size specifier>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .FORM mnemonic in the operation field.

3. Operands: size specifier

Size Specifier	Listing Size
LIN=<line count>	The specified value is set to the number of lines per page.
COL=<column count>	The specified value is set to the number of columns per line.

   These specifications determine the number of lines and columns in the assemble listing.

Description

1. .FORM is the assembler directive that sets the number of lines per page and columns per line in the assemble listing.

2. The line count and column count must be specified as follows:

   - The specifications must be absolute values,
     and,
   - Forward reference symbols must not appear in the specifications.

   The values allowed for the line count are from 20 to 255.

   The values allowed for the column count are from 79 to 255.

3. The .FORM directive can be used any number of times in a given source program.

4. The assembler gives priority to command line option specifications concerning the number of lines and columns in the assemble listing.

   References: Setting the line count in assemble listing
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options" LINES

   Setting the column count in assemble listing
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options" COLUMNS

5. When there is no specification of command line option or .FORM assembler directive specification for the line count or the column count, the following values are used:

   - Line count............... 60 lines
   - Column count......... 132 columns

Coding Example

Note: This example and its description assume that no command line options concerning the assemble listing line count and/or column count are specified.

```
       ~

   .FORM    LIN=60, COL=200        ; Starting with this page, the number of lines
                                   ; per page in the assemble listing is 60 lines.
                                   ; Also, starting with this line, the number of
                                   ; columns per line in the assemble listing is
                                   ; 200 columns.
       ~

   .FORM    LIN=55, COL=150        ; Starting with this page, the number of lines
                                   ; per page in the assemble listing is 55 lines.
                                   ; Also, starting with this line, the number of
                                   ; columns per line in the assemble listing is
                                   ; 150 columns.
       ~
```

```
.HEADING
```

Source Program Listing Header Setting

Syntax

```
.HEADINGΔ"<character string>"
```

Statement Elements

1. **Label**

   The label field is not used.

2. **Operation**

   Enter the .HEADING mnemonic in the operation field.

3. **Operands: character string**

   Enter the header for the source program listing in the operand field.

Description

1. .HEADING is the assembler directive that sets the header for the source program listing.

   A character string of up to 60 characters can be specified as the header.

   References: Character strings
   → Programmer's Guide, 1.7, "Character Strings"

2. The .HEADING directive can be used any number of times in a given source program.

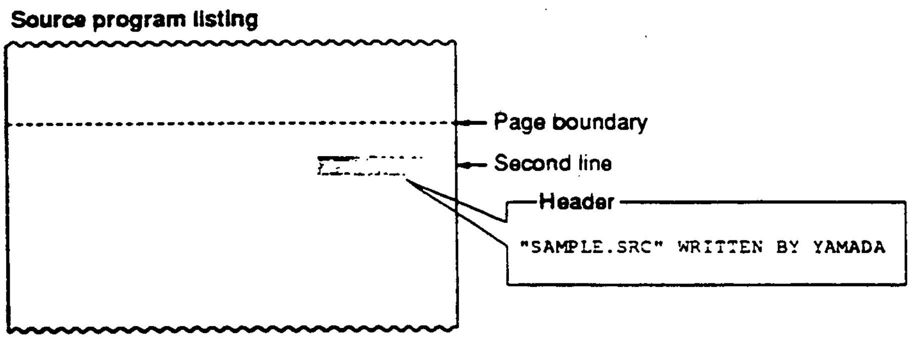
   The range of validity for a given use of the .HEADING directive is as follows:

   - When the .HEADING directive is on the first line of a page, it is valid starting with that page.
   - When the .HEADING directive appears on the second or later line of a page, it is valid starting with the next page.

Coding Example

```
    ~
  .HEADING   """SAMPLE.SRC"" WRITTEN BY YAMADA"
    ~
```

Explanatory Figure for the Coding Example



.PAGE

Source Program Listing New Page Insertion

Syntax

```
.PAGE
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PAGE mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .PAGE is the assembler directive that inserts a new page in the source program listing at an arbitrary point.

2. The .PAGE directive is ignored if it is used on the first line of a page.

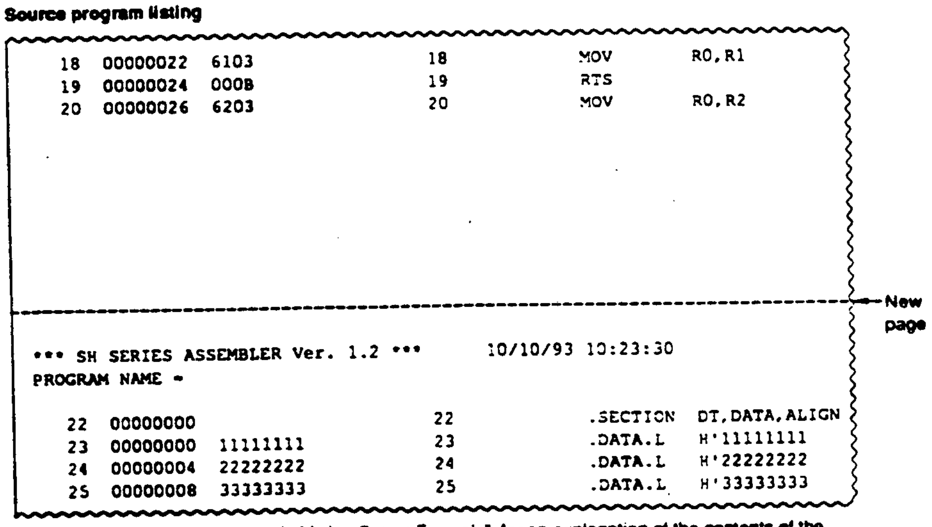
3. .PAGE directive statements themselves are not output to the source program listing.

Coding Example

```
      ~

MOV         R0,R1
RTS
MOV         R0,R2
.PAGE                                ; A new page is specified here since the
                                     ; section changes at this point.
.SECTION    DT,DATA,ALIGN=4
.DATA.L     H'11111111
.DATA.L     H'22222222
.DATA.L     H'33333333

      ~
```

Explanatory Figure for the Coding Example

Source program listing



```
   18  00000022  6103                    18            MOV         R0,R1
   19  00000024  000B                    19            RTS
   20  00000026  6203                    20            MOV         R0,R2

*** SH SERIES ASSEMBLER Ver. 1.2 ***          10/10/93 10:23:30
PROGRAM NAME =

   22  00000000                          22            .SECTION   DT,DATA,ALIGN
   23  00000000  11111111                23            .DATA.L    H'11111111
   24  00000004  22222222                24            .DATA.L    H'22222222
   25  00000008  33333333                25            .DATA.L    H'33333333
```

Note: See appendix C, "Assemble Listing Output Example", for an explanation of the contents of the source program listing.

```
.SPACE
```

Source Program Listing Blank Line Output

Syntax

```
.SPACE[Δ<line count>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .SPACE mnemonic in the operation field.

3. Operands: line count

   Enter the number of blank lines in the operand field.

   A single blank line is output if this operand is omitted.

Description

1. .SPACE is the assembler directive that outputs the specified number of blank lines to the source program listing. Nothing is output for the lines output by the .SPACE directive; in particular line numbers are not output for these lines.

2. The line count must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   Values from 1 to 50 can be specified as the line count.

3. When a new page occurs as the result of blank lines output by the .SPACE directive, any remaining blank lines are not output on the new page.

4. .SPACE directive statements themselves are not output to the source program listing.

Coding Example

```
        .SECTION    DT1,DATA,ALIGN=4
        .DATA.L     H'11111111
        .DATA.L     H'22222222
        .DATA.L     H'33333333
        .DATA.L     H'44444444              ; Inserts five blank lines at the point
        .SPACE  5                           ; where the section changes.
        .SECTION    DT2,DATA,ALIGN=4

            ~
```

Explanatory Figure for the Coding Example

Source program listing

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***       10/10/93 10:23:30
PROGRAM NAME =

    1  00000000                        1            .SECTION   DT1,DATA,ALIGN=4
    2  00000000  11111111              2            .DATA.L    H'11111111
    3  00000004  22222222              3            .DATA.L    H'22222222
    4  00000008  33333333              4            .DATA.L    H'33333333
    5  0000000C  44444444              5            .DATA.L    H'44444444





    6  00000000                        6            .SECTION   DT2,DATA,ALIGN=4

              ~
```

Note: See appendix C, "Assemble Listing Output Example", for an explanation of the contents of the source program listing.

4.2.7 Other Assembler Directives

This assembler provides the following additional assembler directives.

`.PROGRAM`

Sets the name of the object module.

`.RADIX`

Sets the radix in which integer constants with no radix specifier are interpreted.

`.END`

Declares the end of the source program.

Object Module Name Setting

Syntax

```
.PROGRAMΔ<object module name>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PROGRAM mnemonic in the operation field.

3. Operands: <object module name>

   Enter a name that identifies the object module in the operand field.

Description

1. .PROGRAM is the assembler directive that sets the object module name.

   The object module name is a name that is required by the H Series Linkage Editor or the H Series Librarian to identify the object module.

2. Object module naming conventions are the same as symbol naming conventions.

   The assembler distinguishes upper-case and lower-case letter in object module names.

   References: Coding of symbols
   → Programmer's Guide, 1.3.2, "Coding of Symbols"

3. Setting the object module name with the .PROGRAM directive is valid only once in a given program. (The assembler ignores the second and later specifications of the .PROGRAM directive.)

4. If there is no .PROGRAM specification of the object module name, the assembler will set a default (implicit) object module name.

   The default object module name is the file name of the object file (the object module output destination).

   $$\begin{tikzcd}[row sep=small] \text{Example: Object file name} \cdots\cdots\cdots & \boxed{\text{PROG}} \arrow[d, phantom, "\parallel"] & . & \boxed{\text{OBJ}} \arrow[d, phantom, "\parallel"] \\ & \text{File name} \arrow[d] & & \text{File format} \\ \text{Object module name} \cdots\cdots & \text{PROG} & & \end{tikzcd}$$

   References: User's Guide, 1.2, "File Specification Format"

5. The object module name can be the same as a symbol used in the program.

Coding Example

```
        .PROGRAM   PROG1      ; This statement sets the object module name to be
                              ; PROG1.
           ~
```

Default Integer Constant Radix Setting

Syntax

```
.RADIXΔ<radix specifier>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .RADIX mnemonic in the operation field.

3. Operands: radix specifier

Radix Specifier	Radix of Integer Constants with No Radix Specification
B	Binary
Q	Octal
D	Decimal
H	Hexadecimal

The shaded section indicates the default setting when the specifier is omitted.

   This specifier sets the radix (base) for integer constants with no radix specification.

Description

1. .RADIX is the assembler directive that sets the radix (base) for integer constants with no radix specification.

2. When there is no radix specification with the .RADIX directive in a program, integer constants with no radix specification are interpreted as decimal numbers.

3. If hexadecimal (radix specifier H) is specified as the radix for integer constants with no radix specification, integer constants whose first digit is A through F must be prefixed with a 0 (zero). (The assembler interprets expressions that begin with A through F to be symbols.)

4. Specifications with the .RADIX directive are valid from the point of specification forward in the program.

Coding Example

```
          ~
        .RADIX   D
X:      .EQU     100          ; This 100 is decimal.
          ~
        .RADIX   H
Y:      .EQU     64           ; This 64 is hexadecimal.
          ~
```

```
          ~
        .RADIX   H
Z:      .EQU     0F           ; A zero is prefixed to this constant "0F" since it would
                              ; be interpreted as a symbol if it were written as simply
                              ; "F".
          ~
```

Source Program End Declaration

Syntax

```
.END[Δ<start address>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .END mnemonic in the operation field.

3. Operands: start address

   Enter the start address for simulation in the operand field if required.

Description

1. .END is the assembler directive that declares the end of the source program.

   Assembly processing terminates at the point that the .END directive appears.

2. If a start address is specified with the .END directive in the operand field, the simulator/debugger starts simulation from that address.

3. The start address must be specified with either an absolute value or an address value.

4. The value of the start address must be an address in a code section.

```
.END
```

Coding Example

```
         .SECTION   CD,CODE,ALIGN=4
START:
             ~
         .END       START              ; This statement declares the end of the source
                                        ; program.

 ; The simulator/debugger starts simulation from the address indicated by the value of the
 ; symbol START.
```

Section 5 File Inclusion Function

The file inclusion function allows source files to be inserted into other source files at assembly time. The file inserted into another file is called an included file.

This assembler provides the .INCLUDE directive to perform file inclusion. The file specified with the .INCLUDE directive is inserted at the location of the .INCLUDE directive.

Example:

Source program

```
 .INCLUDE   "FILE.H"

 .SECTION CD1,CODE,ALIGN=4
 MOV #ON,R0

    ~
```

Included file FILE.H

```
ON:   .EQU  1
OFF:  .EQU  0
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

File included result (source list)

```
      .INCLUDE   "FILE.H"
ON:   .EQU  1
OFF:  .EQU  0

      .SECTION CD1,CODE,ALIGN=4
      MOV #ON,R0

       ~
```

.INCLUDE

File Inclusion

Syntax

```
.INCLUDEΔ"<file name>"
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .INCLUDE mnemonic in the operation field.

3. Operands

   Enter the file to be included.

Description

1. .INCLUDE is the file inclusion assembler directive.

2. If no file format is specified, only the file name is used as specified (the assembler does not assume any default file format).

   Reference: User's Guide, 1.2, "File Specification Format"

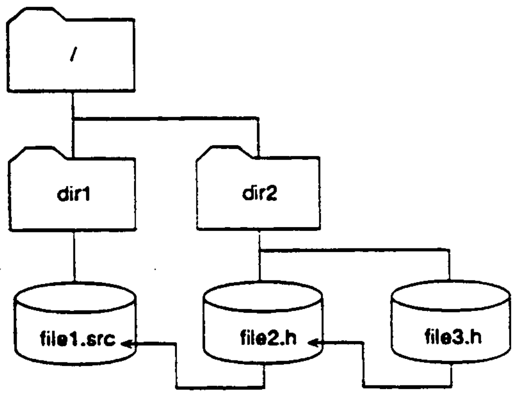
3. The file name can include the directory. The directory can be specified either by the absolute path (path from the route directory) or by the relative path (path from the current directory).

Note: The current directory for the .INCLUDE directive in a source file is the directory where the assembler is initiated. The current directory for the .INCLUDE directive in an included file is the directory where the included file exits.

4. Included files can include other files. The nesting depth for file inclusion is limited to eight levels (multiplex state).

Coding Example

This example assumes the following directory configuration and operations:



- Starts the assembler from the route directory (/)
- Inputs source file /dir1/file1.src
- Inserts file2.h in file1.src
- Inserts file3.h in file2.h

The start command is as follows:

```
/asmsh /dir1/file1.src (RET)
```

file1.src must have the following inclusion directive:

```
    .INCLUDE "dir2/file2.h"     ; / is the current directory (relative path specification).
or
    .INCLUDE "/dir2/file2.h"    ; Absolute path specification
```

file2.h must have the following inclusion directive:

```
    .INCLUDE "file3.h"          ; /dir2 is the current directory (relative path
                                  specification).
or
    .INCLUDE "/dir2/file3.h"    ; Absolute path specification
```

CAUTION!

When using MS-DOS, change the slash in the above example to a backslash (\).

(This page intentionally left blank.)

Section 6 Conditional Assembly Function

6.1 Overview of the Conditional Assembly Function

The conditional assembly function provides the following assembly operations:

- Selects whether or not to assemble a specified part of a source program according to the specified condition.
- Iteratively assembles a specified part of a source program.

6.1.1 Preprocessor variables

Preprocessor variables are used to write assembly conditions. Preprocessor variables are of either integer or character type.

1. Integer preprocessor variables

   Integer preprocessor variables are defined by the .ASSIGNA directive (these variables can be redefined).

   When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

Example:

```
FLAG:  .ASSIGNA 1
   ~
          .AIF \&FLAG EQ 1          ;  MOV R0,R1 is assembled
           MOV R0,R1                ;  when FLAG is 1.
          .AENDI
   ~
```

2. Character preprocessor variables

   Character preprocessor variables are defined by the .ASSIGNC directive (these variables can be redefined).

   When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

Example:

```
FLAG:  .ASSIGNC "ON"
   ~
          .AIF "\&FLAG" EQ "ON"     ;  MOV R0,R1 is assembled
           MOV R0,R1                ;  when FLAG is "ON".
          .AENDI
   ~
```

6.1.2 Conditional Assembly

The conditional assembly function determines whether or not to assemble a specified part of a source program according to the specified condition. A coding example is shown below.

```
      ~
    .AIF  <condition>
      <Statements to be assembled when the condition is satisfied>
   ┌-------------------------------------------------------------┐
   :.AELSE                                                        :
   : <Statements to be assembled when the condition is not satisfied>:
   └-------------------------------------------------------------┘
    .AENDI                          :
      ~                             :
                                    ▼
                  This part can be omitted from the coding.
```

Example:

```
      ~
    .AIF "\&FLAG" EQ "ON"
     MOV R0,R10                ;  These statements
     MOV R1,R11                ;  will be assembled
     MOV R2,R12                ;  when FLAG is "ON".
    .AELSE
     MOV R10,R0                ;  These statements
     MOV R11,R1                ;  will be assembled
     MOV R12,R2                ;  when FLAG is not "ON".
    .AENDI
      ~
```

6.1.3 Iterated Expansion

A part of a source program can be iteratively assembled the specified number of times. A coding example is shown below.

```
      ~
    .AREPEAT <count>
       <Statements to be iterated>
    .AENDR
      ~
```

Example:

```
                              ; This example is a division of 64-bit data by 32-bit data.
                              ; R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
    TST       R0,R0           ; Zero divisor check
    BT        zero_div
    CMP/HS    R0,R1           ; Overflow check
    BT        over_div
    DIV0U                     ; Flag initialization
    .AREPEAT  32
    ROTCL     R2              ; These statements are
    DIV1      R0,R1           ; iterated 32 times.
    .AENDR
    ROTCL     R2              ; R2 = quotient
```

6.1.4 Conditional Iterated Expansion

A part of a source program can be iteratively assembled while the specified condition is satisfied. A coding example is shown below.

```
          ~
        .AWHILE <condition>
           <Statements to be iterated>
        .AENDW
          ~
```

Example:

```
                                    ; This example is a multiply and accumulate
                                    ; operation.
TblSiz: .ASSIGNA  50                ; TblSiz: Data table size
        MOV       A_Tbl1,R1         ; R1: Start address of data table 1
        MOV       A_Tbl2,R2         ; R2: Start address of data table 2
        CLRMAC                      ; MAC register initialization
        .AWHILE   \&TblSize GT 0    ; While TblSiz is larger than 0,
        MAC.W     @R0+,@R1+         ; this statement is iteratively assembled.
TblSiz: .ASSIGNA  \&TblSiz-1        ; 1 is subtracted from TblSiz.
        .AENDW
        STS       MACL,R0           ; The result is obtained in R0.
```

6.2 Conditional Assembly Directives

This assembler provides the following conditional assembly directives.

`.ASSIGNA`	Defines an integer preprocessor variable. The defined variable can be redefined.
`.ASSIGNC`	Defines a character preprocessor variable. The defined variable can be redefined.

`.AIF`
`.AELSE`
`.AENDI`
	Determines whether or not to assemble a part of a source program according to the specified condition. When the condition is satisfied, the statements after the .AIF are assembled. When not satisfied, the statements after the .AELSE are assembled.

`.AREPEAT`
`.AENDR`
	Repeats assembly of a part of a source program (between .AREPEAT and .AENDR) the specified number of times.

`.AWHILE`
`.AENDW`
	Assembles a part of a source program (between .AWHILE and .AENDW) iteratively while the specified condition is satisfied.

`.EXITM`	Terminates .AREPEAT or .AWHILE iterated expansion.

.ASSIGNA

Integer Preprocessor Variable Definition (redefinition is possible)

Syntax

```
<preprocessor variable>[:].ASSIGNAΔ<value>
```

Statement Elements

1. Label

   Enter the name of the preprocessor variable.

2. Operation

   Enter the .ASSIGNA mnemonic in the operation field.

3. Operands

   Enter the value to be assigned to the preprocessor variable.

Description

1. .ASSIGNA is the assembler directive that defines a value for an integer preprocessor variable. The syntax of integer preprocessor variables is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

2. The preprocessor variables defined with the .ASSIGNA directive can be redefined with the .ASSIGNA directive.

3. The values for the preprocessor variables must be the following:

   - Constant (integer constant and character constant)
   - Defined preprocessor variable
   - Expression using the above as terms

4. Defined preprocessor variables are valid from the point of specification forward in the source program.

5. Defined preprocessor variables can be referenced in the following locations:

- .ASSIGNA directive
- .ASSIGNC directive
- .AIF directive
- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

```
\&<preprocessor variable>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

Coding Example

```
                                  ; This example generates a general-purpose multiple-bit
                                  ; shift instruction which shifts bits to the right by the
                                  ; number of SHIFT.
RN:      .REG       R0            ; R0 is set to Rn.
SHIFT:   .ASSIGNA   27            ; 27 is set to SHIFT

         .AIF \&SHIFT GE 16       ; Condition: SHIFT ≥ 16
         SHLR16  Rn               ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
SHIFT:   .ASSIGNA   \&SHIFT-16    ; 16 is subtracted from SHIFT.
         .AENDI

         .AIF \&SHIFT GE 8        ; Condition: SHIFT ≥ 8
         SHLR8   Rn               ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
SHIFT:   .ASSIGNA   \&SHIFT-8     ; 8 is subtracted from SHIFT.
         .AENDI

         .AIF \&SHIFT GE 4        ; Condition: SHIFT ≥ 4
         SHLR2   Rn               ; When the condition is satisfied, Rn is shifted to the right by 4 bits.
         SHLR2   Rn               ;
SHIFT:   .ASSIGNA   \&SHIFT-4     ; 4 is subtracted from SHIFT.
         .AENDI

         .AIF \&SHIFT GE 2        ; Condition: SHIFT ≥ 2
         SHLR2   Rn               ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
SHIFT:   .ASSIGNA   \&SHIFT-2     ; 2 is subtracted from SHIFT.
         .AENDI

         .AIF \&SHIFT EQ 1        ; Condition: SHIFT = 1
         SHLR      Rn             ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
         .AENDI
```

The expanded results are as follows:

```
         SHLR16   R1    ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
         SHLR8    R1    ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
         SHLR2    R1    ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
         SHLR1    R1    ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
```

Character Preprocessor Variable Definition (redefinition is possible)

Syntax

```
<preprocessor variable>[:].ASSIGNCΔ"<character string>"
```

Statement Elements

1. Label

   Enter the name of the preprocessor variable.

2. Operation

   Enter the .ASSIGNC mnemonic in the operation field.

3. Operands

   Enter the character string enclosed with double-quotation marks (").

Description

1. .ASSIGNC is the assembler directive that defines a character string for an character preprocessor variable. The syntax of character preprocessor variables is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

2. The preprocessor variables defined with the .ASSIGNC directive can be redefined with the .ASSIGNC directive.

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

4. Defined preprocessor variables are valid from the point of specification forward in the source program.

5. Defined preprocessor variables can be referenced in the following locations:

   - .ASSIGNA directive
   - .ASSIGNC directive
   - .AIF directive

- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

```
\&<preprocessor variable>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

Coding Example

```
FLAG:  .ASSIGNC  "ON"              ;  "ON" is set to FLAG.
        ~
        .AIF "\&FLAG" EQ "ON"      ;  MOV R0,R1 is assembled
        MOV R0,R1                  ;  when FLAG is "ON".
        .AENDI
        ~
FLAG:  .ASSIGNC  "\&FLAG "         ;  A space (" ") is added to FLAG.
FLAGA: .ASSIGNC  "OFF"             ;  "OFF" is added to FLAGA.
FLAG:  .ASSIGNC  "\&FLAG'AND \&FLAGA"
                                   ;  An apostrophe (') is used to distinguish FLAG and AND.
                                   ;  FLAG finally becomes "ON AND OFF".
        ~
```

Conditional Assembly

Syntax

```
.AIFΔ<term1>Δ<relational operator>Δ<term2>

<Source statements assembled if the condition is satisfied>

[.AELSE

<Source statements assembled if the condition is not satisfied>]

.AENDI
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AIF, .AELSE (can be omitted), or AENDI mnemonic in the operation field.

3. Operands

   .AIF: Enter the condition. Refer to the description below.
   .AELSE: The operand field is not used.
   .AENDI: The operand field is not used.

Description

1. .AIF, .AELSE, and .AENDI are the assembler directives that select whether or not to assemble source statements according to the condition specified. The .AELSE directive can be omitted.

2. The condition must be specified as follows:

   ```
   .AIFΔ<term1>Δ<relational operator>Δ<term2>
   ```

Terms are specified with numeric values or character strings. However, when a numeric value and a character string are compared, the condition always fails.

Numeric values are specified by constants or preprocessor variables.

Character strings are specified by characters or preprocessor variables enclosed by double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks (" ") in succession.

3. The following relational operators can be used:

EQ: term1 = term2
NE: term1 ≠ term2
GT: term1 > term2
LT: term1 < term2
GE: term1 ≥ term2
LE: term1 ≤ term2

Note: Numeric values are handled as 32-bit signed integers. For character strings, only EQ and NE conditions can be used.

Coding Example

```
   ~
  .AIF "\&FLAG" EQ "ON"
   MOV R0,R10          ; These statements
   MOV R1,R11          ; are assembled
   MOV R2,R12          ; when FLAG is "ON".
  .AELSE
   MOV R10,R0          ; These statements
   MOV R11,R1          ; are assembled
   MOV R12,R2          ; when FLAG is not "ON".
  .AENDI
   ~
```

Iterated Expansion

Syntax

```
.AREPEAT <count>

<Source statements iteratively assembled>

.AENDR
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AREPEAT or AENDR mnemonic in the operation field.

3. Operands

   .AREPEAT: Enter the number of iterations.
   .AENDR: The operand field is not used.

Description

1. .AREPEAT and .AENDR are the assembler directives that assemble source statements by iterative expansion the specified number of times.
2. The source statements between the .AREPEAT and .AENDR directives are iterated the number of times specified with the .AREPEAT directive. Note that the source statements are simply copied the specified number of times, and therefore, the operation does not loop at program execution.
3. Counts are specified by constants or preprocessor variables.
4. Nothing is expanded if a value of 0 or smaller is specified.

Coding Example

```
                                  ;  This example is a division of 64-bit data by 32-bit data.
                                  ;  R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
      TST      R0,R0              ;  Zero divisor check
      BT       zero_div
      CMP/HS   R0,R1              ;  Overflow check
      BT       over_div
      DIV0U                       ;  Flag initialization
      .AREPEAT 32
      ROTCL    R2                 ;  These statements are
      DIV1     R0,R1              ;  iterated 32 times.
      .AENDR
      ROTCL    R2                 ;  R2 = quotient
```

Conditional Iterated Expansion

Syntax

```
.AWHILEΔ<term1>Δ<relational operator>Δ<term2>

<Source statements iteratively assembled>

.AENDW
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AWHILE or AENDW mnemonic in the operation field.

3. Operands

   .AWHILE: Enter the condition to iteratively expand source statements.
   .AENDW: The operand field is not used.

Description

1. .AWHILE and .AENDW are the assembler directives that assemble source statements by iterative expansion while the specified condition is satisfied.

2. The source statements between the .AWHILE and .AENDW directives are iterated while the condition specified with the .AWHILE directive is satisfied. Note that the source statements are simply copied iteratively, and therefore, the operation does not loop at program execution.

3. The condition must be specified as follows:

   ```
   .AWHILEΔ<term1>Δ<relational operator>Δ<term2>
   ```

   Terms are specified with numeric values or character strings. However, when a numeric value and a character string are compared, the condition always fails.

Numeric values are specified by constants or preprocessor variables.

Character strings are specified by characters or preprocessor variables enclosed by double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks (" ") in succession.

Conditional iterated expansion terminates when the condition finally fails. An infinite loop occurs if a condition which never fails is specified. Accordingly, the condition for this directive must be carefully specified.

4. The following relational operators can be used:

   EQ: term1 = term2
   NE: term1 ≠ term2
   GT: term1 > term2
   LT: term1 < term2
   GE: term1 ≥ term2
   LE: term1 ≤ term2

Note: Numeric values are handled as 32-bit signed integers. For character strings, only EQ and NE conditions can be used.

Coding Example

```
                                            ; This example is a multiply and accumulate
                                            ; operation.
TblSiz:   .ASSIGNA  50                      ; TblSiz: Data table size
          MOV       A_Tbl1,R1               ; R1: Start address of data table 1
          MOV       A_Tbl2,R2               ; R2: Start address of data table 2
          CLRMAC                            ; MAC register initialization
          .AWHILE   \&TblSize GT 0          ; While TblSiz is larger than 0,
          MAC.W     @R0+,@R1+               ; this statement is iteratively assembled.
TblSiz:   .ASSIGNA  \&TblSiz-1              ; 1 is subtracted from TblSiz.
          .AENDW
          STS       MACL,R0                 ; The result is obtained in R0.
```

Expansion Termination

Syntax

```
.EXITM
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXITM mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .EXITM is the assembler directive that terminates an iterated expansion (.AREPEAT to .AENDR) or a conditional iterated expansion (.AWHILE to .AENDW).

2. Either expansion is terminated when this directive appears.

3. This directive is also used to exit from macro expansions. The location of this directive must be specified carefully when macro instructions and iterated expansion are combined.

   Reference: Macro expansion
   → Programmer's Guide, 7.2, "Macro Function Directives"

Coding Example

```
        ~
COUNT   .ASSIGNA 0              ; 0 is set to COUNT.
        .AWHILE 1 EQ 1          ; An infinite loop (condition is always satisfied) is
                                ; specified.
        ADD     R0,R1
        ADD     R2,R3

COUNT   .ASSIGNA \&COUNT+1      ; 1 is added to COUNT.
        .AIF     \&COUNT EQ 2 ; Condition: COUNT = 2
        .EXITM
        .AENDI
        .AENDW
        ~
```

When COUNT is updated and satisfies the condition specified with the .AIF directive, .EXITM is assembled. When .EXITM is assembled, .AWHILE expansion is terminated.

The expansion results are as follows:

```
        ADD  R0,R1              ; When COUNT is 0
        ADD  R2,R3
        ADD  R0,R1              ; When COUNT is 1
        ADD  R2,R3
```

After this, COUNT becomes 2 and expansion is terminated.

Section 7 Macro Function

7.1 Overview of the Macro Function

The macro function allows commonly used sequences of instructions to be named and defined as one macro instruction. This is called a macro definition. Macro instructions are defined as follows:

```
     ~
    .MACRO <macro name>
       <macro body>
    .ENDM
     ~
```

A macro name is the name assigned to a macro instruction, and a macro body is the statements to be executed as the macro instruction.

Using a defined macro instruction by specifying the name is called a macro call. Macro instructions are called as follows:

```
     ~
    <defined macro name>
     ~
```

An example of macro definition and macro call is shown below.

Example:

```
     ~
    .MACRO SUM              ; Processing to obtain the sum of R0, R1, R2,
     MOV R0,R10             ; and R3 is defined as macro instruction SUM.
     ADD R1,R10
     ADD R2,R10
     ADD R3,R10
    .ENDM
     ~

     SUM                    ; This statement calls macro instruction SUM.
                            ; Macro body MOV R0,R10
                            ;            ADD R1,R10
                            ;            ADD R2,R10
                            ;            ADD R3,R10
                            ; is expanded from the macro instruction.
```

Parts of the macro body can be replaced when expanded by the following procedure:

1. Macro definition

   a. Declare formal parameters in the .MACRO directive.

   b. Use the formal parameters in the macro body. Formal parameters must be identified in the macro body by placing a backslash (\) in front of them.

2. Macro call

   Specify macro parameters in the macro call.

   When the macro instruction is expanded, the formal parameters are replaced with their corresponding macro parameters.

Example:

```
  ~
.MACRO  SUM ARG1          ; Formal parameter ARG1 is defined.
 MOV R0,\ARG1             ; ARG1 is referenced in the macro body.
 ADD R1,\ARG1
 ADD R2,\ARG1
 ADD R3,\ARG1
.ENDM
  ~

 SUM R10                  ; This statement calls macro instruction SUM
                          ; specifying macro parameter R10.
                          ; The formal parameter in the macro body is
                          ; replaced with the macro parameter, and
                          ;          MOV  R0,R10
                          ;          ADD  R1,R10
                          ;          ADD  R2,R10
                          ;          ADD  R3,R10 is expanded.
```

7.2 Macro Function Directives

This assembler provides the following macro function directives.

`.MACRO` `.ENDM`	Defines a macro instruction.
`.EXITM`	Terminates macro instruction expansion.

.MACRO	.ENDM

Macro Definition

Syntax

```
.MACROΔ<macro name>[Δ<formal parameter>[=<default>]
                                      [,<formal parameter>...]]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .MACRO or .ENDM mnemonic in the operation field.

3. Operands

   .MACRO: Enter the name and formal parameters for the macro instruction to be defined. When formal parameters are defined, their defaults can be defined (defaults can be omitted).
   .ENDM: The operand filed is not used.

Description

1. .MACRO and .ENDM are the assembler directives that define a macro instruction (a sequence of source statements that are collectively named and handled together).

2. Macro definition

   Naming as a macro instruction the source statements (macro body) between .MACRO and .ENDM directives is called a macro definition.

3. Macro name

   Macro names are the names assigned to macro instructions.

4. Formal parameters

   Formal parameters are specified so that parts of the macro body can be replaced by specific parameters at expansion time. Formal parameters are replaced with the character strings (macro parameters) specified at macro expansion (macro call).

a. Formal parameter syntax

The syntax for formal parameters is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

b. Formal parameter reference

Formal parameters are used (referenced) at the part to be replaced in the the macro body.

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

```
\<formal parameter name>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

3. Formal parameter defaults

Defaults for formal parameters can be specified in macro definitions. The default specifies the character string to replace the formal parameter when the corresponding macro parameter is omitted in a macro call.

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

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

The assembler inserts defaults at macro expansion by removing the double quotation marks or angle brackets that enclose the character strings.

6. Restrictions on macro definitions

   a. Macros cannot be defined in the following locations:

      - Macro bodies (between .MACRO and .ENDM directives)
      - Between .AREPEAT and .AENDR directives
      - Between .AWHILE and .AENDW directives

   b. The .ENDM directive cannot be used within a macro body.

   c. No symbol can be inserted in the label field of the .ENDM directive. The .ENDM directive is ignored if its label field is not blank, but no error is generated in this case.

Coding Example

```
  ~
.MACRO SUM                  ; Processing to obtain the sum of R0, R1, R2,
 MOV R0,R10                 ; and R3 is defined as macro instruction SUM.
 ADD R1,R10
 ADD R2,R10
 ADD R3,R10
.ENDM
  ~

 SUM                        ; This statement calls macro instruction SUM
                            ; Macro body  MOV  R0,R10
                            ;             ADD  R1,R10
                            ;             ADD  R2,R10
                            ;             ADD  R3,R10 is expanded.
```

Expansion Termination

Syntax

```
.EXITM
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXITM mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .EXITM is the assembler directive that terminates a macro expansion. This directive can be specified within the macro body (between .MACRO and .ENDM directives).

2. Expansion is terminated when this directive appears.

3. This directive is also used to exit from iterated expansions specified with the .AREPEAT or .AWHILE directive. The location of this directive must be specified carefully when macro instructions and iterated expansion are combined.

Coding Example

```
        .MACRO   SUM P1
        MOV      R0,R10          ┐
        ADD      R1,R10          │ (1)
        ADD      R2,R10          ┘
        \P1               -------- (2)
        ADD      R3,R10
        .ENDM

         ~

        SUM      .EXITM
```

.EXITM is expanded at (2) and macro expansion is terminated. Only the statements indicated by (1) are expanded.

7.3 Macro Body

The source statements between the .MACRO and .ENDM directives are called a macro body. The macro body is expanded and assembled by a macro call.

1. Formal parameter reference

   Formal parameters are used to specify the parts to be replaced with macro parameters at macro expansion.

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

   ```
   \<formal parameter name>[']
   ```

   To clearly distinguish the formal parameter name from the rest of the source statement, add an apostrophe (').

Coding example:

```
        .MACRO   PLUS1 P,P1         ; P and P1 are formal parameters.
        ADD      #1,\P1             ; Formal parameter P1 is referenced.
        .SDATA   "\P'1"             ; Formal parameter P is referenced.
        .ENDM
        PLUS1    R,R1               ; PLUS1 is expanded.
```

Expanded results are as follows:

```
        ADD      #1,R1              ; Formal parameter P1 is referenced.
        .SDATA   "R1"               ; Formal parameter P is referenced.
```

2. Preprocessor variable reference

   Preprocessor variables can be referenced in macro bodies.

   The syntax for preprocessor variable reference is as follows:

   ```
   \&<preprocessor variable name>[']
   ```

   To clearly distinguish the formal parameter name from the rest of the source statement, add an apostrophe (').

Coding example:

```
        .MACRO   PLUS1
        ADD      #1,R\&V1          ; Preprocessor variable V1 is referenced.
        .SDATA   "\&V'1"           ; Preprocessor variable V is referenced.
        .ENDM
V       .ASSIGNC "R"               ; Preprocessor variable V is defined.
V1      .ASSIGNA 1                 ; Preprocessor variable V1 is defined.
        PLUS1                      ; PLUS1 is expanded.

  Expanded results are as follows:

        ADD      #1,R1             ; Preprocessor variable V1 is referenced.
        .SDATA   "R1"              ; Preprocessor variable V is referenced.
```

3. Macro generation number

   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 in a macro. This will result in symbols that are unique to each macro call.

   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:

   ```
   \@
   ```

   Two or more macro generation number markers can be written in a macro body, and they will be expanded to the same number in one macro call.

CAUTION!

Because macro generation number markers are expanded to numbers, they must not be written at the beginning of symbol names.

Reference: Programmer's Guide, 1.3.2, "Coding of Symbols"

Coding example:

```
        .MACRO  RES_STR STR, Rn
        MOV.L   #str\@,\Rn
        BRA     end_str\@
        NOP
str\@   .SDATA  "\STR"
        .ALIGN 2
end_str\@
        .ENDM

        RES_STR "ONE",R0
        RES_STR "TWO",R1
```

Different symbols are generated each time RES_STR is expanded.

Expanded results are as follows:

```
        MOV.L   #str00000,R0
        BRA     end_str00000
        NOP
str00000        .SDATA  "ONE"
        .ALIGN 2

        MOV.L   #str00001,R1
        BRA     end_str00001
        NOP
str00001        .SDATA  "TWO"
        .ALIGN 2
```

4. Macro replacement processing exclusion

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

   The syntax for macro replacement processing exclusion is as follows:

   `\(<macro replacement processing excluded character string>)`

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

Coding example:

```
        .MACRO   BACK_SLASH_SET
        \(MOV     #"\",R0)           ; \ is expanded as an ASCII character.
        .ENDM
```

Expanded results are as follows:

```
        MOV       #"\",R0            ; \ is expanded as an ASCII character.
```

5. Comments in macros

   Comments in macro bodies can be coded as normal comments or as macro internal comments. When comments in the macro body are not required in the macro expansion code (to avoid repeating the same comment in the listing file), those comments can be coded as macro internal comments to suppress their expansion.

   The syntax for macro internal comments is as follows:

   `\;<comment>`

Coding example:

```
        .MACRO PUSH Rn
        MOV.L      \Rn,@-R15        \;    \Rn is a register.
        .ENDM
        PUSH       R0
```

Expanded results are as follows (the comment is not expanded):

```
        MOV.L      R0,@-R15
```

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.
.INSTR	Character string search.
.SUBSTR	Character string substring.

References:
.LEN → Programmer's Guide, 7.5, "Character String Manipulation Functions", .LEN
.INSTR → Programmer's Guide, 7.5, "Character String Manipulation Functions", .INSTR
.SUBSTR → Programmer's Guide, 7.5, "Character String Manipulation Functions", .SUBSTR

7.4 Macro Call

Expanding a defined macro instruction is called a macro call. The syntax for macro calls is as follows:

Syntax

```
[<symbol>] <macro name>[<macro parameter>
                                        [,<macro parameter> ...]]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the macro name to be expanded in the operation field. The macro name must have been already defined before a macro call.

3. Operands

   Enter character strings as macro parameters to replace formal parameters at macro expansion. The formal parameters must have been declared in the macro definition with .MACRO.

Description

1. Macro parameter specification

   Macro parameters can be specified by either positional specification or keyword specification.

   a. 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

      Each macro parameter is specified following its corresponding formal parameter, separated by an equal sign (=).

2. Macro parameter syntax

Macro parameters must be enclosed by double quotation marks (") or angle brackets (<>) if any of the following characters are included in the macro parameters:

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

Macro parameters are inserted by removing the double quotation marks or angle brackets that enclose character strings at macro expansion.

Coding Example

```
        .MACRO  SUM FROM=0,TO=9          ; Macro instruction SUM and formal
                                         ; parameters FROM and TO are defined.
        MOV     R\FROM,R10              ┐
COUNT   .ASSIGNA         \FROM+1        │
        .AWHILE \&COUNT LE \TO          │ Macro body is coded
        MOV      R\&COUNT,R10           │ using formal parameters.
COUNT   .ASSIGNA          \&COUNT+1     │
        .AENDW                          │
        .ENDM                           ┘

        SUM      0,5                    ┐ Both will be expanded
        SUM      TO=5                   ┘ into the same statements.
```

Expanded results are as follows (the formal parameters in the macro body are replaced with macro parameters):

```
        MOV      R0, R10
        MOV      R1, R10
        MOV      R2, R10
        MOV      R3, R10
        MOV      R4, R10
        MOV      R5, R10
```

7.5 Character String Manipulation Functions

This assembler provides the following character string manipulation functions.

`.LEN`	Counts the length of a character string.
`.INSTR`	Searches for a character string.
`.SUBSTR`	Extracts a character string.

Character String Length Count

Syntax

```
.LEN[Δ]("<character string>")
```

Description

1. .LEN counts the number of characters in a character string and replaces itself with the number of characters in decimal with no radix.

2. 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.

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

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

4. This function can only be used within a macro body (between .MACRO and .ENDM directives).

Coding Example:

```
        ~
        .MACRO RESERVE_LENGTH P1
        .ALIGN 4
        .SRES    \LEN("\P1")
        .ENDM
        ~

        RESERVE_LENGTH ABCDEF
        RESERVE_LENGTH ABC
```

Expanded results are as follows:

```
        .ALIGN   4
        .SRES    6                 ; "ABCDEF" has six characters.
        .ALIGN   4
        .SRES    3                 ; "ABC" has three characters.
```

.INSTR

Character String Search

Syntax

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

Description

1. .INSTR searches character string 1 for character string 2, and replaces itself with the numerical value of the position of the found string (with 0 indicating the start of the string) in decimal with no radix. .INSTR is replaced with –1 if character string 2 does not appear in character string 1.

2. 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.

3. The <start position> parameter specifies the search start position as a numerical value, with 0 indicating the start of the string. Zero is used as default when this parameter is omitted.

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

   ```
   .INSTR("\<formal parameter>", ...)

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

5. This function can only be used within a macro body (between .MACRO and .ENDM directives).

Coding Example:

```
        ~
        .MACRO FIND_STR P1
        .DATA.W .INSTR("ABCDEFG","\P1",0) .
        .ENDM
        ~

        FIND_STR CDE
        FIND_STR H
```

Expanded results are as follows:

```
        .DATA.W  2            ; The start position of "CDE" is 2 (0 indicating the
                                beginning of the string) in "ABCDEFG"
        .DATA.W  -1           ; "ABCDEFG" includes no "H".
```

.SUBSTR

Character Substring Extraction

Syntax

```
.SUBSTR[Δ]("<character string>",<start position>
                                  ,<extraction length>)
```

Description

1. .SUBSTR extracts from the specified character string a substring starting at the specified start position of the specified length. .SUBSTR is replaced with the extracted character string enclosed by double quotation marks (").
2. Character strings are specified by enclosing the desired characters in double quotation marks ("). To specify a double quotation mark in a character string, enter 2 double quotation marks in succession.
3. The value of the extraction start position must be 0 or greater. The value of the extraction length must be 1 or greater.
4. If illegal or inappropriate values are specified for the <start position> or <extraction length> parameters, this function is replaced with a blank space (" ").
5. Macro formal parameters and preprocessor variables can be specified in the character string, and as the start position and extraction length parameters as shown below.

   ```
   .SUBSTR("\<formal parameter>", ...)

   .SUBSTR("\&<preprocessor variable>", ...)
   ```
6. This function can only be used within a macro body (between .MACRO and .ENDM directives).

Coding Example:

```
~
.MACRO RESERVE_STR P1=0 P2
.SDATA   .SUBSTR("ABCDEFG",\P1,\P2)
.ENDM
~

RESERVE_STR 2,2
RESERVE_STR ,3        ; Macro parameter P1 is omitted.
```

Expanded results are as follows:

```
.SDATA   "CD"
.SDATA   "ABC"
```

(This page intentionally left blank.)

Section 8   Automatic Literal Pool Generation Function

8.1 Overview of Automatic Literal Pool Generation

To move 2-byte or 4-byte constant data (referred to below as a "literal") to a register, a literal pool (a collection of literals) must be reserved and referred to in PC relative addressing mode. For literal pool location, the following must be considered:

- Is data stored within the range that can be accessed by data move instructions?
- Is 2-byte data aligned to a 2-byte boundary and is 4-byte data aligned to a 4-byte boundary?
- Can data be shared by several data move instructions?
- Where should the literal pool be located in the program?

The assembler automatically generates from a single instruction a .DATA directive and a PC relative MOV or MOVA instruction, which moves constant data to a register.

For example, this function enables program (a) below to be coded as (b):

(a)

```
         MOV.L    DATA1,R0
         MOV.L    DATA2,R1

         ~

         .ALIGN 4
DATA1    .DATA.L H'12345678
DATA2    .DATA.L 50000
```

(b)

```
         MOV.L    #H'12345678,R0
         MOV.L    #500000,R1

         ~
```

8.2 Extended Instructions Related to Automatic Literal Pool Generation

The assembler automatically generates a literal pool corresponding to an extended instruction (MOV.W #imm, Rn; MOV.L #imm, Rn; or MOVA #imm, R0) and calculates the PC relative displacement value.

An extended instruction source statement is expanded to an executable instruction and literal data as shown in table 8-1.

Table 8-1 Extended Instructions and Expanded Results

Extended Instruction	Expanded Result
MOV.W #imm, Rn	MOV.W @(disp, PC), Rn and 2-byte literal data
MOV.L #imm, Rn	MOV.L @(disp, PC), Rn and 4-byte literal data
MOVA #imm, R0	MOVA @(disp, PC), R0 and 4-byte literal data

8.3 Literal Pool Output

The literal pool is output to one of the following locations:

- After an unconditional branch (after the delay slot instruction following BRA, JMP, RTS, or RTE)
- Where a .POOL directive has been specified by the programmer

The assembler outputs the literal corresponding to an extended instruction to the nearest output location following the extended instruction. The assembler gathers the literals to be output as a literal pool.

CAUTION!

When a label is specified in a delay slot instruction, no literal pool will be output to the location following the delay slot.

8.3.1 Literal Pool Output after Unconditional Branch (BRA, JMP, RTS, RTE)

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
        MOV.L    #H'FFFF0000,R0
        MOV.W    #H'FF00,R1
        MOV.L    #CD1_START,R2
        MOV      #FF,R3
        RTS
        MOV      R0,R10
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
 1 0000F000                 1        .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000                 2   CD1_START
 3 0000F000 5004            3        MOV.L    #H'FFFF0000,R0
 4 0000F002 1103            4        MOV.W    #H'FF00,R1
 5 0000F004 5205            5        MOV.L    #CD1_START,R2
 6 0000F006 6300            6        MOV      #FF,R3
 7 0000F008 000B            7        RTS
 8 0000F00A 6A03            8        MOV      R0,R10
 9 [illegible]                **** BEGIN-POOL ****
10 0000F00C FF00                DATA FOR SOURCE-LINE 4
11 0000F00E 0000                ALIGNMENT CODE
12 0000F010 FFFF0000            DATA FOR SOURCE-LINE 3
13 0000F014 0000F000            DATA FOR SOURCE-LINE 5
14                              ****  END-POOL   ****
15                          9        .END
```

8.3.2 Literal Pool Output to the .POOL Location

If literal pool output location after unconditional branches is not available within the valid displacement range (because the program has a small number of unconditional branches), the assembler outputs error message 402. In this case, a .POOL directive must be specified within the valid displacement range.

The valid displacement range is as follows:

- Word-size operation: 0 to 511 bytes
- Long word-size operation: 0 to 1023 bytes

When a literal pool is output to a .POOL location, a branch instruction is also inserted to jump over the literal pool.

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START
        MOV.L    #H'FFFF0000,R0
        MOV.W    #H'FF00,R1
        MOV.L    #CD1_START,R2
        MOV      #FF,R3
        .POOL
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
 1 0000F000                1       .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000                2  CD1_START:
 3 0000F000 5012           3       MOV.L    #H'FFFF0000,R0
 4 0000F002 110E           4       MOV.W    #H'FF00,R1
 5 0000F004 5216           5       MOV.L    #CD1_START,R2
 6 0000F006 6300           6       MOV      #H'FF,R3
 7 0000F008                7       .POOL
 8                            **** BEGIN-POOL ****
 9 0000F008 A006                  BRA TO END-POOL
10 0000F00A 0009                  NOP
11 0000F00C FF00                  DATA FOR SOURCE-LINE 4
12 0000F00E 0000                  ALIGNMENT CODE
13 0000F010 FFFF0000              DATA FOR SOURCE-LINE 3
14 0000F014 0000F000              DATA FOR SOURCE-LINE 5
15                            ****  END-POOL  ****
16                         8       .END
```

8.4 Literal Sharing

When the literals for several extended instructions are gathered into a literal pool, the assembler makes the extended instructions share identical immediate data.

The following operand forms can be identified and shared:

- Symbol
- Constant
- Symbol ± constant

In addition to the above, expressions that are determined to have the same value at assembly processing may be shared.

However, extended instructions having different operation sizes do not share literal data even when they have the same immediate data.

An example of literal data sharing among extended instructions is shown below.

Source program

```
         .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
         MOV.L    #H'FFFF0000,R0
         MOV.W    #H'FF00,R1
         MOV.L    #H'FFFF0000,R2
         MOV      #H'FF,R3
         RTS
         MOV      R0,R10
         .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
 1 0000F000                    1       .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000                    2  CD1_START:
 3 0000F000 5004               3       MOV.L    #H'FFFF0000,R0
 4 0000F002 1103               4       MOV.W    #H'FF00,R1
 5 0000F004 5204               5       MOV.L    #H'FFFF0000,R2
 6 0000F006 6300               6       MOV      #H'FF,R3
 7 0000F008 000B               7       RTS
 8 0000F00A 6A03               8       MOV      R0,R10
 9                                 **** BEGIN-POOL ****
10 0000F00C FF00                     DATA FOR SOURCE-LINE 4
11 0000F00E 0000                     ALIGNMENT CODE
12 0000F010 FFFF0000                 DATA FOR SOURCE-LINE 3,5
13                                 ****  END-POOL  ****
14                             9       .END
```

8.5 Literal Pool Output Suppression

When a program has too many unconditional branches, the following problems may occur:

- Many small literal pools are output
- Literals are not shared

In these cases, suppress literal pool output as shown below.

```
             ~
  <delayed branch instruction>
            <delay slot instruction>
            .NOPOOL
             ~
```

Example

Source program

```
              ~
CASE1:
        MOV.L    #H'FFFF0000,R0    --+---- Extended instruction 1
        RTS                          |
        NOP                          |
        .NOPOOL                    --+---- No literal pool is output here
CASE2:                               |
        MOV.L    #H'FFFF0000,R0    --+---- Extended instruction 2
        RTS                          |
        NOP                          |
                                   --+---- Literal pool is output here
              ~
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
                 ~
 20 0000F000                20 CASE1:
 21 0000F000 5001           21       MOV.L    #H'FFFF0000,R0
 22 0000F002 000B           22       RTS
 23 0000F004 0009           23       NOP
 24                         24       .NOPOOL
 25 0000F006                25 CASE2:
 26 0000F006 5001           26       MOV.L    #H'FFFF0000,R0
 27 0000F008 000B           27       RTS
 28 0000F00A 0009           28       NOP
 29                            **** BEGIN-POOL ****
 30 0000F00B 0000                   ALIGNMENT CODE
 31 0000F00C FFFF0000               DATA FOR SOURCE-LINE 21,26
 32                            ****  END-POOL  ****
                 ~
```

8.6 Notes on Automatic Literal Pool Output

1. If an error occurs when an extended instruction is written

   — Extended instructions must not be specified in delay slots (error 151).
   — Extended instructions must not be specified in relative sections having a boundary alignment value of less than 2 (error 152).
   — MOV.L #imm, Rn or MOVA #imm, R0 must not be specified in relative sections having a boundary alignment value of less than 4 (error 152).

2. If an error occurs when a .POOL directive is written

   .POOL directives must not be written after unconditional branches (error 522).

3. If an error occurs when a .NOPOOL directive is written

   .NOPOOL directives are valid only when written after delay slot instructions. If written at other locations, the .NOPOOL directive causes error 521.

4. If the displacement of an executable instruction exceeds the valid range when an extended instruction is expanded

   The assembler generates a literal pool and outputs error message 402 for the instruction having a displacement outside the valid range.

   Solution: Move the literal pool output location by the .NOPOOL directive, or change the location or addressing mode of the instruction causing the error.

5. If the literal pool output location cannot be found

   If the assembler cannot find a literal pool output location satisfying the following conditions in respect to the extended instruction,

   — Same file
   — Same section
   — Forward direction

   the assembler outputs, at the end of the section which includes the extended instruction, the literal pool and a BRA instruction with a NOP instruction in the delay slot to jump around the literal pool, and outputs warning message 876.

6. If the displacement from the extended instruction exceeds the valid range

   If the displacement of the literal pool from the extended instruction exceeds the valid range, error 402 is generated.

   Solution: Output the literal pool within the valid range using the .POOL directive.

(This page intentionally left blank.)

User's Guide

(This page intentionally left blank.)

Section 1 Executing the Assembler

1.1 Command Line Format

To start the assembler, enter a command line with the following format when the host computer operating system is in the input wait state.

```
> asmsh Δ <input source file>[,<input source file>...][[Δ] <command line options> ...]
  (1)          (2)                                             (3)
```

(1) Assembler start command.
(2) Name of input source file. Multiple source files can be specified at the same time.
(3) Command line options, which specify the assembly method in more detail.

CAUTION!

When multiple source files are specified on the command line, the unit of assembly processing will be the concatenation of the specified files in the specified order.

In this case, the .END directive must appear only in the last file.

Supplement:

The assembler returns the operating system a return code that reports whether or not the assembly processing terminated normally. The return value indicates the level of the errors occurred as follows.

Normal termination		0
Warnings occurred		0
Errors occurred	MS-DOS:	2
	UNIX:	1
Fatal error occurred	MS-DOS:	4
	UNIX:	1

1.2 File Specification Format

Files handled by the assembler are specified in the following format.

```
<file name>.[<file format>]
```

The term "file name" as used in this manual normally refers to both the file name and the file format.

Example:

```
(File name)
file.src ..... A file with the file name file and the file format src.
prog.obj ..... A file with the file name prog and the file format obj.
```

The file format is used as an identifier to distinguish the contents of the file. Thus two files with differing formats are different files even if the file name is the same.

Example:

```
file.src  ⎫
file.obj  ⎭ These file names specify different files.
```

The assembler handles the following types of file.

- Source file
  This is a source program file. If a source program file is specified without the file format, the file format src will be supplied.

- Object file
  This is an output destination file for object modules. If an object file is specified without the file format, the file format obj will be supplied. If an object file is not specified to the assembler, a file with the same name as the source file (the first specified source file) and with the file format obj will be used.

- Listing file
  This is an output destination file for assemble listings. If a listing file is specified without the file format, the extension lis will be supplied. If a listing file is not specified to the assembler, a file with the same name as the source file (the first specified source file) and with the file format lis will be used.

Note: The PC system treats all-file names, command lines, and subcommand lines as capital letters.

Section 2 Command Line Options

2.1 Overview of Command Line Options

Command line options are detailed specifications of the assembly processing. Table 2-1 shows an overview of the command line options.

Table 2-1 Command Line Options

Type	Command Line Option	Function
Object module specifications	OBJECT NOOBJECT	Control output of an object module.
	DEBUG NODEBUG	Control output of debug information.
Assemble listing specifications	LIST NOLIST	Control output of an assemble listing.
	SOURCE NOSOURCE	Control output of a source program listing.
	CROSS_REFERENCE NOCROSS_REFERENCE	Control output of a cross-reference listing.
	SECTION NOSECTION	Control output of a section information listing.
	SHOW NOSHOW	Control output of the source program listing.
	LINES	Sets the number of lines in the assemble listing.
	COLUMNS	Sets the number of columns in the assemble listing.

CAUTION!

When starting the assembler on MS-DOS, enter a slash (/) instead of a hyphen (-) before the command line options.

Supplement:

The assemble listing is a listing to which the results of the assembly processing are output, and consists of a source program listing, a cross-reference listing, and a section information listing.

References: See appendix C, "Assemble Listing Example", for a detailed description of the assemble listing.

2.2 Command Line Option Reference

2.2.1 Object Module Command Line Options

This assembler provides the following command line options concerned with object modules.

OBJECT	NOOBJECT

These command line options control output of an object module.

DEBUG	NODEBUG

These command line options control output of debug information.

Note: The syntaxes are written for a UNIX system; use a slash (/) instead of a hyphen (-) for an MS-DOS system.

Object Module Output Control

Syntax

```
OBJECT [= <object output file>]
NOOBJECT
```

The abbreviated forms are indicated by shading.

Description

1. The OBJECT option specifies output of an object module.

   The NOOBJECT option specifies no output of an object module.

2. The object output file specifies the output destination for the object module.

3. When the object output file parameter is omitted, the assembler takes the following actions:

   - If the file format is omitted:
     The file format obj is supplied.

   - If the specification is completely omitted:
     The file format obj is appended to the name of the input source file (the first specified source file).

4. Do not specify the same file for the input source file and the output object file.

OBJECT	NOOBJECT

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
OBJECT	(regardless of any specification)	An object module is output.
NOOBJECT	(regardless of any specification)	An object module is not output.
(no specification)	.OUTPUT OBJ	An object module is output.
	.OUTPUT NOOBJ	An object module is not output.
	(no specification)	An object module is output.

Debug Information Output Control

Syntax

```
-DEBUG
-NODEBUG
                                   The abbreviated forms are indicated by shading.
```

Description

1. The DEBUG option specifies output of debug information.

   The NODEBUG option specifies no output of debug information.

2. The DEBUG and NODEBUG options are only valid in cases where an object module is being output.

   References: Object module output
   → Programmer's Guide, 4.2.5, "Object Module Assembler Directives", .OUTPUT
   → User's Guide, 2.2.1, "Object Module Command Line Options", OBJECT NOOBJECT

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
DEBUG	(regardless of any specification)	Debug information is output.
NODEBUG	(regardless of any specification)	Debug information is not output.
(no specification)	.OUTPUT DBG	Debug information is output.
	.OUTPUT NODBG	Debug information is not output.
	(no specification)	Debug information is not output.

DEBUG	NODEBUG

Supplement:

Debug information is information required when debugging a program using the simulator/debugger or the emulator, and is part of the object module. Debug information includes information about source statement lines and information about symbols.

2.2.2 Assemble Listing Command Line Options

This assembler provides the following command line options concerned with the assemble listing.

`LIST` `NOLIST`

These command line options control output of an assemble listing.

`SOURCE` `NOSOURCE`

These command line options control output of a source program listing.

`CROSS_REFERENCE` `NOCROSS_REFERENCE`

These command line options control output of a cross-reference listing.

`SECTION` `NOSECTION`

These command line options control output of a section information listing.

`SHOW` `NOSHOW`

These command line options control output of the source program listing.

`LINES`

This command line option sets the number of lines in the assemble listing.

`COLUMNS`

This command line option sets the number of columns in the assemble listing.

LIST	NOLIST

Assemble Listing Output Control

Syntax

```
LIST [ =<listing output file> ]
NOLIST
```

The abbreviated forms are indicated by shading.

Description

1. The LIST option specifies output of an assemble listing.

   The NOLIST option specifies no output of an assemble listing.

2. The listing output file specifies the output destination file for the assemble listing.

3. When the listing output file parameter is omitted, the assembler takes the following actions:

   - If the file format is omitted:
     The file format lis is supplied.

   - If the specification is completely omitted:
     The file format lis is appended to the name of the input source file (the first specified source file).

4. Do not specify the same file for the input source file and the listing output file.

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
LIST	(regardless of any specification)	An assemble listing is output.
NOLIST	(regardless of any specification)	An assemble listing is not output.
(no specification)	.PRINT LIST	An assemble listing is output.
	.PRINT NOLIST	An assemble listing is not output.
	(no specification)	An assemble listing is not output.

SOURCE	NOSOURCE

Source Program Listing Output Control

Syntax

```
SOURCE
NOSOURCE
                                   The abbreviated forms are indicated by shading.
```

Description

1. The SOURCE option specifies output of a source program listing to the assemble listing.

   The NOSOURCE option specifies no output of a source program listing to the assemble listing.

2. The SOURCE and NOSOURCE options are only valid in cases where an assembly listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
SOURCE	(regardless of any specification)	A source program listing is output.
NOSOURCE	(regardless of any specification)	A source program listing is not output.
(no specification)	.PRINT SRC	A source program listing is output.
	.PRINT NOSRC	A source program listing is not output.
	(no specification)	A source program listing is output.

Cross-Reference Listing Output Control

Syntax

```
-CROSS_REFERENCE
-NOCROSS_REFERENCE
```

The abbreviated forms are indicated by shading.

Description

1. The CROSS_REFERENCE option specifies output of a cross-reference listing to the assemble listing.

   The NOCROSS_REFERENCE option specifies no output of a cross-reference listing to the assemble listing.

2. The CROSS_REFERENCE and /NOCROSS_REFERENCE options are only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
CROSS_REFERENCE	(regardless of any specification)	A cross-reference listing is output.
NOCROSS_REFERENCE	(regardless of any specification)	A cross-reference listing is not output.
(no specification)	.PRINT CREF	A cross-reference listing is output.
	.PRINT NOCREF	A cross-reference listing is not output.
	(no specification)	A cross-reference listing is output.

Section Information Listing Output Control

Syntax

```
SECTION
NOSECTION
```

The abbreviated forms are indicated by shading.

Description

1. The SECTION option specifies output of a section information listing to the assemble listing.

   The NOSECTION option specifies no output of a section information listing to the assemble listing.

2. The SECTION and NOSECTION options are only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

SECTION	NOSECTION

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
SECTION	(regardless of any specification)	A section information listing is output.
NOSECTION	(regardless of any specification)	A section information listing is not output.
(no specification)	.PRINT SCT	A section information listing is output.
	.PRINT NOSCT	A section information listing is not output.
	(no specification)	A section information listing is output.

SHOW	NOSHOW

Source Program Listing Output Control

Syntax

```
<UNIX>
-SHOW [= <output type>[,<output type> ...]]
-NOSHOW [= <output type>[,<output type> ...]]

<MS-DOS>
/SHOW [=(<output type>[,<output type> ...])]
/NOSHOW [=(<output type>[,<output type> ...])]
```

When only one output type is specified, the parentheses can be omitted.

```
Output type:   {CONDITIONALS|DEFINITIONS|CALLS|EXPANSIONS|CODE}
```

The abbreviated forms are indicated by shading.

Description

1. The SHOW option specifies output of preprocessor function source statements and object code lines in the source program listing.

   The NOSHOW option suppresses output of specified preprocessor function source statements and object code display lines in the source program listing.

2. The items specified by output types will be output or suppressed depending on the option. When no output type is specified, all items will be output or suppressed.

   -SHOW: Output
   -NOSHOW: Not output (suppress)

3. The following output types can be specified:

Output Type	Object	Description
CONDITIONALS	Failed condition	Condition-failed .AIF statements
DEFINITIONS	Definition	Macro definition parts, .AREPEAT and .AWHILE definition parts, .INCLUDE directive statements .ASSIGNA and .ASSSIGNC directive statements
CALLS	Call	Macro call statements, .AIF and .AENDI directive statements
EXPANSIONS	Expansion	Macro expansion statements .AREPEAT and .AWHILE expansion statements
CODE	Object code lines	The object code lines exceeding the source statement lines

References: Assemble listing output
→ Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST SOURCE NOSOURCE

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
SHOW=<output type>	(regardless of any specification)	The object code is output.
NOSHOW=<output type>	(regardless of any specification)	The object code is not output.
(no specification)	.LIST <output type> (output)	The object code is output.
	.LIST <output type> (suppress)	The object code is not output.
	(no specification)	The object code is output.

LINES

Sets the Number of Lines in the Assemble Listing

Syntax

```
LINES=<line count>
```

The abbreviated form is indicated by shading.

Description

1. LINES is the command line option that sets the number of lines on a single page of the assemble listing. The range of valid values for the line count is from 20 to 255.

2. The LINES specification is only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
LINES=<line count>	(regardless of any specification)	The number of lines on a page is given by the LINES specification.
(no specification)	.FORM LIN=<line count>	The number of lines on a page is given by the .FORM specification.
	(no specification)	The number of lines on a page is 60 lines.

Sets the Number of Columns in the Assemble Listing

Syntax

```
COLUMNS=<column count>
```

The abbreviated form is indicated by shading.

Description

1. COLUMNS is the command line option that sets the number of columns in a single line of the assemble listing. The range of valid values for the column count is from 79 to 255.

2. The COLUMNS specification is only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
COLUMNS=<column count>	(regardless of any specification)	The number of columns in a line is given by the COLUMNS specification.
(no specification)	.FORM COL=<column count>	The number of columns in a line is given by the .FORM specification.
	(no specification)	The number of columns in a line is 132 columns.

(This page intentionally left blank.)

Appendix

(This page intentionally left blank.)

Appendix A Limitations and Notes on Programming

Table A-1 Limitations and Notes on Programming

No.	Item	Limitation	
1	Character types	ASCII characters	
2	Upper/lower-case letter distinction	Symbols (including section names) Object module names	Distinguished
		Reserved words Executable instruction mnemonics Assembler directive mnemonics Operation sizes Integer constant radixes	Not distinguished
3	Line length	Up to 255 bytes	
4	Program length (in lines)	Up to 65,535 lines	
5	Character constants	Up to 4 characters	
6	Symbol length	Up to 32 characters	
7	Number of symbols	Up to 65,535 symbols	
8	Number of import symbols	Up to 65,535 symbols	
9	Number of export symbols	Up to 65,535 symbols	
10	Section size	Up to H'FFFFFFFF bytes	
11	Number of sections	Up to 65,535 sections	
12	Number of macro generation numbers	Up to 100,000 numbers	
13	Number of literals	Up to 100,000 literals	

Appendix B Sample Program

This appendix presents a sample program written for this assembler.

Sample Program Specifications:

Functional Specification

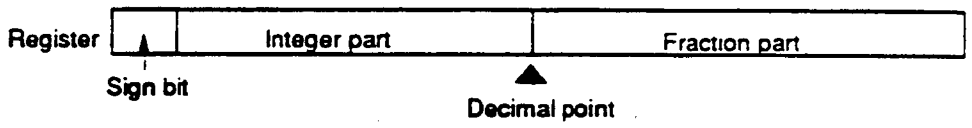
Macros and subroutines for addition, subtraction, multiplication, and division of fixed-point data in the following format:

<parameter 1> OP <parameter 2> → result

OP: +, −, ×, ÷

Note: Operation results are rounded off. Neither underflow nor overflow is checked.

Data Format



The location of the decimal point is set in preprocessor variable POINT as the number of bits from the MSB.

Inputs and Outputs

Inputs: Set parameter 1 in register Parm1.
Set parameter 2 in register Parm2.
For addition and subtraction, parameters 1 and 2 can be specified as macro parameters.

Output: The result is stored in register Parm 1.

Macro and Subroutine Usage

Addition (+): Macro call FIX_ADD [parameter 1], [parameter 2]

Subtraction (−): Macro call FIX_SUB [parameter 1], [parameter 2]

Multiplication (×): Subroutine call FIX_MUL

Division(÷): Subroutine call FIX_DIV

Registers to be Used

Define the following registers with the .REG directive:
Parm1, Parm 2, WORK 1, WORK2, WORK3, WORK4

Supplement

An example of using this sample program is shown in appendix C.

Coding Example:

```
        .MACRO   FIX_ADD Rs=Parm2, Rd=Parm1
        ADD      \Rs,\Rd
        .ENDM

        .MACRO   FIX_SUB Rs=Parm2,Rd=Parm1
        SUB      \Rs,\Rd
        .ENDM

FIX_MUL:
        DIV0S    Parm1,Parm2              ;
        MOVT     WORK1                    ;  Stores the sign of the result in WORK1.
        CMP/PZ   Parm1                    ;  —
        BT       MUL01                    ;  |If (Parm1 < 0), Parm1 = –Parm1
        NEG      Parm1,Parm1              ;  —
MUL01   CMP/PZ   Parm2                    ;  —
        BT       MUL02                    ;  |If (Parm2 < 0), Parm2 = –Parm2
        NEG      Parm2,Parm2              ;  —
MUL02   MULU     Parm1,Parm2              ;  Parm1 (low) * Parm2 (low)
        SWAP.W   Parm1,Parm1              ;
        STS      MACL,WORK2               ;
        MULU     Parm1,Parm2              ;  Parm1 (high) * Parm2 (low)
        SWAP.W   Parm1,Parm1              ;
        SWAP.W   Parm2,Parm2              ;
        STS      MACL,WORK3               ;
        MULU     Parm1,Parm2              ;  Parm1 (low) * Parm2 (high)
        SWAP.W   Parm1,Parm1              ;
        STS      MACL,WORK4               ;
        MULU     Parm1,Parm2              ;  Parm1 (high) * Parm2( high)
        CLRT                              ;
        STS      MACL,Parm1               ;
        MOV      WORK3,Parm2              ;  —
        SHLR16   WORK3                    ;  |
        SHLL16   Parm2                    ;  |
        ADDC     Parm2,WORK2              ;  |
        ADDC     WORK3,Parm1              ;  |  Sums 16-bit multiplication results.
        MOV      WORK4,Parm2              ;  |
        SHLR16   WORK4                    ;  |
        SHLL16   Parm2                    ;  |
        ADDC     WORK2,Parm2              ;  |
        ADDC     WORK4,Parm1              ;  —
        .AREPEAT \&POINT                  ;  —
        SHLL     Parm2                    ;  |  Corrects decimal point location.
        ROTCL    Parm1                    ;  |
        .AENDR                            ;  —
        SHLR     WORK1                    ;  —
        BF       MUL03                    ;; |  Adds the sign.
        NEG      Parm1,Parm1              ;  —
MUL03   RTS
        NOP
```

(Continued on following page.)

```
FIX_DIV:
        MOV      #0,WORK1             ;  →
        DIV0S    WORK1,Parm1          ;  | If dividend is a negative value,
        SUBC     WORK1,Parm1          ;  → converts to 1's complement.
        .AREPEAT \&POINT              ;  →
        SHAR     Parm1                ;  | Corrects decimal point location.
        ROTCR    WORK1                ;  |
        .AENDR                        ;  →
        DIV0S    Parm2,Parm1          ;
        .AREPEAT 32                   ;  →
        ROTCL    WORK1                ;  | Parm1:WORK1/Parm2 —> WORK1
        DIV1     Parm2,Parm1          ;  |
        .AENDR                        ;  →
        ROTCL    WORK1                ;
        MOV      #0,Parm1             ;  →
        ADDC     Parm1,WORK1          ;  | Converts to 2's complement.
        MOV      WORK1,Parm1          ;  →
        RTS
        NOP
```

Appendix C Assemble Listing Output Example

The assemble listing shows the result of the assemble processing. The assemble listing consists of a source program listing, a cross-reference listing, and a section information listing.

This appendix describes the content and output format of the assemble listing using the assembly of the source program shown below as an example. This uses the sample program shown in appendix B to calculate the following:

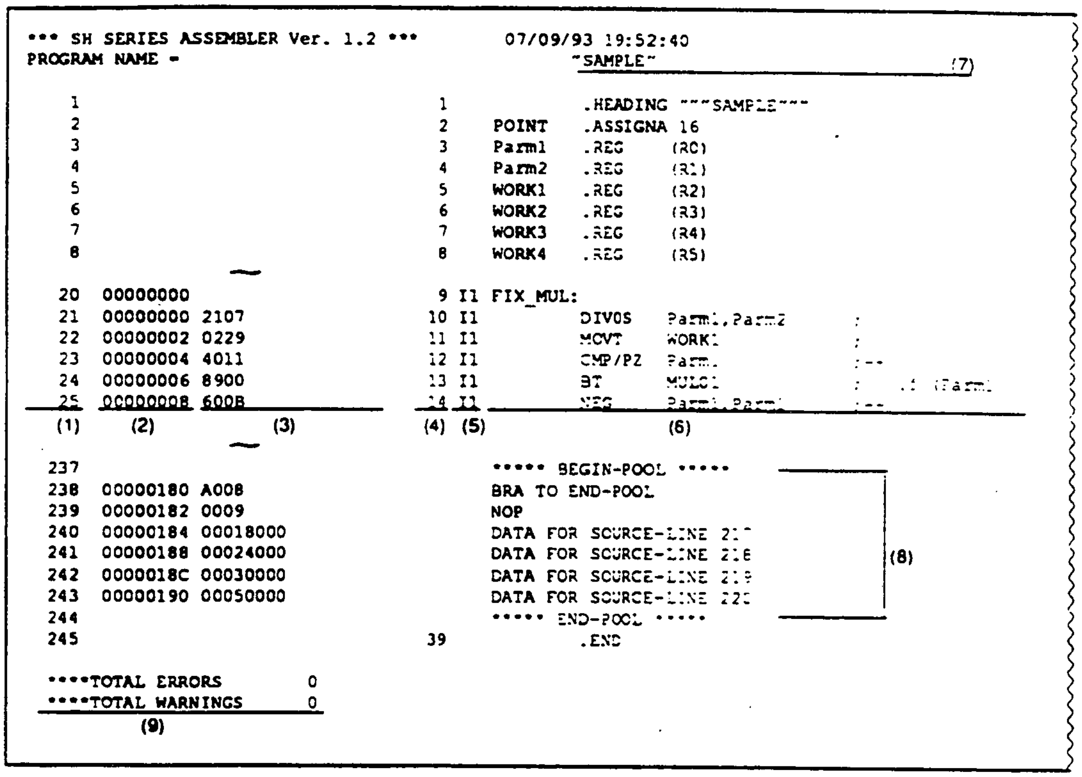
$$1.5 \times 2.25 + 3 + 5$$

```
POINT    .ASSIGNA 16
Parm1    .REG     (R0)
Parm2    .REG     (R1)
WORK1    .REG     (R2)
WORK2    .REG     (R3)
WORK3    .REG     (R4)
WORK4    .REG     (R5)

         .SECTION SAMPLE,CODE,ALIGN=4
         .INCLUDE "appendix B"

a        .REG     (R8)
b        .REG     (R9)
c        .REG     (R10)
d        .REG     (R11)

start
         STS      PR,@-SP
         MOV.L    #H'00018000,a
         MOV.L    #H'00024000,b
         MOV.L    #H'00030000,c
         MOV.L    #H'00050000,d

         MOV      a,Parm1
         MOV      b,Parm2
         BSR      FIX_MUL
         NOP
         MOV      Parm1,a
         MOV      c,Parm1
         MOV      d,Parm2
         BSR      FIX_DIV
         NOP
         FIX_ADD a
         MOV      Parm1,a
         LDS      @SP+,PR
         RTS
         NOP
         .END
```

C.1 Source Program Listing

The source program listing lists information related to the source statements, including the line number and the corresponding object code.

Figure C-1 shows an example of a source program listing.



```
*** SH SERIES ASSEMBLER Ver. 1.2 ***          07/09/93 19:52:40
PROGRAM NAME =                                 "SAMPLE"                    (7)

   1                          1          .HEADING """SAMPLE"""
   2                          2   POINT  .ASSIGNA 16
   3                          3   Parm1  .REG     (R0)
   4                          4   Parm2  .REG     (R1)
   5                          5   WORK1  .REG     (R2)
   6                          6   WORK2  .REG     (R3)
   7                          7   WORK3  .REG     (R4)
   8                          8   WORK4  .REG     (R5)

  20  00000000                9 I1 FIX_MUL:
  21  00000000 2107          10 I1       DIVOS   Parm1,Parm2      ;
  22  00000002 0229          11 I1       MOVT    WORK1            ;
  23  00000004 4011          12 I1       CMP/PZ  Parm1            ;--
  24  00000006 8900          13 I1       BT      MUL01            ;    [illegible] (Parm1
  25  00000008 600B          14 I1       NEG     Parm1,Parm1      ;--
 (1)     (2)        (3)      (4) (5)             (6)

 237                              ***** BEGIN-POOL *****
 238  00000180 A008               BRA TO END-POOL
 239  00000182 0009               NOP
 240  00000184 00018000           DATA FOR SOURCE-LINE 21[illegible]
 241  00000188 00024000           DATA FOR SOURCE-LINE 218               (8)
 242  0000018C 00030000           DATA FOR SOURCE-LINE 219
 243  00000190 00050000           DATA FOR SOURCE-LINE 220
 244                              ***** END-POOL *****
 245                         39          .END

 ****TOTAL ERRORS        0
 ****TOTAL WARNINGS      0
         (9)
```

Figure C-1 Source Program Listing Output Example

(1) Line numbers (in decimal)

(2) The value of the location counter (in hexadecimal)

(3) The object code (in hexadecimal). The size of the reserved area in bytes is listed for areas reserved with the .RES, .SRES, .SRESC, and .SRESZ assembler directives.

(4) Source line numbers (in decimal)

(5) Expansion type. Whether the statement is expanded by file inclusion, conditional assembly function, or macro function is listed.

In: File inclusion (n indicates the nest level).

C: Satisfied conditional assembly, performed iterated expansion, or satisfied conditional iterated expansion

M: Macro expansion

(6) The source statements
(7) The header setup with the .HEADING assembler directive.
(8) The literal pool
(9) The total number of errors and warnings. Error messages are listed on the line following the source statement that caused the error.

C.2 Cross-Reference Listing

The cross-reference listing lists information relating to symbols, including the attribute and the value.

Figure C-2 shows an example of a cross-reference listing.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***        07/09/93 19:52:40

*** CROSS REFERENCE LIST

NAME                                 SECTION  ATTR VALUE            SEQUENCE

FIX_DIV                              SAMPLE        00000088    94*   229
FIX_MUL                              SAMPLE        00000000    20*   224
MAN03                                         UDEF 00000000    89
MUL01                                SAMPLE        0000000A    24     26*
MUL02                                SAMPLE        00000010    27     29*
Parm1                                         REG               3*    21    23    25
                                                               37     37    39    41
                                                               69     71    73    75
                                                               96     97   102   104
                                                              122    124   126   128
                                                              150    152   154   156
                                                              174    176   178   180
                                                              198    200   202   204
Parm2                                         REG               4*    21    26    28
                                                               44     45    47    49
                                                               70     72    74    76
                                                              144    146   148   150
                                                              168    170   172   174
  ___________________________________  _______  ____ ________  ______________________
   (1)                                  (2)     (3)   (4)        (5)
```

Figure C-2 Cross-Reference Listing Output Example

(1) The symbol name
(2) The name of the section that includes the symbol (first eight characters)
(3) The symbol attribute

EXPT	Export symbol
IMPT	Import symbol
SCT	Section name
REG	Symbol defined with the .REG assembler directive
ASGN	Symbol defined with the .ASSIGN assembler directive
EQU	Symbol defined with the .EQU assembler directive
MDEF	Symbol defined two or more times
UDEF	Undefined symbol

No symbol attribute (blank)....A symbol other than those listed above

(4) The value of symbol (in hexadecimal)

(5) The list line numbers (in decimal) of the source statements where the symbol is defined or referenced. The line number marked with an asterisk is the line where the symbol is defined.

C.3 Section Information Listing

The section information listing lists information related to the sections in a program, including the section type and section size.

Figure C-3 shows an example of a section information listing.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***      07/09/93 19:52:40

*** SECTION DATA LIST

SECTION                                ATTRIBUTE   SIZE            START

SAMPLE                                 REL-CODE    00000194
  (1)                                    (2)         (3)             (4)
```

Figure C-3 Section Information Listing Output Example

(1) The section name

(2) The section type

REL	Relative address section
ABS	Absolute address section
CODE.................	Code section
DATA	Data section
COMMON	Common section
STACK..............	Stack section
DUMMY...........	Dummy section

(3) The section size (in hexadecimal, byte units)

(4) The start address of absolute address sections

Appendix D Error Messages

D.1 Error Types

(1) Command Errors

These are errors related to the command line that starts the assembler. These errors can occur, for example, in cases where there are errors in the source file or command line option specifications.

The assembler outputs the error message to standard error output (usually the display). The format of these messages is as follows:

```
<error number><message>
```

Example:

```
10 NO INPUT FILE SPECIFIED
```

(2) Source Program Errors

These are syntax errors in the source program.

The assembler outputs the error message to standard output (usually the display) or the source program listing. (If a source program listing is output during assembly, these messages are not output to standard output.)

The format of these messages is as follows:

```
"<source file name>",line <line number>: ERROR <error number>
"<source file name>",line <line number>: WARNING <error number>
```

Example:

```
"PROG.SRC",line 25: ERROR   300
"PROG.SRC",line 33: WARNING 811
```

The source program error numbers are classified as follows:

100's	General source program syntax errors
200's	Errors in symbols
300's	Errors in operations and/or operands
400's	Errors in expressions
500's	Errors in assembler directives
600's	Errors in file inclusion, conditional assembly, or macro function
800's	General source program warnings

(3) Fatal Errors

These are errors related to the assembler operating environment, and can occur, for example, if the available memory is insufficient.

The assembler outputs a message to standard error output. The format of these messages is as follows:

```
FATAL ERROR (<error number>)
```

Example:

```
FATAL ERROR (902)
```

Assembly processing is interrupted when a fatal error occurs.

D.2 Error Message Tables

Table D-1 Command Error Messages

10	Message:	NO INPUT FILE SPECIFIED
	Meaning:	There is no input source file specified.
	Recovery procedure:	Specify an input source file.
20	Message:	CANNOT OPEN FILE <file name>
	Meaning:	The specified file cannot be opened.
	Recovery procedure:	Check and correct the file name and directory.
30	Message:	INVALID COMMAND PARAMETER
	Meaning:	The command line options are not correct.
	Recovery procedure:	Check and correct the command line options.
40	Message:	CANNOT ALLOCATE MEMORY
	Meaning:	All available memory is used up during processing.
	Recovery procedure:	This error only occurs when the amount of available user memory is extremely small. If there is other processing occurring at the same time as assembly, interrupt that processing and restart the assembler. If the error still occurs, check and correct the memory management employed on the host system.
50	Message:	COMPLETED FILE NAME TOO LONG <file name>
	Meaning:	The file name including the directory is too long.
	Recovery procedure:	Shorten the total length of the file name and directory path.
	Supplement:	It is possible that the object module output by the assembler after this error has occurred will not be usable with the simulator/debugger.

Table D-2 Source Program Error Messages

General Source Program Syntax Errors		
100	Error description:	Too complex operation.
	Recovery procedure:	Simplify the expression for the operation.
101	Error description:	Syntax error in executable instruction source statement.
	Recovery procedure:	Check and correct the whole source statement.
102	Error description:	Syntax error in assembler directive source statement.
	Recovery procedure:	Check and correct the whole source statement.
103	Error description:	Program does not end with .END directive.
	Recovery procedure:	Add .END directive.
104	Error description:	The value of location counter exceeded its maximum value.
	Recovery procedure:	Reduce the size of the program.
105	Error description:	Executable instruction or assembler directive that reserves data in stack section.
	Recovery procedure:	Remove the instruction or directive in the stack section.
106	Error description:	Error display terminated due to too many errors.
	Recovery procedure:	Check and correct the whole source statement.
108	Error description:	Illegal continuation line.
	Recovery procedure:	Check and correct continuation line.
109	Error description:	The number of lines being assembled exceeded 65,535 lines.
	Recovery procedure:	Subdivide the program into multiple files.
150	Error description:	Illegal executable instruction placed following delayed branch instruction in memory.
	Recovery procedure:	Change the order of the instruction so that the instruction does not immediately follow a delayed branch instruction.
151	Error description:	Extended instruction placed following a delayed branch instruction in memory.
	Recovery procedure:	Place an executable instruction following the delayed branch instruction.
152	Error description:	Illegal boundary alignment value specified for a section including extended instructions.
	Recovery procedure:	Specify 2 or a larger multiple of 2 as a boundary alignment value.
153	Error description:	Executable or extended instruction placed at an odd address.
	Recovery procedure:	Place executable and extended instructions at even addresses.

Table D-2 Source Program Error Messages (cont)

Symbol Errors		
200	Error description:	Undefined symbol reference.
	Recovery procedure:	Define the symbol.
201	Error description:	Reserved word specified as symbol (or section name).
	Recovery procedure:	Correct the symbol or section name.
202	Error description:	Illegal symbol (or section name).
	Recovery procedure:	Correct the symbol or section name.
Operation and Operand Errors		
300	Error description:	Illegal operation.
	Recovery procedure:	Correct the operation.
301	Error description:	Too many operands of executable instruction, or illegal comment format.
	Recovery procedure:	Check and correct the operands and comment.
304	Error description:	Too few operands.
	Recovery procedure:	Correct the operands.
307	Error description:	Illegal addressing mode in operand.
	Recovery procedure:	Correct the operand.
308	Error description:	Syntax error in operand.
	Recovery procedure:	Correct the operand.
Expression and Operation Errors		
400	Error description:	Character constant is longer than 4 characters.
	Recovery procedure:	Correct the character constant.
402	Error description:	Operand value out of range for this instruction.
	Recovery procedure:	Change the value.
403	Error description:	Attempt to perform multiplication, division, or logic operation on relative value.
	Recovery procedure:	Correct the expression.
407	Error description:	Memory overflow during expression calculation.
	Recovery procedure:	Simplify the expression.
408	Error description:	Attempt to divide by 0.
	Recovery procedure:	Correct the expression.

Table D-2 Source Program Error Messages (cont)

409	Error description:	Register name in expression.
	Recovery procedure:	Correct the expression.
411	Error description:	STARTOF or SIZOF specifies illegal section name.
	Recovery procedure:	Correct the section name.
450	Error description:	Illegal displacement value. (Negative value is specified.)
	Recovery procedure:	Correct the displacement value.
452	Error description:	PC-relative data move instruction specifies illegal address for data area.
	Recovery procedure:	Access a correct address according to the instruction operation size. (4-byte boundary for MOV.L and MOVA, and 2-byte boundary for MOV.W.)
453	Error description:	More than 510 extended instructions exist that have not output literals.
	Recovery procedure:	Output literal pools using .POOL.
Assembler Directive Errors		
500	Error description:	Label not defined in directive that requires label.
	Recovery procedure:	Insert a label.
501	Error description:	Illegal specification of the start address or the value of location counter in section.
	Recovery procedure:	Correct the start address or value location counter.
502	Error description:	Illegal value (forward reference symbol, import symbol, or relative address symbol) specified in operand.
	Recovery procedure:	Correct the operand.
503	Error description:	Symbol declared for export symbol not defined in the file.
	Recovery procedure:	Define the symbol. Alternatively, remove the export symbol declaration.
504	Error description:	Illegal value (forward reference symbol or import symbol) specified in operand.
	Recovery procedure:	Correct the operand.
505	Error description:	Misspelled operand.
	Recovery procedure:	Correct the operand.
506	Error description:	Illegal element specified in operand.
	Recovery procedure:	Correct the operand.

Table D-2 Source Program Error Messages (cont)

508	Error description:	Operand value out of range for this directive.
	Recovery procedure:	Correct the operand.
510	Error description:	Illegal boundary alignment value.
	Recovery procedure:	Correct the boundary alignment value.
512	Error description:	Illegal execution start address.
	Recovery procedure:	Correct the execution start address.
513	Error description:	Illegal register name.
	Recovery procedure:	Correct the register name.
514	Error description:	Symbol declared for export symbol that cannot be exported.
	Recovery procedure:	Remove the declaration for the export symbol.
516	Error description:	Inconsistent directive specification.
	Recovery procedure:	Check and correct all related directives.
517	Error description:	Illegal value (forward reference symbol, an import symbol, or relative-address symbol) specified in operand.
	Recovery procedure:	Correct the operand.
518	Error description:	Symbol declared for import defined in the file.
	Recovery procedure:	Remove the declaration for the import symbol.
521	Error description:	.NOPOOL placed at illegal position.
	Recovery procedure:	Place .NOPOOL following a delayed branch instruction.
522	Error description:	.POOL placed following a delayed branch instruction.
	Recovery procedure:	Place an executable instruction following the delayed branch instruction.
File Inclusion, Conditional Assembly, and Macro Errors		
600	Error description:	Illegal character.
	Recovery procedure:	Correct it.
601	Error description:	Illegal delimiter character.
	Recovery procedure:	Correct it.
602	Error description:	Character string error.
	Recovery procedure:	Correct it.
603	Error description:	Source statement syntax error.
	Recovery procedure:	Reexamine the entire source statement.

Table D-2 Source Program Error Messages (cont)

604	Error description:	Illegal operand specified in a directive.
	Recovery procedure:	No symbol or location counter ($) can be specified as an operand of this directive.
610	Error description:	Macro name reused in macro definition (.MACRO directive).
	Recovery procedure:	Correct the macro name.
611	Error description:	Macro name not specified (.MACRO directive).
	Recovery procedure:	Specify a macro name in the name field of the .MACRO directive.
612	Error description:	Macro name error (.MACRO directive).
	Recovery procedure:	Correct the macro name.
613	Error description:	.MACRO directive appears in macro body (between .MACRO and .ENDM directives), between .AREPEAT and .AENDR directives, or between .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .MACRO directive.
614	Error description:	Identical formal parameters repeated in formal parameter declaration in macro definition (.MACRO directive).
	Recovery procedure:	Correct the formal parameters.
615	Error description:	.END directive appears in macro body (between .MACRO and .ENDM directives).
	Recovery procedure:	Remove the .END directive.
616	Error description:	An .ENDM directive appears without a preceding .MACRO directive, or an .EXITM directive appears outside of a macro body (between .MACRO and .ENDM directives), outside of .AREPEAT and .AENDR directives, or outside of .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .ENDM or .EXITM directive.
618	Error description:	Line with over 255 characters generated by macro expansion.
	Recovery procedure:	Correct the definition or call so that the line is less than or equal to 255 characters.
619	Error description:	Macro parameter name error in macro call, or error in formal parameter in a macro body (between .MACRO and .ENDM directives).
	Recovery procedure:	Correct the formal parameter.
	Supplement:	When there is an error in a formal parameter in a macro body, the error will be detected and flagged during macro expansion.
620	Error description:	Reference to an undefined preprocessor variable.
	Recovery procedure:	Define the preprocessor variable.

Table D-2 Source Program Error Messages (cont)

621	Error description:	.END directive in macro expansion.
	Recovery procedure:	Remove the .END directive.
622	Error description:	Matching parenthesis missing in macro processing exclusion.
	Recovery procedure:	Add the missing macro processing exclusion parenthesis.
623	Error description:	Syntax error in character string manipulation function.
	Recovery procedure:	Correct the character string manipulation function.
624	Error description:	Too many macro parameters for positional specification in macro call.
	Recovery procedure:	Correct the number of macro parameters.
630	Error description:	Syntax error in structured assembly directive operand.
	Recovery procedure:	Reexamine the directive.
631	Error description:	Terminating preprocessor directive does not agree with matching directive.
	Recovery procedure:	Reexamine the preprocessor directives.
640	Error description:	Syntax error in conditional assembly directive operand.
	Recovery procedure:	Reexamine the entier source statement.
641	Error description:	Error in conditional assembly directive relational operator.
	Recovery procedure:	Correct the relational operator.
642	Error description:	.END directive appears between .AREPEAT and .AENDR directives or between .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .END directive.
643	Error description:	.AENDR or .AENDW directive does not form a proper pair with .AREPEAT or .AWHILE directive.
	Recovery procedure:	Re-examine the preprocessor directives.
644	Error description:	.AENDW or .AENDR directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .AENDW or .AENDR directive.
645	Error description:	Line with over 255 characters generated by .AREPEAT or .AWHILE expansion.
	Recovery procedure:	Correct the .AREPEAT or .AWHILE to generate lines of less than or equal to 255 characters.
650	Error description:	Error in .INCLUDE file name.
	Recovery procedure:	Correct the file name.
651	Error description:	Could not open .INCLUDE file.
	Recovery procedure:	Correct the file name.

Table D-2 Source Program Error Messages (cont)

652	Error description:	File inclusion nesting exceeded 8 levels.
	Recovery procedure:	Limit the nesting to 8 or fewer levels.
653	Error description:	Syntax error in .INCLUDE operand.
	Recovery procedure:	Correct the operand.
660	Error description:	Missing .ENDM directive following .MACRO.
	Recovery procedure:	Insert an .ENDM directive.
662	Error description:	.END directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .END directive.
663	Error description:	.END directive appears in included file.
	Recovery procedure:	Remove the .END directive.
664	Error description:	.END directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .END directive.
General Source Program Warnings		
800	Error description:	A symbol exceeded 32 characters.
	Recovery procedure:	Correct the symbol.
	Supplement:	The assembler ignores the characters starting at the 33rd character.
801	Error description:	Symbol already defined.
	Recovery procedure:	Remove the symbol redefinition.
	Supplement:	The assembler ignores the second and later definitions.
807	Error description:	Illegal operation size.
	Recovery procedure:	Correct the operation size.
	Supplement:	The assembler ignores the incorrect operation size specification.
808	Error description:	Illegal notation of integer constant.
	Recovery procedure:	Correct the notation.
	Supplement:	The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.
810	Error description:	Too many operands or illegal comment format.
	Recovery procedure:	Correct the operand or the comment.
	Supplement:	The assembler ignores the extra operands.
811	Error description:	Specified label in assembler directive that cannot have a label.
	Recovery procedure:	Remove the label specification.
	Supplement:	The assembler ignores the label.

Table D-2 Source Program Error Messages (cont)

812	Error description:	Section or object module name exceeded 32 characters.
	Recovery procedure:	Correct the section or object module name.
	Supplement:	The assembler ignores the 33rd and later characters.
813	Error description:	A different section type is specified on section restart (reentry), or, a section start address is respecified at the restart of absolute section.
	Recovery procedure:	Do not respecify the section type or start address on section reentry.
	Supplement:	The specification of starting section remains valid.
815	Error description:	Respecification of object module name.
	Recovery procedure:	Specify the object module name once in a program.
	Supplement:	The assembler ignores the second and later object module name specifications.
816	Error description:	Illegal allocation of data or data area.
	Recovery procedure:	Locate the word data or data area on the even address. Locate the long word data or data area on an address of a multiple of 4.
	Supplement:	The assembler corrects the location of the data or data area according to the size of it.
817	Error description:	A boundary alignment value less than 4 specified for a code section.
	Recovery procedure:	The specification is valid, but if an executable instruction or extended instruction is located at an odd address, error 153 occurs.
	Supplement:	Special care must be taken when specifying 1 for code section boundary alignment value.
825	Error description:	Executable instruction or assembler directive that reserves data or data area in dummy section.
	Recovery procedure:	Remove the instruction or directive.
	Supplement:	The assembler ignores the instruction or directive.
832	Error description:	Symbol P already defined before a default section is used.
	Recovery procedure:	Do not define P as a symbol if a default section is used.
	Supplement:	The assembler regards P as the name of the default section, and ignores other definitions of the symbol P.
835	Error description:	Operand value out of range for this instruction.
	Recovery procedure:	Correct the value.
	Supplement:	The assembler generates object code with a value corrected to be within range.

Table D-2 Source Program Error Messages (cont)

836	Error description:	Illegal notation of integer constant.
	Recovery procedure:	Correct the notation.
	Supplement:	The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.
837	Error description:	The length of a source statement exceeded 255 bytes.
	Recovery procedure:	Rewrite the source statement to be within 255 bytes by, for example, rewriting the comment. Alternatively, rewrite the statement as a multi-line statement.
	Supplement:	The assembler ignores byte number 256, and regards the characters starting at byte 257 as the next statement.
850	Error description:	Symbol specified in label field.
	Recovery procedure:	Remove the symbol.
851	Error description:	Macro generation counter exceeded 99999.
	Recovery procedure:	Reduce the number of macro calls.
852	Error description:	Characters appear after the operands.
	Recovery procedure:	Correct the operand(s).
870	Error description:	Illegal displacement value. (Either the displacement value is not an even number when the operation size is word, or the displacement value is not a multiple of 4 when the operation size is long word.)
	Recovery procedure:	Take account of the fact that the assembler corrects the displacement value.
	Supplement:	The assembler generates object code with the displacement corrected according to the operation size. (For a word size operation the assembler discards the low order bit of the displacement to create an even number, and for a long word size operation the assembler discards the two low order bits of the displacement to create a multiple of 4.)
871	Error description:	Executable instruction with PC relative addressing mode operand is located following delayed branch instruction.
	Recovery procedure:	Take account of the fact that the value of PC is changed by a delayed branch instruction.
	Supplement:	The assembler generates object code exactly as specified in the program.

Table D-2 Source Program Error Messages (cont)

872	Error description:	Executable instruction is located on the odd address in absolute address section.
	Recovery procedure:	Locate the instruction on the even address.
	Supplement:	The assembler only outputs this message for the first illegal instruction in the section.
874	Error description:	Cannot check data area boundary for PC-relative data move instructions.
	Recovery procedure:	Note carefully the data area boundary at linkage process.
	Supplement:	The assembler only outputs this message when a data move instruction is included in a relative section, or when an import symbol is used to indicate a data area.
875	Error description:	Cannot check displacement size for PC-relative data move instructions.
	Recovery procedure:	Note carefully the distance between data move instructions and data area.
	Supplement:	The assembler only outputs this message for the first illegal instruction in the section.
876	Error description:	The assembler automatically outputs a BRA instruction.
	Recovery procedure:	Specify a literal pool output position using .POOL, or check that the program to which a BRA instruction is added can run normally.
	Supplement:	When a literal pool output location is not available, the assembler automatically outputs literal pool and a BRA instruction to jump over the literal pool.

Table D-3 Fatal Error Messages

901	Error description:	Source file input error.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
902	Error description:	Insufficient memory. (Unable to process the temporary information.)
	Recovery procedure:	Subdivide the program.
903	Error description:	Output error on the list file.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
904	Error description:	Output error on the object file.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
905	Error description:	Insufficient memory. (Unable to process the line information.)
	Recovery procedure:	Subdivide the program.
906	Error description:	Insufficient memory. (Unable to process the symbol information.)
	Recovery procedure:	Subdivide the program.
907	Error description:	Insufficient memory. (Unable to process the section information.)
	Recovery procedure:	Subdivide the program.
908	Error description:	The number of sections exceeded 65,535.
	Recovery procedure:	Subdivide the program.
909	Error description:	The number of symbols exceeded 65,535.
	Recovery procedure:	Subdivide the program.
910	Error description:	The number of source program lines exceeded 65,535.
	Recovery procedure:	Subdivide the program.
911	Error description:	The number of import symbols exceeded 65,535.
	Recovery procedure:	Reduce the number of import symbols.
912	Error description:	The number of export symbols exceeded 65,535.
	Recovery procedure:	Reduce the number of export symbols.
950	Error description:	Insufficient memory.
	Recovery procedure:	Separate the source program.
951	Error description:	More than 16 sections exist that have not output literal pools.
	Recovery procedure:	Output literal pools using .POOL before terminating section processing.

Please contact your Hitachi, Ltd., sales representative if a problem cannot be resolved using the indicated recovery procedure, or if an error message that does not appear in the manual is displayed.

Appendix E ASCII Code Table

Table E-1 ASCII Code Table

	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	/	?	O	_	o	DEL

(This page intentionally left blank.)

Index